Orsay
N^o d'ordre:

UNIVERSITE PARIS XI
UFR SCIENTIFIQUE D'ORSAY

THESE

présentée pour obtenir

Le GRADE de DOCTEUR EN SCIENCES

DE L'UNIVERSITÉ PARIS XI ORSAY

SPÉCIALITÉ : INFORMATIQUE

PAR

Frédéric Magniette

Sujet : **Preuves d'algorithmes auto-stabilisants**

Rapporteurs :

Mme	Carole	DELPORTE
M	Marc	BUI

Soutenue le 19 juin 2002 devant la Commission d'examen :

Membres du jury :	M.	Joffroy	BEAUQUIER (Directeur)
	M.	Marc	BUI
	Mme	Carole	DELPORTE
	M.	Laurent	FRIBOURG (Codirecteur)
	Mme	Laurence	PUEL
	M.	André	SCHIPER
Invité :	M.	Ajoy	DATTA

Table des matières

1 Introduction **9**

1.1 Cadre général 9

1.2 Problématique 10

1.2.1 Preuves par fonction décroissante 11

1.2.2 Preuves par attracteurs 11

1.2.3 Simplification des méthodes classiques 12

1.3 Automatisation des preuves 13

1.3.1 Le model-checking 13

1.3.2 Les systèmes de vérification formelle 14

1.4 Contributions 15

1.5 Plan du mémoire 16

2 Préliminaires **17**

2.1 Systèmes répartis 17

2.1.1 Caractéristiques des processus 18

2.1.2 Caractéristiques des liens de communications 19

2.1.3 Caractéristiques liées aux vitesses relatives 19

2.2 La tolérance aux défaillances 20

2.2.1 La classification des défaillances 20

2.2.2 L'approche classique 21

2.3 L'auto-stabilisation 21

3 Modèle **23**

3.1 Références 23

3.2 Notions préliminaires 23

3.2.1 Automates et Langages 24

3.2.2 Produit synchronisé d'automates 25

3.3 éléments de base 26

3.3.1 Processus 26

3.3.2 Liens 28

3.3.3 La lecture d'état 28

3.3.4 Le passage de message 28

3.4 Systèmes répartis 29

3.5 Validation du modèle 33

3.5.1 Algorithme 33

3.5.2 Topologie 34

3.5.3 Passage du modèle automate au modèle topologie/algorithme 36

3.6 Propriétés des exécutions 36

3.6.1 Non-déterminisme 36

3.6.2 Equité 37

3.6.3 Equité des communications 37

3.6.4 Atomicité 38

3.7 Démon 38

3.8 Auto-stabilisation 40

3.9 Résumé 41

4 LME **43**

4.1 Références 43

4.2 Définitions 44

4.2.1 Orientation Virtuelle 45

4.2.2 Renversement d'arêtes 45

4.3 ULME 47
4.3.1 Algorithme ULME 47
4.3.2 Lemmes préliminaires 49
4.3.3 Preuve de l'algorithme ULME 50
4.3.4 Index d'équité et temps de service 51
4.4 BLME 53
4.4.1 Algorithme BLME 55
4.4.2 Principe de l'algorithme 56
4.4.3 Preuve de l'algorithme 57
4.5 Composition croisée 61
4.6 Résumé 63

5 Réécriture **65**

5.1 Modélisation 65
5.1.1 Les processus 66
5.1.2 Les transitions 66
5.1.3 Réduction close 67
5.1.4 Une caractérisation de l'auto-stabilisation pour les systèmes clos 68
5.2 Premier Ordre 69
5.2.1 Unification 69
5.2.2 Réduction minimale 70
5.2.3 Chaînes de réductions minimales dans Top_S 71
5.3 Auto-stabilisation 72
5.4 La φ-optimisation 75
5.5 Schémas 76
5.5.1 Génération des graphes de dérivations généralisés 78
5.5.2 Les cycles finis 80
5.5.3 Exhibition de contre-exemples 81
5.6 Exemples 81

5.6.1 L'algorithme Beauquier-Debas 81
5.6.2 L'algorithme à quatre états de Ghosh 82
5.6.3 L'algorithme d'orientation d'anneau de Hoepman 85
5.7 Résumé 87

6 Poulet **89**
6.1 Introduction 89
6.2 Présentation du logiciel 90
6.3 Structures 90
6.3.1 Graphes 90
6.3.2 Automates 94
6.3.3 Expressions régulières 96
6.4 Règles de réécriture 97
6.4.1 Le format des règles 97
6.4.2 Traduire les règles 97
6.4.3 Exemple de l'algorithme à quatre états de Gosh 98
6.5 Les structures de stockage 99
6.5.1 La table des objets nommés 99
6.5.2 La pile 100
6.5.3 La liste des cycles 100
6.6 Algorithmes 101
6.6.1 La fonction d'initialisation 101
6.6.2 La fonction "next" 101
6.6.3 La fonction "addreg" 102
6.6.4 Les fonctions de calcul des cycles 102
6.7 Prouver un algorithme 102
6.8 Fonctionnalités additionnelles 108
6.8.1 Production de Coq 110
6.8.2 Sauvegarde de contexte 110

6.8.3 Les scripts 110
6.8.4 Oracle de schémas 111
6.9 Résumé 112

7 Conclusion 113
7.1 Passage de messages 114
7.1.1 La taille de l'anneau 114
7.1.2 Modélisation des transitions 116
7.1.3 Système abstrait 116
7.1.4 Preuve de l'auto-stabilisation 117
7.1.5 Conclusion 119
7.2 Autres Topologies 119
7.3 Les perspectives de Poulet 121

Chapitre 1

Introduction

1.1 Cadre général

Un très grand nombre de systèmes informatiques sont composés de plusieurs ordinateurs reliés par des moyens de communication rapides. En effet, le développement des grands réseaux de données a facilité l'inter-connection des machines au sein des grands parcs informatiques. Ces systèmes, appelés systèmes répartis, permettent de faire des calculs avec de très bonnes performances car les ordinateurs qui participent se divisent le travail et coopèrent via le réseau.

Toutefois, les systèmes d'exploitation classiques ne sont pas conçus pour supporter ce genre de comportement coopératif. C'est pourquoi, il est nécessaire d'utiliser une algorithmique dédiée au calcul réparti, pour pouvoir garantir que les calculs seront effectués sans problèmes. C'est pourquoi, l'algorithmique distribuée est un sujet très étudié aujourd'hui.

Les grands réseaux comme l'Internet sont entièrement basés, au niveau des protocoles, sur des mécanismes de type client-serveur. C'est-à-dire que des relations privilégiées s'établissent entre la machine qui demande un service et celle qui fournit ce service. Ce type d'architecture n'est pas optimal dans bien des cas. En effet, il est fréquent que les solutions centralisées entraînent des congestions si les services sont très sollicités. De plus, la résistance au pannes est quasiment nulle dans ce genre d'architecture. C'est pourquoi, il est souvent intéressant de considérer des solutions complètement réparties, c'est-à-dire, où les machines du système ont un rôle symétrique. Cette voie de recherche est explorée dans la majorité des projets de distribution de calculs sur l'Internet. Au niveau des protocoles à mettre en oeuvre, cela implique qu'on ne peut plus se baser sur un ensemble de connections asymétriques mais qu'il est nécessaire de considérer toutes les connections inter-machines pour concevoir ou prouver des algorithmes.

De plus, l'interaction du système avec son environnement induit une difficulté supplémentaire. En effet, l'asynchronisme des systèmes conduit à un non-déterminisme intrinsèque qui rend très difficile la modèlisation des comportements des systèmes. Les processus participants aux calculs n'exécutent pas leurs instructions à la même vitesse et les liens de communications ont des délais variables pour l'acheminement des messages. Par conséquent, il est nécessaire de considérer toutes les exécutions possibles du système en les considérant comme possibles. Le problème est que rien ne garantit un ordre quelconque sur les différentes actions des machines et par conséquent, ce nombre d'exécutions peut être très grand, voire infini. C'est une fonction qui croît exponentiellement en fonction du nombre de machines participantes et en fonction de la complexité des algorithmes (nombre de variables et nombre d'actions possibles). Lorsque l'on veut vérifier de tels algorithmes, on va donc se heurter à un problème connu sous le nom d'explosion combinatoire. Il est donc nécessaire de chercher des solutions pour simplifier la conception et la preuve d'algorithmes répartis.

Dans le cadre des systèmes qui peuvent tomber en panne, le problème est encore accru. En effet, les défaillances induisent des dysfonctionnements tant au niveau des processus que des liens de communications, ce qui augmente encore le non-déterminisme. La question que nous nous sommes posée est : comment simplifier la conception et la preuve d'algorithmes dans ce type d'environnement?

Nous avons développé deux approches pour y répondre : l'une est de composer les algorithmes avec des modules qui permettent de restreindre le non-déterminisme inhérent au système. La seconde approche est une tentative d'automatisation des preuves via des systèmes formalisés.

Nous nous sommes intéressé à ces questions dans un cadre particulier : les systèmes corruptibles. Ces systèmes sont sujets à des défaillances très graves qui peuvent corrompre à la fois les données manipulées par les processus mais également le contenu des canaux de communications.

1.2 Problématique

Dans les systèmes corruptibles, on considère que les corruptions (les plus graves des défaillances) ne se produisent que rarement. En effet, il est facile de montrer que si ces corruptions ont une fréquence élevée, tout calcul réparti est impossible.

L'auto-stabilisation est une propriété des algorithmes qui garantit un fonctionnement correct au bout d'un temps fini dans ce type de systèmes. L'idée est la suivante : la corruption de la mémoire place le système dans un état indéfini. On va donc essayer de concevoir des algorithmes qui vont converger vers un comportement stable au bout d'un temps fini. Un tel algorithme est défini par rapport à un ensemble d'états qui sont dits légitimes. Dans la phase qui suit la corruption, appelée phase de convergence, le système va osciller entre plusieurs états illégitimes. A la fin de cette phase, il va atteindre un état légitime. Il entre alors dans la phase dite de correction où il va respecter sa spécification, c'est-à-dire résoudre sans erreur le problème spécifié par son algorithme.

Les algorithmes auto-stabilisants se prouvent en deux phases : la phase de convergence et la phase de correction. Prouver la correction est une chose classique en algorithmique répartie : on peut par exemple, utiliser des méthodes d'invariants ou plus simplement si on considère la clôture aux états légitimes, une simple étude des transitions peut prouver la clôture de ces états. Dans le cas ou la clôture n'est pas suffisante, on prouve deux propriétés : la vivacité et la sûreté. La vivacité est la propriété qui traduit l'absence de blocage dans le calcul. La sûreté est la propriété qui garanti que le prédicat est toujours vérifié (dans la phase convergée). Ces propriétés sont en général prouvées par des méthodes d'invariants qui consistent à exhiber une expression qui reste vraie quelle que soit l'exécution. En revanche, la preuve de convergence est plus difficile, il est nécessaire de montrer qu'il n'existe pas d'exécution ne contenant que des états illégitimes. La méthode utilisée la plus fréquemment est celle de la fonction décroissante ([Kes88]) ou sa généralisation : la technique des attracteurs ([GM91]).

1.2.1 Preuves par fonction décroissante

Pour démontrer la convergence, l'idée est d'associer une mesure M à chaque état. Cette mesure doit être nulle (ou minimale) pour les états légitimes et positive pour les états illégitimes. Si cette mesure est bien choisie, on peut prouver que pour toute transition du système $e_1 \to e_2$, $M(e_1) > M(e_2)$. Ainsi on prouve que la mesure décroît strictement au long des exécutions qui commencent par un état illégitime et par conséquent qu'un état légitime est toujours atteint puisqu'il possède la mesure minimale du système. Cela revient donc à trouver un ordre noetherien (celui des entiers positifs) et à l'appliquer sur les états illégitimes, ce qui est très classique dans les preuves de convergence (en théorie des langages notamment). Le problème de cette méthode est que la mesure dépend très fortement de l'algorithme et par conséquent, la méthode ne présente aucune généricité. De plus, il est souvent difficile de trouver une telle mesure, surtout si l'algorithme est compliqué (il possède beaucoup de variables).

1.2.2 Preuves par attracteurs

La méthode des attracteurs est une généralisation de la méthode de la fonction décroissante qui s'applique lorsqu'il est trop difficile de trouver intuitivement une mesure. Un attracteur est un ensemble de configurations tel qu'à partir d'un autre ensemble de configurations, toute exécution atteint les éléments de l'attracteur. Cela permet de découper la convergence en une suite d'ensembles qu'il faut atteindre successivement.

Par exemple, la figure 1.1 représente une preuve par attracteur, les ensembles de configurations L_1, L_2, ..., L_{n-1} sont les buts intermédiaires de la preuve. Ln est le but final : c'est l'ensemble des états légitimes. La preuve est découpée en $n-1$ sous-preuves qui démontrent qu'à partir des états de L_i on atteint forcément un état de L_{i+1} au bout d'un temps fini pour i allant de 1 à $n-1$. Ceci se prouve via $n-1$ mesures qui sont plus simples qu'une mesure globale.

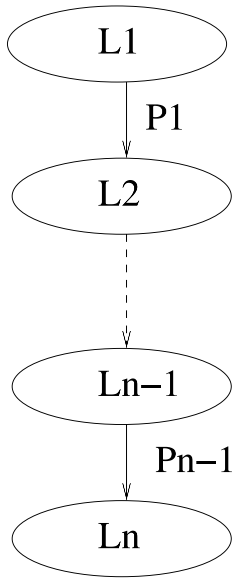

FIG. 1.1 – *Exemple de preuve par attracteur*

1.2.3 Simplification des méthodes classiques

L'inconvénient majeur de ce type de preuve est sa difficulté et son manque de généricité. En effet, il peut se révéler très ardu de trouver une ou plusieurs mesures pour faire la preuve de convergence. Dans le cas où l'algorithme manipule beaucoup de variables où beaucoup de communications différentes, il faut prendre en compte tous les comportements possibles du système pour découvrir l'élément clé de la mesure. De plus, ces techniques ne présentent aucune généricité : c'est-à-dire qu'il est impossible de réutiliser les mesures pour un autre algorithme. Cela vient de la dépendance très forte entre l'algorithme proprement dit et la mesure. En effet, toute mesure se base sur les valeurs possibles de différentes variables et de leur évolution au cours du temps les unes par rapport aux autres. Cette évolution est précisément ce qui différencie les algorithmes entre eux. Il en résulte que les mesures sont toujours ad'hoc et donc ne peuvent en aucun cas être réutilisées.

Dans un premier temps, nous avons donc cherché des solutions pour simplifier ce processus de recherche de mesures. Une des solutions envisagées est de contraindre de manière automatique l'environnement (que nous appelons le démon) pour qu'il fournisse des exécutions avec des propriétés particulières qu'il devient inutile d'exprimer par le biais des mesures. En effet, comme nous l'avons dit, les mesures sont liées à l'évolution des valeurs des différentes variables. Dans la cas d'un démon fort, comme par exemple le démon distribué, il est nécessaire de considérer des actions simultanées sur tous les processus en même temps. L'utilisation d'un démon faible, comme par exemple le démon centralisé, où les actions des processus sont séquentialisées, simplifie grandement les preuves en fournissant un cadre où les actions peuvent être considérées une à une. Or, dans un système réparti, le démon que l'on considère est généralement un démon fort, ce qui correspond mieux aux exigences de l'informatique pratique. Par conséquent, il est intéressant de trouver des moyens automatiques de transformation pour qu'un algorithme quelconque fonctionnant sous un démon faible puisse fonctionner avec la même spécification sous un démon fort. Cette transformation est généralement envisagée comme une composition de l'algorithme original (que l'on appelle module faible) avec un module fort qui est en général un algorithme d'exclusion mutuelle qui simule le démon faible.

Cette idée de composition de modules a été introduite dans [GH91] par Gouda et Herman puis par Varghese dans [Var97]. La composition décrite dans ces articles est dite "par selection". Les modules composés sont indépendants mais peuvent modifier les mêmes variables de sortie. La sélection se fait par un prédicat qui vaut vrai pour le module qui a le droit d'écriture sur les variables et faux sur les autres. Dans [BGJ99], les auteurs introduisent un autre type de composition : la composition croisée qui est asymétrique et permet au module faible d'être commandé par le module fort. C'est-à-dire que le module fort décide quand le module faible exécute ses actions ce qui simule un démon de façon fonctionnelle.

1.3 Automatisation des preuves

Une autre idée pour simplifier les preuves des algorithmes auto-stabilisants est d'automatiser au maximum les processus de preuves. Nous avons vu que les méthodes basées sur des mesures sont très difficiles à automatiser à cause du manque de généricité des mesures. Par conséquent, il est nécessaire de chercher dans la direction des méthodes énumératives classiques. En effet, une manière standard de montrer des propriétés sur des exécutions est d'énumérer de façon plus ou moins complète les configurations atteignables. Il existe de nombreux outils qui font ce genre de vérification pour les systèmes répartis : leurs différences proviennent essentiellement des formalismes utilisés qui influent sur le type de propriétés qu'il est possible de démontrer.

1.3.1 Le model-checking

Le model-checking est une technique qui consiste à contruire le graphe associé au système. Ce graphe est une représentation de tous les états globaux du système et des transitions entre ces états. Suivant les outils, le graphe peut être construit complètement ou partiellement et par conséquent, il peut être exploré exhaustivement ou selectivement.

Le model-checking est basé sur une structure de représentation qui modélise le système réparti. On trouve fréquemment trois types de structures : les structures d'évènements, les systèmes de transitions et les réseaux de Petri.

Il existe de nombreux outils qui implémentent du model-checking : Cadp(lotos), Spin et SpinULG (promela), ObjectGeode(sdl), Marella (estelle), PeP(réseaux de Petri). Le nom entre parenthèses désigne le langage de spécification utilisé pour décrire le système avant la preuve. Les prédicats à vérifier sont généralement exprimés dans des logiques modales avec des opérateurs temporels.

Cette technique est très performante pour démontrer les propriétés de sûreté et de vivacité mais elle présente néanmoins des défauts majeurs pour le problème qui nous intéresse. Le problème principal auquel nous sommes confrontés est l'explosion combinatoire du nombre d'états. En effet, la représentation des exécutions possibles a une taille exponentielle en fonction du nombre de processus du système. Le choix de l'état initial du système, qui peut parfois limiter cette explosion combinatoire, n'est pas possible dans le cadre de l'auto-stabilisation. De plus, contrairement à la méthode des mesures décroissantes, la preuve est limitée à des configurations avec un nombre de processus fixé à l'avance.

1.3.2 Les systèmes de vérification formelle

Les systèmes de vérification formelle sont des moteurs d'inférences qui permettent de prouver des prédicats logiques. Après une modélisation du système, c'est à dire une description de l'espace des états dans lequel évolue le système, on peut étudier des propriétés sur celui-ci, notamment les propriétés en rapport avec le problème considéré. Les programmes qui implémentent ces méthodes formelles permettent de faire des preuves automatiques à partir d'une spécification logique du problème et de leurs bibliothèques d'axiomes.

Il est possible d'utiliser de tels outils pour faire des preuves associées à des systèmes répartis. Il suffit pour cela de trouver un formalisme adapté et de pouvoir exprimer les prédicats recherchés dans la logique correspondante.

Coq

Coq est un assistant interactif de preuve pour le calcul sur les constructions inductives. Il permet de faire des preuves mathématiques basées sur la théorie des types et plus précisément sur l'isomorphisme de Curry-Howard qui permet d'associer des types sur des objets avec des preuves sur ces mêmes objets. Il utilise un grand nombre de bibliothèques qui permettent de manipuler des objets différents depuis de simples entiers jusqu'à des programmes impératifs. A ces objets sont associés des tactiques de preuves pour simplifier le travail de l'opérateur.

C'est un programme qui fonctionne sur de nombreuses architectures et surtout sur les machines UNIX. Il peut être utilisé avec de nombreuses interfaces utilisateur comme par exemple, l'interface pour GNU Emacs. Il est basé sur un langage de spécification, nommé Gallina, qui permet de développer des axiomatisations formelles et sur un noyau qui calcule les inductions. Il peut être trouvé à l'adresse suivante : `http://pauillac.inria.fr/coq/`

Il est possible d'utiliser Coq pour faire des preuves sur des algorithmes répartis. Dans [Cou02], Pierre Courtieu a démontré que notre méthode de preuves exposée dans le chapitre 5 était valide. De plus, il a écrit une base générique de preuves de l'auto-stabilisation pour Coq qui permet de certifier les résultats fournis par notre programme Poulet exposé dans le chapitre 6.

PVS

PVS (Prototype Verification System) est un programme qui fournit un environnement pour le développement et l'analyse de spécifications formelles. Il permet de créer et d'analyser des théories et des preuves. On peut le trouver à l'adresse suivante : `http://pvs.csl.sri.com`

De même que Coq, PVS peut être utilisé pour faire des preuves d'algorithmes répartis. Dans [Rus00], John Rushby montre comment montrer des propriétés sur les systèmes répartis en utilisant des systèmes à induction. Ces systèmes permettent de montrer des propriétés de sûreté exprimées sous la forme de prédicats sur les exécutions induites.

1.4 Contributions

Dans cette thèse, nous présentons nos contributions dans le domaine de la simplification et de l'automatisation des preuves d'algorithmes auto-stabilisants.

Afin de simplifier la conception et la preuve d'algorithmes, nous présentons un algorithme d'exclusion mutuelle locale. Cet algorithme peut etre composé avec n'importe quel autre algorithme auto-stabilisant sous le démon quasi-central au moyen de la technique de composition croisée afin de le rendre auto-stabilisant sous un démon distribué. De plus, la composition peut être faite à un niveau d'atomicité très fin (atomicité lecture/écriture). Ce travail a donné lieu à une publication : [BDGM00] à la conférence DISC'2000 puis à une version journal [BDGM02] dans le Chicago Journal of Theoritical Computer Science.

Dans le domaine de l'automatisation des preuves d'algorithmes auto-stabilisants, nous avons mis au point une méthode de preuves pour les topologies linéaires qui n'utilise pas de fonction strictement décroissante. Cette méthode s'appuie sur l'analogie entre une topologie linéaire et un mot. L'algorithme est alors vu comme un système de réécriture sur des mots dont on peut prouver la terminaison en utilisant des résultats connus sur la surréduction. Cette méthode a donné lieu à une publication : [BBFM01] dans le journal Distributed Computing.

Cette technique est assez facile à automatiser. Nous avons tout d'abord tenté d'utiliser l'assistant de preuve Coq pour automatiser les preuves mais Coq n'est pas muni de librairies sur les expressions régulières, ce qui empèche une automatisation complète. Nous avons donc développé un outil : Poulet qui automatise la partie inférence des calculs.

1.5 Plan du mémoire

Ce mémoire est organisé en six chapitres.

Le chapitre 2 (Préliminaires) est un aperçu du domaine de l'algorithmique répartie.

Dans le chapitre 3 (Modèle), les concepts évoqués dans l'aperçu sont formalisés de manière à fournir un cadre théorique aux résultats présentés dans les chapitres suivants.

Le chapitre 4 (Exclusion mutuelle locale) présente deux algorithmes d'exclusion mutuelle locale. Ce travail permet, par composition croisée, de fournir à des algorithmes auto-stabilisants un démon quasi-central.

Le chapitre 5 (Réécriture) présente une nouvelle méthode de preuve des algorithmes auto-stabilisants. Nous n'y considérons que les topologies linéaires (anneaux et chaînes) et nous proposons une méthode de preuve qui n'est basée que sur la mesure induite par les mots formés par les configurations. Cette étude théorique est complétée par des exemples tirés de la littérature.

Enfin, le chapitre 6 (Poulet) présente le logiciel qui implémente la méthode de preuve décrite au chapitre 5. Ce programme est un prouveur semi-automatique interactif d'algorithmes auto-stabilisants. Il permet d'automatiser les phases de génération évitant ainsi un travail fastidieux et supprimant au maximum les sources d'erreurs.

Enfin, le mémoire se termine sur des conclusions et des perspectives dans lesquelles nous présentons deux extensions de la méthode de preuve décrite dans le chapitre 5 basées sur une technique d'abstraction.

Chapitre 2

Préliminaires

Dans ce chapitre, nous présentons les éléments de base qui sous-tendent notre travail : les systèmes répartis, la tolérance aux défaillances, ainsi que l'auto-stabilisation.

2.1 Généralités sur les systèmes répartis

Dans cette section, nous introduisons les systèmes répartis, leurs propriétés et les problèmes qui leurs sont associés.

Informellement, un système réparti est un ensemble d'unités de calcul (ordinateurs, processeurs, processus) reliées entre elles par un réseau de communication. Ces unités peuvent exécuter des calculs et communiquer avec les autres via le réseau. Ces liens de communication sont des mémoires partagées ou des canaux dans lesquels transitent des messages. Chaque unité de calcul (que nous appellerons processus dans la suite) exécute sa propre tâche (son algorithme) et manipule sa mémoire locale qui est constituée d'un ensemble de variables.

Les systèmes répartis sont devenus une réalité dans le monde informatique avec l'avènement des réseaux de données auxquels sont connectés un grand nombre d'ordinateurs. Ceux-ci peuvent communiquer via un grand nombre de protocoles réseaux tout en restant indépendants : ils ont leur propre matériel, leur système d'exploitation et leurs propres applications. Le but des systèmes répartis est de fournir des services partagés :

- Partager les ressources : les ressources informatiques (imprimantes, disques dur,...) restent relativement chères et il est toujours intéressant de pouvoir les partager entre plusieurs utilisateurs. De plus, il est souvent utile de pouvoir partager des informations entre des utilisateurs distants.
- Fiabiliser les calculs : il est fréquent que les ressources de calcul soient victimes de défaillances. Une bonne façon de fiabiliser des calculs longs et coûteux est

d'utiliser la redondance. Pour cela, il est nécessaire que les machines se communiquent les résultats intermédiaires ou que des machines spécialisées gèrent les différentes ressources afin d'optimiser les temps de calculs.

– Communiquer : au-delà des contingences systèmes liées aux ressources, les utilisateurs ont des besoins de communications rapides et fiables. Les systèmes répartis peuvent fournir les protocoles associés.

On peut classer les systèmes répartis en fonction du type de ressources qu'ils gèrent. Nous allons détailler les caractéristiques des processus et des liens de communications qui les caractérisent.

2.1.1 Caractéristiques des processus

Un processus est une unité de calcul au sein du système. Les caractéristiques individuelles des processus induisent des propriétés des systèmes entiers. D'après [Lyn96], les systèmes répartis peuvent être classés dans les trois groupes suivants :

– Un système réparti est anonyme si tous les processus sont strictement identiques : c'est-à-dire qu'ils exécutent tous le même code et qu'ils ont les mêmes propriétés. Cela inclut le fait que les processus ne peuvent se distinguer des autres par un identifiant unique. A l'opposé, un système réparti peut être non anonyme si les processus peuvent se distinguer.

– Un système réparti est uniforme si tous les processus exécutent le même code mais qu'ils sont capables de se reconnaître. L'uniformité est une propriété plus faible que l'anonymat. Un système est non uniforme si au moins un processus exécute un code différent des autres (on l'appelle le processus distingué).

– Un système réparti peut être déterministe : les processus exécutent un code déterministe localement. Au contraire, un système est non déterministe si au moins un processus exécute un code non déterministe (par exemple, probabiliste).

La plupart des algorithmes sont conçus pour des systèmes non-anonymes voire non-uniformes. L'inconvénient de tels algorithmes est qu'ils sont peu résistants aux défaillances. Par exemple, si un processus est distingué et que tout l'algorithme repose sur ses communications, sa déconnection empêche le système de remplir son rôle convenablement. C'est pourquoi il peut être avantageux d'essayer de trouver des algorithmes uniformes.

Quelques auteurs ont essayé de trouver des systèmes de transformation pour passer automatiquement d'un système non-uniforme à des systèmes uniformes. Malheureusement, il existe de nombreux résultats d'impossibilité. Par exemple, [Ang80] montre que pour les systèmes anonymes et uniformes, il existe une classe non négligeable de problèmes qui ne peuvent avoir de solution déterministe.

2.1.2 Caractéristiques des liens de communications

Les processus communiquent via un réseau de communication. Celui-ci est constitué de liens qui sont des registres partagés ou des canaux de communication permettant de faire transiter des messages. Ce réseau de communication ne relie pas forcément tous les processus entre eux.

On distingue deux grands types de réseau ([Lyn96]):

- Communication globale : tous les processus sont connectés via un médium qui leur permet de communiquer tous entre eux (diffusion). C'est l'équivalent du bus des machines parallèles.
- Communication point à point : les processus sont connectés via des liens qui relient deux processus entre eux. Le réseau peut être plus ou moins connecté.

Dans le cadre d'une communication point à point, on distingue deux types de liens :

- Lien unidirectionnel : le lien possède un émetteur et un récepteur. Les messages vont donc dans un seul sens.
- Lien bidirectionnel : les deux processus connectés au lien ont un rôle symétrique dans la communication. Les messages sont acheminés dans les deux sens.

Dans la suite, nous n'envisagerons que des communications point à point. Dans ce cas, les liens forment un graphe de communications qui est la topologie du réseau. Ce graphe est orienté si les liens sont unidirectionnels, non orienté sinon.

Enfin, ces liens peuvent être divisés en deux catégories :

- Les canaux de communication : ce sont des tubes dans lesquels transitent des messages.
- Les registres partagés : ce sont des mémoires exportées qui sont accessibles en lecture et en écriture par les processus qu'elles relient.

2.1.3 Caractéristiques liées aux vitesses relatives

Dans les réseaux hétérogènes, les processus ne traitent pas l'information à la même vitesse. De même, les liens de communications ne sont pas tous aussi rapides. Cela nous amène à une troisième classification des systèmes répartis.

- Un système réparti est dit synchrone quand les processus exécutent leur algorithme local à la même vitesse et que les liens véhiculent l'information à la même vitesse. C'est-à-dire qu'on peut borner les temps d'exécution et de transmission.

- Un système réparti est dit asynchrone lorsque les vitesses relatives des processus et des liens ne peuvent être bornées.

Entre ces deux extrêmes, il existe des catégories intermédiaires. On parlera de système partiellement synchrone lorsque le synchronisme n'est pas parfait, mais que certaines bornes existent, qu'elles soient connues ou pas. On peut également avoir des bornes qui sont vérifiées au bout d'un temps fini.

2.2 La tolérance aux défaillances

Dans les systèmes informatiques, les défaillances sont fréquentes et multi-formes. Dans le cadre de l'algorithmique répartie, le problème de supporter des défaillances tout en maintenant un comportement globalement cohérent a mené à l'étude d'algorithme tolérant aux défaillances.

2.2.1 La classification des défaillances

Les défaillances peuvent être classées suivant leur caractéristiques :

- Défaillance définitive : une défaillance est dite définitive si elle se prolonge indéfiniment dans le temps (pannes crash). Pour un processus, cela signifie qu'il n'exécutera plus jamais son algorithme. Pour un lien, cela signifie que plus aucun message ne transitera par lui dans le futur.

- Défaillance byzantine : un processus byzantin est un processus qui peut exécuter un autre code que celui de son algorithme original. Un lien byzantin est un lien qui peut modifier des messages en transit, en créer de nouveaux ou perdre des messages existants.

- Défaillance transitoire : une défaillance transitoire est une défaillance qui se produit très rarement mais qui peut corrompre les mémoires des processus et le contenu des liens.

Les défaillances byzantines représentent un ensemble de défaillances très différentes. Nous distinguerons les défaillances par omission des défaillances strictement byzantines. Les défaillances par omission sont celles où les composants n'exécutent pas certaines actions mais ne peuvent pas exécuter des instructions qui ne sont pas dans l'algorithme original. Pour un lien, cela signifie qu'il peut perdre des messages mais pas les dupliquer ou les modifier. Pour un processus, cela signifie qu'il peut ne pas exécuter certaines actions.

2.2.2 L'approche classique

L'approche classique de la tolérance aux défaillances est basée sur une vision pessimiste : on considère qu'il est nécessaire de vérifier constamment le comportement du système. Cette approche a donné de très bons résultats pour les défaillances définitives et les défaillances par omission. Toutefois, il existe des théorèmes d'impossibilité. Par exemple, le théorème de Fisher, Lynch et Paterson ([FLP85]) montre qu'il est impossible de résoudre le problème du consensus (un problème central de l'algorithmique répartie) en présence d'un seul processus qui fait une défaillance définitive dans un système asynchrone. Néanmoins, des approches utilisant des synchronismes partiels offrent des solutions pour ce problème et pour ses nombreux dérivés. Cette approche classique est très efficace dans le cadre des défaillances définitives et byzantines mais est un peu coûteuse en performance lorsqu'il s'agit de tolérer des défaillances transitoires. Une approche alternative, introduite dans [CHT96] et qui donne également de bons résultats est celle des détecteurs de défaillance. L'idée est de caractériser précisément les propriétés du système nécessaires sous forme de services disponibles auprès des processus. De nombreuses améliorations ont été apportées au concept de base notamment au niveau de la tolérance à d'autres types de défaillances comme par exemple [ADGFT01].

2.3 L'auto-stabilisation

L'auto-stabilisation est une approche alternative pour écrire des algorithmes tolérants aux défaillances transitoires. Cette notion a été introduite par Dijkstra en 1974 dans [Dij74]. Le principe de conception est une vision optimiste où on considère que les défaillances sont suffisamment rares pour ne pas être significatives. En revanche, comme ces défaillances peuvent corrompre complètement le système, il est nécessaire de considérer l'état initial comme étant quelconque, contrairement à l'algorithmique répartie classique où on choisit son état initial. Avec les algorithmes auto-stabilisants, le système démarre son exécution dans un état quelconque et passe par une phase de stabilisation dans laquelle le système converge vers un état dit légitime en un temps fini. A partir de cet état, le système va vérifier sa spécification, c'est-à-dire exécuter sans erreur la tâche qui lui est assignée.

La figure 2.1 présente le schéma de principe des algorithmes auto-stabilisants. Dans un premier temps, l'exécution part d'un état illégitime. Pendant un phase de convergence, le système oscille entre différents états illégitimes, puis, le système atteint un état légitime. C'est le début de la phase de correction. Dans cette phase infinie (ou jusqu'à l'apparition d'une défaillance transitoire), le système n'atteindra plus jamais d'état illégitime. L'algorithme est auto-stabilisant si la phase de correction est atteinte en un temps fini à partir de n'importe quel état illégitime.

Les preuves de tels algorithmes se font donc en deux temps : une preuve de convergence finie qui fait intervenir des mesures sur les états du système suivie d'une preuve

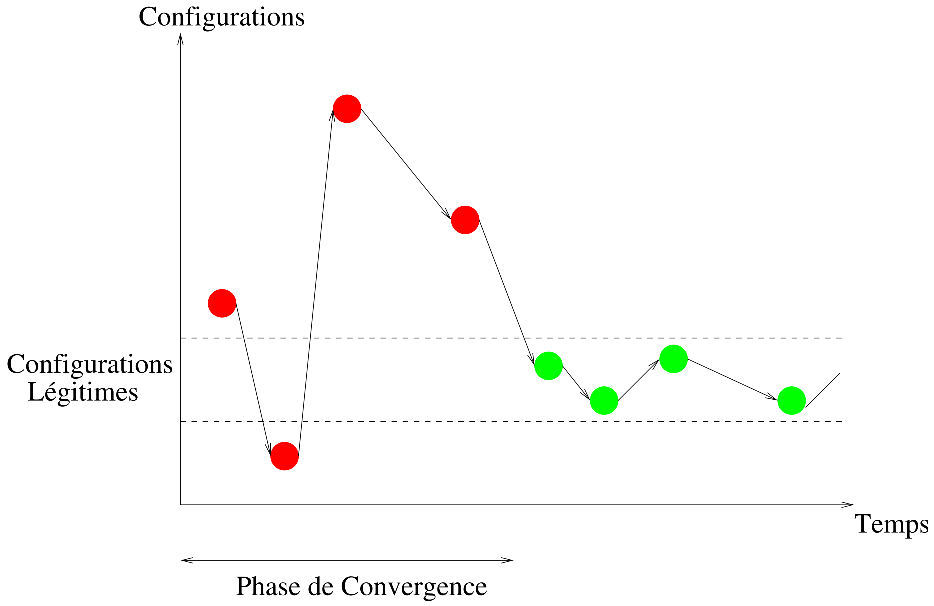

FIG. 2.1 – *Auto-stabilisation*

de correction qui montre la clôture des états légitimes et le respect de la spécification dans toute exécution démarrant dans un état légitime.

Chapitre 3

Modèle

Dans ce chapitre, nous formaliserons la notion de systèmes répartis à partir des éléments qui les constituent : les processus et les liens de communication. Nous développerons le concept d'auto-stabilisation.

3.1 Références

Les systèmes répartis sont modélisés sous la forme de graphes dont les sommets sont les processus et les liens de communication sont les arêtes. Les processus peuvent être représentés par des systèmes de transitions comme dans [Tel94] et [AW98], ou comme des automates à entrées/sorties [Lyn96].

Dans le cadre plus spécifique de l'auto-stabilisation, les modèles les plus utilisés sont basés sur des travaux spécifiques : [Her91] utilise un système de transitions à atomicité fine, [Pet98] un système de transitions à atomicité composite. [Tix00] et [VG00] utilisent un système de transitions à atomicité fine, globalement similaire à celui de [AW98]. C'est aussi le modèle que nous utilisons dans ce mémoire.

Pour l'auto-stabilisation , nous utiliserons les définitions les plus communément acceptées comme celle de [Dol00] et [Tel94]. Des détails sur l'auto-stabilisation et sur ses applications pourront être trouvés dans [Gou95] ou encore [Shn93].

3.2 Notions préliminaires

Dans cette section, nous introduisons la notion d'automate ainsi que la composition de ceux-ci. En effet, les systèmes répartis sont modélisés par des automates.

3.2.1 Automates et Langages

Il est très pratique de pouvoir représenter un système distribué sous la forme d'un automate. En effet, cette notation permet d'exprimer la sémantique des algorithmes d'une manière axée sur les exécutions. En effet, chaque transition de l'automate désigne une action du système et donc l'automate reconnait le langage formé par les exécutions du système.

Définition 1 (Automate) *Un automate est un quintuplet A=(Q,I,E,T,F) où Q est l'ensemble des états.* $I \subseteq Q$ *est l'ensemble des états initiaux, E est l'ensemble des actions, T est l'ensemble des transitions avec* $T \subset Q \times E \times Q$ *et F est l'ensemble des états terminaux. Une transition est un triplet (q1,q2,e) élément de* $Q \times Q \times E$*. Elle indique que l'on peut passer de l'état q1 (appelé la source) à l'état q2 (appelé la destination) en exécutant l'action e.*

Dans la suite, nous considérerons les automates comme des triplets (Q,E,T) en supposant que $I = F = Q$.

La figure 3.1 montre un exemple d'automate. Les états sont $Q = \{1, 2, 3\}$ et les actions sont $E = \{a, b\}$. Les transitions sont représentées par des flèches. L'état 1 est le seul état initial et l'état 2 est le seul état terminal (comme le montre les deux petites flèches). Cet automate est dit non déterministe car l'état 3 est la source de deux transitions portant la même action b.

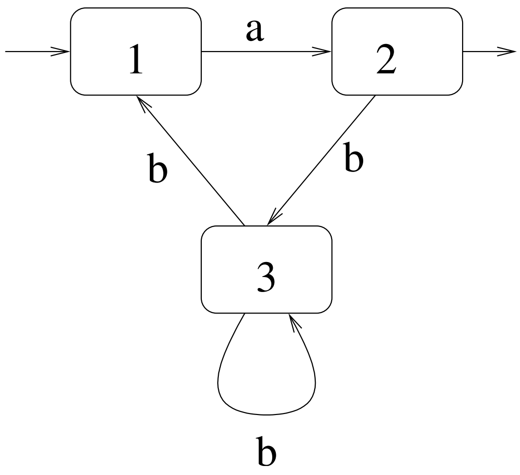

FIG. 3.1 – *Exemple d'automate*

Définition 2 (Automate déterministe) *Un automate est déterministe si et seulement si il n'existe pas deux transitions dans T* (q_1, q_2, e_1) *et* (q_3, q_4, e_2) *telles que* $q_1 = q_3$*,* $e_1 = e_2$ *et* $q_2 \neq q_4$*.*

Informellement, un automate est déterministe si et seulement si à partir de n'importe quel état q, il n'existe pas deux transitions ayant pour origine q et portant la même action.

Définition 3 (Grammaire) *Une grammaire est un quadruplet G=(Vn,Vt,P,S).* V_n *représente les variables,* V_t *les terminaux. P est l'ensemble des productions et S est l'élément de départ. Par définition,* V_n *et* V_t *sont disjoints et on note* $V = V_t \cup V_n$*. P est un ensemble d'expressions de la forme* $\alpha \rightarrow \beta$ *où* α *est un élément de* V^+ *et* β *est un élément de* V^**.* V^+ *désigne l'ensemble des mots sur l'alphabet* V *et ayant au moins une lettre.* V^* *désigne l'ensemble des mots sur l'alphabet* V*, y compris le mot vide. S est un élément de* V_n*. Une grammaire permet d'engendrer un langage noté L(G) obtenu en appliquant des productions à S.*

Définition 4 (Langage régulier) *Un langage régulier sur un alphabet* Σ *est un langage engendré par la grammaire suivante :*
$V_n = \{BASE, L\}$
$V_t = \Sigma \cup \{\varepsilon\}$
$S = L$
et P est l'ensemble des productions suivantes :

- $L \rightarrow BASE$ *: c'est l'instanciation de base. Un expression peut être une lettre.*
- $L \rightarrow LL$ *: c'est l'opérateur de concaténation sur les chaînes de caractères.*
- $L \rightarrow L*$ *: c'est l'opérateur de concaténation multiple.*
  Formellement, $L* = \{\epsilon, L, LL, LLL, ...\}$.
- $L \rightarrow L+$ *: c'est l'opérateur de concaténation multiple non nulle.*
  Formellement, $L+ = \{L, LL, LLL, ...\}$.
- $L \rightarrow L|L$ *: c'est l'opérateur de choix. Par exemple,* $a|b$ *peut produire* a *ou* b*.*

De plus, P contient toutes les productions du type :
$BASE \rightarrow a$ *où* a *est un élément de* V_t*.*

Un langage régulier est également appelé une expression régulière.

3.2.2 Produit synchronisé d'automates

Considérons deux automates : ils ont différents états et différentes transitions (actions). La modélisation de l'interaction entre ces deux automates peut être vue comme la donnée des liaisons entre leurs différentes actions. Par exemple, on peut souhaiter que les deux automates soient synchronisés sur une action particulière. Pour modéliser ces interactions, nous utilisons le produit synchronisé d'automates. Il permet de décrire le comportement du système composé de ces deux automates et de contraindre certaines synchronisations.

Pour pouvoir définir des synchronisations, nous définissons l'action vide. Cette action est associée à des transitions qui sont du type (q, q, e). C'est-à-dire que l'exécution de cette action ne change pas l'état courant de l'automate : c'est une boucle sur l'état courant.

Définition 5 (Produit Synchronisé) *Le produit synchronisé d'un ensemble de n automates $A = \{A_1, A_2, ..., A_n\}$, où chaque $A_i = (Q_i, E_i, T_i)$ avec l'ensemble de synchronisation $S \subset \Pi_{1 \leq i \leq n}(E_i \cup \{-\})$, est un automate P=(Q,E,T) où :*

- $Q = Q_1 \times Q_2 \times ... \times Q_n$
- $E = S$
- $T = \{(q_1, q_2, ..., q_n), (q'_1, q'_2, ..., q'_n), (e_1, e_2, ..., e_n)\}$ *où* $(e_1, e_2, ..., e_n) \in E$, *et* $(e_i = - \Rightarrow q_i = q'_i)$ *ou* $(e_i \neq - \Rightarrow (q_i, q'_i, e_i) \in T_i)$

La figure 3.2 donne un exemple de produit synchronisé entre deux automates avec un ensemble de synchronisation très restreint : l'ensemble de synchronisation spécifie les transitions autorisées. L'ensemble des états à considérer est le produit cartésien de tous les états des automates à synchroniser. Toutes les transitions autorisées sont représentées. Sur cet exemple, les actions a et b des deux automates sont synchronisées entre elles. En revanche, l'action c n'est pas contrainte (c'est pourquoi elle est synchronisée avec l'action vide).

3.3 Les processus et les liens de communications

Les processus sont les éléments de calcul d'un système réparti. Pour mener à bien leurs calculs, ils utilisent des liens de communication qui leur permettent d'échanger des informations. Nous modélisons ces deux composants de base par des automates. Au sein d'un système réparti, les actions de communication des processus et des liens devront être synchronisées.

3.3.1 Processus

Définition 6 (Processus) *Un processus est un automate dont les états indiquent les différentes valeurs des variables utilisées et dont les transitions indiquent les différentes actions qu'il peut effectuer.*

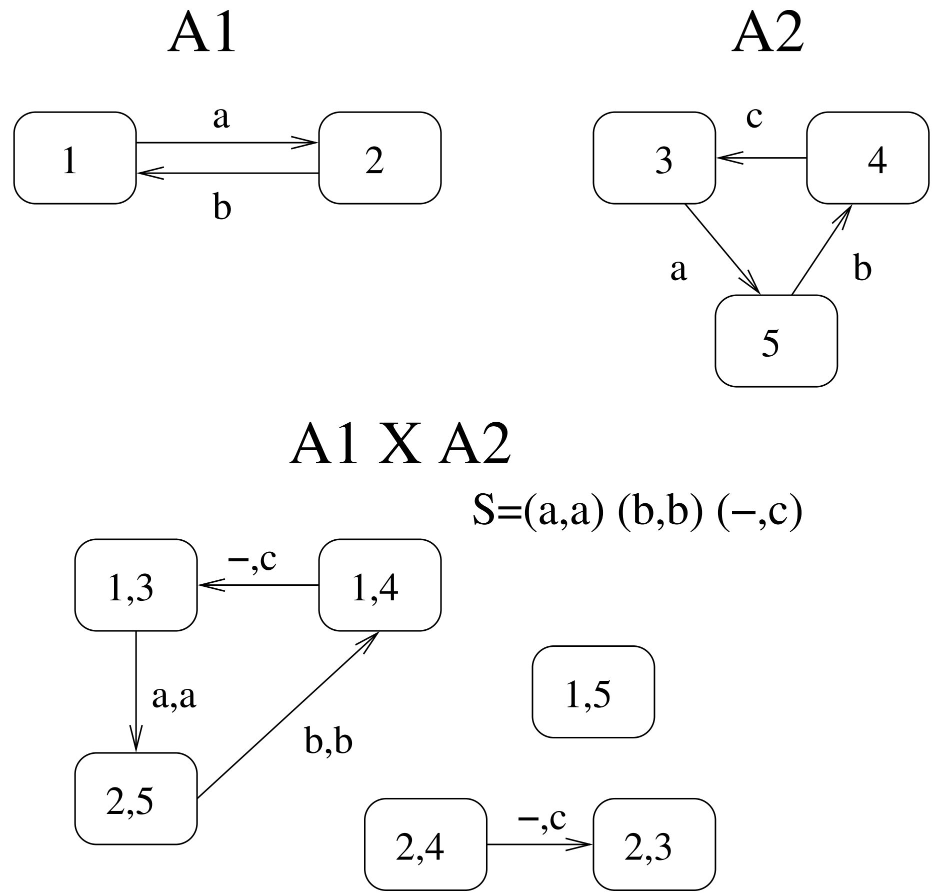

FIG. 3.2 – *Produit synchronisé de deux automates*

Ces transitions sont divisées en trois ensembles disjoints :

- L : ensemble des actions de lecture. Elles correspondent aux actions de lecture des variables partagées dans le cas d'un système à lecture d'état ou aux actions de réception dans un canal dans le cas d'un système à passage de messages. Ces actions seront donc synchronisées avec les liens de communications.

- I : ensemble des actions internes aux processus : elles correspondent à un changement de valeur sur l'une des variables du processus. Elles n'ont pas à être synchronisées sur les liens de communications.

- E : ensemble des actions d'écriture. Elles correspondent aux actions d'émission de messages dans un canal dans le cas d'un système à passage de messages. Elles correspondent à l'écriture d'un nouvelle valeur dans un registre partagé dans un système à lecture d'état. Ces actions seront donc synchronisées avec les liens de communications.

3.3.2 Liens

Les liens de communication sont des automates qui, synchronisés avec les processus, fournissent un support de communication. On distingue deux grands types de liens :

- les registres partagés qui sont utilisés dans les systèmes à lecture d'état
- les canaux qui sont utilisés dans les systèmes à passage de messages.

Définition 7 (Lien) *Un lien est un automate qui est indifféremment un registre ou un canal.*

3.3.3 La lecture d'état

Dans les systèmes à lecture d'état, chaque processus peut lire la valeur des registres de ses voisins. Informellement, cela signifie qu'un processus partage des variables avec ses voisins et que l'algorithme n'est pas obligé de spécifier explicitement la lecture.

Définition 8 (Registre) *Un registre est un automate dont les états correspondent aux valeurs de la variable partagée correspondante. Les actions sont de type lecture ou écriture et seront synchronisées avec les actions correspondantes au niveau des processus.*

La figure 3.3 présente un registre à quatre états. Les transitions marquées $w(i)$ pour $i = 1..4$ sont les transitions d'écriture et les transitions marquées $r(i)$ pour $i = 1..4$ sont les transitions de lecture. Une transition d'écriture $w(i)$ a pour effet de faire passer dans l'état i. L'action de lecture est toujours une boucle (elle ne change pas l'état de l'automate).

3.3.4 Le passage de message

Définition 9 (Canal) *Un canal est un automate dont les états sont des multi-ensembles appelés messages. Ses transitions sont de type envoi ou réception et sont synchronisées avec les actions correspondantes au niveau des processus.*

On peut classer les canaux suivant plusieurs types de propriétés. Tout d'abord, on trouve les canaux bornés qui ne peuvent contenir qu'un nombre fini de message. Les canaux non bornés peuvent contenir une infinité de messages : les automates associés sont infinis. De même, les canaux peuvent contenir des messages de taille fixe ou variable. Enfin, on peut classer les canaux suivant les propriétés d'ordre de

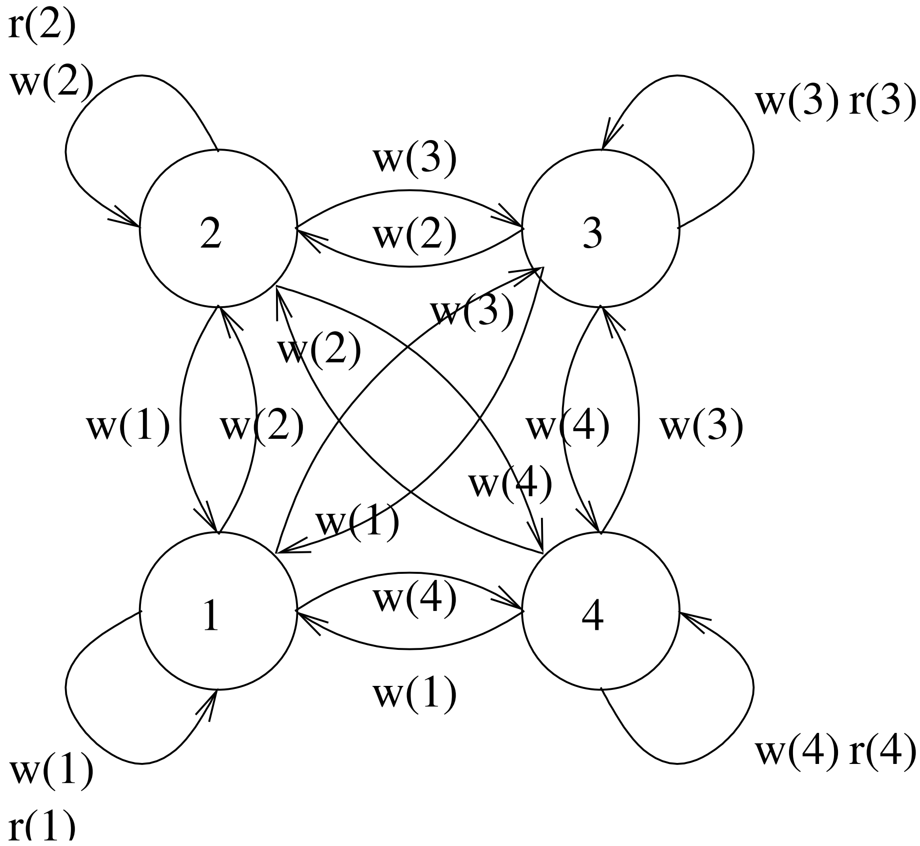

FIG. 3.3 – *Exemple de registre*

réception des messages. Un canal FIFO assure que les messages sont reçus dans l'ordre de leur émission. Les autres canaux peuvent déséquencer les messages, c'est-à-dire qu'ils peuvent être reçus dans un ordre différent de celui de leur envoi. On peut aussi envisager des canaux qui perdent les messages, c'est-à-dire qu'ils possèdent des transitions particulières qui font "disparaître" un message. On peut aussi envisager des canaux qui dupliquent ou modifient les messages.

La figure 3.4 donne un exemple de canal de taille 2 et pouvant manipuler deux messages m_1 et m_2. La figure 3.5 donne un exemple de canal FIFO : c'est-à-dire que les messages sont reçus dans l'ordre de leur émission.

3.4 Systèmes répartis

Définition 10 (Système réparti) *Un système réparti est le produit synchronisé de deux ensembles d'automates : P l'ensemble des processus du système et L l'ensemble des liens. La synchronisation est effectuée entre les transitions des liens et les transitions de communications correspondantes au niveaux des processus. L'ensemble de*

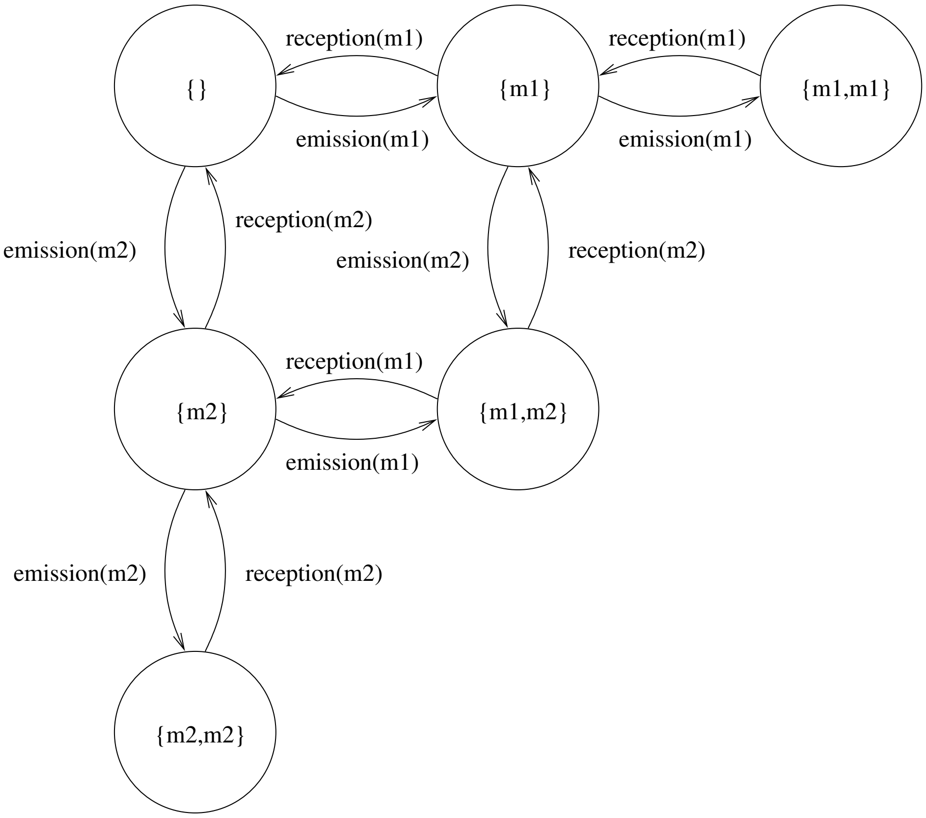

FIG. 3.4 – *Exemple de canal*

synchronisation contient donc toutes les actions internes de tous les processus associées à l'action vide pour tous les autres automates, ainsi que les actions de communication des processus, synchronisées avec les actions des liens correspondants.

Définition 11 (Configuration) *Une configuration du système est un état du produit synchronisé. Par construction, c'est donc un vecteur d'états de tous les automates du système.*

Définition 12 (Action du système) *Une action du système est un élément de son ensemble de synchronisation.*

L'ensemble de synchronisation permet de forcer les actions de communication entre les liens et les processus connectés par ces liens à s'exécuter simultanément. Cet ensemble, tel qu'il est décrit dans [Arn92], contient des actions de la forme :

- Exactement une action interne d'un processus et l'action vide pour tous les autres automates.
- Une action de lecture pour un processus, une action de lecture pour le lien associé et l'action vide pour tous les autres automates.

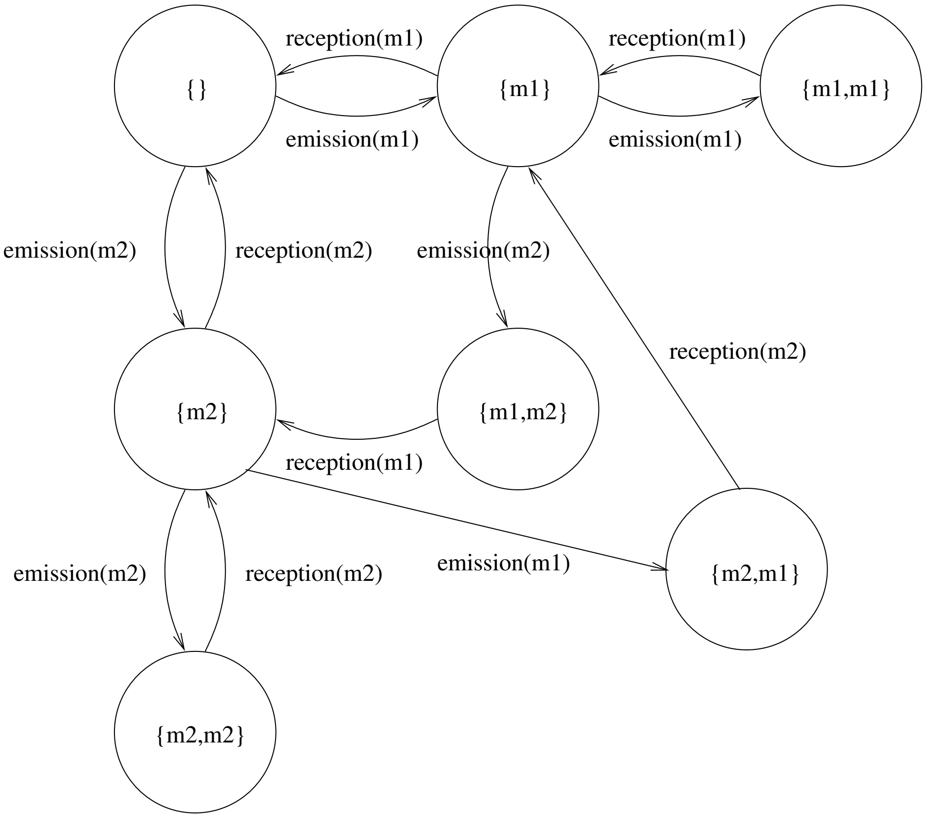

FIG. 3.5 – *Exemple de canal FIFO*

– Une action d'écriture pour un processus, une action d'écriture pour le lien associé et l'action vide pour tous les autres automates.

Exemple Considérons un système réparti constitué de deux processus P_1 et P_2 et d'un registre R. Les deux automates de P_1 et de P_2 ne comportent que deux actions : lecture et écriture dans le registre de la valeur 1 pour P_1 et 2 pour P_2. Les automates ne possèdent que deux états : 1 et 2 et les actions de lecture et d'écriture associées $(lit(1), ecrit(1), ecrit(2), lit(2))$. Les figures 3.6, 3.7 et 3.8 montre les automates P_1, P_2 et R. L'ensemble de synchronisation est semblable à la description ci-dessus :
$S = \{(lit(1), -, lit(1)),(-, lit(2), lit(2)),(ecrit(1), -, ecrit(1)),(-, ecrit(2), ecrit(2))\}$.
La figure 3.9 montre l'automate produit synchronisé.

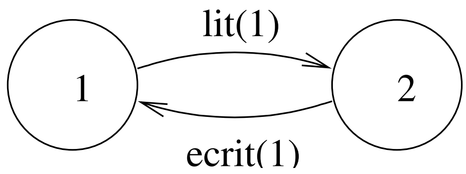

FIG. 3.6 – *Automate P_1*

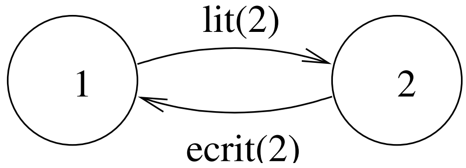

FIG. 3.7 – *Automate* P_2

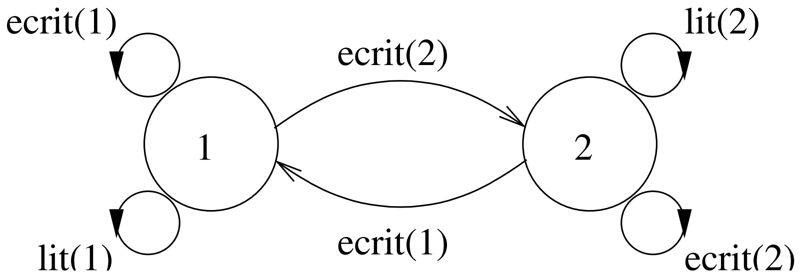

FIG. 3.8 – *Automate* R

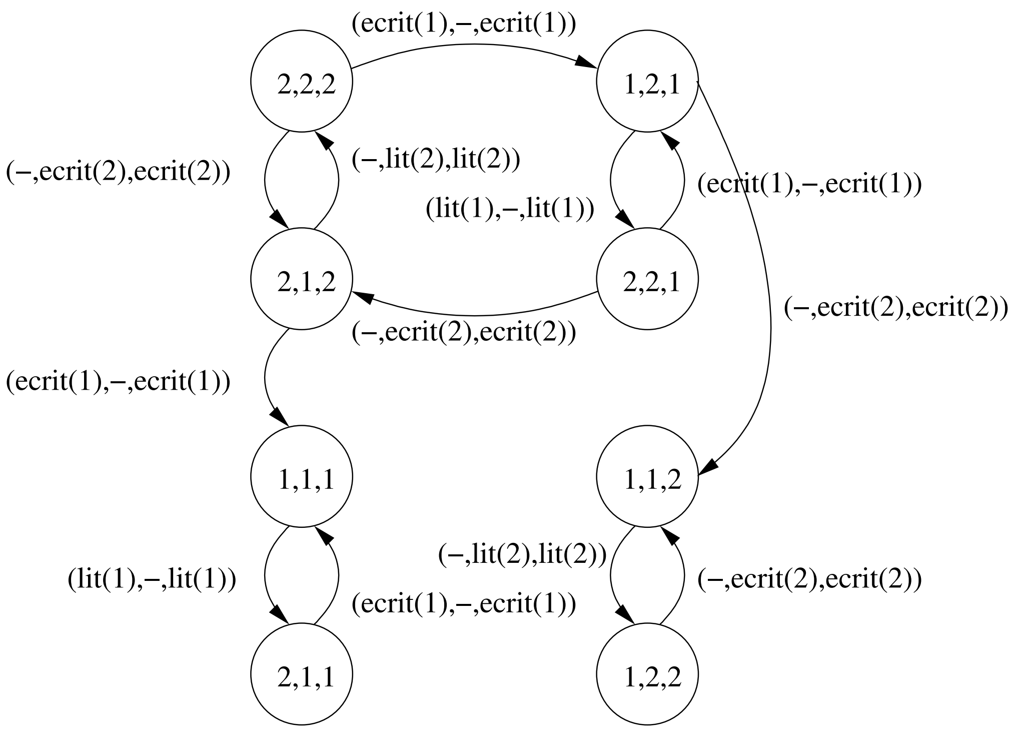

FIG. 3.9 – *Produit synchronisé de* P_1, P_2 *et* R

Définition 13 (Exécution du système) *Une exécution d'un système réparti est une séquence de configurations et d'actions , éventuellement infinie, notée* $e = e_1 a_1 e_2 a_2 ...$ e_{i+1} *est la configuration atteinte à partir de la configuration* e_i *en exécutant l'action*

a_i. La configuration c_1 est appelée configuration initiale de e.

Il est souvent pratique de projeter les exécutions sur des espaces plus petits pour pouvoir caractériser des propriétés. Nous présentons ici deux projections classiques : la trace et le journal.

Définition 14 (Trace) *La trace d'une exécution est sa projection sur les actions qui la constituent.*

Par exemple, la trace de l'exécution $e_1a_1e_2a_2e_3a_3$... est $a_1a_2a_3$... On dira que a_1 est l'action initiale de cette trace.

Définition 15 (Journal) *Le journal d'une exécution est sa projection sur les états qui la constituent.*

Par exemple, le journal de l'exécution précédente est $e_1e_2e_3$.... On dira que e_1 est l'état initial de ce journal.

3.5 Validation du modèle

Le modèle basé sur des automates est très puissant pour faire des preuves sur les systèmes répartis. En revanche, il est très peu synthétique et ne permet donc pas une bonne description : il décrit la sémantique mais pas la syntaxe. Les algorithmes sont implémentés par des programmes qui sont des représentations syntaxiques des automates associés. Nous introduisons donc deux notions : la topologie et l'algorithme, qui permettent de spécifier complètement un système d'une manière syntaxique. Naturellement, il existe une manière canonique de passer d'un modèle à l'autre.

3.5.1 Algorithme

Un algorithme est une spécification des processus. Elle est donnée sous la forme d'un ensemble de règles gardées du type : <condition>$\rightarrow$ action. La condition est appelée garde et l'action est appelée instruction. La garde est une suite d'actions internes ou de lectures dont le résultat conditionne l'exécution de l'instruction. L'instruction est une suite d'actions internes et d'écriture. Il est possible que la garde soit vide : on parle alors d'une action spontanée. On peut transformer directement un algorithme en un automate de la façon suivante :

– Les états de l'automate sont formés par les différentes valeurs des variables de l'algorithme, y compris le compteur de programme.

– Les transitions de l'automate correspondent aux instructions effectuées qui modifient le contenu des variables.

Exemple Voyons un exemple d'algorithme

R1		→	X=0
R2	X=0	→	X=1
			diffuse(X)
R3	receive(X')	→	X=X'

Cet algorithme tout à fait fantaisiste ne résout aucun problème mais permet d'illustrer les différents types de gardes que l'on peut rencontrer. La règle R1 est une règle dite spontanée, c'est-à-dire qu'elle peut se produire à n'importe quel moment. Son instruction est interne : elle met à 0 la valeur de la variable X.
La règle R2 n'est exécutée que lorsque la condition est vraie, c'est-à-dire que la valeur de la variable X est égale à 0. Les deux instructions de sa partie droite sont exécutées séquentiellement, c'est-à-dire que la valeur de X est mise à 1 puis cette valeur est diffusée à tous les voisins du processus.
La règle R3 est exécutée lorsque le processus reçoit un message. La variable X est alors mise à la valeur spécifiée dans le message.

3.5.2 Topologie

La notion de voisin est essentielle dans un système réparti, elle permet d'établir avec quelles autres entités un processus peut communiquer. Pour l'obtenir, nous introduisons la notion de topologie qui indique comment l'interconnection entre les processus et les liens. La modélisation de la topologie se fait par la notion de graphe, c'est pourquoi nous les définissons ici.

Définition 16 (Graphe) *Un graphe est un couple (V,E) où V est un ensemble non vide de noeuds et E est un ensemble d'arêtes où* $E \subset V \times V$.

Définition 17 (Graphe orienté) *Un graphe est dit orienté si et seulement si les éléments de E sont des paires ordonnées* (x, y). *On appelle* x *l'origine de l'arête et* y *la destination.*

Notation 1 *Soit* (x, y) *une arête d'un graphe, on dit que* x *et* y *sont voisins.*

Définition 18 (Chemin) *On appelle chemin dans un graphe un ensemble d'arêtes consécutives du graphe. C'est donc une séquence du type :*
$(x_1, x_2), (x_2, x_3), ...(x_{n-2}, x_{n-1}), (x_{n-1}, x_n)$. *Le neoud* x_1 *est appelé l'origine du chemin et le noeud* x_n *la destination du chemin, on dit également que le chemin va de* x_1 *à* x_n. *Le nombre de noeuds traversés (ici, n) est appelé longueur du chemin.*

Définition 19 (Diamètre) *Soit G=(V,E) un graphe. Soit Ch l'ensemble des chemins sur G. On appelle diamètre la longueur du plus long des plus courts chemin de Ch.*

Définition 20 (Connexité) *Un graphe G=(V,E) est fortement connexe si pour tout couple (x, y) de $V \times V$, il existe un chemin allant de x à y et un chemin allant de y à x. Un graphe est dit connexe s'il existe un chemin allant de x à y ou un chemin allant de y à x.*

Définition 21 (Cycle) *On appelle cycle tout chemin allant de x_1 à x_n et tel que $x_1 = x_n$.*

Définition 22 (Arbre) *On appelle arbre tout graphe connexe sans cycle.*

Définition 23 (Sous-graphe) *Soit G=(V,E) une graphe quelconque. Un sous-graphe G'=(V',E') de G est une graphe tel que V' soit un sous-ensemble de V et E' est l'ensemble des arêtes (x,y) de E tel que x et y soient éléments de V'.*

Définition 24 (Topologie) *Une topologie est un graphe dont les noeuds sont des processus et dont les arêtes sont des liens de communication. Ce graphe est également appelé graphe de communications.*

Définition 25 (Voisins) *Deux processus sont dits voisins si et seulement si il existe, dans la topologie du système, une arête entre les deux noeuds qui les représentent.*

La figure 3.10 montre quelques topologies courantes. De gauche à droite : une topologie en anneau, une topologie en chaîne et une topologie en arbre. On appellera topologie maillée une topologie dont le graphe est quelconque : voir un exemple sur la figure 3.11.

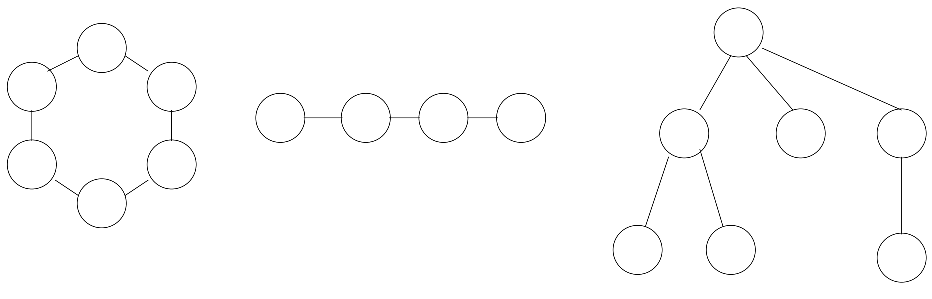

FIG. 3.10 – *Exemples de topologies*

Au niveau de l'automate du système, on dira que deux processus sont voisins si et seulement s'ils ont des actions synchronisées avec le même lien.

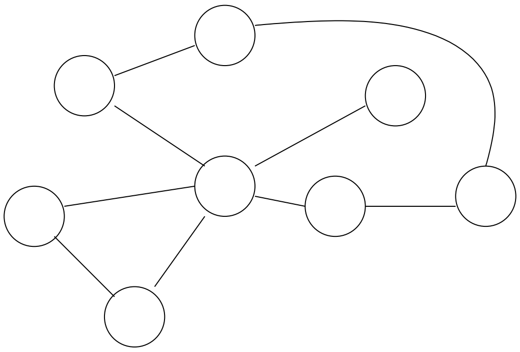

FIG. 3.11 – *Topologie maillée*

3.5.3 Passage du modèle automate au modèle topologie/algorithme

L'avantage de ces deux modélisations est que l'on peut passer canoniquement de l'un à l'autre.

Considérons le cas où l'on passe d'un algorithme et d'une topologie à un automate. Les automates associés aux canaux sont produits en fonction de leurs propriétés. Les automates associés aux processus sont construits de la manière suivante : on construit tous les états possibles en fonction des valeurs de toutes les variables, puis, pour chaque état, on calcule si la partie gauche de chaque règle est vraie ou non. Si la règle est vraie, on ajoute une arête qui relie cet état à l'état obtenu en appliquant la règle. Enfin, l'ensemble de synchronisation est obtenu à partir de la topologie. Le résultat de la synchronisation est l'automate du système réparti.

Pour la partie converse, on déduit la topologie de la synchronisation. L'algorithme est extrapolé à partir des transitions suivant le processus inverse à celui décrit dans le paragraphe précédent.

3.6 Propriétés des exécutions

Dans cette section nous étudierons les propriétés des exécutions.

3.6.1 Non-déterminisme

Il est fréquent qu'à partir d'un état d'un système réparti, plusieurs actions soient possibles. C'est-à-dire qu'il soit possible d'exécuter plusieurs actions à partir du même état. Ce non-déterminisme induit un choix au niveau de chaque état. Ce choix va permettre de construire plusieurs exécutions (souvent une infinité) à partir d'un même état initial. Ces diverses exécutions vont présenter des propriétés remarquables.

Définition 26 (Processus activable) *Un processus est dit activable si une de ses gardes est activable, c'est-à-dire que sa partie gauche est vraie. Au niveau d'un état d'un système, cela indique qu'il existe au moins une action exécutable à partir de cet état.*

3.6.2 Equité

Nous avons vu qu'à chaque état, un choix doit être fait pour déterminer quelle action va être exécutée. Par exemple, imaginons un système avec deux processus qui sont activables dans chaque état. On peut imaginer une exécution telle que ce soit toujours le même processus qui exécute son action. Dans ce cas, l'autre processus ne pourra jamais exécuter la moindre action. Ce cas de figure est typique d'un comportement des systèmes qui est appelé famine. L'équité des exécutions est une propriété qui permet de caractériser l'apparition des famines.

Il existe des formes d'équité plus ou moins fortes.

Définition 27 (équité faible) *Soit S un système et e une exécution de S. L'exécution e est dite faiblement équitable si et seulement si, au bout d'un temps fini, la trace de e contient l'action d'un processus activable infiniment souvent.*

Définition 28 (équité forte) *Soit S un système et e une exécution de S. L'exécution e est dite fortement équitable si et seulement si la trace de e contient une infinité d'actions d'un processus infiniment activable.*

Définition 29 (équité totale) *Soit S un système et e une exécution de S. L'exécution e est dite totalement équitable si et seulement si la trace de e contient infiniment souvent des actions de tous les processus.*

Définition 30 (k-équité) *Soit S un système et e une exécution de S. L'exécution e est dite k-équitable si et seulement si (a) la trace de e contient infiniment souvent des actions de tous les processus et (b) entre deux actions successives d'un processus, les actions de tout autre processus apparaissent au plus k fois.*

3.6.3 Equité des communications

Dans le cas du passage de messages, des choix similaires peuvent apparaître. En effet, si les canaux sont non ordonnés, il est possible qu'un message ne soit jamais reçu dans une exécution particulière.

Définition 31 (Communication équitable) *Soit S un système et e une exécution de S. Les communications de e sont dites équitables si et seulement si pour toute action d'émission de message dans e, il existe l'action de réception correspondante dans e.*

La communication équitable signifie qu'un message envoyé infiniment souvent doit être reçu infiniment souvent.

3.6.4 Atomicité

Lors du passage de l'algorithme à l'automate, on peut considérer différents niveaux d'actions comme étant atomiques. Par exemple, on peut considérer que toutes les actions sont atomiques, qu'elles soient internes ou de communication. C'est le grain le plus fin que l'on puisse utiliser dans la transformation. Toutefois, on peut également choisir des grains plus gros en groupant certaines actions entre elles. Par exemple, on peut considérer que la partie droite des règles est atomique quel que soit le nombre d'actions qu'elle comporte.

Dans la littérature, on trouve deux sortes d'atomicités qui influent sur la traduction de l'algorithme en un automate : l'atomicité composite et l'atomicité lecture/écriture. L'atomicité est la donnée de ce qu'on considère comme action atomique : c'est-à-dire la plus petite action que le système peut exécuter sans être interrompu. Les définitions données ici sont celles de [DIM93] qui sont acceptées largement dans la communauté. Dans le cas de l'atomicité composite, on considère qu'une action atomique est composée de plusieurs actions d'entrée (lecture de registre ou réception de message), de plusieurs actions internes et d'une action de sortie (écriture de registre ou émission de message). Dans le cas de l'atomicité lecture/écriture, on considère qu'une action atomique est composée de plusieurs actions internes et d'une seule action d'entrée ou de sortie. L'atomicité lecture/écriture est le grain d'atomicité le plus fin utilisé en pratique. En effet, une atomicité plus fine est envisageable (une action quelconque comme pas atomique) mais cette approche rend les preuves très difficiles et ne présente pas de réel intérêt.

3.7 Démon

Les démons sont les modélisations des propriétés de l'environnement. Un système réparti est l'expression d'un algorithme exécuté par des processus et des liens de communications. Or, il existe des propriétés de ces composants de bases qui ne peuvent pas s'exprimer dans les automates. C'est notamment le cas des propriétés sur les exécutions qui sont des conditions dynamiques sur des suites de transitions le plus souvent infinies. La notion de démon permet de modéliser ces propriétés en restreignant l'ensemble des exécutions à un sous-ensemble dont tous les éléments vérifient une spécification dynamique exprimée sous la forme d'un prédicat.

Définition 32 (Prédicat) *Un prédicat est une formule logique (éventuellement en logique temporelle) portant sur les états et les actions d'une exécution.*

Définition 33 (Démon) *Un démon est un prédicat sur les exécutions d'un système.*

On dit que le système s'exécute sous le démon d de prédicat P_d si et seulement si toutes les exécutions du système vérifient le prédicat P_d. Cela revient à caractériser un ensemble d'exécutions du système et à s'y restreindre pour faire les preuves. Il existe également une approche fonctionnelle qui permet de définir le démon d'une manière dynamique : le démon choisit à chaque pas de l'exécution quelle va être l'action suivante qui sera exécutée parmi celles qui sont disponibles. Cette approche présente l'avantage de la simplicité d'écriture mais n'est pas aussi expressive que la forme prédicat.

Définition 34 (Démon central) *Le démon central est le prédicat suivant : toute action est l'action d'un seul processus. C'est-à-dire qu'on ne considère que les systèmes synchronisés où les actions des processus sont synchronisées avec les actions des liens mais deux actions de deux processus différents ne peuvent être synchronisées.*

Sous forme fonctionnelle, cela revient à dire qu'à chaque pas de l'exécution, le démon choisit un seul processus activable et lui fait exécuter une action atomique.

Définition 35 (Démon quasi-central) *Le démon quasi-central est le prédicat suivant : deux actions de deux processus voisins (au sens de la topologie) ne peuvent être synchronisées.*

Définition 36 (Démons équitables) *Un démon équitable est un démon sous lequel toutes les exécutions vérifient la propriété d'équité forte. De même pour les démons faiblement équitables ou k-équitables.*

Définition 37 (Démon distribué) *A chaque étape, au moins une action d'un processus est effectuée.*

Sous forme fonctionnelle, le démon choisit à chaque pas de l'exécution un ensemble quelconque de processus activables et leur fait exécuter une action atomique.

Propriété 1 (Classification des démons) *Il est possible de classer les différents démons en comparant l'ensemble des exécutions possibles associées au prédicat. En* *effet, à partir d'un système réparti sous forme d'automate, il est possible de construire l'ensemble E de toutes les exécutions possibles. Les ensembles d'exécutions associés aux démons sont des sous-ensembles de E. On peut classer ses sous-ensembles par simple inclusion. Si toutes les exécutions associées à un démon d1 sont incluses dans les exécutions associées à un démon d2 : on dit que d2 est plus fort que d1 et on le note d1<d2.*

Il est démontré que la classification suivante est valide : démon central<démon quasi-central<démon distribué.

3.8 Auto-stabilisation

Dans les systèmes répartis habituels, les exécutions envisagées démarrent toutes par une configuration initiale connue. Dans les systèmes auto-stabilisants, on considère que tout état du système peut être une configuration initiale. Les exécutions des algorithmes auto-stabilisants se découpent donc en deux parties : une partie de convergence où, à partir d'un état quelconque, le système va atteindre un état dit légitime. A partir de cet état, l'exécution rentre dans une phase dite de correction où elle va vérifier sa spécification originale.

Définition 38 (Auto-stabilisation) *Un système S est dit auto-stabilisant pour la spécification P si et seulement si il existe un ensemble L de configurations dit ensemble de configurations légitimes tel que : (convergence) toute exécution de S contient au moins une configuration de L, et (correction) toute exécution de S partant d'un état de L vérifie la spécification P.*

Il existe une forme un peu plus faible de la définition qui peut être utilisée dans le cadre de problèmes relativement statiques : la correction s'exprime sous la forme d'une clôture aux états légitimes. Plus formellement, toute exécution de S partant d'un état légitime ne contient que des états légitimes. Le concept a été étendu par celui de pseudo-stabilisation dans [BGM93]. La pseudo-stabilisation est un affaiblissement de l'auto-stabilisation destiné à s'affranchir des états légitimes qui ne sont pas adaptés à tous les problèmes.

Définition 39 (Pseudo-stabilisation) *Un système S est dit pseudo-stabilisant pour la spécification P si et seulement si toutes les exécutions du système possèdent un suffixe non vide qui vérifie P.*

Enfin, il existe pour certaines classes de problèmes, des algorithmes qui sont dits instantanément stabilisants. Ce concept a été introduit pour la première fois dans [Vil98]. C'est la propriété pour un algorithme auto-stabilisant d'avoir un temps de stabilisation nul. Autrement dit un algorithme instantanément stabilisant conserve un comportement correspondant à sa spécification quel que soit l'état initial du système.

Définition 40 (Stabilisation instantanée) *Le système S est dit instantanément stabilisant pour la spécification P si et seulement si toute exécution du système vérifie P.*

3.9 Résumé

Nous modélisons un système réparti par un produit synchronisé d'automates de base représentant les processus et les liens de communications. Ces composants de base sont spécifiés par un algorithme et une topologie. Sur ces systèmes, on peut exhiber des exécutions ainsi que leurs projections (traces, journaux, ...). Il est possible de montrer que ces exécutions vérifient une spécification. Ces exécutions sont classées dans des ensembles qui sont définis par les prédicats des démons et qui caractérisent le système d'un point de vue fonctionnel (équité des processus et des liens de communication, granularité,...).

Un système réparti est auto-stabilisant s'il vérifie sa spécification au bout d'un temps fini et indépendamment de sa configuration initiale.

Chapitre 4

Exclusion Mutuelle Locale et composition d'algorithmes

La conception et la preuve d'algorithmes auto-stabilisants est simplifiée si le démon que l'on considère est faible, c'est-à-dire qu'il ne contient que des exécutions présentant des bonnes propriétés. Par exemple, il est assez facile d'écrire des algorithmes qui fonctionnent sous le démon central qui permet de s'affranchir des actions simultanées de processus impliqués dans une action commune.

Il existe une technique, la composition croisée, qui permet de simuler un démon faible sous un démon fort. L'idée de base est de composer l'algorithme original que l'on a démontré auto-stabilisant sous un démon faible, avec un filtre d'exécution (module fort). Cet algorithme composé a l'intéressante propriété d'être auto-stabilisant pour la même spécification, y compris sous un démon fort.

Nous présentons ici un filtre d'exécutions qui garantit le démon quasi-central sous un démon quelconque même dans un environnement à grain très fin de type atomicité lecture/écriture. Ce filtre est basé sur un algorithme d'exclusion mutuelle locale. Une fois composé avec tout algorithme auto-stabilisant sous un démon quasi-central, le composé devient auto-stabilisant sous le démon distribué même si l'algorithme original a un grain d'exécution fin (atomicité lecture/écriture).

4.1 Références

L'exclusion mutuelle locale est un problème qui s'est rendu célèbre en informatique sous le nom de "dîner des philosophes". C'est un paradigme des systèmes gestionnaires de ressources qui a été utilisé autant en théorie des systèmes d'exploitation qu'en algorithmique répartie. La problématique est la suivante : des philosophes sont assis autour d'une table ronde. Ils ont devant eux une assiette dans laquelle ils peuvent manger. Malheureusement, il n'y a pas assez de fourchettes. Il y a exactement une fourchette entre chaque paire de philosophes. Ceux-ci ont besoin de deux fourchettes pour pouvoir manger. Les philosophes ne mangent pas tout le temps, à tout moment, ils peuvent se mettre à réfléchir, relâchant ainsi les fourchettes. L'objectif est de gérer l'accès aux fourchettes de façon à éviter deux problèmes : la famine (quand un philosophe ne peut jamais manger) et l'interblocage (chaque philosophe détient une fourchette et aucun ne peut manger).

Cette définition s'applique sur des processus formant une topologie d'anneau. Sa forme généralisée à un graphe quelconque porte le nom d'exclusion mutuelle locale. Dans le cadre de l'algorithmique répartie, on trouve des solutions non auto-stabilisantes dans [CM84] et [AS90].

Les premières solutions auto-stabilisantes peuvent être trouvées dans [Gou87] et [HP89]. Ces solutions utilisent la technique de circulation de jeton. Le processus qui détient le jeton est privilégié et peut exécuter sa section critique. Ces solutions nécessitent un processus distingué.

La transformation des algorithmes depuis des démons faibles vers des démons forts a été publiée pour la première fois par Gouda et Hadix. Dans [GH97] les auteurs proposent le concept d'alternateur linéaire, étendu dans [GH99] à des topologies quelconques. Ils utilisent des variables bornées qui ne peuvent atteindre une valeur critique au même instant sur des processus voisins. Le problème pour la transformation automatique est que cette valeur dépend de la taille du réseau.
Une autre approche, proposée par Mizuno et Nesterenko dans [MN97] utilise des horloges sans bornes afin d'ordonner les actions d'un algorithme auto-stabilisant.
Dans le cadre des arbres, on trouve des solutions de ce problème dans [JADT99], [AS98] et [AS99].
Dans [Hua00], Huang propose une nouvelle solution pour l'exclusion mutuelle locale sous un démon quelconque. Il fait l'hypothèse que les arêtes du graphe de communications peuvent être colorées préalablement et il utilise cette coloration.
Enfin, Arora et Nesterenko, dans [AN99] utilisent l'exclusion mutuelle locale pour raffiner l'atomicité des algorithmes. Leur méthode s'appuie sur un démon faiblement équitable.

4.2 Définitions et idée de base

Le problème de l'exclusion mutuelle locale s'exprime sous la forme de deux prédicats : la sûreté garantit qu'un processus n'exécute sa section critique (on dit qu'il est privilégié) que si ses voisins ne l'exécutent pas; la vivacité garantit que tous les processus peuvent exécuter leur section critique infiniment souvent.

Définition 41 (Exclusion mutuelle locale) *Un système vérifie la spécification du problème d'exclusion mutuelle locale si et seulement si ses exécutions vérifient en* *même temps les deux prédicats suivants :*

- *Sûreté : Dans toute configuration, il existe au moins un processus privilégié et si un processus est privilégié, alors aucun de ses voisins ne détient le privilège.*
- *Vivacité : chaque processus qui demande le privilège infiniment souvent est privilégié infiniment souvent.*

Dans la suite, nous supposerons que tout processus demande le processus infiniment souvent et d'une manière automatique : c'est-à-dire qu'un processus est toujours considéré comme demandeur du privilège s'il ne l'a pas.

Définition 42 (Index d'équité) *Soit A un algorithme qui résoud le problème d'exclusion mutuelle locale. On dit que A a pour index d'équité k si dans toute exécution de A, entre deux sections critiques d'un même processus, les autres processus peuvent entrer au plus k fois en section critique.*

Définition 43 (Temps de service) *Soit p un processus quelconque. Entre deux sections critiques de p, l'ensemble des autres processus peuvent exécuter au plus t sections critiques (d'une façon cumulée). Cette valeur t est appelée le temps de service.*

4.2.1 Orientation Virtuelle

Nous définissons ici une notion d'orientation virtuelle qui sera la clef des algorithmes que nous allons présenter. Pour cela, nous définissons une relation d'ordre, notée $\triangleright$, sur les variables partagées des processus voisins.

Soit x.p et x.q les deux variables partagées de deux processus voisins p et q. Dans le graphe de communications, l'arête entre p et q est virtuellement orientée de p vers q si et seulement si $x.p \triangleright x.q$. On dira que cette arête est sortante pour p et entrante pour q.

Définition 44 (Processus privilegié) *Un processus est dit privilégié dans une configuration si et seulement si dans le graphe de communications, toutes les arêtes sont orientées vers lui au sens de l'orientation virtuelle (elles sont entrantes).*

Par exemple, dans la figure 4.1, les processus p3 et p5 sont privilégiés.

4.2.2 Renversement d'arêtes

La technique de renversement d'arêtes a été étudiée dans le cadre de l'algorithmique distribuée dans [CM84] et [BG89]. Informellement, cette technique exploite une

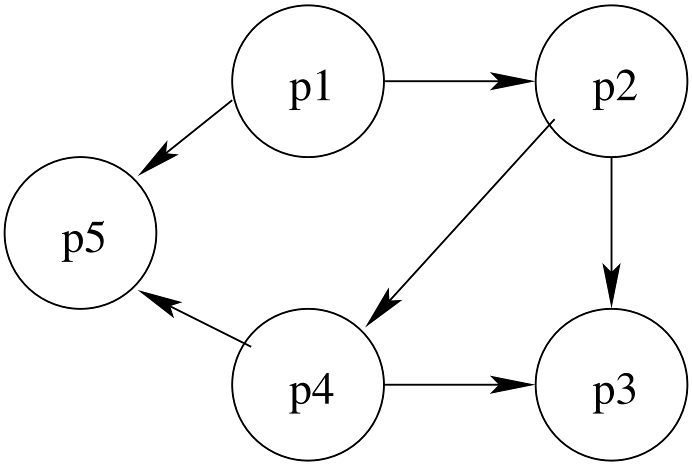

FIG. 4.1 – *Exemple d'orientation virtuelle*

propriété des graphes acycliques : dans un tel graphe, il existe toujours un noeud qui a toutes ses arêtes orientées vers lui. De plus, si toutes ses arêtes changent de sens, le graphe reste acyclique. Nous allons utiliser cette propriété pour garantir l'exclusion mutuelle locale : en effet, deux noeuds voisins ne peuvent avoir toutes les arêtes orientées vers eux car ils en partagent une.

Définition 45 (Noeud puits) *Dans un graphe orienté, un noeud puits est un noeud dont toutes les arêtes sont orientées vers lui.*

Propriété 2 *Dans tout graphe orienté acyclique, il existe au moins un noeud puits.*

La preuve de cette propriété peut être trouvée dans [CM84].

Lemme 1 *Soit G un graphe orienté acyclique, soit p un noeud puits de G. Le graphe G' obtenu en renversant les arêtes de p est acyclique.*

Preuve Soit G un graphe acyclique. Soit x un noeud puits. Supposons que x renverse ses arêtes et que le graphe résultant possède au moins un cycle. Montrons tout d'abord par l'absurde que x fait forcément partie du cycle : supposons que x ne fasse pas partie du cycle. Cela implique qu'il existe un sous-graphe de G qui possède un cycle. Cela contredit le fait que G soit acyclique. Or, x vient de renverser ses arêtes par conséquent toutes ses arêtes sont sortantes, ce qui implique qu'il ne peut pas faire partie d'un cycle. Par conséquent, le graphe obtenu est acyclique. □

La figure 4.2 montre le résultat du renversement des arêtes des noeuds p3 et p5 : le noeud p4 devient un noeud puits et le graphe reste acyclique.

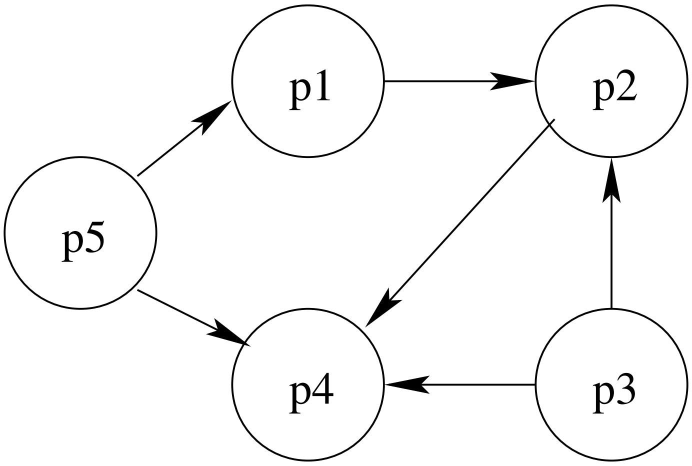

FIG. 4.2 – *Après le renversement*

Dans les deux sections suivantes, nous présentons deux algorithmes qui résolvent le problème de l'exclusion mutuelle locale sous un démon quelconque. La première solution est une variante simplifiée de la deuxième : elle permet de comprendre le principe de l'algorithme. La deuxième solution présente l'intéressante propriété d'utiliser une mémoire bornée sur chaque processus. L'index d'équité de ces deux algorithmes est n-1.

4.3 Exclusion mutuelle locale utilisant une mémoire sans borne (ULME)

De manière informelle, cet algorithme est basée sur l'idée suivante : les variables partagées des processus définissent une orientation virtuelle. Cette orientation garantit l'acyclicité du graphe et donc la présence de noeuds puits. Ces noeuds puits seront dits privilégiés et pourront exécuter leur section critique. Une fois qu'ils auront terminé leur section critique, ils retourneront leurs arêtes (via la règle A1). Afin de garantir un algorithme correct quel que soit le grain d'atomicité envisagé, notre algorithme utilise des copies des variables des processus voisins (atomicité lecture/écriture). Pour maintenir ses copies, il vérifie régulièrement (via la règle CP1) que les copies sont bien cohérentes avec ses propres valeurs, si ce n'est pas le cas, il recopie la vraie valeur.

4.3.1 Algorithme ULME

Algorithme 1 (ULME pour le processus p)

Constantes *:*

$id.p$: Identifiant entier unique de p;
$N.p$: L'ensemble des voisins de p;

Variables partagées *:*
$L.p$: entier non borné;

Variables locales *:*
$L_{copy}[|N.p|]$: tableau d'entiers non bornés contenant les copies des variables L des voisins de p;
CS : flag booléen utilisé pour signaler si p est en section critique ou non.

Fonction *:*
$(\forall q \in N.p) : p \prec q \equiv L_{copy}[q] \rhd L.p$

Actions *:*
A1 : $\forall q \in N.p, L_{copy}[q] \rhd L.p \longrightarrow$
$CS = 1;$ *exécuter la section critique;*
$L.p = max\{L_{copy}[q'] | q' \in N.p\} + 1;$
CP1 : $\exists q \in N.p, L.p \rhd L_{copy}[q] \wedge L_{copy}[q] \neq L.q \longrightarrow$
$CS = 0; L_{copy}[q] = L.q;$

Définissons maintenant la relation $\rhd$. Elle est basée sur la différence entre les variables L des processus voisins. Une arête est orientée du processus qui a la plus grande valeur de L vers celui qui a la plus petite valeur de L. En cas d'égalité, on utilise la différence entre les identifiants uniques des processus.

Définition 46 (Orientation d'une arête) *Pour toute paire de processus voisins p_1 et p_2 qui exécutent l'algorithme ULME, p_2 est virtuellement orienté vers p_1 dénoté $L.p_2 \rhd L.p_1$ si et seulement si $(L.p_1 < L.p_2) \vee ((L.p_1 = L.p_2) \wedge (id.p_1 < id.p_2))$ où $L.p_i$ est la variable partagée L du processus p_i et $id.p_i$ est son identifiant unique.*

Propriété 3 *Du fait de l'unicité des identifiants, la relation $\rhd$ est une relation d'ordre total (anti-reflexive, anti-symétrique et transitive).*

Preuve

- Anti-reflexive : si $L.p \rhd L.p$, cela implique que $id.p < id.p$ ce qui est impossible.
- Anti-symétrique : si $L.p_1 \rhd L.p_2$ et $L.p_2 \rhd L.p_1$, cela implique que $L.p_1 < L.p_2 < L.p_1$ ce qui est impossible ou que $L.p_1 = L.p_2$ ce qui implique $id.p_1 < id.p_2 < id.p_1$, ce qui est impossible.

– Transitive : supposons que $L.p_1 \rhd L.p_2$ et que $L.p_2 \rhd L.p_3$. Soit $L.p_1 < L.p_2$ et $L.p_2 < L.p_3$: dans ce cas, la relation est bien transitive. Soit $L.p_1 = L.p_2$ et $L.p_2 < L.p_3$, par conséquent, $L.p_1 < L.p_3$ et la relation est bien transitive. Soit $L.p_1 < L.p_2$ et $L.p_2 = L.p_3$, par conséquent, $L.p_1 < L.p_3$ et la relation est bien transitive. Soit enfin, $L.p_1 = L.p_2 = L.p_3$ dans ce cas, on a $id.p_1 < id.p_2$ et $id.p_2 < id.p_3$, par conséquent, $L.p_1 < L.p_3$ et la relation est bien transitive.

□

Propriété 4 *La règle A1 a pour effet de renverser les arêtes du processus privilégié qui l'exécute.*

Preuve Lors de l'exécution de la règle $A1$, la valeur de $L.p$ passe à $max\{L_{copy}[q'] | q' \in N.p\} + 1$. Soit q le voisin de p qui possède la plus grande valeur. $L.p$ devient $L.q + 1$. En effet, nous verrons que la copie ne peut que refléter la vraie valeur de la variable partagée. Par conséquent, quel que soit un voisin p' de p, on a $L.p \rhd L.p'$ car $L.p' < L.p$ et donc l'arête correspondante est bien sortante pour p. □

4.3.2 Lemmes préliminaires

Nous allons montrer quelques lemmes préliminaires qui nous servirons dans la preuve de correction de l'algorithme.

Définition 47 (états légitimes) *Une configuration est dite légitime pour l'algorithme ULME si et seulement si : (i) au moins un processus est privilégié; (ii) deux processus voisins ne sont pas privilégiés.*

Lemme 2 *Soit G le graphe de communications représentant le système qui exécute l'algorithme ULME. G est acyclique.*

Preuve Nous prouvons ce lemme par l'absurde : Supposons que G comporte un cycle, alors il existe G1=(V1,E1) un sous-graphe de G avec V1=p_1,...,p_m tel que les noeuds de V1 forment un cycle. Par la définition de l'orientation virtuelle (définition 46), on sait que les processus p_i satisfont la relation suivante : $L.p_1 \rhd L.p_2 \rhd ... \rhd L.p_n \rhd L.p_1$. Par conséquent $L.p_1 \rhd L.p_1$ ce qui est impossible car la relation $\rhd$ est anti-reflexive (propriété 3). □

Lemme 3 *Dans toute configuration de l'algorithme ULME, il existe au moins un processus privilégié.*

Preuve La preuve est directe par l'application du lemme 2 et de la propriété 2. □

Lemme 4 *Dans toute configuration de l'algorithme ULME, deux voisins ne peuvent être privilégiés en même temps.*

Preuve Supposons que deux processus voisins p_1 et p_2 soient tous les deux privilégiés. Par conséquent, on a la relation $L.p_1 \rhd L.p_2$ et $L.p_2 \rhd L.p_1$ ce qui est impossible car la relation $\rhd$ est anti-symétrique. □

Lemme 5 *Entre deux sections critiques d'un même processus p, ses voisins exécutent leur section critique exactement une fois.*

Preuve Soit e une exécution de l'algorithme ULME et soit f un fragment de e compris entre deux configurations c_1 et c_2. Supposons que le processus p soit privilégié dans les deux configurations c_1 et c_2 et qu'il n'existe aucune autre configuration de f où p soit privilégié. Selon la définition des processus privilégiés (définition 44), toutes les arêtes de p sont entrantes dans c_1 comme dans c_2. Quand p exécute sa section critique, toutes ses arêtes sont renversées, c'est-à-dire qu'elles deviennent sortantes pour p. Pour que p puisse exécuter sa section critique à nouveau, il est nécessaire que toutes ses arêtes soient renversées une fois de plus. Or, chaque renversement fait suite à une section critique. Par conséquent, entre deux sections critiques de p, tous ses voisins exécutent au moins une fois leur section critique. De plus, après la section critique d'un de ces voisins, l'arête commune est entrante pour p. Par conséquent, ce voisin ne pourra exécuter une nouvelle section critique qu'après p. En conclusion, entre deux sections critiques d'un processus, chacun de ses voisins exécute exactement une fois sa section critique. □

4.3.3 Preuve de l'algorithme ULME

Nous allons maintenant montrer la validité de cet algorithme. Tout d'abord, nous montrerons que cet algorithme converge puis qu'il est correct.

Lemme 6 *Toute exécution de l'algorithme ULME est infinie.*

Preuve Soit e une exécution finie de l'algorithme ULME et soit c sa configuration finale. D'après le lemme 3, il existe un processus privilégié dans la configuration c, par conséquent, ce processus peut exécuter la règle $A1$, ce qui contredit le fait que c soit une configuration finale. □

Lemme 7 (Vivacité) *Tout processus est privilégié infiniment souvent dans toute exécution de l'algorithme ULME.*

Preuve Soit e une exécution de l'algorithme ULME telle qu'il existe un processus p qui est privilégié un nombre fini de fois. Par le lemme 5, les voisins de p sont privilégiés également un nombre fini de fois. Or, le graphe de communications est connexe et acyclique, donc tous les processus sont privilégiés un nombre fini de fois. Cela implique que l'exécution est finie ce qui est en contradiction avec le lemme 6. □

Lemme 8 (Péremption des copies) *Lorsqu'un processus exécute sa règle $A1$, hormis la première fois, les valeurs contenues dans ses variables de copies sont égales aux vraies valeurs : c'est-à-dire celles contenues dans les registres des processus voisins.*

Preuve Supposons qu'un processus p ait exécuté sa section critique, sa valeur de L devient plus grande que toutes les copies de ses voisins. Par conséquent, il ne peut exécuter à nouveau sa section critique que si les valeurs de tous ses voisins ont changé et qu'il les a toutes lues. En effet, il ne pourra effectuer sa nouvelle section critique que si sa valeur de L est plus petite que toutes ses copies. Par conséquent, au moment de l'exécution de la règle $A1$ par p, toutes les valeurs auront été lues au moins une fois après leur modification par ses voisins. □

Lemme 9 (Convergence) *Toute exécution de l'algorithme ULME atteint une configuration légitime.*

Preuve Nous avons montré que toute configuration où la valeur des copies est identique aux valeurs des registres de voisins est légitime par les propositions 3 et 4. Par le lemme précédent, nous avons montré que si tous les processus ont effectué au moins une fois leur section critique, les valeurs des copies sont en accord avec les valeurs réelles de variables L. On sait également par le lemme 7 que tous les processus exécutent leur section critique infiniment souvent. Par conséquent, un état légitime (c'est-à-dire une configuration telle que tous les processus ont exécuté au moins une section critique) est toujours atteint au bout d'un temps fini. □

Théorème 1 (Auto-stabilisation) *L'algorithme ULME est auto-stabilisant pour l'ensemble des ses états légitimes.*

Preuve La correction résulte des lemmes 3 et 4 pour la sûreté et du lemme 7 pour la vivacité. La convergence est démontrée par le lemme 9. □

4.3.4 Index d'équité et temps de service

Dans cette sous-section, nous calculons l'index d'équité et le temps de service de l'algorithme ULME.

Définition 48 (Distance) *Soit p et q deux processus. La distance entre p et q, notée $distance(p, q)$, est le nombre de processus sur le plus court chemin de p à q plus 1* *dans le graphe de communications. Nous notons $SPdist_i.p$ l'ensemble des processus à distance i de p et $|SPdist_i.p|$ le cardinal de cet ensemble.*

Lemme 10 *Dans toute exécution de l'algorithme ULME, entre deux exécutions de la section critique d'un processus p, les éléments de $SPdist_i.p$ exécutent au plus i fois leur section critique.*

Preuve Nous montrons ce résultat par récurrence sur la distance i.

- Cas de base : $i = 1$
  Les processus de $SPdist_1.p$ sont les voisins directs de p. Par le lemme 5, ils exécutent exactement une fois leur section critique.
- Induction :
  Supposons que le lemme soit vrai pour une distance i de p et montrons qu'il est vrai pour une distance $i+1$. Soit q un processus de $SPdist_{i+1}.p$. Par construction, il existe un chemin entre p et q de distance $i+1$. De plus, ce chemin est minimal. Soit q' l'avant-dernier processus de ce chemin : il est à distance 1 de q et à distance i de p. Par conséquent, pour tout processus q de $SPdist_{i+1}.p$, il existe un processus q' de $SPdist_i.p$ tel que q soit dans l'ensemble des voisins de q'. En utilisant l'hypothèse inductive, nous savons qu'entre deux sections critiques de p, q' exécute au plus i sections critiques. Par le lemme 5, nous savons qu'entre deux sections critiques de q', q exécute exactement une section critique. Par conséquent, si q' exécute i sections critiques, q peut exécuter au plus $i+1$ sections critiques. Par conséquent, entre deux sections critiques de p, q peut exécuter au plus $i+1$ section critique.

□

Lemme 11 (Index d'équité) *L'index d'équité de l'algorithme ULME est* $(n-1)$.

Preuve Soit e une exécution de l'algorithme ULME. Le lemme 7 montre que tout processus exécute infiniment souvent sa section critique. Soit f un fragment de e compris entre deux configurations où un processus p est privilégié. Supposons qu'il n'existe pas d'autre configuration de f où p exécute sa section critique. Conformément au lemme 10, les processus à distance i de p exécutent au plus i sections critiques dans f. Or la distance maximale d'un processus à p dans le graphe de communications est $(n-1)$ où n est le nombre de processus du système. Par conséquent, dans f, tout processus peut exécuter au plus $(n-1)$ fois sa section critique et donc l'index d'équité de l'algorithme est $(n-1)$. □

Lemme 12 *La borne supérieure du temps de service pour un processus p est la valeur maximale de l'expression*

$$\Sigma_{i=1}^{n-1} i.neigh_i$$

où $neigh_i = |SPdist_i.p|$.

Preuve Par le lemme 10, nous savons qu'entre deux sections critiques d'un processus p, ses voisins à distance i peuvent exécuter au plus i sections critiques. Par conséquent, entre deux sections critiques de p, ses voisins à distance i peuvent exécuter $i.neigh_i$ sections critiques. □

Lemme 13 (Temps de service) *Le temps de service de l'algorithme ULME est borné par* $\frac{n(n-1)}{2}$.

Preuve La valeur maximale de $\Sigma_{i=1}^{n-1} i.neigh_i$ est atteinte pour $\forall i, neigh_i = 1$. En effet, si il existe i tel que $neigh_i > 1$ alors on a $neigh_{n-1} = 0$ par construction. Par conséquent, l'expression diminue de $n-1$ et augmente de $i < n-1$.

Par conséquent, la valeur maximale de l'expression est $\Sigma_{i=1}^{n-1} i = \frac{n(n-1)}{2}$ et elle est atteinte.

□

4.4 Exclusion mutuelle locale utilisant une mémoire bornée

L'inconvénient principal de l'algorithme ULME est sa mémoire non bornée. Dans cette section, nous proposons une version bornée de cet algorithme. Pour cela, nous utilisons une comparaison cyclique pour les variables L. Cette comparaison implique l'utilisation d'un mécanisme de réinitialisation. Ce nouvel algorithme est auto-stabilisant sous un démon réparti, maintient l'index d'équité et fonctionne en atomicité lecture/écriture.

La relation d'adjacence est définie sur les variables L bornées. L'idée de base est que les valeurs de L ne sont jamais éloignées de plus de $n-1$ entre deux voisins. Par conséquent, il est possible de comparer deux valeurs non plus suivant l'ordre des entiers mais suivant leur position relative dans un intervalle borné.
Nous introduisons la constante B (définie dans l'algorithme) comme étant n^2 si n est pair et n^2+1 sinon. Cette constante nous sert de borne pour les variables L. Sa valeur sera justifiée dans la suite.

Définition 49 (Comparaison cyclique) *Soit x et y deux entiers positifs compris entre 0 et B-1. Nous définissons la relation $\rhd$ comme suit :*

- $\forall x \in [0, \frac{B}{2}]$
  - $y \rhd x$ *ssi* $y \in [x+1, x+\frac{B}{2}]$
  - $x \rhd y$ *ssi* $y \in [x+\frac{B}{2}+1, B-1] \cup [0, x-1]$

- $\forall x \in [\frac{B}{2}+1, B-1]$
  - $y \rhd x$ *ssi* $y \in [x+1, B-1] \cup [0, x+\frac{B}{2}]$
  - $x \rhd y$ *ssi* $y \in [x+\frac{B}{2}+1, x-1]$

La figure 4.3 montre la comparaison cyclique. A gauche, on a $x \rhd y1$, $y2 \rhd x$ et $y3 \rhd x$. A droite, on a $x \rhd y1$, $x \rhd y2$ et $y3 \rhd x$. Cela signifie qu'on divise l'espace borné en deux parties : la partie qui va de x à $x + B/2$ et toutes les valeurs y dans cette partie sont plus grandes que x : $x \rhd y$. Dans la deuxième partie, qui va de $x+B/2+1$ à $x-1$, toutes les valeurs y sont plus petites que x : $y \rhd x$. Il faut noter que cette relation ne définit pas un ordre total, en effet, il est possible d'exhiber des valeurs x, y1 et y2 telles que $x \rhd y1$, $y1 \rhd y2$ et $y2 \rhd x$ ce qui implique que $\rhd$ n'est pas anti-reflexive.

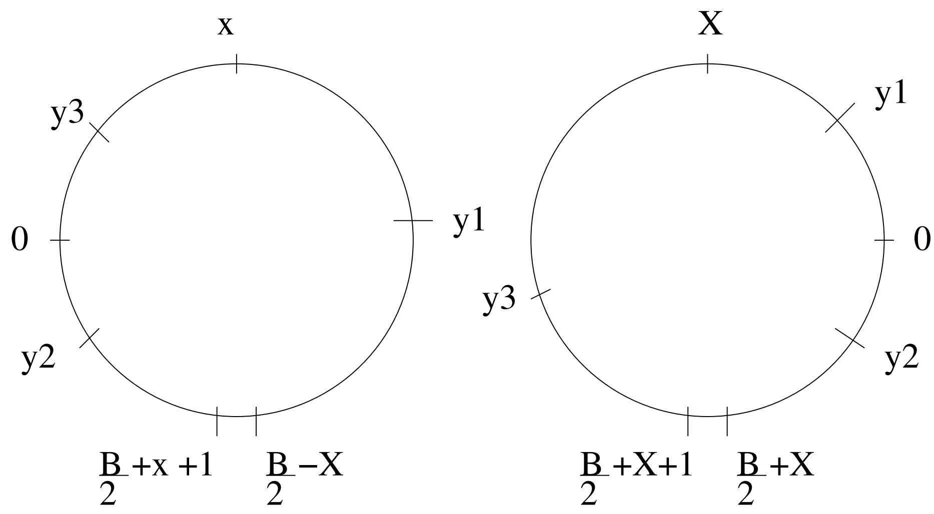

FIG. 4.3 – *Comparaison cyclique*

A partir de cette relation, nous redéfinissons l'orientation virtuelle du graphe de communications basées sur les variables L bornées par B.

Définition 50 (Orientation virtuelle bornée) *Soit p et q deux processus voisins qui exécutent l'algorithme BLME. p est virtuellement orienté vers q si et seulement si* $L.p \rhd L.q$ *pour la définition 49.*

Comme $\rhd$ n'est pas un ordre total, il est nécessaire de contrôler la différence entre les valeurs de L pour des processus voisins. Nous dirons que deux processus sont équilibrés si la différence entre leurs variables L n'excède pas $(n-1)$.plus formellement :

Définition 51 (Processus équilibrés) *Soit p et q deux processus voisins exécutant l'algorithme BLME. On dit que p et q sont équilibrés, noté* $p \sim q$*, si et seulement si*

$|L.p_1 - L.p_2| < n \wedge ((L.p_1 \neq L.p_2) \vee (L.p_1 = L.p_2 = 0)$
Une configuration où tous les processus voisins sont équilibrés deux à deux est appelée configuration équilibrée.
Si les deux processus ne sont pas équilibrés, on dit qu'il sont déséquilibrés et on le note $p \not\sim q$.
Si au moins deux processus voisins ne sont pas équilibrés dans une configuration, celle-ci est dite déséquilibrée.

4.4.1 Algorithme BLME

Algorithme 2 (BLME pour le processus p)

Constantes *:*
B *: vaut* n^2 *si* n *est pair,* n^2+1 *si* n *est impair*
$id.p$ *: Identifiant entier unique de* p*;*
$N.p$ *: L'ensemble des voisins de* p*;*

Variables partagées *:*
$L.p$ *: entier* $\in [0..B-1]$*;*
$R.p$ *: booléen indicateur de réinitialisation.*
Variables locales *:*
$L_{copy}[|N.p|]$ *: tableau d'entiers contenant les copies des variables* L *des voisins de* p*;*
$R_{copy}[|N.p|]$ *: tableau de booléens contenant les copies des variables* R *des voisins de* p*;*
CS *: flag booléen utilisé pour signaler si* p *est en section critique ou non.*

Fonction *:*
$MaxL(p) = L.i$ *tel que* $|L\_copy[i] - L.p| = max(|L\_copy[j] - L.p|, \forall j \in \mathcal{N}.p)$;
$(\forall i \in \mathcal{N}.p) : p \prec i \equiv (L.p \lhd L\_copy[i]) \vee ((L.p = L\_copy[i] = 0) \wedge (id.p < id.i))$;
$(\forall i \in \mathcal{N}.p) : p \sim i \equiv |L.p - L\_copy[i]| < n \wedge (L.p \neq L\_copy[i]) \vee (L.p = L\_copy[i] = 0 \wedge id.p < id.i)$;
$(\forall i \in \mathcal{N}.p) : p \not\sim i \equiv |L.p - L\_copy[i]| \geq n \vee ((L.p = L\_copy[i]) \wedge (L.p \neq 0))$;

Actions *:*
$\mathcal{A}_1$: $(R.p = 0) \wedge (\forall i \in \mathcal{N}.p)\ (p \prec i) \wedge (p \sim i) \wedge (R\_copy[i] = 0) \longrightarrow$
  CS=1;
  exécuter la section critique;
  $L.p = MaxL(p) + 1$;
$\mathcal{R}_1$: $(R.p = 0) \wedge (\exists i \in \mathcal{N}.p\ (p \not\sim i) \vee ((R.i = 1)\ \wedge((L.i \neq 0) \vee (L.p \neq 0)))) \longrightarrow$
  CS=0;
  R.p=1;
$\mathcal{R}_2$: $(R.p = 1) \wedge (L.p \neq 0) \wedge (\forall i \in \mathcal{N}.p)\ (R\_copy[i] = 1) \longrightarrow$

$CS{=}0;$
$L.p = 0;$
$\mathcal{R}_3$: $(R.p = 1) \wedge (L.p = 0) \wedge (\forall i \in \mathcal{N}.p) L\_copy[i] = 0 \longrightarrow$
$CS{=}0;$
$R.p{=}0;$
$\mathcal{CP}_1$: $R.p = 0 \wedge (\forall i \in \mathcal{N}.p, (p \sim i) \wedge R\_copy[i] = 0) \wedge (\exists i \in \mathcal{N}.p, i \prec p \wedge L\_copy[i] \neq L.i) \longrightarrow$
$CS{=}0;$
$L\_copy[i]{=}L.i;$
$\mathcal{CP}_2$: $R.p = 1 \wedge L.p \neq 0 \wedge (\exists i \in \mathcal{N}.p, R\_copy[i] = 0 \wedge R.i = 1) \longrightarrow$
$CS{=}0;$
$R\_copy[i]{=}R.i;$
$\mathcal{CP}_3$: $R.p = 1 \wedge L.p = 0 \wedge (\exists i \in \mathcal{N}.p, L\_copy[i] \neq 0 \wedge L.i = 0) \longrightarrow$
$CS{=}0;$
$L\_copy[i]{=}L.i;$

4.4.2 Principe de l'algorithme

Notre algorithme se déroule en deux phases distinctes : la phase de convergence et la phase de correction. La phase de correction commence lorsque le système atteint une configuration légitime, c'est-à-dire une configuration équilibrée. En effet, nous montrons dans la section suivante que si la configuration est équilibrée alors, l'orientation virtuelle bornée est acyclique et donc qu'il existe un noeud puits dans le système. Nous montrons également que lorsqu'un processus privilégié exécute sa section critique, et donc renverse ses arêtes, la configuration atteinte reste légitime.

Lors de la phase de convergence, le système va passer d'un état quelconque, c'est-à-dire potentiellement déséquilibrée à un état légitime. Cette convergence est mise en oeuvre via un mécanisme de réinitialisation. Chaque processus dispose d'une variable R qui est un marqueur pour indiquer à ses voisins qu'il est déséquilibré ou qu'il a détecté un déséquilibre dans la configuration courante. Dans une configuration équilibrée, les variables R de tous les processus doivent être à 0.

La réinitialisation a pour but de faire passer les variables L de tous les processus à 0 afin de repartir sur une configuration où les différences entre les variables L sont maîtrisées. Lorsqu'il détecte un déséquilibre avec un de ses voisins, un processus met sa variable R à 1. C'est la première étape de la réinitialisation. Ensuite, ce processus ne fait plus rien jusqu'à ce que tous ses voisins aient leur variable R à 1 également. A ce moment, le processus considéré fait passer sa variable L à 0 tout en maintenant sa variable R à 1. C'est la deuxième phase de la réinitialisation. Ensuite, le processus ne fait rien tant que tous ses voisins n'ont pas L à 0 et R à 1. Enfin, quand tous ses voisins sont identiques, il met sa variable R à 0. La réinitialisation est terminée pour ce processus.

Nous montrons dans la section suivante que dans toute exécution, chaque processus fait au plus une réinitialisation. Informellement, la réinitialisation va se propager comme une vague grâce au mécanisme d'attente. Ainsi, un processus va lancer la vague qui va se propager de tous les côtés à la fois et ceci jusqu'aux extémités du système. L'algorithme est conçu pour que si deux vagues différentes sont lancées en même temps, elles fusionnent au niveau des processus frontières.

Notre algorithme contient trois ensembles de règles : la règle $A1$ qui est la règle de base, c'est-à-dire la règle de retournement d'arêtes. Les règles $R1$, $R2$ et $R3$ correspondent respectivement aux trois phases successives de la réinitialisation, elles sont appelées au plus une fois sur chaque processus. Les règles $CP1$, $CP2$ et $CP3$ permettent de recopier les valeurs des voisins afin de permettre à l'algorithme de fonctionner en atomicité lecture/écriture.

Comme pour l'algorithme ULME, la péremption des copies n'est pas un problème car les processus doivent attendre que leurs voisins changent leurs valeurs pour pouvoir changer eux-mêmes. Par conséquent, une valeur ne peut être changée avant d'avoir été lue.

4.4.3 Preuve de l'algorithme

Dans cette section, nous prouvons des lemmes techniques qui permettent d'établir la convergence et la correction de l'algorithme.

Lemme 14 *Soit G le graphe de communications d'un système qui exécute l'algorithme BLME. Dans toute configuration équilibrée du système, le graphe G est acyclique.*

Preuve Nous montrons ce lemme par l'absurde. Supposons que dans une configuration équilibrée, il existe un sous-graphe $G1 = (V1, E1)$ tel que les noeuds de $V1 = \{p_1, p_2, ..., p_m\}$ forment un cycle simple. Nous ne considérons que le cas où les valeurs de la variable L des processus de $V1$ sont différentes de 0. En effet, dans ce cas, l'unicité des identifiants garantit l'acyclicité du graphe. Par la définition de l'orientation virtuelle (définition 46), les processus correspondants aux noeuds de $V1$ satisfont la relation

$$L.p_1 \triangleright L.p_2 \triangleright ... \triangleright L.p_m \triangleright L.p_1 \tag{4.1}$$

puisque tout processus p_i, $i \in [1..m]$ est équilibré avec ses voisins, en appliquant la définition des processus équilibrés (définition 51).

$$|L.p_i - L.p_{i+1}| < n \tag{4.2}$$

cela implique

$$|L.p_1 - L.p_2| + |L.p_2 - L.p_3| + ... + |L.p_{m-1} - L.p_m| + |L.p_m - L.p_1| < B \tag{4.3}$$

L'équation 4.1 peut être écrite de la manière suivante :

$$|L.p_1 - L.p_2| + |L.p_2 - L.p_3| + ... + |L.p_{m-1} - L.p_m| + |L.p_m - L.p_1| = k * B \quad (4.4)$$

où k est un entier positif et non nul. Les deux équations précédentes se contredisent mutuellement et donc, dans toute configuration équilibrée, le graphe de communications a une orientation acyclique. □

Définition 52 (Configurations légitimes) *Une configuration légitime du système exécutant l'algorithme BLME est une configuration équilibrée où la variable R de tous les processus est égale à 0.*

Lemme 15 (Correction) *Soit e une exécution de l'algorithme BLME commençant par une configuration légitime. Toutes les configurations de e sont des configurations légitimes.*

Remarque Si on s'abstrait des règles de copies, un processus ne peut exécuter que la règle $A1$ dans une configuration équilibrée.

Preuve Soit c la configuration initiale de l'exécution e. Soit p un processus qui exécute la règle $A1$ dans c. Soit c' la configuration qui suit c dans e. Cela n'est possible que si toutes les arêtes de p sont entrantes. C'est-à-dire :

$$\forall i \in N.p, L.i \triangleright L.p \quad (4.5)$$

Soit i_{max} et i_{min} les voisins de p représentant les processus qui ont les valeurs maximales et minimales pour leurs variables L. Soit $L.p_{old}$ et $L.p_{new}$ les valeurs de la variable L de p dans c et c'. Puisque p a exécuté la règle $A1$, alors on a $L.p_{new} = L.i_{max} + 1$. L'équation 4.5 implique $L.i_{min} \triangleright L.p_{old}$, c'est-à-dire $|L.p_{old} - L.i_{max}| < n$.

Ces relations impliquent que la configuration c' est équilibrée. Formellement :

$$|L.p_{new} - L.i_{min}| = |L.i_{max} - L.p_{old}| < n \quad (4.6)$$

□

Pour démontrer la convergence de l'algorithme, nous définissons des ensembles d'arêtes dont l'union est l'ensemble des arêtes du graphe de communications de toute configuration.

1. $M1 = \{(p, q)|$ p est q sont équilibrés et $(R.p = 0 et R.q = 0)$
2. $M2 = \{(p, q)|$ p est q sont équilibrés et $(R.p = 1 ou R.q = 1)$
3. $M3 = \{(p, q)|$ p est q sont déséquilibrés et $(R.p = 0 ou R.q = 0)$
4. $M4 = \{(p, q)|$ p est q sont déséquilibrés et $(R.p = 1 et R.q = 1)$

5. $MT = \cup_{i=1}^{4} M_i$

6. $\forall i \in \{2, 4\} : M_i^0 = \{(p, q) | (p, q) \in M_i et(L.p = 0 ou L.q = 0)\}$

7. $M^{(0,0)} = \{(p, q) | L.p = 0 et L.q = 0\}$

Dans toute configuration légitime, toutes les arêtes sont dans l'ensemble $M1$ ($MT = M1$) et la seule règle exécutable est $A1$. Quand le système est dans une configuration illégitime, alors $MT \neq M1$ et selon l'algorithme, les règles exécutables sont les règles de type R (c'est-à-dire celles qui assurent la réinitialisation des variables L). Par la suite, nous montrons qu'à partir d'une configuration où $MT \neq M1$, toutes les arêtes deviennent éléments de l'ensemble $M1$ suivant un schéma de migration d'arêtes représenté dans la figure 4.4.

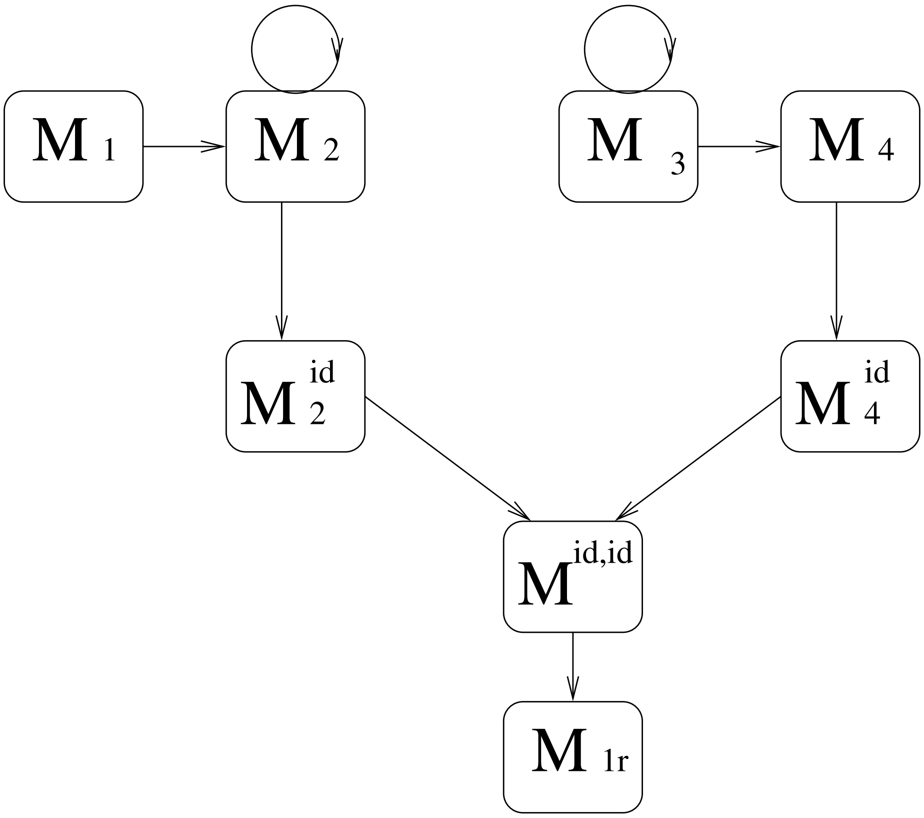

FIG. 4.4 – *Schéma de migration des arêtes.*

La preuve de la convergence sera établie en deux temps. Tout d'abord, nous montrerons que dans toute configuration illégitime, une phase de réinitialisation est amorcée. Dans un deuxième temps, nous montrerons que les arêtes suivent le schéma de migration représenté par la figure 4.4. C'est-à-dire que :

– Toute arête de M_i devient élément de $M_i^0 (i \neq 0)$ ou de $M^{(0,0)}$

– Toute arête de $M_i^0 (i \neq 0)$ devient élément de $M^{(0,0)}$

– Toute arête de $M^{(0,0)}$ devient une arête de M_1

Lemme 16 *Soit e une exécution de l'algorithme BLME. Soit c la configuration initiale de e. Si c est une configuration illégitime c'est-à-dire que $MT \neq M_1$, au bout d'un temps fini, une des règles R_i sera exécutée.*

Preuve Il existe dans la configuration c un processus capable d'exécuter une des règles R_i sinon la configuration serait légitime. Soit SR l'ensemble des processus qui peuvent exécuter une de ces règles dans c. Par construction de l'algorithme, aucun de ces processus n'est en mesure d'exécuter la règle A_1. Supposons que tout au long de l'exécution e, tous les processus exécutent uniquement la règle A_1, cela implique qu'il existe un sous-graphe correspondant aux processus équilibrés qui soit cyclique. Le lemme 14 montre que le graphe de communications est acyclique si tous les processus sont équilibrés. Par conséquent, ce sous-graphe n'existe pas et donc il existe une configuration dans e telle qu'au moins un des processus de SR exécute une règle R_i. $\square$

Définition 53 *M_{1r} est équivalent à l'ensemble M_1. La différence entre les deux est que l'ensemble M_{1r} ne peut être obtenu qu'après la convergence. En effet, nous montrons que la correction se fait par une seule vague qui parcourt tout le graphe. Par conséquent, si une arête est équilibrée (ses deux processus sont équilibrés), elle peut se déséquilibrer mais une seule fois. C'est pourquoi nous faisons une distinction entre M_{1r} et M_1.*

Lemme 17 *Soit G le graphe de communications d'un système qui exécute l'algorithme BLME. Soit e une exécution de ce système commençant par une configuration c où $MT \neq M_1$ Toutes les arêtes seront éléments de M_{1r} au bout d'un temps fini. Si tous les processus sont équilibrés avec leurs voisins, nous considérerons que toutes les arêtes sont dans M_{1r}.*

Preuve Soit (p,q) une arête de G qui n'est pas élément de M_{1r} dans c. Nous considérons les cas suivant :

– (p,q) $\in M_1$. On distingue deux cas :

1. p est en équilibre avec tous ses voisins. p peut exécuter uniquement la règle A_1 et (p,q) reste dans l'ensemble M_1. Notons qu'il n'est pas possible que le système reste indéfiniment dans cet état, en effet, le lemme 16 montre que si la configuration est illégitime, une règle R_i finira par être appliquée. Par propagation, toutes les arêtes seront concernées par ce changement que nous décrivons ci-dessous.

2. p a un voisin s qui se déséquilibre, c'est-à-dire que sa variable R.s devient 1. La seule règle que p pourra appliquer sera la règle R_1. C'est-à-dire qu'il positionnera sa variable R à 1. L'arête (p,q) va donc passer de l'ensemble M_1 à l'ensemble M_2.

– (p,q) $\in M_2$. Les deux processus sont en première phase de réinitialisation. Un des deux processus va alors exécuter sa règle R_2 (la seule dont la partie gauche soit valide). Cela va forcer l'autre processus à exécuter également sa règle R_2. Les deux processus seront alors dans la position où ils ont tous deux remis leur variable L à 0 mais leur variable R est toujours à 1. L'arête est donc passée dans l'ensemble M_2^0.

– (p,q) $\in M_2^0$ Les deux processus ne peuvent exécuter qu'une seule règle : R_3. Celle-ci va donc s'exécuter sur un des deux processus, puis sur l'autre. La situation sera la fin de la réinitialisation : c'est-à-dire que les deux processus ont leurs variables R et L à 0. L'arête passe donc dans l'ensemble $M^{(0,0)}$. L'arête est orientée par les identifiants uniques des deux processus. C'est d'ailleurs le seul cas où ceux-ci seront utilisés.

– (p,q) $\in M^{(0,0)}$ Une fois la réinitialisation terminée, l'arête sera équilibrée à la première action A_1 qui sera exécutée par p ou q. L'arête devient donc élément de M_{1r}.

Si on part d'un arête (p,q) non équilibrée (i.e. $\in M_3$), le raisonnement est strictement identique : le passage des variables L à 0 rééquilibre l'arête. □

Remarque Les lemmes 11 et 13 sont valables pour l'algorithme BLME puisque l'orientation des arêtes change de manière identique dans les deux algorithmes.

Théorème 2 *L'algorithme BLME est un algorithme auto-stabilisant pour la spécification de l'exclusion mutuelle locale sous un démon distribué avec un index d'équité égal à* $(n-1)$ *et un temps de service égal à* $\frac{n(n-1)}{2}$.

Preuve Les lemmes 16 et 17 prouvent la convergence de l'algorithme. Le lemme 15 prouve la correction de l'algorithme. Les valeurs de l'index d'équité et du temps de service sont fournies par les lemmes 11 et 13. □

4.5 Exclusion mutuelle locale et composition croisée

Dans cette section, nous proposons une application de l'exclusion mutuelle locale dans le cadre de la composition croisée. La composition croisée est une technique qui permet de "fortifier" un algorithme auto-stabilisant dans son interaction avec le démon. D'une manière informelle, un algorithme A, auto-stabilisant sous un démon faible (par exemple, le démon central) peut être transformé en un algorithme A' auto-stabilisant pour la même spécification sous un démon plus fort. Cette transformation est une composition avec un algorithme B (appelé module fort) dont les exécutions vérifient certaines propriétés (équité ou centralité locale dans notre cas) sous un démon fort (par exemple, le démon distribué). L'idée de base est que les actions de l'algorithme initial (le module faible) sont exécutées en même temps que certaines actions du module fort. Or, le module fort est choisi précisément pour les propriétés de ses exécutions qu'il transmet par composition au module faible.

Dans notre cadre, l'algorithme d'exclusion mutuelle locale sert de module fort. Nous utiliserons de préférence BLME qui n'utilise qu'une mémoire bornée. Le module faible ne sera autorisé à exécuter une action que lorsque le module fort sera en exclusion mutuelle, simulant ainsi un démon quasi-central.

Formellement, on peut exprimer la composition croisée de la manière suivante :

Définition 54 (Composition croisée) *Soit A un algorithme quelconque auto-stabilisant sous le démon quasi-central du type :*
$\forall i \in \{1,..,n\}, <$ *garde* $a_i >\rightarrow<$ *action* $a_i >$.
Soit B l'algorithme d'exclusion mutuelle locale du type :
$\forall i \in \{1,..,m\}, <$ *garde* $b_i >\rightarrow<$ *action* $b_i >$.
La composition croisée des algorithmes A et B, notée $A \diamond B$ *est l'algorithme à* $(n+1)\times m$ *règles :*
$\forall i \in \{1,..,n\}, \forall j \in \{1,..,m\}$
$<$ *garde* $a_i > \wedge <$ *garde* $b_i >\rightarrow<$ *action* $b_j >$; *si CS=1 alors* $<$ *action* $a_i >$
$\forall j \in \{1,..,m\}$,
$\neg <$ *garde* $a_1 > \wedge ... \wedge \neg <$ *garde* $a_n > \wedge <$ *garde* $b_j >\rightarrow<$ *action* $b_j >$

Cette composition garantit au module faible une exécution avec les propriétés du démon quasi-central comme le démontre le lemme suivant.

Lemme 18 *La projection de toute exécution de l'algorithme composé sur le module faible a un suffixe maximal où les processus voisins n'exécutent pas leurs actions en même temps.*

Preuve Soit e une exécution de l'algorithme composé. Comme nous l'avons démontré précédemment, l'exécution de la section critique sur deux processus voisins ne peut être simultanée. Soit p la projection de e sur le module faible. Supposons que dans cette projection, il existe une configuration telle que deux actions de deux processus voisins sont exécutées conjointement. Cela implique que dans la projection de e sur le module fort, il existe une configuration telle que les valeurs de la variables CS soient 1 sur deux processus voisins. Ceci est en contradiction avec la propriété d'exclusion mutuelle locale du module fort. □

4.6 Résumé

Dans ce chapitre, nous présentons deux algorithmes d'exclusion mutuelle locale. Le premier, qui utilise une mémoire non bornée, est une version simplifiée du second, qui utilise une mémoire bornée. Ces deux algorithmes sont auto-stabilisants sous le démon distribué et supportent l'atomicité lecture/écriture. Dans la dernière section, nous présentons une application de ces algorithmes. Ils sont utilisés comme filtres dans le contexte de la composition croisée pour assurer des exécutions simulant le démon quasi-central.

Chapitre 5

Preuves de convergence par réécriture

Dans ce chapitre, nous proposons une nouvelle technique de preuves pour les algorithmes auto-stabilisants. En effet, les techniques habituelles présentent quelques inconvénients : trouver un fonction décroissante pour l'algorithme considéré est souvent une tâche ardue et toujours à refaire car elle ne présente aucune généricité. Comme cette fonction est basée sur les variables et des propriétés intrinsèques de l'algorithme étudié, il est impossible d'utiliser la même fonction pour différents algorithmes. Dans ce chapitre, nous proposons de prouver la convergence des algorithmes par une technique basée uniquement sur l'alphabet formé par les états des processus et sans fonction décroissante. Nous l'étudierons ici pour des topologies linéaires (chaîne, anneau) et dans le cadre de la lecture d'états. Cette méthode est basée sur la similitude qu'il existe entre un système à topologie linéaire et un mot, ce qui permet de considérer les algorithmes comme des systèmes de réécriture du premier ordre. La convergence de l'algorithme se réduit donc à une preuve de terminaison dudit système. Cette technique est complétée par un système de généralisation des graphes de dérivations associés au système de réécriture par des expressions régulières. Le but de ce travail est d'automatiser au maximum le travail de preuve de convergence. Dans la méthode que nous proposons, nous automatisons complètement le calcul des graphes de dérivations ainsi que la recherche de cycles dans ces graphes. En revanche, la partie permettant de limiter la taille de ces graphes par généralisation est strictement intuitive et doit être effectuée par un opérateur humain.

5.1 Modélisation

Dans cette section, nous verrons comment on peut modéliser des systèmes répartis à topologie linéaire par des mots dans un alphabet fini; et des systèmes de transitions par des systèmes de réécriture de mots.

5.1.1 Les processus

Considérons une topologie linéaire (chaîne, anneau) sur laquelle des processus exécutent un algorithme. Les différents états des processus individuels forment un alphabet Σ que nous imposons fini (c'est-à-dire que la mémoire utilisée par chacun des processus est bornée). Nous l'enrichissons d'un ensemble de symboles de variables $\nu = \{W, X, Y ...\}$. Une chaîne est un élément de $\Sigma*$ avec ϵ pour le mot vide. Nous introduisons un symbole spécial, absent de Σ, noté $\#$, et qui sert à délimiter les mots. Un mot clos est donc un élément de $\#\Sigma * \#$. Un mot du premier ordre est un élément de $\#(\Sigma \cup \nu) * \#$. Les mots ainsi définis sont appelés des mots d'état et décrivent le système à un instant donné : complètement pour les mots clos, en partie pour les mots du premier ordre.

Considérons un exemple très simple : quatre ordinateurs sont reliés par une topologie en anneau. Ils exécutent un algorithme très simple qui n'utilise qu'une seule variable booléenne notée X. La figure 5.1 donne un exemple de configuration d'un tel système.

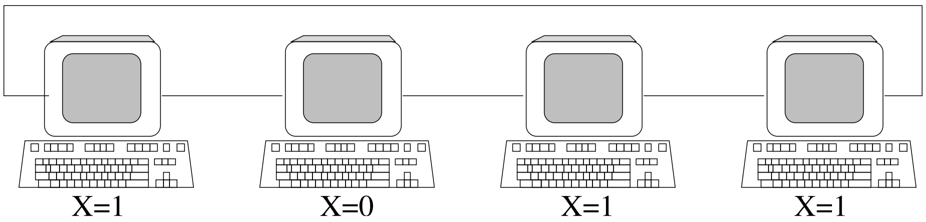

FIG. 5.1 – *exemple de configuration*

On peut modéliser cette configuration par le mot clos $\#1011\#$ avec $\Sigma = \{0, 1\}$. On peut également la modéliser par des mots du premier ordre comme par exemple $\#1X1\#$, $\#X11\#$ ou même $\#X\#$.

5.1.2 Les transitions

Une transition de l'algorithme fait changer l'état d'un processus en fonction de l'état de ses voisins. Nous nous plaçons dans l'optique d'un démon central ou quasi-central. Il est alors possible de modéliser une transition comme une réécriture d'un mot d'état en un autre. Un algorithme peut ainsi être modélisé par un système S de règles de réécriture. Notons que ces transitions préservent la longueur du mot.

On classe les différentes règles en trois ensembles en fonction de leurs interactions avec les bords. Top_S contient les règles qui s'appliquent à la partie la plus à droite des mots, $Bottom_S$ contient les règles qui s'appliquent à la partie la plus à gauche des mots ou éventuellement aux deux extrémités simultanément (dans le cas des anneaux). Enfin, $Middle_S$ contient toutes les autres règles. Plus formellement, soit S

un système de règles. Soit l et r (respectivement l_i,r_i pour i=1,2) des mots clos de $\Sigma*$ de même taille et X,Y des variables.

- $Middle_S$ est composé des règles du type : $\#XlY\# \rightarrow \#XrY\#$.
- Top_S est composé des règles du type : $\#Xl\# \rightarrow \#Xr\#$.
- $Bottom_S$ est composé des règles du type : $\#lX\# \rightarrow \#rX\#$ ou bien $\#l_1Xl_2\# \rightarrow \#r_1Xr_2\#$

Considérons un exemple simple sur la topologie de la figure 5.1. Chaque processus du système exécute un algorithme qui possède les deux règles suivantes :

- Si mon voisin de droite a sa variable X à 1 et que j'ai ma variable X à 0, alors, je mets ma variable X à 1.
- Si je suis le plus à droite et que ma variable X est à 0, alors je mets ma variable X à 1.

Cet algorithme peut être modélisé par l'ensemble de règles S suivant :

- T : 0# $\rightarrow$ 1#
- M : 01 $\rightarrow$ 11

Dans ce système, T est une règle de Top_S et M est une règle de $Middle_S$.

5.1.3 Réduction close

Définition 55 (Réduction) *Un mot clos w est dit réductible via une règle du type $\#Xl_1Y\# \rightarrow \#Xr_1Y\#$ si et seulement si il est de la forme $w = \#ul_1v\#$ où u et v sont des mots clos éléments de $\Sigma*$. La forme réduite de w est $w' = \#ur_1v\#$. On dit aussi que w est une instance de la partie gauche de la règle via la substitution close $\{X/u, Y/v\}$.*

On définit de manière similaire la réduction via une règle de type $\#l_1Xl_2\# \rightarrow \#r_1Xr_2\#$.

Notation 2 *La réduction est notée $w \rightarrow_S w'$ ou plus simplement $w \rightarrow w'$*

Par exemple, prenons un état de notre système d'exemple : #0100#. On voit que l'on peut appliquer la règle T. En effet, la chaîne 0# est bien présente dans le mot d'état. On peut donc appliquer cette règle et réécrire sous la forme :
#0100# $\rightarrow$ #0101#
De la même façon, on voit que la partie gauche de la règle M : 01 est présente dans le mot d'état originel . On peut le réécrire :
#0100# $\rightarrow$ #1100#

Définition 56 (Terminaison) *On dira que le système S ne termine pas si et seulement si il existe une suite infinie de réductions via S à partir d'un mot clos quelconque. Dans le cas contraire, on dira que S termine.*

Définition 57 (Cycle clos) *On appelle cycle clos toute suite infinie de réductions du type $w_1 \rightarrow w_2 \rightarrow ... \rightarrow w_n$ tel que $w_1 = w_n$.*

Définition 58 (Dérivation close) *Une dérivation close est une suite de réductions closes s'appliquant consécutivement.*

Définition 59 (Graphe de dérivations) *Le graphe de dérivations basé sur le mot m, est le graphe formé par toutes les dérivations possibles à partir de m. Chaque embranchement du graphe correspond aux différentes réductions possibles à partir du mot correspondant à l'embranchement. Ces graphes sont éventuellement infinis.*

5.1.4 Une caractérisation de l'auto-stabilisation pour les systèmes clos

A partir des définitions précédentes, nous caractérisons l'auto-stabilisation pour les systèmes clos. Soit L l'ensemble des états légitimes du système.

Un système de réécriture S est auto-stabilisant par rapport à L si et seulement si :

- Tout mot clos, non élément de L, est réductible via S.
- L est clos via S c'est-à-dire $\forall w, w'\ w \in L \wedge w \rightarrow_S w' \Rightarrow w' \in L$.
- Il n'existe pas de cycle clos du type $w_1 \rightarrow w_2 \rightarrow ... \rightarrow w_n = w_1$ avec $w_1 \notin L$.

La première propriété exprime l'absence inter-blocage. La deuxième exprime la clôture aux états légitimes. Enfin la dernière exprime la propriété de convergence.

Reprenons l'exemple de notre algorithme simple, la figure 5.2 montre le graphe de dérivations correspondant (pour des raisons de simplicité, nous avons réduit à trois le nombre de processus). On voit qu'il n'y a pas de blocage : en effet, à partir de tout état non légitime, il existe une règle applicable. L est clos via S dans la mesure où l'état légitime ne peut plus être réécrit. Enfin on a bien convergence car le graphe ne comporte pas de cycle. De plus, le système termine.

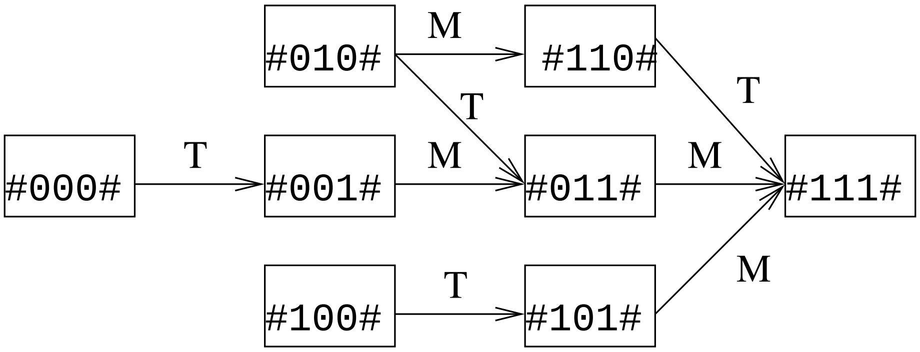

FIG. 5.2 – *exemple de graphe de dérivations*

5.2 Caractérisation du premier ordre des cycles

Le problème de la caractérisation des cycles clos est habituel dans le cadre de la vérification : le nombre d'états augmente exponentiellement avec le nombre de noeuds dans la topologie. De plus, cette caractérisation ne permet pas de faire des vérifications sur des topologies de taille quelconque. Pour nous affranchir de ce deuxième problème, nous introduisons dans cette section la réécriture sur des mots du premier ordre, c'est-à-dire contenant des variables, ce qui nous permet de nous affranchir de la taille du graphe de communications.

5.2.1 Unification

Considérons des mots contenant une seule variable, c'est-à-dire de la forme $\#uWv\#$ avec u et v des éléments de $\Sigma*$ et W un élément de ν. La notion de réduction close peut être trivialement étendue à ce type de mot : il suffit de considérer la lettre W comme une nouvelle constante. Nous dirons dans ce cas que la réduction est faite avec une substitution identitaire : $\sigma = id$.

A partir de ce même mot t du premier ordre, il est également possible d'envisager des instanciations non triviales qui rendent ce mot réductible via une règle. Considérons une règle R : $l \to r$. Ce problème est une cas particulier du problème de l'unification : trouver des instances communes dans t et l.

Définition 60 (unificateur) *Soit a et b deux mots du premier ordre. Un unificateur est un ensemble de substitution S=(s1,s2,..,sn) tels que s1(s2(..sn(a)...))=s1(s2(..sn(b)...))*

Le problème général de l'unification sur des mots est très complexe et a été résolu par Makanin [Mak77]. Toutefois, notre cas particulier est très simple car t et l ne partagent pas de variables et sont linéaires (ils contiennent au plus une occurrence de la même variable). Dans ce cas, il existe un ensemble complet et fini d'unificateurs minimaux : c'est-à-dire qu'il suffit de considérer tous les recouvrements possibles entre t et l en fonction de toutes les instanciations possibles de toutes leurs variables. Nous supprimons de cet ensemble les instanciations qui créent une partie droite de règle à partir d'une variable (i.e. qui créent une partie gauche à position variable). Nous ne considérons donc que les instanciations σ du type : $aW \rightarrow_\sigma abcW$ ou $abW \rightarrow_\sigma abcW$ ou encore $aW \rightarrow_\sigma abW$ et leurs symétriques à droite de W. Nous ne considérons pas les instanciations du type $W \rightarrow_\sigma abcW$. En effet, ces instanciations n'ont pas de raison d'être dans notre contexte car elles créent une partie gauche de règle à une position variable. Cette opération de réduction minimale est une adaptation de l'opération de "narrowing" ou "surréduction" décrite dans [Sla74], [Fay79] ou encore [Hul80].

Exemple Considérons la règle $\#X11Y\# \rightarrow \#X22Y\#$. Le mot $\#1W1\#$ est unifié avec la partie gauche $\#X11Y\#$ via l'ensemble d'unificateurs suivants :
$\mu_1 : \{W/1W'\} \cup \{X/\epsilon, Y/W'1\}$
$\mu_2 : \{W/W'1\} \cup \{X/1W', Y/\epsilon\}$
$\mu_3 : \{W/\epsilon'\} \cup \{X/\epsilon, Y/\epsilon\}$
$\mu_4 : \{W/W_1 11 W_2\} \cup \{X/1W_1, Y/W_2 1\}$

L'unificateur μ_4 est supprimé car il crée une partie gauche à position variable. Par conséquent, les réductions minimales de t correspondant à μ_1,μ_2 et μ_3 sont : $\#11W'1\# \rightarrow \#22W'1\#$, $\#1W'11\# \rightarrow \#1W'22\#$ et $\#11\# \rightarrow \#22\#$.

5.2.2 Réduction minimale

Nous allons définir plus précisément ce qu'est une réduction minimale. Nous distinguons deux cas : soit la substitution est identitaire ($\sigma = id$), soit elle ne l'est pas ($\sigma \neq id$). Considérons une règle de $Middle_S$ R : $\#XlY\# \rightarrow \#XrY\#$ où l est de la forme $a_1...a_n$ (avec $a_i \in \Sigma$). L'ensemble des substitutions non identitaires impliquées dans les réductions minimales de R est de la forme $D_R = A_R \cup B_R \cup C_R$ avec

- A_R est l'ensemble des substitutions α_i : $\{W/W'a_1..a_i\}$ pour i allant de 1 à n-1.
- B_R est l'ensemble des substitutions β_i : $\{W/a_{i+1}..a_n W'\}$ pour i allant de 1 à n-1.
- C_R est l'ensemble des substitutions (closes) γ_i : $\{W/a_{i+1}..a_j\}$ pour i et j allant de 1 à n-1 avec la convention $\{W/\epsilon\}$ si i=j.

Plus simplement, les trois ensembles de substitutions sont tels qu'il est nécessaire qu'au moins une lettre de l soit déjà instanciée pour que la substitution donne une réduction.

Un tel ensemble D_R peut être défini de manière similaire pour une règle de Top_S ou $Bottom_S$.

Définition 61 (Réduction minimale) *Un mot t est réductible de manière minimale en u via une règle R: $l \to p$ en utilisant la substitution $\sigma \in D_R \cup \{id\}$ si et seulement si :*

- *$\sigma = id$: dans ce cas, t est une instance de l et u est l'instance correspondante de p.*
- *R : $\#XlY\# \to \#XrY\#$ où l est de la forme $a_1...a_n$ (avec $a_i \in \Sigma$) et t : $\#t_1Wt_2\#$:*
  - *t_2 commence par la chaîne $a_{i+1}..a_n$ et u est obtenu en remplaçant $Wa_{i+1}..a_n$ par $W'r$.*
  - *t_1 finit par la chaîne $a_1..a_i$ et u est obtenu en remplaçant $a_1..a_iW$ par rW'.*
  - *t2 commence par $a_{j+1}...a_n$ et t1 finit par $a_1...a_i$. u est obtenu en remplaçant $a_1...a_iWa_{j+1}...a_n$ par r.*
- *On définit de même pour $R \in Top_S$ ou $R \in Bottom_S$*

Exemple Considérons la règle $M_1 : \#X10Y\# \to \#X01Y\#$ et le mot $t_1 = \#W02\#$. Il existe une seule substitution : $\{W/W'1\}$ qui conduit à une réduction minimale. Considérons la règle $M_2 : \#X100Y\# \to \#X110Y\#$ et le mot $t_2 = \#W003\#$. Il existe deux instanciations possibles : $\{W/W'10\}$ et $\{W/W'1\}$ qui conduisent au réductions minimales suivantes : $\#W10003\# \to \#W11003\#$ et $\#W1003\# \to \#W1103\#$.

5.2.3 Chaînes de réductions minimales dans Top_S

Plutôt que d'exhiber toutes les chaînes afin d'y trouver des cycles, nous allons exhiber un type particulier de chaînes : les chaînes top. Ces chaînes ont la particularité de commencer par une règle Top. Nous montrerons dans la section suivante qu'il est possible de se restreindre à ce type de chaînes lors de la recherche de cycles.

Définition 62 (Chaîne Top) *Une chaîne de réductions minimales dans Top_S, aussi appelée chaîne Top, est définie inductivement de la façon suivante :*

- *Toute règle de Top_S est une chaîne Top.*
- *Si C : $t_0 \to t_1 \to ... \to t_n$ est une chaîne Top et R une règle de S telle que $t_n \to_{\sigma,R} u$ alors $t_0 \to t_1 \to ... \to t_n \to u$ est une chaîne Top appelée successeur de C via R utilisant σ.*

Définition 63 (chaîne Top quasi-cyclique) *Une chaîne Top $t_0 \rightarrow t_1 \rightarrow ... \rightarrow t_n$ est dite quasi-cyclique si il existe $0 \leq i < n$ tel que $t_i = t_n$ et $t_p \neq t_q$ pour tout p et q inférieurs à n.*

5.3 Caractérisation du premier ordre de l'auto-stabilisation

Dans cette section, nous démontrons un théorème qui permet de démontrer l'auto-stabilisation d'un système de réécriture en observant ses chaînes Top.

Théorème 3 *Soit $S = Middle_S \cup Top_S \cup Bottom_S$ un système de réécriture et L un ensemble de configurations. Si :*

- *tout mot clos non élément de L est réductible via S,*
- *L est clos via S, et*
- *$S - Top_S$ termine.*

alors, S est auto-stabilisant par rapport à L si et seulement si il n'existe pas de chaîne Top quasi-cyclique $t_0 \rightarrow t_1 \rightarrow ... \rightarrow t_n$ via S telle que $t'_n \notin L$ pour toute instance close t'_n de t_n.

Ce résultat est une adaptation du théorème de Dershowitz [Der81] qui fait intervenir la notion de pas actifs ou inactifs. Nous allons maintenant démontrer des lemmes techniques afin de démontrer ce théorème.

Définition 64 (Partie active) *La partie active d'un mot clos w_i dans une dérivation $w_1 \rightarrow w_2 \rightarrow ... \rightarrow w_n$ est la partie qui est créée par la partie close des parties droites des règles qui ont été appliquées. Seule la lettre la plus à droite du mot est considérée active pour w_1.*

Plus précisément, considérons une règle du type $\#XlY\# \rightarrow \#XrY\#$ qui est appliquée à un mot clos w de la forme $\#u_1l_1u_2\#$ pour obtenir un mot clos w' de la forme $\#u_1r_1u_2\#$. Dans w', toutes les lettres de r_1 sont actives si au moins une lettre de l_1 est active. De plus, toutes les lettres de u_1 et u_2 qui étaient déjà actives dans w, le restent dans w'. Nous dirons qu'un mot est actif si sa lettre la plus à droite est active. Cela implique que toute application de règle de Top_S est active.

Définition 65 (Dérivation close active) *Une dérivation close $w_1 \rightarrow w_2 \rightarrow ... \rightarrow w_n$ est dite active si toutes les réductions sont faites sur des parties actives des mots. Dans le cas contraire, la dérivation sera dite inactive. On notera $\rightarrow_{act}$ (respectivement $\rightarrow_{inact}$) l'application d'une règle sur une partie active (respectivement inactive) d'un mot clos.*

Prenons un exemple : soit un mot clos # 2 0 1 **2** #. La lettre la plus à droite (celle en gras) est active. Nous allons lui appliquer successivement deux règles :

- T4 : # X 1 2 # → # X 2 1 #.
- B1 : # 2 X 1 # → # 1 X 2 #.

Nous obtenons la dérivation close active suivante :
2 0 1 **2** # $\rightarrow_{act}$ # 2 0 **2 1** # $\rightarrow_{act}$ # **1** 0 **2 2**#

Nous allons tout d'abord prouver un lemme qui permet de généraliser toute dérivation close à une chaîne du premier ordre.

Lemme 19 *Pour toute dérivation close active Δ : $w_1 \rightarrow_{act} w_2 \rightarrow_{act} \ldots \rightarrow_{act} w_n$, via S, il existe une top chaîne C : $t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_n$ via S telle que Δ soit une instance de C.*

La preuve de ce lemme peut être trouvée dans un cadre plus général dans la partie converse de la relation entre les séquences de réductions et les séquences de narrowing dans la théorie des termes du premier ordre. Celle-ci peut être trouvée dans [Hul80].

Reprenons l'exemple précédent : Δ : # 2 0 1 **2** # $\rightarrow_{act}$ # 2 0 **2 1** # $\rightarrow_{act}$ # **1** 0 **2 2**# est un instance via la substitution $\{W/0\}$ de la top chaîne : C : # 2 W 1 2 # $\rightarrow_{act}$ # 2 W 2 1# $\rightarrow_{act}$ # 1 W 2 2 #.

Concrètement, il suffit pour obtenir cette chaîne Top, de remplacer les éléments inactifs de la chaîne par des variables.

Nous allons maintenant introduire le lemme de semi-commutation qui est un cas particulier de celui démontré dans [Der81].

Lemme 20 (Lemme de semi-commutation) *Soit w_1,w_2 et w_3 trois mots clos. si $w_1 \rightarrow_{act} w_2 \rightarrow_{inact} w_3$ alors il existe un mot clos w_2' tel que $w_1 \rightarrow_{inact} w_2' \rightarrow_{act} w_3$*

Cela signifie qu'il est toujours possible de permuter les transitions inactives et les transitions actives. Cela implique que si les transitions inactives sont en nombre fini, elles peuvent être groupées au début de la dérivation. Nous allons donc prouver que le nombre de transitions inactives sont en nombre fini dans les systèmes qui nous intéressent.

Proposition 1 *Soit S une système de réécriture tel que $S - Top_S$ termine. Toute dérivation infinie via S comporte un nombre fini de transitions inactives.*

Preuve Considérons une dérivation close infinie $\Delta : w_1 \to w_2 \to ... \to w_n \to ...$ via S. L'application d'une règle sur une partie active d'un mot ne peut pas produire de lettre inactive par construction. De plus, l'application d'une règle sur une partie inactive d'un mot ne peut que remplacer une partie inactive par une autre partie inactive de même taille. On a donc deux comportements possibles : (a) soit toutes les parties inactives de Δ disparaissent après un nombre fini d'applications de règles actives, (b) soit au moins une partie inactive le reste mais entre deux positions fixées dans les mots. Considérons le cas (b), il existe une dérivation Δ' qui est une sous-partie de Δ qui est de la forme $w_i \to ... \to w_n \to ...$ et telle que chaque w_i soit de la forme $x_i v_i y_i$ tous les x_i respectivement (y_i, v_i) ont la même taille et tous les v_i sont toujours inactifs. Par conséquent, les applications de règles actives ne concernent jamais les v_i. On peut donc extraire de Δ' une dérivation close infinie inactive Δ'' n'affectant que les v_i. Or cette dérivation infinie ne contient pas de transitions de Top_S. Ceci est en contradiction avec le fait que $S - Top_S$ termine et par conséquent, Δ'' est finie. Par conséquent, le seul cas envisageable est le cas (a) c'est-à-dire que toutes les parties inactives de Δ disparaissent après un nombre fini d'applications de règles actives. □

Par conséquent, il est possible de grouper toutes les actions inactives au début des dérivations et donc d'obtenir des dérivations infinies qui ont un suffixe purement actif. Ce suffixe est l'instance d'une chaîne Top. Plus formellement :

Proposition 2 *Si $S - Top_S$ termine et L est clos via S, alors il existe une dérivation close infinie via S ne contenant pas de mot de L si et seulement si il existe une chaîne Top quasi-cyclique via S commençant par un mot t tel qu'il n'existe pas de substitution θ telle que $t_\theta \notin L$.*

Preuve La partie converse est triviale, en effet, une chaîne quasi-cyclique dont les mots ne sont pas dans L implique directement une dérivation close infinie dont les mots ne sont pas dans L. Pour la partie directe, considérons une dérivation close infinie Δ dont les mots ne sont pas dans L. Par la proposition 1 et comme $S - Top_S$ termine, les pas inactifs de cette dérivation sont en nombre fini. En appliquant de manière itérative le lemme de semi-commutation, il est possible d'obtenir une dérivation Δ' telle que tous les pas inactifs soient placés au début de celle-ci. Il existe donc un point particulier de Δ' à partir duquel tous les pas sont actifs. Appelons Δ'' ce suffixe actif. On sait par la propriété de Δ qu'aucun des mots de Δ'' n'est dans L. Or, on sait que la taille des mots est finie et que les règles préservent la longueur. Par conséquent, Δ'' est forcément quasi-cyclique, c'est-à-dire de la forme $w_i \to w_{i+1} \to ... \to w_n = w_i$. Par le lemme 19, il existe une chaîne C : $t_i \to t_{i+1} \to ... \to t_n = t_i$ telle que pour toute substitution close θ, $C\theta = \Delta$. C'est-à-dire que C est une chaîne quasi-cyclique telle qu'il n'existe pas de substitution θ telle que $t_n\theta = w_n \notin L$. □

La démonstration du théorème 3 résulte directement de la proposition 2 et de la définition de l'auto-stabilisation.

Remarque Afin d'utiliser le théorème 3 d'une manière automatique, il est nécessaire de démontrer que $S - Top_S$ termine. Pour cela, on pourra démontrer cette partie en montrant que $S - Top_S - Bottom_S$ termine ce qui est généralement facile et en appliquant le théorème sur $S - Top_S$ avec les chaînes Bottom.

Le théorème nous suggère la procédure de preuve :

- Générer toutes les chaînes Top quasi-cycliques.
- si les toutes les chaînes Top quasi cycliques : $w_1 \to w_i \to ... \to w_n = w_i$ sont telles que toutes les instances de w_n sont dans L, alors le système est auto-stabilisant, sinon il ne l'est pas.

5.4 La φ-optimisation

La méthode précédente implique la manipulation de nombreuses règles dès que l'algorithme est un peu compliqué. Ce grand nombre de règles implique la construction de beaucoup de graphes dotés de nombreuses arêtes. Dans cette section, nous introduisons une optimisation qui permet de réduire le nombre des règles à manipuler grâce à une mesure.

Considérons une mesure sur les mots qui ne croît jamais lors de l'application d'une règle. formellement, $w \to_S w' \Rightarrow \varphi(w) \geq \varphi(w')$ On peut alors classer les règles dans deux catégories :

- Celles qui préservent la mesure : $w \to_S w' \Rightarrow \varphi(w) = \varphi(w')$
- Celles qui ne la préservent pas : $w \to_S w' \Rightarrow \varphi(w) > \varphi(w')$

Cette mesure n'a de sens que si ses valeurs sont bornées inférieurement. Il faut noter que dans les preuves classiques de l'auto-stabilisation, on cherche une mesure telle que toutes les règles soient dans la deuxième catégorie. Par conséquent, les règles que nous cherchons sont, en général, plus faciles à trouver.

Supposons qu'il existe une dérivation close infinie, il est clair qu'elle ne peut contenir un nombre infini de règles de la deuxième catégorie (car la mesure possède une borne inférieure). Par conséquent, l'application de ces règles peut être faite au début puis se limiter à un suffixe ne contenant que des applications de règles qui préservent la mesure.

Nous en déduisons une nouvelle définition de l'auto-stabilisation : soit S le système à prouver. Soit S' le système constitué des règles qui préservent la mesure. S est auto-stabilisant si et seulement si S' est auto-stabilisant. Ceci nous donne donc une réduction possible du nombre de règles qui implique une simplification de la preuve.

Nous illustrons cette méthode sur l'exemple de l'algorithme à quatre états de Gosh (tous les détails sur la preuve de cet algorithme sont donnés dans la section "exemples"

de ce chapitre). Cet algorithme contient deux règles Middle :
$M_1 : \quad \#X(q+1)qY\# \rightarrow \#X(q+1)(q+1)Y\#$
$M_2 : \quad \#Xq(q+1)Y\# \rightarrow \#X(q+1)(q+1)Y\#$
avec $q \in \{0,1,2,3\}$ et '$+$' est l'addition modulo 4.

La preuve originale de convergence propose une mesure sur les états sous la forme (Br,Ds) telle que les fonctions Br et Ds sont des fonctions décroissantes non strictement. La preuve est basée sur le fait qu'à chaque pas, Br ou Ds décroît strictement (c'est la somme des deux qui sert de mesure). La fonction Ds est une fonction très subtile et difficile à trouver, mais la fonction Br est simplement le nombre de "breaks", c'est-à-dire le nombre de couples de processus voisins dont la valeur diffère au plus de un. Notre méthode permet de n'utiliser que la mesure Br en simplifiant les règles. Ainsi, chaque règle de notre système S' devient la composition de deux règles de S :
$M_1 : \quad \#X(q+1)qqY\# \rightarrow \#X(q+1)(q+1)qY\#$
$M_2 : \quad \#Xqq(q+1)Y\# \rightarrow \#Xq(q+1)(q+1)Y\#$

Cette réduction permet de réduire le nombre de contextes sur lesquels s'appliquent les règles.

Remarque Notons que l'on peut utiliser une mesure non décroissante (au lieu d'une mesure non croissante) à condition que la valeur de celle-ci soit bornée supérieurement.

5.5 Généralisation par schémas

Le problème de la méthode de génération de chaînes Top présentée dans le section précédente est que leur nombre est souvent infini même en utilisant la φ-optimisation. De plus, il est souvent possible de trouver des motifs qui se répètent dans les dérivations. L'idée qui est développée dans cette section est de regrouper les états obtenus dans les dérivations via des expressions régulières. Cette approche augmente le nombre d'états considéré par rapport à la dérivation du premier ordre et par conséquent, elle ne fournit qu'une condition suffisante de l'auto-stabilisation.

Définition 66 (Schéma) *Un schéma du premier ordre est un langage de la forme $\#LWM\#$ où L et M sont des langages réguliers sur Σ^* et W est une variable du premier ordre. Un schéma clos est un langage de la forme $\#L\#$ où L est un langage régulier sur Σ^*.*

Notons qu'un mot du premier ordre du type $\#uWv\#$ peut être vu comme une schéma du premier ordre où $L = u$ et $M = v$. De même, un mot clos peut être vu comme un schéma clos.

Définition 67 (Réduction généralisée) *Soit S un schéma, R une règle. S est ré-*

duit en S' via R si et seulement si $\forall s' \in S', \exists s \in S$ *tel que* $s \to_R s'$. *On note cette réduction* $S \to_R S'$. *Ce type de réduction appliquée aux schémas est appelée réduction généralisée.*

Le concept de chaîne Top utilisé dans la preuve est étendu au chaînes Top généralisées.

Définition 68 (Généralisation) *Soit A un graphe de dérivations. On dira que A' est la généralisation de A si et seulement si*

- *quel que soit n un noeud de A, il existe un noeud n' de A' tel que n soit inclus dans n'.*
- *soit* $n_1 \to n_2$ *une transition du graphe A (i.e. une arête), il existe* n'_1 *dans A' tel que* $n_1 \subseteq n'_1$ *et il existe* n'_2 *dans A' tel que* $n_2 \subseteq n'_2$ *et il existe une transition dans A' de* n'_1 *vers* n'_2.

La figure 5.3 montre une généralisation. Le graphe de droite A' est une généralisation du graphe A. Tous les états de A sont inclus dans les états de A'. Les transitions sont respectées.

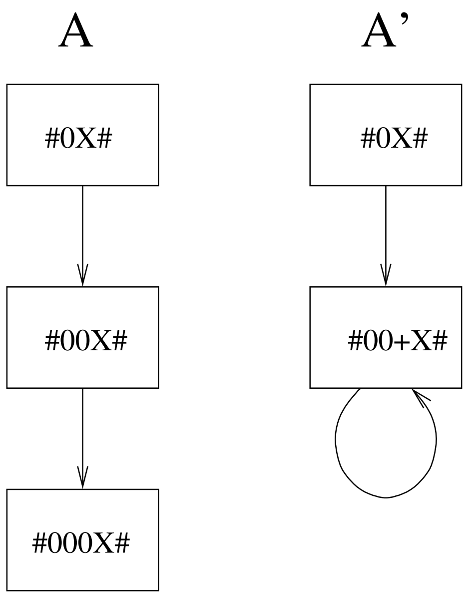

FIG. 5.3 – *Généralisation d'un graphe de dérivations*

Comme on le voit sur la figure, la généralisation induit une surgénération de termes par rapport à la dérivation non généralisée : les termes éléments de $\#0000 + X\#$ ont été rajoutés. Par conséquent, le graphe de dérivations généralisé ne permet d'obtenir qu'une condition suffisante de l'auto-stabilisation.

Théorème 4 *Soit un système de réécriture S. Soit L l'ensemble d'états légitimes associé à S. Si :*

- *tout mot clos non élément de L est réductible via S,*
- *L est clos via S, et*
- *$S - Top_S$ termine.*

et qu'il n'existe pas de chaîne Top généralisée quasi-cyclique $t_0 \rightarrow t_1 \rightarrow \ldots \rightarrow t_n$ via S telle que $t'_n \notin L$ pour toute instance close t'_n de t_n, alors l'algorithme est auto-stabilisant par rapport à L.

En effet, toute chaîne Top a son équivalent dans toute généralisation. Par conséquent, si une chaîne Top est quasi-cyclique, le graphe généralisé contiendra au moins un cycle. D'une manière pratique, cela revient à dire qu'il n'existe pas de cycle infini dans les graphes de dérivations de S issus des chaînes Top de S.

5.5.1 Génération des graphes de dérivations généralisés

A partir du théorème précédent, nous déduisons une méthode de preuve pratique par génération de graphes de dérivations généralisés. Cette méthode permet de déterminer l'absence de chaîne Top généralisée quasi-cyclique dans les graphes de dérivations.

Pour toute règle Top $t_0 \rightarrow t_1$ de S, on génère le graphe de dérivations A de la façon suivante :

$A = \emptyset$
ajouter à A les noeuds t_0 et t_1 et l'arête $t_0 \rightarrow t_1$
$P = \emptyset$
Pour tout $s \in S$, si t_1 est réductible via s alors
    ajoute(P,(t_1,s))
fin_pourtout
Tant que $P \neq \emptyset$ faire
    choisir et supprime (e,s) de P.
    calculer les successeurs e_1, e_2, ..., e_n de e via s.
    pour i allant de 1 à n
        Pour tout $s \in S$, si e_i est réductible via s alors
            si $\exists n$, un noeud de A tel que $e_i \subset n$
                alors ajouter l'arête $e \rightarrow n$ à A si elle n'existe pas.
            sinon
                ajouter à P (e_i, s).
                ajouter à A le noeud e_i et l'arête $e \rightarrow e_i$.
                si il existe un noeud n tel que $n \subset e_i$
                fin_si
                    pour toutes les arêtes du type $n' \rightarrow n$
                        supprimer l'arête et ajouter $n' \rightarrow e$ à A.
                    fin_pourtoutes.
                    pour toutes les arêtes du type $n \rightarrow n'$
                        supprimer l'arête et ajouter $e \rightarrow n'$ à A.
                    fin_pourtoutes.
            fin_si.
        fin_pourtout.
    fin_pour.
    Si c'est nécessaire : généraliser A.
fin_tantque.

A est le graphe de dérivations généralisé en cours de construction. P est l'ensemble des actions restant à traiter.

Pour la généralisation, l'algorithme est le suivant :

Tant que la généralisation n'est pas finie
    proposer une expression régulière e.
    s'il existe un noeud n de A tel que $e \subset n$ alors continuer avec une nouvelle expression.
    ajouter à A le noeud e.
    Pour tout noeud n de A tel que $n \subset e$
        pour toutes les arêtes du type $n' \to n$
            supprimer l'arête et ajouter $n' \to e$ à A.
        fin_pourtoutes.
        pour toutes les arêtes du type $n \to n'$
            supprimer l'arête et ajouter $e \to n'$ à A.
        fin_pourtoutes.
        supprimer les entrées de P du type (n,s) pour tout s.
    fin_pourtout.
fin_tantque.

Une fois ces graphes construits : les chaînes Top généralisées quasi-cycliques sont des cycles dans les graphes que nous avons construits. Si le graphe ne contient pas de cycle infini, hormis ceux dans L, l'algorithme est auto-stabilisant modulo les trois points préliminaires. Dans le cas contraire, il n'est pas possible de décider si l'algorithme n'est pas auto-stabilisant ou si la surgénération associée à la généralisation a engendré des cycles.

5.5.2 Les cycles finis

Lors de la construction des graphes de dérivations, des cycles peuvent apparaître. Toutefois, un certain nombre d'entre eux ne peuvent pas être des cycles infinis. Pour le démontrer, nous utilisons des méta-arguments.

Le méta-argument le plus fréquemment utilisé concerne la substitution associée à chacune des arêtes. En effet, comme nous l'avons vu, on peut distinguer deux sortes de réductions : celles qui ont une substitution identitaire et celles qui ont une substitution non identitaire. Nous avons vu dans la partie modélisation que les règles préservent la longueur du mot d'origine. Cela s'applique également aux règles appliquées à un mot du premier ordre. Cela signifie qu'on ne peut pas instancier infiniment souvent une variable. Lorsqu'un cycle apparaît, il est donc intéressant de regarder sa substitution globale. Pour qu'un cycle puisse être infini, il est nécessaire que sa substitution globale soit identitaire. C'est-à-dire que toutes les réductions qui le composent soient associées à une substitution identitaire. Si ce n'est pas le cas, un cycle infini impliquerait qu'au moins une variable est instanciée infiniment souvent, ce qui contredit la propriété de conservation de la longueur des règles.

5.5.3 Exhibition de contre-exemples

Une propriété intéressante de cette méthode est qu'elle permet d'exhiber des contre-exemples lorsque la preuve échoue. En effet, s'il reste des cycles après la suppression des cycles finis, ceux-ci traduisent la plupart du temps une erreur dans l'algorithme ou dans la traduction des règles. L'observation de ces cycles nous fournit une exécution problématique. Il est intéressant de noter que si une telle exécution existe dans l'ensemble des exécutions associées a l'algorithme, elle apparaîtra forcément dans le graphe de dérivations du fait de l'aspect exhaustif de la construction. En revanche, il est possible que les cycles infinis qui apparaissent soient des artefacts dus à la généralisation. Dans ce cas, il est facile de s'en rendre compte : une généralisation abusive conduit à des transitions qui ne peuvent pas avoir lieu dans l'algorithme. Par conséquent, l'échec de la preuve n'est pas si négatif que l'on pourrait le croire au vu du théorème. Soit il fournit des contre-exemples qui permettent de corriger l'algorithme, soit il fournit des indications pour trouver une généralisation plus appropriée.

5.6 Exemples

Dans cette section, nous présentons trois exemples d'algorithme tirés de la littérature. Ils ont été choisis afin de présenter les différentes techniques présentées dans ce chapitre. Le premier exemple illustre la réécriture sur plusieurs lettres en même temps (sans pour autant violer l'hypothèse de démon central), le deuxième exemple est tout à fait standard et montre un cas réel. Enfin, le dernier exemple illustre un cas compliqué qui est résolu en utilisant la φ-optimisation.

5.6.1 L'algorithme Beauquier-Debas

L'algorithme de Beauquier-Debas est décrit dans sa forme originale dans [BD95]. C'est une adaptation du troisième algorithme de Dijkstra que l'on peut trouver dans [Dij74]. Dans notre formalisme, il correspond au système de réécriture S suivant :

Bottom	B_1	$\#2X1\#$	$\rightarrow$	$\#1X2\#$	
Top	T_1	$\#X00\#$	$\rightarrow$	$\#X21\#$	
	T_2	$\#X10\#$	$\rightarrow$	$\#X01\#$	
	T_3	$\#X20\#$	$\rightarrow$	$\#X11\#$	
	T_4	$\#X12\#$	$\rightarrow$	$\#X21\#$	
	T_5	$\#X22\#$	$\rightarrow$	$\#X01\#$	
Middle	M_1	$\#X10Y\#$	$\rightarrow$	$\#X01Y\#$	(avec $Y \neq \varepsilon$)
	M_2	$\#X11Y\#$	$\rightarrow$	$\#X02Y\#$	(avec $Y \neq \varepsilon$)
	M_3	$\#X12Y\#$	$\rightarrow$	$\#X00Y\#$	(avec $Y \neq \varepsilon$)
	M_4	$\#X02Y\#$	$\rightarrow$	$\#X20Y\#$	(avec $Y \neq \varepsilon$)
	M_5	$\#X22Y\#$	$\rightarrow$	$\#X10Y\#$	(avec $Y \neq \varepsilon$)

L est défini comme : #0*20*1# ∪ #0*10*2# .

Dans l'article original, chaque processus maintient une seule variable ce qui conduirait à des règles ne modifiant qu'un seule lettre à la fois. Toutefois, les auteurs basent leur preuve non pas directement sur les valeurs de ces variables mais sur leur différence (qu'ils appellent gap). Ils définissent un gap entre deux processus voisins comme étant la valeur absolue de la différence entre les deux valeurs des variables des processus considérés. Nous reprenons cette notation qui simplifie les preuves en nous fournissant une mesure pour la φ-optimisation. Comme le démon est considéré central, une seule variable peut être modifiée pour toute action de processus, toutefois, cette modification entraîne la modification des gaps entre ce processus et ses deux voisins. C'est pourquoi chaque règle modifie deux lettres en même temps.

Cette formalisation en gap implique une propriété sur les mots que nous considérons : la somme des éléments de toute configuration modulo trois est toujours nulle. Cette propriété est conservée par toute application des règles de S. Il est facile de montrer que tout mot clos est réductible via S et que L est clos via S. Ces deux propriétés sont démontrées dans l'article original. Par conséquent, l'algorithme est auto-stabilisant si et seulement si il n'existe pas de dérivation close cyclique via S contenant un élément $w \notin L$.

Considérons la mesure φ comme le nombre d'éléments non nuls contenus dans un mot. Cette mesure n'augmente pas pour toutes les règles excepté les trois règles T_1 et T_3. Toutefois, les auteurs de [BD95] ont remarqué que les trois règles T_1, T_2 et T_3 ne sont applicables qu'une seule fois. Par conséquent, le système est auto-stabilisant si et seulement si il n'existe pas de dérivations closes cycliques pour le système S, formé des quatre règles B_1,M_1,M_4 et T_4 qui préservent la mesure.

La figure 5.4 montre le graphe de dérivations complet du système S' basé sur la règle T_4. On constate que les seuls cycles sont de substitution non nulle, c'est-à-dire qu'ils ne peuvent pas être infinis. Par conséquent, nous avons montré que l'algorithme de Beauquier-Debas est auto-stabilisant.

5.6.2 L'algorithme à quatre états de Ghosh

L'algorithme à quatre états de Ghosh [Gho93] est une variante de l'algorithme à quatre états de Dijkstra [Dij74]. Le système est un anneau de N machines où chaque machine peut avoir quatre états : $\{0, 1, 2, 3\}$ sauf la machine la plus à gauche (respectivement la plus à droite) qui peut prendre deux états $\{1, 3\}$ (respectivement $\{0, 2\}$). La configuration considérée est le mot constitué de la valeur de chaque processus, délimité par le caractère #.

Le système S a la forme suivante :

$Middle$

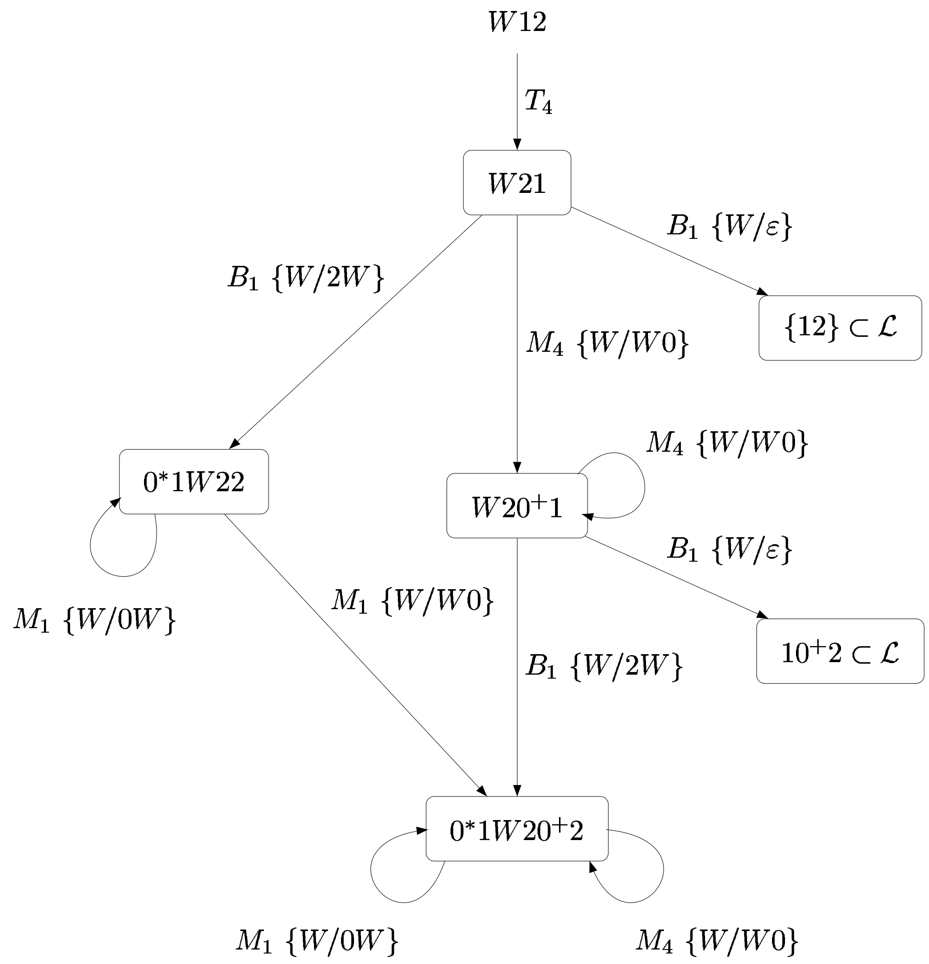

FIG. 5.4 – *Graphe de dérivations pour la règle T_4*

$$\begin{array}{ll} M_1: & \#X(q+1)qY\# \rightarrow \#X(q+1)(q+1)Y\# \\ M_2: & \#Xq(q+1)Y\# \rightarrow \#X(q+1)(q+1)Y\# \end{array}$$

où $q \in \{0,1,2,3\}$ et '$+$' est l'addition modulo 4.

Top

$$\begin{array}{ll} T_1: & \#X32\# \rightarrow \#X30\# \\ T_2: & \#X10\# \rightarrow \#X12\# \end{array}$$

Bottom

$$\begin{array}{ll} B_1: & \#12X\# \rightarrow \#32X\# \\ B_2: & \#30X\# \rightarrow \#10X\# \end{array}$$

$\mathcal{L}$ est défini comme : $\#\{1^+,3^+\}\{0^+,2^+\}\#$.

Comme il est expliqué dans la section 5.4, Ghosh prouve la convergence au moyen d'une mesure (B_r, D_s) tel que $B_r + D_s$ décroît strictement à chaque pas de calcul.

En appliquant la φ-optimisation comme elle est expliquée dans la section 5.4, nous pouvons remplacer les deux règles Middle par :

$$
\begin{array}{ll}
M_1 : & \#X(q+1)qqY\# \rightarrow \#X(q+1)(q+1)qY\# \\
M_2 : & \#Xqq(q+1)Y\# \rightarrow \#Xq(q+1)(q+1)Y\#
\end{array}
$$

La figure 5.5 montre le graphe de dérivations associé à la règle T_1. Celui obtenu avec la règle T_2 est très similaire. Sur cette figure, on obtient plusieurs cycles. Mais on y applique le méta-argument de la conservation de la longueur et comme aucun d'entre eux n'a une substitution globale identitaire, l'algorithme est auto-stabilisant.

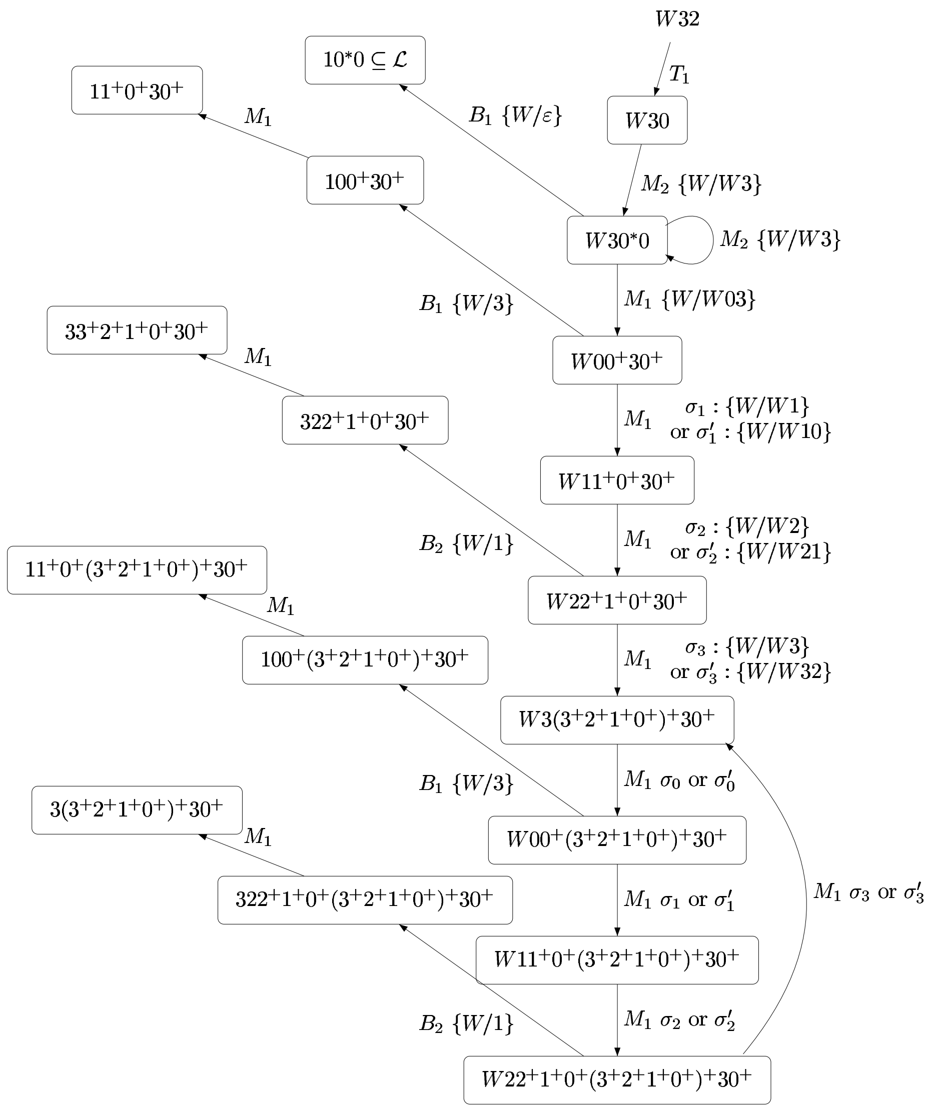

FIG. 5.5 – *Graphe de dérivations pour la règle T1*

5.6.3 L'algorithme d'orientation d'anneau de Hoepman

L'algorithme d'orientation d'anneau décrit par Hoepman dans [Hoe94] est un algorithme uniforme, c'est-à-dire qu'aucun des processus n'est distingué. Dans ce contexte, la distinction des règles Top n'a plus lieu d'être. En revanche, nous utilisons une notion d'équité (définie dans l'article original) qui garantit que toute dérivation infinie change infiniment souvent l'état de tous les processus. Par conséquent, nous considérons toutes les règles comme des règles Top et nous distinguons un bord d'une manière arbitraire.

L'algorithme d'Hoepman est basé sur 16 règles appliquées sur les mots de l'alphabet : $\Sigma = \{1_-, 1_+, 0_-, 0_+\}$

On utilisera les notations suivantes :
$L_0 = \Sigma^* = \{0, 1\}^*$ avec $0 = \{0_+, 0_-\}$, $1 = \{1_+, 1_-\}$,
$L_1 = (O_1\ I_1)^*$ avec $O_1 = 0 \cup \{0_+0_-0_+,\ \ 0_-0_+,\ \ 0_+0_-\}$
et $I_1 = 1 \cup \{1_+1_-1_+,\ \ 1_-1_+,\ \ 1_+1_-\}$,
$L_2 = (O_2\ I_2)^*$ avec $O_2 = 0 \cup \{0_+0_-\}$ et $I_2 = 1 \cup \{1_+1_-\}$.

règle	p	q	r	q'
a	0	0	0	1_-
b	0	1	0	1_-
c	1	1	1	0_-
d	1	0	1	0_-

règle	p	q	r	q'
e	0_+	0_-	1_-	1_+
e'	1_-	0_-	0_+	1_+
f	1_+	1_-	0_-	0_+
f'	0_-	1_-	1_+	0_+

règle	p	q	r	q'
g	0_-	0_-	1	0_+
g'	1	0_-	0_-	0_+
h	0_+	0_+	1	0_-
h'	1	0_+	0_+	0_-

règle	p	q	r	q'
i	1_-	1_-	0	1_+
i'	0	1_-	1_-	1_+
j	1_+	1_+	0	1_-
j'	0	1_+	1_+	1_-

Les règles peuvent être appliquées sur n'importe quel processus, c'est-à-dire à toute position dans un mot d'état. Formellement, toute règle transformant pqr en $pq'r$ correspond à trois règles : $\#XpqrY\# \to \#Xpq'rY\#$, $\#rXpq\# \to \#rXpq'\#$ et $\#qrXp\# \to \#q'rXp\#$. Hoepman prouve que l'algorithme converge vers L_2 par un attracteur à trois ensembles : tout d'abord, il exhibe une mesure qui décroît strictement lors de l'application des règles sur des mots de L_0. Cette décroissance mène à un état de L_1. Ensuite, il exhibe une mesure qui décroît strictement lors de l'application des règles sur des mots de L_1. Cette décroissance mène à un état de L_2. Il montre ensuite que le système converge de L_2 vers L_3 mais nous ne le montrerons pas ici. Notre méthode montre directement la convergence vers L_2 sans passer par des ensembles intermédiaires. Nous considérons L_2 comme les états légitimes.

Dans un premier temps, nous utilisons la φ-optimisation pour réduire le nombre de règles. Celle-ci nous permet de supprimer les règles (a) et (c) en comptant le nombre de sous-séquences maximales de même valeur dans les mots en considérant que la partie gauche et la partie droite sont contiguës. Par exemple, la mesure sur le mot $\#000100W1100\#$ est 4. Les règles (a) et (c) font croître strictement cette mesure et peuvent donc être supprimées. Le système obtenu est de la forme :

règle	p	q	r	p'	q'	r'
E	0_+	0_-	1	0	1_+	1_-
E'	1	0_-	0_+	1_-	1_+	0
F	1_+	1_-	0	1	0_+	0_-
F'	0	1_-	1_+	0_-	0_+	1

Nous avons profité de la transformation pour réexprimer les règles dans leur notation détaillées et réduire les similitudes. La figure 5.6 montre le graphe de dérivations associé à la règle E. Cette figure est assez compliquée mais elle peut être réduite par une généralisation qui est représentée par la figure 5.7.

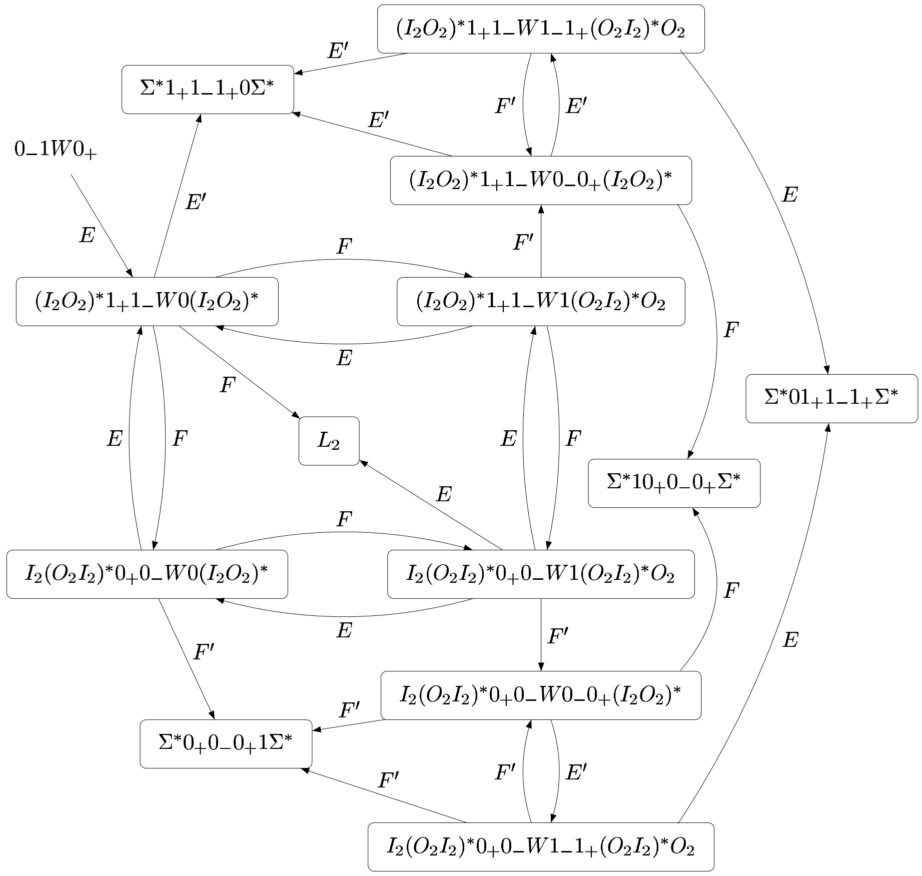

FIG. 5.6 – *Graphe de dérivations détaillé*

Les cycles que nous observons sont de trois types :

- Les mots considérés contiennent $0_+0_-0_+$ ou $1_+1_-1_+$. Or il n'existe pas de règle qui permette de réécrire le caractère central. Par conséquent, ces cycles ne peuvent être infinis car ils sont en contradiction avec l'hypothèse d'équité qui implique que toute lettre est réécrite infiniment souvent dans toute dérivation infinie.
- Les cycles qui ont une instanciation globalement non identitaire et qui ne peuvent être infinis pour les raisons exposées précédemment.
- Les cycles dans L_2 qui sont une condition de l'auto-stabilisation.

On peut construire des graphes similaires sur lesquels la même analyse s'applique pour les règles E', F et F'et par conséquent, l'algorithme est auto-stabilisant.

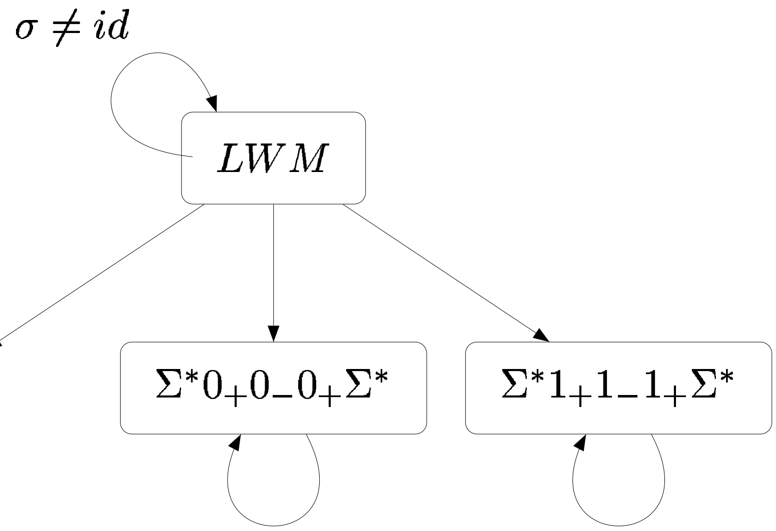

FIG. 5.7 – *Une généralisation du graphe de dérivations*

5.7 Résumé

Dans ce chapitre, nous proposons une nouvelle méthode de preuves de l'auto-stabilisation qui n'utilise pas de mesure strictement décroissante et qui s'applique aux topologies linéaires. Tout d'abord, nous avons introduit une formalisation des systèmes répartis à topologie linéaire et des algorithmes qui s'y appliquent sous forme de réductions. Puis, nous avons montré que le problème de l'auto-stabilisation se ramène à un problème de terminaison d'un système de réécriture. Enfin, nous proposons une méthode qui permet de généraliser ce résultat au moyen d'expressions régulières et qui fournit une méthodologie pratique pour faire des démonstrations d'auto-stabilisation. Cette méthode est illustrée sur trois exemples d'algorithmes tirés de la littérature.

Chapitre 6

Poulet, un outil pour les preuves par réécritures

6.1 Introduction

Poulet est un outil qui permet d'automatiser la méthode de preuve par réécriture exposée dans le chapitre précédent. Nous avons vu que la méthode est basée sur deux algorithmes principaux : la construction du graphe de dérivations qui se fait de manière complètement automatique, et la généralisation de ce même graphe qui doit être faite d'une manière instinctive par l'utilisateur.

Poulet conserve la même approche : il permet, au moyen d'une interface graphique et instinctive de s'affranchir de tous les calculs fastidieux et permet à l'utilisateur de chercher les généralisations. Les algorithmes que l'on souhaite prouver doivent être spécifiés sous la forme de systèmes de réécritures dans une syntaxe facile à utiliser. Le procédé de preuve en lui-même est très simple et ne requiert que l'habitude de lire des expressions régulières. En effet, le programme affiche les graphes d'un façon conviviale qui permet de trouver facilement les régularités.

Une fois que les graphes complets sont calculés, Poulet calcule les cycles dans ces graphes, applique certains meta-arguments que nous avons vus au chapitre précédent et décide si le protocole est auto-stabilisant. Si la preuve n'aboutit pas, les graphes sont disponibles pour une analyse plus étendue qui permet de trouver les contre-exemples ou de comprendre une généralisation exagérée.

Cet outil est disponible gratuitement à l'adresse suivante :

`http://www.lri.fr/~magniett/poulet/index.html`

Il est très facile à compiler pour une machine Linux. Il est également possible de le compiler sur une machine Sun sur laquelle sont installées les librairies GTK.

6.2 Présentation du logiciel

Poulet est un programme écrit en C avec une interface graphique GTK. Il permet de construire graphiquement les preuves d'auto-stabilisation par réécriture. Il est d'un usage simple et graphique et permet de gagner beaucoup de temps sur la rédaction des preuves ainsi que d'assurer leur validité.

La figure 6.1 montre l'interface de Poulet au cours d'une preuve : la fenêtre graphique en haut à gauche montre le graphe de dérivations en cours de construction. La fenêtre au milieu à gauche est la console, elle contient les messages générés par le programme. Juste en dessous, une ligne de saisie permet de taper des commandes. En effet, Poulet accepte un langage de script spécialisé pour recevoir des indications de constructions.

En haut à droite se situe la zone de commande : des onglets permettent d'accéder à une matrice de boutons qui appellent les fonctionnalités courantes.

Enfin, en bas à gauche se trouve la fenêtre d'informations : des onglets permettent d'accéder aux informations relatives à la preuves en cours : les règles de l'algorithme, la pile d'actions possibles, la liste des objets du système et la liste des cycles détectés dans le graphe construit (à la fin de la preuve).

Les fonctionnalités de Poulet sont :

- Construire le graphe de dérivations du premier ordre d'un algorithme.
- Intégrer des expressions régulières (schémas) dans le graphe.
- Sauvegarder et imprimer les résultats.
- Calculer les cycles dans les graphes résultats.
- Recevoir des schémas d'un oracle connecté via le réseau Internet.

6.3 Structures

Poulet manipule de nombreux types de données : chacun de ces types est représenté par une structure munie de fonctions.

6.3.1 Graphes

Les graphes sont les éléments de base du programme, en effet, le but principal de Poulet est de construire un graphe de dérivations. Un graphe est une liste chaînée de noeuds qui possèdent une liste chaînée d'arêtes. Ils sont décrits dans les fichiers “graphe.h” et “graphe.c”.

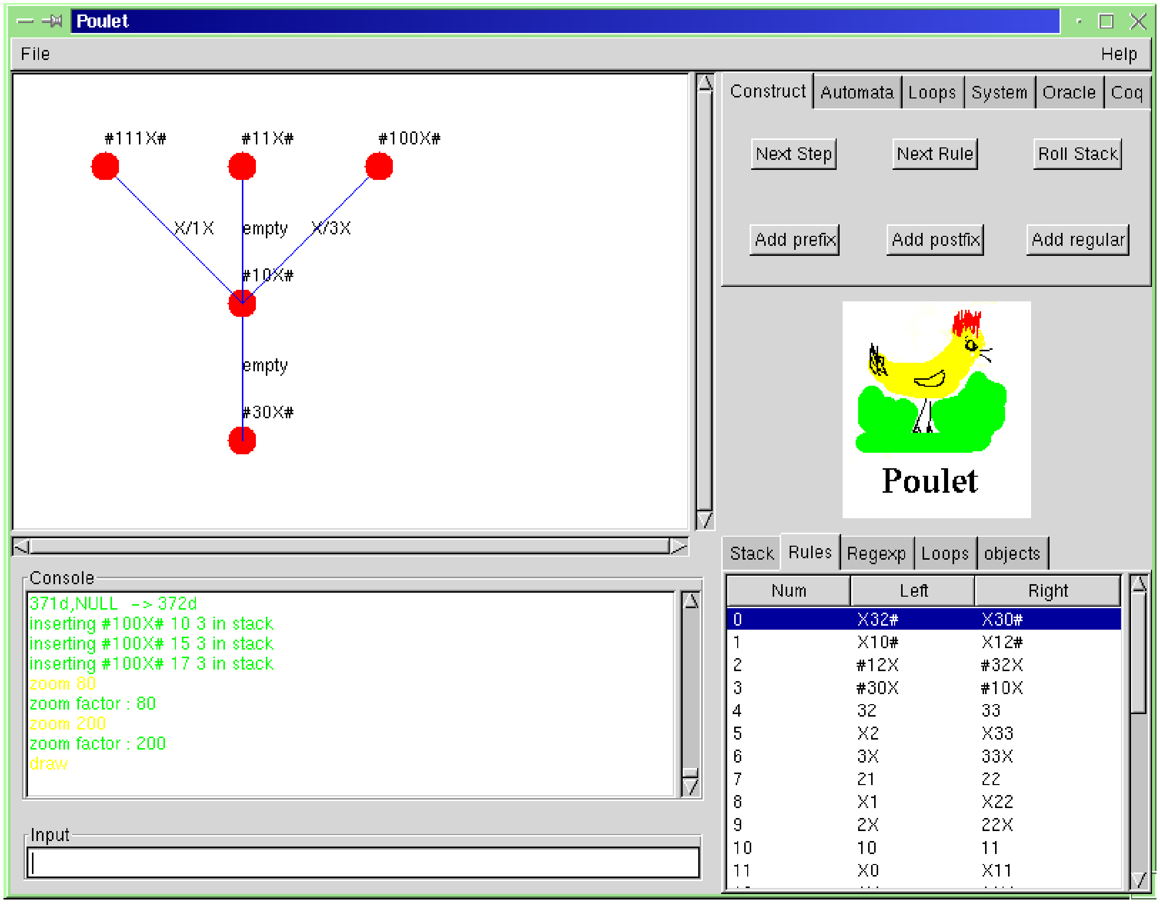

Fig. 6.1 – *Capture d'écran de Poulet*

Structure

Les structures C correspondantes sont définies dans le fichier “graphe.h”:

```
typedef struct edge
{
  char * subst;
  int rulenum;
  struct node * next;
  struct edge * ne;
} edge;

typedef struct node
{
  int nu;
  char * val;
  float x;
```

```
  float y;
  void * assoc;
  struct node * nn;
  struct edge * ne;
} node;

typedef struct graph
{
  struct node * nn;
  int placed;
  int maxx,maxy;
} graph;
```

Un noeud est représenté par la structure “node” qui contient les champs suivants:

- `nu` : numéro unique dans le graphe.
- `val` : chaîne de caractères. C'est le nom du noeud, c'est-à-dire l'expression régulière qu'il représente.
- `x` et `y` : coordonnées du noeud pour sa représentation graphique.
- `assoc` : pointeur vers l'automate associé à l'expression régulière `val`.
- `nn` : c'est le noeud suivant dans la liste chaînée des noeuds du graphe.
- `ne` : c'est le pointeur vers la liste chaînée des arêtes qui ont ce noeud pour origine.

Une arête est représentée par la structure “edge” qui contient les champs suivants :

- `subst` : chaîne de caractères représentant la substitution qui correspond à la transition associée.
- `rulenum` : entier indiquant le numéro de la transition associée.
- `next` : noeud destination de l'arête.
- `ne` : arête suivante dans la liste chaînée.

Enfin, les graphes sont représentés par la structure “graph” qui contient les champs suivants :

- `nn` : liste chaînée des noeuds du graphe.
- `placed` : indique si les coordonnées de chaque noeud ont été calculées ou non.
- `maxx`, `maxy` : coordonnées maximum des noeuds du graphe : ces variables sont utilisées pour que le dessin du graphe prenne le plus de place possible sur la représentation graphique (autofit).

Les fonctions

De nombreuses fonctions sont disponibles pour manipuler les graphes en voici la liste :

```
graph_init         : permet de créer un nouveau graphe vide.
graph_free         : détruit un graphe.
graph_save         : enregistre un graphe dans un fichier.
graph_load         : charge un graphe à partir d'un fichier.
graph_serialize    : sérialise un graphe (sauvegarde).
graph_deserialize  : désérialise un graphe (chergement).
graph_place        : calcule les coordonnées des noeuds d'un graphe.
ajout_arete        : ajoute une arête à un graphe.
ajout_noeud        : ajoute un noeud à un graphe.
search             : cherche une noeud dans un graphe par son nom.
graph_copy         : copie une graphe.
freenode           : détruit un noeud.
freedge            : détruit une arête.
suppr_noeud        : supprime une noeud.
suppr_arete        : supprime une arête dont on connaît le nom.
suppr_une_arete    : supprime une arête.
search_arete       : cherche une arête dont on connaît le nom.
ajout_prefixe      : ajoute un préfixe à tous les noms des noeuds du graphe.
graph_zoom         : multiplie les coordonnées des noeuds par un facteur.
ajoute_expr        : ajoute une expression régulière au graphe.
graph_calc_loops   : calcule les boucles dans un graphe.
```

La fonction `ajoute_expr` est l'une des fonctions les plus importantes de Poulet, c'est elle qui permet d'ajouter une expression régulière dans un graphe. Pour cela, elle calcule les automates associés aux expressions régulières de tous les noeuds du graphe, les compare et décide des inclusions éventuelles. Elle s'occupe également de supprimer les entrées correspondantes aux expressions détruites dans la pile d'actions. Cette fonction implémente l'algorithme principal décrit dans le chapitre 3.

Les fichiers

Les graphes, comme toutes les structures de Poulet, sont sérialisables, c'est-à-dire qu'ils sont munis d'une fonction qui permet, d'une manière générique, de les sauvegarder sur tout support. Le format utilisé a été conçu pour être lisible et éventuellement créé par un utilisateur. Pour cette raison, le format est ASCII et parfaitement compréhensible. De plus, le format de graphe est très similaire à celui utilisé par le logiciel *graphplace* ([Eij]) qui sert de base au placement des noeuds sur le dessin.

Les fichiers de graphes commencent par un préfixe : #graph qui sert à indiquer le type de fichier. Ensuite, le fichier contient deux sortes de lignes :

- lignes de noeuds : décrit un noeud du graphe avec le format : "(nom) nom node". où nom est le nom du noeud (son expression régulière) écrit deux fois de suite : la première est cosmétique et la deuxième est le nom interne. node est un mot clé.

- lignes d'arêtes : décrit une arête du graphe avec le format : "(nom) node1 node2 edge", où nom est le nom de l'arête (sa substitution), node1 est le nom du noeud origine, node2 est le nom du noeud destination et edge est un mot clé.

Par exemple, le noeud représenté par la figure 6.2 sera sérialisé sous la forme :

```
#!graph
(a) a node
(b) b node
(c) c node
(ab) a b edge
(ac) a c edge
```

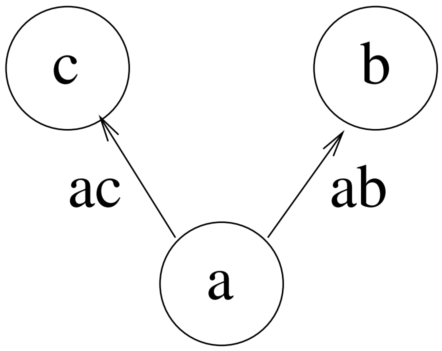

Fig. 6.2 – *Graphe sérialisé*

6.3.2 Automates

Avec les graphes, les automates sont les structures les plus utilisées dans Poulet. En effet, les expressions régulières qui modèlisent le système sont transformées en automates pour pouvoir être comparées et transformées. Ils sont décrits dans les fichiers "automata.h" et "automata.c"

La structure

Ils sont représentés par la structure C suivante décrite dans le fichier automata.h:

```
struct automata
{
  struct node * first;
  struct node * last;
  struct graph * g;
}
```

Le champ `g` contient le graphe associé à l'automate. Les champs `first` et `last` désignent les noeuds correspondent respectivement à l'état initial et à l'état final. En effet, les expressions régulières que nous manipulons sont toujours de la forme $\#\Sigma^*\#$, où Σ est l'alphabet. Cela implique qu'il n'existe qu'un seul état initial et qu'un seul état final dans les automates considérés.

Les fonctions

Voici la liste des fonctions disponibles sur les automates.

```
autom_alloc        : crée un automate vide.
autom_free         : détruit un automate.
autom_copy         : copie un automate.
inclusion          : décide si un automate est inclus dans un autre.
subinclusion       : décide si un automate est inclus en partie dans un autre.
suppreps           : déterminise un automate
    (et supprime les epsilon transitions).
coherent_output    : décide si un automate engendre tous les mots engendrés
    par un autre automate.
autom_deserialize : désérialise un automate.
autom_load         : charge un automate à partir d'un fichier.
autom_serialize    : sérialise un automate.
autom_save         : enregistre un automate dans un fichier.
```

Les fichiers

Le format de sérialisation des automates commence par le préfixe : "#autom", puis suivent deux lignes comportant les noms de l'état initial et de l'état final. Enfin, le graphe est décrit suivant la sérialisation décrite dans la sous-section consacrée aux graphes.

Par exemple, un automate ayant le graphe de la figure 6.2 comme graphe de base. a comme état initial et b comme état final sera sérialisé comme suit :

```
#!autom
a
b
#!graph
(a) a node
(b) b node
(c) c node
(ab) a b edge
(ac) a c edge
```

Autre exemple, un automate représentant l'expression régulière "# 1+ #" sera sérialisé comme suit :

```
#!autom
5d
8d
#!graph
(5d) 5d node
(6d) 6d node
(7d) 7d node
(8d) 8d node
(#) 5d 6d edge
(1) 6d 7d edge
(1) 7d 7d edge
(#) 7d 8d edge
```

6.3.3 Expressions régulières

Pour transformer les expressions régulières en automates, Poulet implémente la fonction "expr2auto" qui est basée sur la génération d'automates de Thompson ([Tho68]) utilisée dans l'utilitaire *GNU grep*.

La grammaire utilisée est la suivante :

```
ground : [^+*|()]
expr : ( expr ) *        |
       ( expr ) +        |
       ( expr | expr )   |
       ground            |
       expr expr
final : \# expr \#
```

Le terminal `ground` désigne tout caractère de l'alphabet. Le non terminal `expr` désigne les constructions possibles : la concaténation, l'étoile (*), le plus (+) et le ou (|). Le nom terminal `final` désigne la forme construite de toute expression régulière, c'est-à-dire une expression entourée par des symboles #.

Par exemple on peut écrire les expressions suivantes :

```
#(1)+#
#(1|2)#
#(2)*#
#(((1)*|(2)*)+#
```

6.4 Règles de réécriture

Les règles de réécriture sont la traduction de l'algorithme à prouver dans le formalisme de Poulet. Pour cette raison, il est nécessaire que cette formalisation soit faite avec beaucoup de soin pour éviter de prouver quelque chose qui ne reflète pas exactement l'algorithme original. Cette partie est strictement manuelle, car il existe trop d'aspects sémantiques pour automatiser cette opération.

6.4.1 Le format des règles

Les fichiers de règles ont pour préfixe : "#rules i" où i est un entier égal au nombre de règles présentes dans le fichier. Ensuite, chaque ligne est soit une règle, soit un commentaire si elle commence par le caractère '%'.

Une règle compte quatre champs : le redex, le résultat de l'application, le sens de la règle par rapport à X (la variable) et la substitution correspondante.

Par exemple, si la règle est du type 01 → 11, on aura pour redex 01, pour résultat 11, pour sens 'n' qui signifie que X n' intervient pas dans l'application de la règle et pour substitution "empty" pour signifier que la variable X n'a pas a être instanciée pour appliquer cette règle, soit l'expression suivante :"01 11 n empty".

Prenons un autre exemple, reprenons la règle précédente avec comme redex X1. On aura comme résultat X11, pour sens 'd' car l'instanciation se fait à droite de X. Enfin la substitution sera "X/X0" pour retrouver le redex de l'exemple précédent, soit l'expression suivante : "X1 X11 d X/X0".

6.4.2 Traduire les règles

La première étape de cette traduction est d'établir un alphabet : à chaque état d'un processus doit correspondre une lettre. Par exemple, si l'algorithme ne manipule qu'une seule variable, ce peut être la valeur de cette variable.

Une fois que l'alphabet est fixé, il faut traduire chaque transition de l'algorithme en une règle de réécriture. Pour cela, on regarde un processus et ses voisins concernés dans la règle avant et après son application. Cela nous donne un redex et un résultat d'application.

Ensuite, il faut trier les règles en trois catégories : les règles Top, Bottom et Middle. Il faut ensuite ajouter le # à droite (respectivement à gauche) des règles Top (respectivement Bottom).

Prenons une règle $ab \to ab'$. Il faut maintenant exprimer cette règle dans le formalisme de la réécriture :

- Règle Top :
  - règle sans instanciation : ab# ab'# n empty
  - règle avec instanciation : Xb# Xab'# d X/Xa
- Règle Bottom :
  - règle sans instanciation : #ab #ab' n empty
  - règle avec instanciation : #aX #ab'X g X/bX
- Règle Middle
  - règle sans instanciation : ab ab' n empty
  - règle avec instanciation à gauche : aX ab'X g X/bX
  - règle avec instanciation à droite : Xb Xab' d X/Xa

6.4.3 Exemple de l'algorithme à quatre états de Gosh

L'alphabet est composé de quatre lettres 0,1,2,3 pour les lettres centrales et 0,2 (respectivement 1,3) pour la machine la plus à droite (respectivement à gauche).
Les états légitimes sont du type $\#\{1,3\} + \{0,2\} + \#$.
Enfin, l'algorithme dispose de six règles :

- Règles Middle :
  - M1 : $(q+1)q \to (q+1)(q+1)$
  - M2 : $q(q+1) \to (q+1)(q+1)$

  pour tout $q \in \{0,1,2,3\}$ et $+$ l'addition modulo 4

- Règles Top :
  - T1 : 32# $\rightarrow$ 30#
  - T2 : 10# $\rightarrow$ 12#
- Règles Bottom :
  - B1 : #12 $\rightarrow$ #32
  - B1 : #30 $\rightarrow$ #10

Nous écrirons les règles Top sous la forme de quatre lignes :

```
X32# X30# d empty
X10# X12# d empty
32# 30# n empty
10# 12# n empty
```

Les deux premières sont les règles qui serviront de base aux graphes de dérivations et les deux suivantes seront appliquées quelle que soit la position de X.

Les règles Bottom seront écrites sous cette forme :

```
#X #32X g empty
#X #10X g empty
#12 #32 n empty
#30 #10 n empty
```

Les deux premières sont les règles qui permettent d'instancier la partie des mots qui se situent à gauche de X et les deux dernières seront exécutées quelle que soit la position de X.

6.5 Les structures de stockage

Pour faire ses preuves, le programme dispose d'un certain nombre de structures qui lui permettent de stocker des informations. Ces structures sont en général représentées à l'écran dans les onglets qui se situent en bas à droite de l'interface.

6.5.1 La table des objets nommés

Poulet manipule des objets qui peuvent être des graphes ou des automates. En réalité, la plupart des fonctions sont appelées avec pour argument un objet particulier qui s'appelle “current” et qui est un objet générique qui sert à toutes les opérations courantes.

Il est toutefois possible de manipuler d'autres objets et notamment de les sauvegarder pour une utilisation ultérieure. Pour cela, une structure appelée table des objets nommés est disponible. C'est une table d'association qui permet d'associer un objet quelconque et un nom unique (indépendant du nom de fichier). Cette table est notamment utilisée au cours de la preuve pour stocker les différents graphes de dérivations associés à chacune des règles Top.
Elle est décrite dans les fichiers “namingt.h” et “namingt.c”

6.5.2 La pile

La pile est une table qui recense les actions qui restent à effectuer pour compléter les différents graphes de réductions. C'est l'implémentation de la structure P dans l'algorithme de preuve du chapitre 3. Chacune de ses entrées comporte l'expression régulière sur laquelle porte l'action, un entier qui est le numéro de la règle à appliquer et un autre entier qui indique sur quel graphe de dérivations doit porter cette dérivation. Lors de l'exécution d'une dérivation, le premier élément de la pile est choisi, la dérivation est calculée puis son résultat est injecté dans le graphe de dérivations correspondant. Le terme de pile est légèrement impropre dans la mesure où les éléments qui la constituent sont triés à chaque nouvelle introduction ou suppression d'éléments. En effet, de ce tri dépend l'efficacité de la preuve. Les règles sont triées en fonction de leur numéro d'apparition dans le fichier original. Les actions sont ensuite classées sur le numéro du graphe de dérivations correspondant puis sur le numéro de la règle à appliquer. Ce tri évite que les dérivations soient appliquées dans n'importe quel ordre et garantit une unité dans les actions successives. Cela a pour effet de réduire le nombre d'expressions régulières présentes en même temps à l'écran car les schémas apparaissent plus vite et sont généralisés au fur et à mesure.
La pile est décrite dans les fichiers “pile.h” et “pile.c”

6.5.3 La liste des cycles

Lorsque la construction de la preuve est finie, il est nécessaire de calculer tous les cycles présents dans les graphes de dérivations afin de déterminer si l'algorithme est auto-stabilisant. Pour cela, Poulet utilise une table des cycles dans laquelle il répertorie tous les cycles détectés.

Ensuite, il applique le meta-argument des instanciations nulles, ce qui lui permet d'éliminer les cycles non infinis. Ceux-ci sont marqués dans la table comme étant non infinis ce qui les excluent de la preuve ainsi qu'il est décrit dans le chapitre précédent.

6.6 Algorithmes

L'algorithme principal de la preuve contient deux fonctions principales : la fonction "next" qui permet d'exécuter un pas de calcul de l'algorithme et la fonction "addreg" qui permet d'ajouter une expression régulière dans le graphe courant. Il existe également une fonction d'initialisation qui est exécutée lors du chargement d'un jeu de règles. Une fois les graphes de dérivations construits, il est nécessaire de calculer les cycles présents dans ces graphes puis de déterminer si l'algorithme est auto-stabilisant ou non. Pour cela, le programme dispose de deux fonctions : "calcule_loops" et "ss".

6.6.1 La fonction d'initialisation

Pour commencer une preuve, la première chose à faire est de charger un ensemble de règles. Pour cela, on utilise la fonction "load" dont l'argument est le nom du fichier contenant les règles, formaté comme expliqué dans la section précédente. Lors de ce chargement, la table des règles est remplie avec les règles du fichier. Puis, la fonction cherche les règles Top afin de créer les graphes correspondants. Un graphe de dérivations est créé pour chacune de ces règles : il ne contient qu'une seule expression régulière qui correspond à la partie gauche de la règle Top correspondante. Ces graphes sont rangés dans la table des objets nommés sous le nom "g_rules_i" où i est un entier qui va de 0 au nombre de règles Top moins un. Enfin, la fonction remplit la pile avec les expressions régulières engendrées et leur numéro de règle associé.

6.6.2 La fonction "next"

La fonction "next" est celle qui permet d'avancer d'un pas dans un graphe de dérivations. Pour cela, cette fonction prend le premier élément de la pile et calcule la dérivation. Le résultat de cette dérivation est une expression régulière qu'il faut injecter dans le graphe. Pour cela, la fonction appelle le fonction "addreg".

Le calcul de la dérivation se fait de la façon suivante : tout d'abord, l'expression régulière est transformée en automate d'une manière automatique au moyen de la méthode des automates de Thompson. Cette transformation automatique est suivie d'une déterminisation de l'automate (avec suppression des epsilon transitions). Puis la partie gauche de la règle est étendue à une expression régulière décrivant un mot complet : par exemple, une règle $ab \to ac$ verra sa partie gauche transformée en $\#\Sigma^* ab \Sigma^* \#$. Ensuite le programme calcule l'intersection de l'automate cible et de l'automate de la règle. L'automate obtenu est alors retransformé en expression régulière. Celle-ci contient la partie gauche de règle complètement instanciée et il suffit alors de la remplacer par la partie droite.

6.6.3 La fonction "addreg"

La fonction "addreg" est une fonction très centrale dans Poulet, elle permet d'ajouter une expression régulière dans un graphe. Elle est appelé soit par la fonction "next" lors du calcul d'une nouvelle expression issue du graphe soit directement par l'utilisateur ou l'oracle pour proposer un schéma unificateur.

Tout d'abord la fonction calcule l'automate associé à l'expression régulière qu'il doit ajouter. Ensuite, il parcourt le graphe et cherche s'il existe un noeud tel que le nouvel automate soit inclus dans l'expression régulière de ce noeud. Si ce noeud existe, il faut, si c'est nécessaire, lui rajouter la nouvelle arête correspondante à la transition. La fonction rajoute cette arête et termine.

Si ce noeud n'existe pas, la fonction cherche un noeud tel que son expression régulière soit incluse dans le nouvel automate. Si un tel noeud existe, la fonction rajoute ses arêtes au nouveau noeud et supprime les entrées correspondantes dans la pile. Enfin, elle ajoute le nouveau noeud ainsi que ses entrées correspondantes dans la pile. Pour cela, elle calcule si l'intersection du nouvel automate et de l'automate de chacune des règles est vide ou non. Si l'intersection est non vide, une entrée correspondante est créée dans la pile.

6.6.4 Les fonctions de calcul des cycles

Lorsque le calcul des graphes de dérivations est fini, il est nécessaire de calculer les cycles qui y sont présents afin de déterminer si l'algorithme est auto-stabilisant. La fonction "calcule_loops" se charge de cette tâche. De plus, elle examine les substitutions au long de ces cycles. Si la substitution globale le long d'un cycle est nulle, cela signifie qu'il peut être infini. Cela implique qu'il existe une chaîne top quasi-cyclique dans le graphe de dérivations.

La fonction "ss" observe chacun de ces cycles et si tous ont une substitution globale non nulle, elle décide que l'algorithme est auto-stabilisant.

6.7 Prouver un algorithme

Pour illustrer la méthode utilisée pour montrer un algorithme, nous allons détailler un exemple très simple qui permet de se faire une bonne idée du mode opératoire.

Le fichier de règles est le suivant : c'est un algorithme dont les états légitimes sont inclus dans l'expression régulière $\#0(1)*\#$.

```
#!rules 16
% top rules with instanciation
```

```
X0# X01# d X/X0
X0# X11# d X/X1
%top rules without instanciation
00# 01# n empty
10# 11# n empty
% bottom rules with final instanciation
#X #01X g X/0X
#X #01X g X/1X
#X #11X g X/0X
% bottom rules wo instanciation
#00 #01 n empty
#11 #01 n empty
#10 #11 n empty
% middle rules wo instanciation
00 01 n empty
10 11 n empty
% middle rules with right instanciation
X0 X11 d X/X1
X0 X01 d X/X0
% middle rules with left instanciation
0X 01X g X/0X
1X 11X g X/0X
```

Pour plus de clarté, nous avons supprimé les règles d'instanciation de X qui compliquent l'algorithme pour un intérêt assez faible.

La première chose à faire est d'introduire les règles dans le programme. Ceci se fait en tapant dans la console la commande “load demo.rules” où demo.rules est le nom du fichier de règles précédent. Ceci a pour effet de créer deux objets nommés : g_rules_0 et g_rules_1, ainsi que les deux éléments de pile correspondants.

Ensuite, il faut faire exécuter quelques pas à l'algorithme pour avoir une idée générale de la forme du graphe de dérivations. Ceci se fait en tapant la commande “next” dans la console ou plus simplement en cliquant sur le bouton “Next Step” de l'onglet de commande “Construct”.

Au bout de quelques pas, on obtient la configuration décrite par la figure 6.3.

On constate que les deux états entourés dont les expressions régulières sont respectivement $\#X111\#$ et $\#X1111\#$ sont généralisables par un schéma du type $\#X1(1)*\#$. Si on souhaite le vérifier, on peut avancer de quelques pas supplémentaires pour voir apparaître les suivants. Pour ajouter ce schéma, il suffit de taper dans la console la commande “addreg $\#X1(1)*\#$”. On obtient la configuration décrite par la figure 6.4. On peut voir que les deux états ont été remplacés par notre généralisation.

Sur cette deuxième figure, on constate deux autres régularités généralisables par les expressions régulières $\#01X1(1)*\#$ et $\#01X0(1)*\#$. Nous ajoutons ces expressions

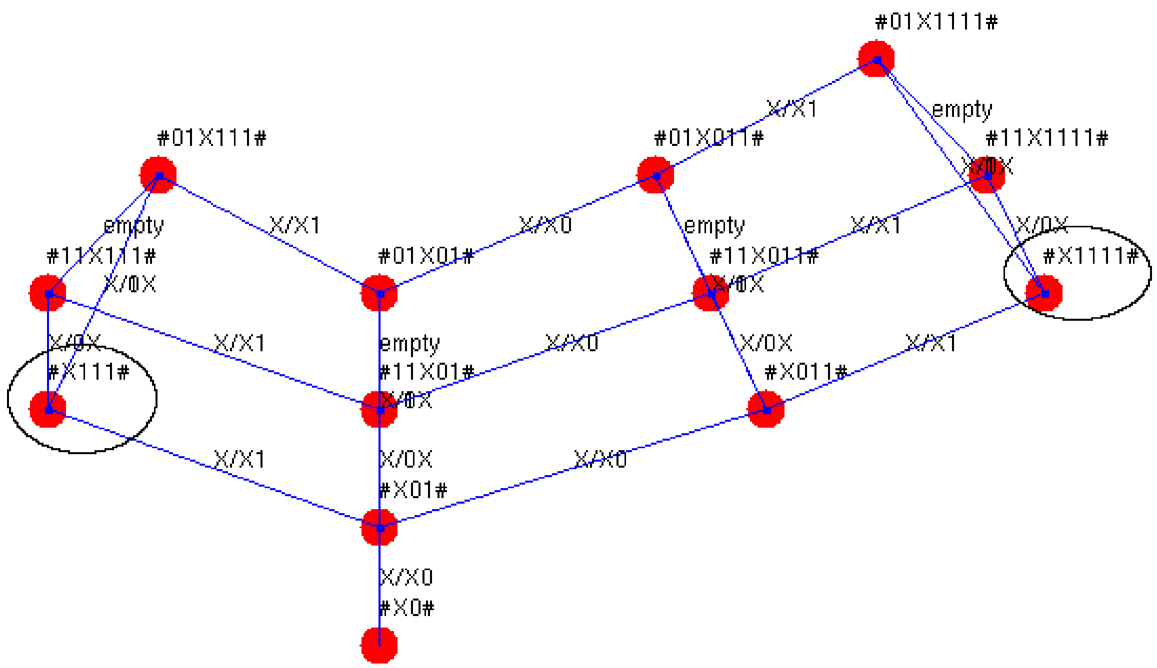

FIG. 6.3 – *Généralisation par* $\#X1(1)*\#$

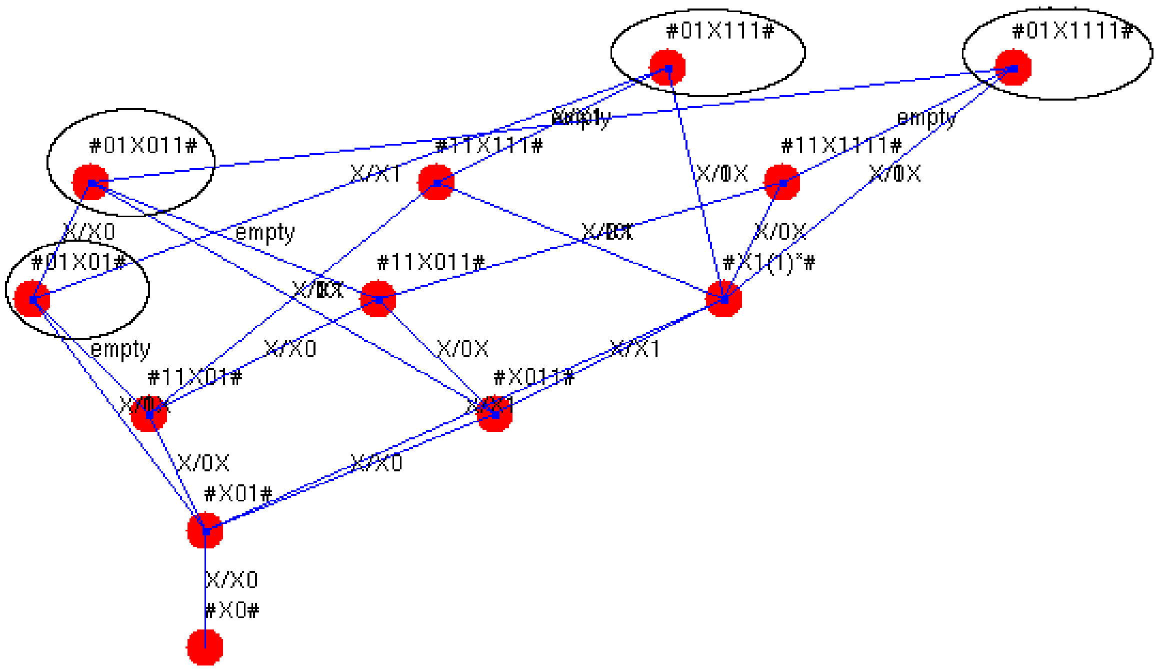

FIG. 6.4 – *Généralisation par* $\#01X1(1)*\#$ *et* $\#01X0(1)*\#$

régulières avec la commande “addreg” puis nous exécutons quelques pas de calculs pour voir l’effet produit. Nous obtenons la configuration décrite par la figure 6.5.

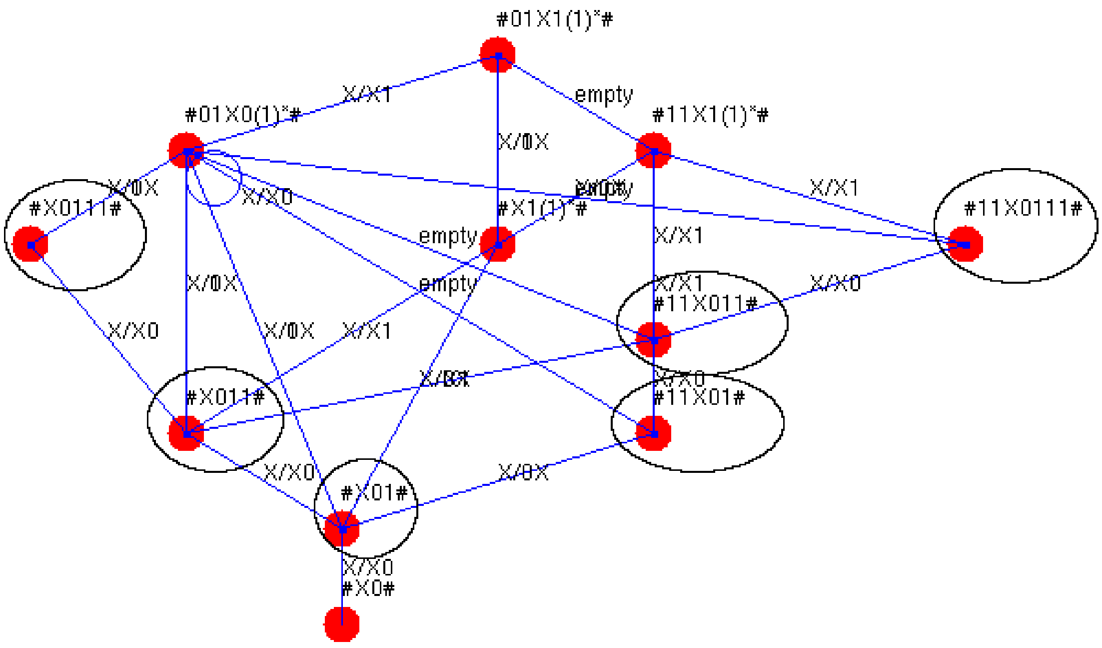

Fig. 6.5 – *Généralisation par* $\#X0(1)*\#$ *et* $\#11X0(1)*\#$

Nous constatons à nouveau deux régularités généralisables par les expressions régulières $\#X0(1)*\#$ et $\#11X0(1)*\#$. Une fois ces schémas ajoutés, nous obtenons la configuration décrite par la figure 6.6.

Sur cette figure, nous constatons que tous les états précédemment engendrés par l’algorithme ont été généralisés par des schémas. Toutefois, après quelques pas de plus, nous pouvons constater que la partie gauche des mots (à gauche de X) est de la même forme que la partie droite, c’est-à-dire généralisable par $0(1)*$ et $(1)*$ comme le montre la figure 6.7.

Nous introduisons les expressions régulières correspondantes et nous obtenons la forme définitive du graphe comme le montre la figure 6.8.

Une fois le premier graphe de dérivations déterminé, nous passons au deuxième, qui correspond à la deuxième règle Top. Après quelques pas de calcul, nous obtenons la figure 6.9.

Deux généralisations sont visibles : $\#0(1)*1X11\#$ et $\#(1)*1X11\#$. Une fois cette généralisation faite, nous obtenons la forme définitive du deuxième graphe de dérivations comme le montre la figure 6.10.

Il ne reste plus qu’à cliquer sur le bouton “Calcule” de l’onglet “Loops”. La liste

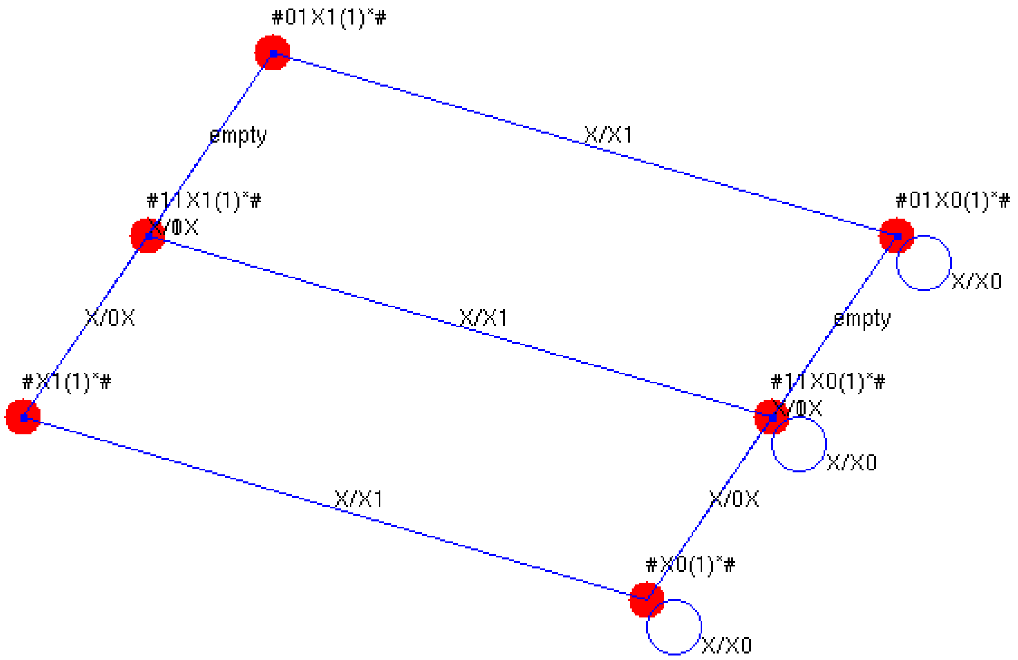

FIG. 6.6 – *Sous-graphe complètement généralisé*

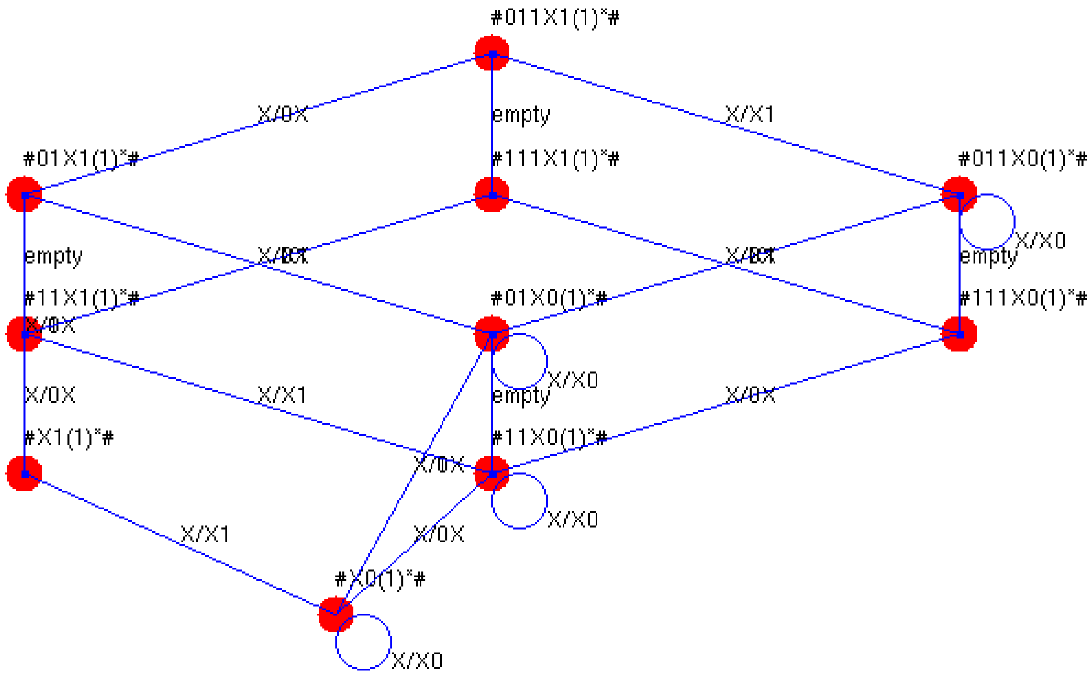

FIG. 6.7 – *Même généralisation sur la partie gauche*

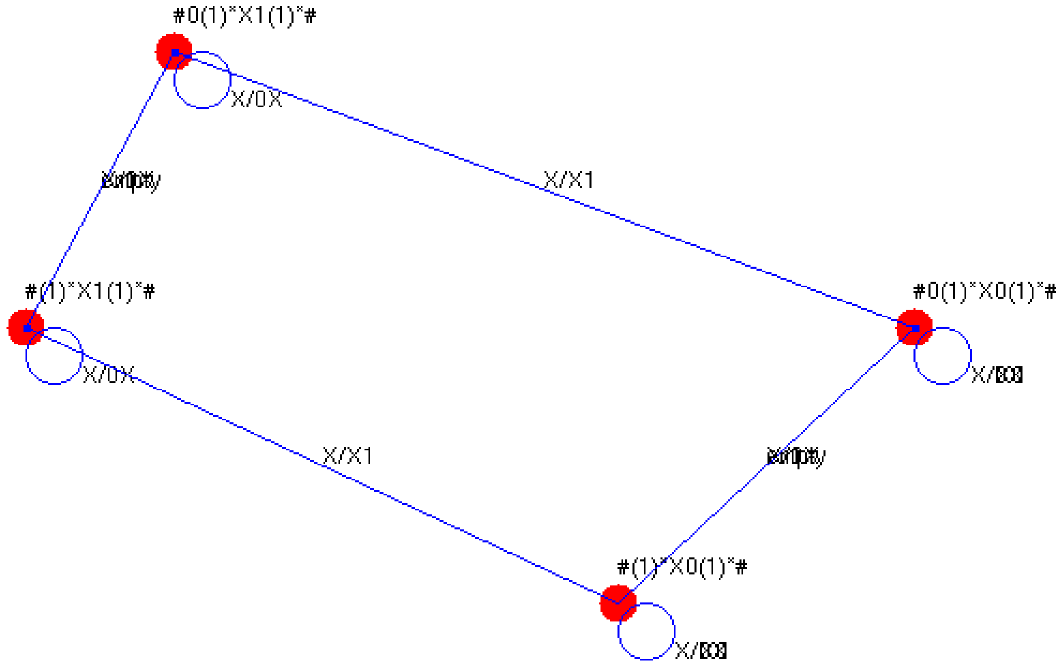

FIG. 6.8 – *Forme finale du premier graphe de dérivations*

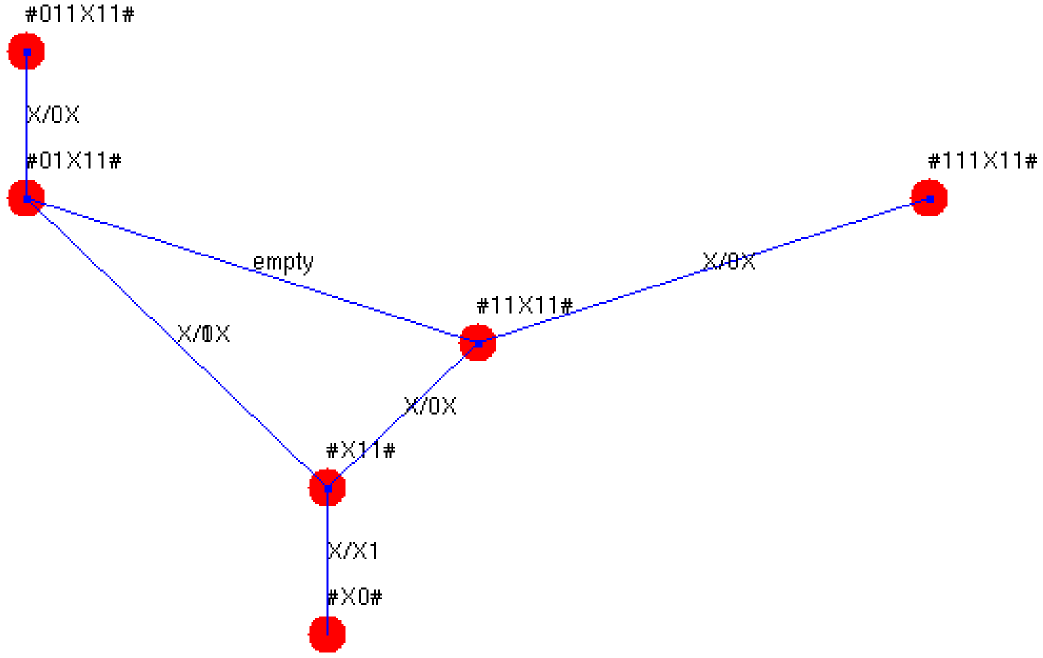

FIG. 6.9 – *Généralisation par $\#0(1)*1X11\#$ et $\#(1)*1X11\#$*

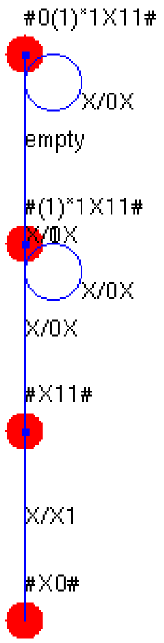

FIG. 6.10 – *Forme finale du deuxième graphe de dérivations*

des cycles détectés apparaît alors dans la table des cycles. En cliquant sur le bouton “SS?” de l’onglet “Loops”, le programme analyse les cycles et écrit dans la console : “The protocol is self stabilizing!!!”.
La preuve est terminée.

Dans cette preuve, les instanciations finales ne sont pas prises en compte. Voici le résultat final de la preuve de l’algorithme auquel on a rajouté les deux règles suivantes qui permettent l’instanciation finale de la variable X pour former un mot clos.

```
X 1 f X/1
X 0 f X/0
```

Les deux figures 6.11 et 6.12 montrent les deux graphes de dérivations obtenus : on constate que cette instanciation supplémentaire ajoute beaucoup d’états et nuit un peu à la clarté de la preuve. C’est pourquoi nous l’avons scindée de cette manière.

6.8 Fonctionnalités additionnelles

Pour faciliter la manipulation des preuves, Poulet dispose de quelques fonctionnalités additionnelles.

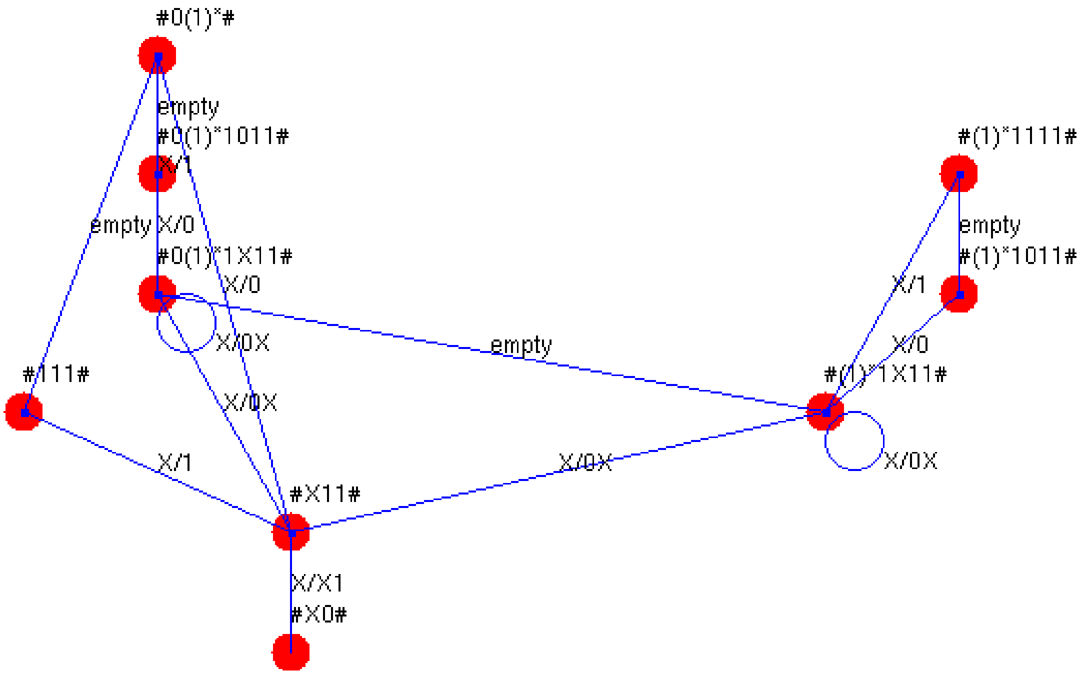

FIG. 6.11 – *Forme finale du premier graphe de dérivations*

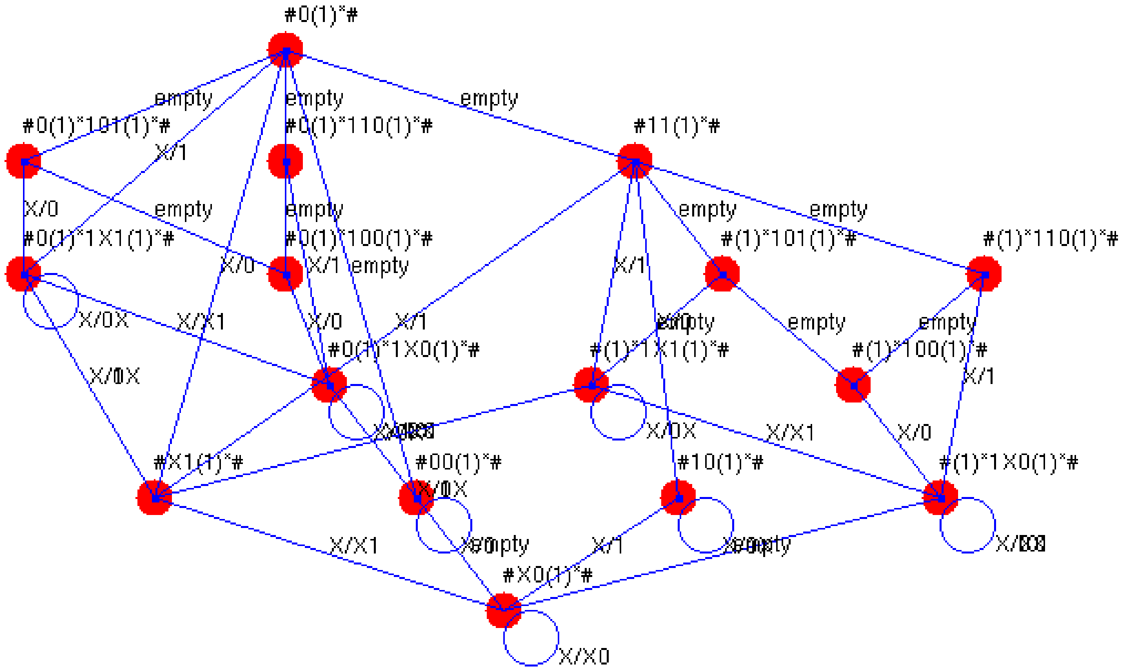

FIG. 6.12 – *Forme finale du deuxième graphe de dérivations*

6.8.1 Production de Coq

Le but originel de Poulet était de générer des preuves pour le programme de vérification formelle Coq (c'est de là que vient son nom). Coq est un système de manipulation de preuves mathématiques formelles. Les preuves qu'il réalise sont mécaniquement vérifiées par le noyau qui est certifié sans erreur.

L'interfaçage entre les deux programmes se fait au moyen de génération de code Coq à partir de la preuve terminée. Ces programmes comportent deux parties : la partie de définition des règles et la partie de la preuve proprement dite.

La partie des définitions contient autant de définitions inductives que de règles. Il faut également spécifier l'ensemble des états légitimes ainsi que l'alphabet.

La partie de preuve est une série de lemmes qui indiquent quelles sont les transitions du graphe de dérivations. Une série de lemmes génériques sont associés à ces lemmes de preuve pour produire une preuve Coq qui peut ensuite être vérifiée. La faiblesse de Coq vis-à-vis de ce problème est qu'il ne dispose pas d'une librairie d'expressions régulières. Par conséquent, la preuve ne peut être complètement automatique car il est nécessaire de rajouter des petits axiomes ad'hoc pour pouvoir obtenir la preuve finale.

6.8.2 Sauvegarde de contexte

Afin de pouvoir revenir sur certaines configurations qui apparaissent au cours de la preuve où il est nécessaire de prendre des décisions qui ne seront peut-être pas les bonnes, il est possible à tout moment de sauvegarder le contexte complet de Poulet. En effet, toutes les structures (objets et structures de stockage) sont sérialisables. Ainsi, il est possible de sauvegarder tout ce que manipule Poulet. En particulier, la sauvegarde de contexte permet de retourner à un état sauvegardé de la preuve en cours. Cela est également pratique lorsqu'on fait sa preuve en plusieurs fois.

Pour sauvegarder le contexte, il suffit de taper dans la console "save context". Le context est alors sauvegardé dans un fichier qui s'appelle "poulet.context". Pour retrouver un contexte, il faut taper dans la console la commande "load context".

6.8.3 Les scripts

Afin d'automatiser certaines tâches fastidieuses, il est possible d'écrire des scripts de commandes qui seront interprétées par Poulet comme si elles avaient été tapées dans la console par un utilisateur. Cela permet par exemple de faire une preuve incomplète sans avoir à retaper les expressions régulières avant d'essayer différentes méthodes pour la fin.

Voyons un exemple de script :

```
#!script
load demo.rules
FOR 2
next
addreg #(0|1)(1)*1X1(1)*#
addreg #(0|1)(1)*1X0(1)*#
addreg #X1(1)*#
addreg #X0(1)*#
FOR 25
next
addreg #(0|1)(1)*1X11#
FOR 2
next
```

La première ligne contient le mot clé #!*script* qui indique au programme qu'il s'agit d'un script (comme tous les fichiers sont chargés avec la même commande "load", cet en-tête permet au programme de savoir quel type de fichier il doit traiter). Les autres commandes sont telles qu'elles sont tapées dans la console. Le mot clé "FOR" permet de répéter une action un nombre défini de fois sans avoir à la retaper. Par exemple,

```
FOR 25
next
```

indique que l'action next va être exécutée vingt cinq fois.

6.8.4 Oracle de schémas

Il peut être intéressant d'utiliser un oracle pour engendrer les expressions régulières qui généralisent les graphes de dérivations. De nombreux articles traitent de l'inférence de langages à partir d'instances dans des cadres différents. Le problème général de l'inférence se pose de la façon suivante : à partir d'un ensemble fini d'instances d'un langage, et éventuellement d'un ensemble fini de contre-exemples, on doit identifier une grammaire incluant tous les éléments positifs. Notre problème se pose d'une façon légèrement différente : à partir d'un ensemble de mots clos, il faut déterminer une grammaire qui regroupe certains éléments qui présentent une régularité commune.

Ce problème est très difficile et n'admet pas de solution exacte dans la majorité des cas. Par exemple, dans [Gol67], Gold montre que le problème ne peut être correctement résolu en l'absence de contre-exemples (ce qui est malheureusement notre cas). Néanmoins, Angluin a proposé une méthode d'apprentissage basée sur la possibilité de décider de l'inclusion d'un mot dans le langage ([Ang87]). D'autres solutions ont été proposées en raffinement de cette méthode : [PH93], [PH96] et [Par98].

Des études ont été menées dans le cadre des algorithmes génétiques qui donnent des résultats intéressants : [DMV94] et [Dup96] proposent de l'inférence à partir d'éléments positifs et négatifs.

Enfin, des résultats intéressants ont été obtenus au moyen de réseaux de neurones artificiels de type perceptrons multi-couches. Par exemple, [GMC$^+$91] propose une méthode d'entraînement pour ce type de réseau qui permet d'inférer des langages réguliers.

Ce problème aux multiples solutions n'est pas résolu dans Poulet. En revanche, le programme dispose d'un oracle vide qui permet de programmer toutes les approches. Le programme d'oracle se connecte au programme principal via une socket Internet. Il est informé de toutes les actions du programme et de l'utilisateur. En fonction de son rôle consultatif ou exécutif, il peut proposer des expressions ou directement les soumettre au programme principal qui les intégrera dans les graphes de dérivations.

6.9 Résumé

Dans ce chapitre, nous avons présenté le programme Poulet qui est un assistant pour les preuves de convergence des algorithmes auto-stabilisants sur des topologies linéaires. Le programme calcule toutes les configurations des graphes de dérivations et intègre dans ces graphes les généralisations proposées par l'utilisateur. Grâce à une interface graphique, cette opération se fait relativement facilement et évite toute erreur de calcul.

Chapitre 7

Conclusions et perspectives

Concevoir et prouver des algorithmes auto-stabilisants sont des activités difficiles surtout si les contraintes de démons sont fortes ou si l'algorithme maintient beaucoup de variables. La méthode des mesures décroissantes comme sa généralisation, la méthode des attracteurs, ne présentent aucune généricité d'où l'impossibilité de les automatiser. De plus, les mesures sont souvent assez ardues à découvrir. Il est donc intéressant de vouloir simplifier ou automatiser la phase de preuve des algorithmes auto-stabilisants.

Afin de simplifier la conception et la preuve d'algorithmes, nous avons présenté un algorithme d'exclusion mutuelle locale. Cet algorithme peut être composé avec n'importe quel autre algorithme auto-stabilisant sous le démon quasi-central au moyen de la technique de composition croisée afin de le rendre auto-stabilisant sous un démon distribué. De plus, la composition peut être faite à un niveau d'atomicité très fin (atomicité lecture/écriture).

Au dela de la simplification des preuves, il peut être intéressant d'automatiser celles-ci. Les méthodes habituelles de preuves automatiques comme le model-checking ne s'applique pas facilement car on est confronté au problème de l'explosion combinatoire. De plus, ces techniques permettent de faire des preuves pour des systèmes avec un nombre fixé de processus. De même, les systèmes formels inférentiels ne sont pas conçus pour faire ce genre de preuves et il est nécessaire de programmer des librairies additionnelles pour pouvoir les utiliser.

Nous avons donc présenté notre propre méthode d'automatisation des preuves de convergence. Celle-ci fonctionne pour les topologies linéaires en modélisant les états du système par des mots et l'algorithme par un système de réécriture. La preuve de convergence est alors réduite à une preuve de terminaison du système de réécriture, ce qui est beaucoup plus facile à automatiser.

Enfin, nous avons programmé un outil qui implémente cette méthode de preuve par réécriture. Il permet de prouver de manière interactive des algorithmes et éventuellement de les exporter vers le système inférentiel Coq.

Les perspectives de ces différents travaux sont nombreuses, surtout dans les extensions de la méthode de preuves par réécriture. Nous en présentons deux qui permettent d'envisager d'autres topologies et d'autres moyens de communications. En effet, les hypothèses que nous avons faites tant au niveau de la topologie que de la lecture des registres peuvent s'avérer trop restrictives. Nous montrons ici qu'il est possible de relaxer ces hypothèses et d'étendre le cadre des preuves en utilisant une méthode d'abstraction.

7.1 Le passage de messages

Jusqu'ici, nous n'avons considéré que des communications par lecture d'états. Nous allons voir dans cette section qu'il est parfois possible d'intégrer le passage de message dans une preuve par abstraction. Nous illustrons notre propos par un petit exemple : l'algorithme de la taille de l'anneau.

7.1.1 La taille de l'anneau

Soit un anneau de N processus tel que les canaux de communications soient unidirectionnels. Chacun des noeuds souhaite connaître la taille de cet anneau au moyen d'un algorithme auto-stabilisant.

L'algorithme

Chaque processus possède une constante que nous appelerons id et qui est un identifiant unique dans le réseau, et une variable que nous appelerons cpt et qui recevra la solution du problème. Les processus peuvent exécuter des actions de communication via deux primitives : rcp pour recevoir un message et emit pour envoyer un message. Comme l'anneau est unidirectionnel, un processus reçoit toujours un message de son voisin de gauche et envoie tous ses messages à son voisin de droite. Nous supposerons ici que les canaux sont équitables, c'est-à-dire qu'un message finit toujours par être reçu par son processus destination. Chaque message à la forme (idm,tm) où idm est l'identifiant de la machine qui l'a envoyé et tm est un compteur.

L'algorithme ne contient qu'un seule règle :

$$\begin{array}{lll} <\mathrm{rcp(idm,tm)}> & \rightarrow & \mathrm{si\ (id=idm)} \\ & & \mathrm{alors\ cpt=tm+1;\ emit(id,0)} \\ & & \mathrm{sinon\ emit(idm,tm+1)} \end{array}$$

Le principe de l'algorithme est que chaque processus envoie un message contenant son identifiant et un compteur initialisé à 0. A chaque fois qu'un processus reçoit un message il en examine l'identifiant. Si ce n'est pas le sien, il se contente de renvoyer le message en incrémentant le compteur. Si c'est le même que le sien, le processus sait que le message vient de lui et par conséquent, chacun des autres processus ont incrémenté le compteur. Par conséquent, il prend la valeur de ce compteur et le considère comme la valeur de la taille. Si des messages originaux contenaient des valeurs inexactes, cela n'a pas d'importance car la valeur de résultat est constamment remise à jour. Par conséquent, au bout d'un temps fini, tous les processus connaîtront la vraie valeur de la taille de l'anneau même s'il sont incapables de savoir à quel moment cette valeur sera bonne.

En réalité, cet algorithme n'est pas tout à fait auto-stabilisant. En effet, si une exécution commence avec tous les canaux vides, il n'y aura jamais de réception et donc d'émission de messages. On peut rajouter à l'algorithme une règle spontanée qui émettra ou supprimera des messages. Pour simplifier, nous supposerons que dans tout état initial et pour tout identifiant de processus i, il existe au moins un message dans le système dont l'identifiant est i.

Modélisation des états

Un anneau unidirectionnel peut être modélisé par un mot construit sur la grammaire suivante :

système	:	#élément système
élément	:	processus canal
processus	:	(identifiant,compteur)#
canal	:	message#
message	:	(identifiant,compteur),

C'est-à-dire que chaque canal est décrit comme une succession de messages et que canaux et processus sont placés dans l'ordre naturel induit par la topologie.

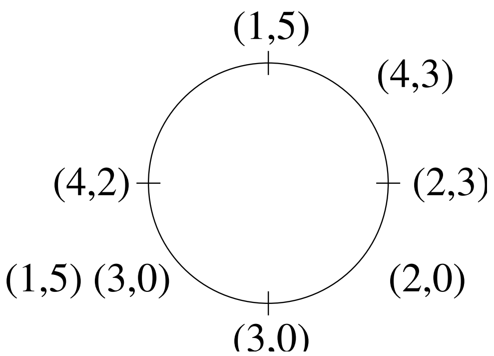

FIG. 7.1 – *Exemple de configuration*

Par exemple, la figure 7.1 présente un anneau dont la représentation est la suivante :

(4,2) # # (1,5) # (4,3) # (2,3) # (2,0) # (3,0) # (1,5) (3,0)

7.1.2 Modélisation des transitions

A partir de la modélisation précédente, chaque transition peut être modélisée par une réécriture de mots. Le système S suivant possède quatre règles modélisant notre algorithme.

R1: #X(id,cpt)#(id,a)#Y $\rightarrow$ #X#(id,cpt+1)#(id,0)Y#...#
R2: #X(id2,cpt2)#(id,a)#Y $\rightarrow$ #X#(id,a)#(id2,cpt2)Y#...#
R3: #Y#...#X(id,cpt)#(id,a)# $\rightarrow$ #(id,0)Y#...#X#(id,cpt+1)#
R4: #Y#...#X(id2,cpt2)#(id,a)# $\rightarrow$ #(id2,cpt2+1)Y#...#X#(id,a)#

Les règles R1 et R2 sont des règles de $Middle_S$ et R3 et R4 sont des règles de $Bottom_S$. Le système S sera appelé le système de réécriture concret. Ses règles sont centrées sur des messages. Nous appellerons message affecté par la transition le message noté (id,cpt) dans R1 et R3 et les messages notés (id2,cpt2) dans R2 et R4.

7.1.3 Système abstrait

La méthode de preuve définie dans le chapitre 3 ne suffit pas à prouver un tel algorithme car le déplacement des messages impose une instanciation des processus qui est trop contraignante pour obtenir des preuves indépendantes de la taille du système. Pour cette raison, nous allons définir un niveau d'abstraction dans lequel il sera possible de faire la preuve. Pour cela, nous allons focaliser les exécutions sur un message particulier par une abstraction.

Définition 69 (Système abstrait) *On appellera système de réécriture abstrait basé sur le message m, un système de réécriture dont les mots ne sont composés que de ce message et de son plus proche voisin processus gauche. On le notera A(m) (ou plus simplement A en supposant m quelconque).*

Notation 3 *Un mot u de A(m) est de la forme* $\#(p_i, p_{cpt})(m_i, m_{cpt})\#$.
p_i *et* p_{cpt} *représentent respectivement l'identifiant et la variable de taille du processus le plus à gauche de m.*
m_i *et* m_{cpt} *représentent respectivement l'identifiant et le compteur du message m. On note process(u) (respectivement message(u)) la partie gauche (respectivement droite) du mot.*
On note cpt(process(u)) la valeur de la variable du processus et id(process(u)) son identifiant. On définit de même cpt(message(u)) et id(message(u)). Enfin on note - une valeur indéfinie c'est-à-dire que la valeur n'a pas d'importance vis à vis de l'application de la règle et qu'elle n'est pas modifiée.

Nous allons construire le système abstrait pour notre algorithme de taille d'anneau basé sur un message quelconque d'identifiant i.

A1: # (i-1,-)(i,I) # $\rightarrow$ # (i,I+1)(i,0) #
A2: # (j-1,-)(i,I) # $\rightarrow$ # (j,-)(i,I+1) # où j $\neq$ i
A3: # (N-1,-)(0,I) # $\rightarrow$ # (0,I+1)(0,0) #
A4: # (N-1,-)(i,I) # $\rightarrow$ # (0,-)(i,I+1) # où i $\neq$ 0

A3 et A4 sont des règles de Top_A et A1 et A2 sont des règles $Middle_A$.

On appellera $\alpha(m)$ la fonction qui à tout état de S fait correspondre l'état de A(m) correspondant.

Soit t une suite de réécritures concrètes : $t_1 \rightarrow_{s_1} t_2 \rightarrow_{s_2} t_3 \rightarrow_{s_3}$ ... où les s_i sont les règles de S qui sont appliquées.

Soit $\alpha(m)(t)$ la suite des réécritures abstraites basées sur m : $\alpha(m)(t_1) \rightarrow_{a_1} \alpha(m)(t_2) \rightarrow_{a_2} \alpha(m)(t_3) \rightarrow_{a_3}$ ... où les a_i sont les règles de A correspondant aux t_i ou l'identité.

Formellement, on définit l'abstraction de la façon suivante : si $t_i \rightarrow_{R_i} t_{i+1}$ dans t tel que le message affecté par la transition soit m, on a $\alpha(m)(t_1) \rightarrow_{A_1} \alpha(m)(t_2)$.

En revanche, si le message affecté n'est pas m, on a $\alpha(m)(t_1) = \alpha(m)(t_2)$.

Propriété 5 *Par construction, $\alpha(m)(t)$ existe. De plus, l'équité des communications garantit que chaque message sera affecté infiniment souvent. Par conséquent, si t est infinie, $\alpha(m)(t)$ est également infinie au sens où il n'existe pas de suffixe infini de transitions identité.*

7.1.4 Preuve de l'auto-stabilisation

Lemme 21 *le système de réécriture abstrait A est auto-stabilisant pour l'ensemble d'états légitimes L suivant : # (j,-) (i,(j-i)[N]) # , # (i,N)(i,0) # pour j $\neq$ i.*

Preuve Montrons les trois points préliminaires à la preuve de l'auto-stabilisation.

- Tout mot clos non élément de L est réductible par A : les règles couvrent tous les cas de figure quelles que soient les valeurs du message ou du processus gauche.
- L est clos via A :
  - # (i,N)(i,0) # se réécrit en # (j=i+1[N],-)(i,1) # via A2
    or $(j - i)[N] = i + 1 - i[N] = 1$ donc cet état est bien légitime.
  - # (i,N)(i,0) # se réécrit en # (0,-)(i,1) via A4 (si i=N-1 et j=0)
    or $0 - (N - 1)[N] = 1$ donc cet état est bien légitime.
  - # (j,-) (i,(j-i)[N]) # se réécrit en # (i,(j-i)[N]+1) (i,0) # via A1 si j=i-1 donc $(j - i)[N] + 1 = ((i - 1) - i)[N] + 1 = (N - 1) + 1 = N$ et par conséquent se réécrit en # (i,N)(i,0) # qui est bien un état légitime.

  – # (j,-) (i,(j-i)[N]) # se réécrit en # (j+1[N],-)(i,(j-i)+1)# via A2
  $j - i + 1[N] = j + 1 - i[N]$ donc cet état est bien légitime.
  – # (j,-) (i,(j-i)[N]) # se réécrit en # (i,N)(i,0) # si (i=0;j=i-1)
  via A3 qui est un état légitime.

  Par conséquent tout état légitime produit un état légitime et donc L est bien clos via A.

– $A{-}Top_A$ termine : par l'équité des communications, un message passe infiniment souvent par le processus le plus à gauche : donc toute dérivation infinie contient une infinité d'applications de règles de Top_A.

Comme dans le chapitre trois, nous partons des règles Top pour construire les graphes de dérivations nous obtenons les deux figures 7.2 et 7.3. Nous constatons que les deux figures contiennent une chaîne quasi-cyclique, mais cette chaîne ne contient que des états légitimes. Par conséquent, l'algorithme est bien auto-stabilisant pour L.
□

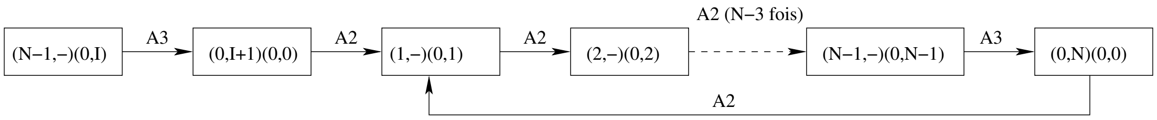

FIG. 7.2 – *Dérivation abstraite pour un message d'identifiant nul*

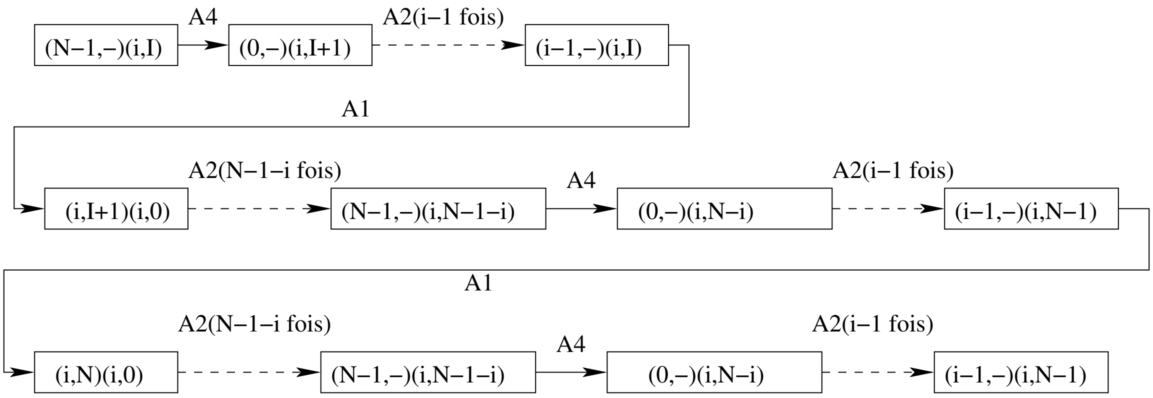

FIG. 7.3 – *Dérivation abstraite pour un message d'identifiant non nul*

Théorème 5 *L'algorithme est auto-stabilisant pour l'ensemble d'états légitimes : quel que soit p un processus, cpt(p)=N.*

Preuve Prouvons tout d'abord la convergence. Soit $\alpha(m_1)$, $\alpha(m_2)$, ... , $\alpha(m_m)$ l'ensemble des exécutions abstraites associées à tous les messages du système. Nous avons vu que ces systèmes sont auto-stabilisants. Par conséquent, il existe c_1, c_2, ..., c_m les premiers états légitimes abstraits atteints par chacune de ces exécutions. Soit c_i le dernier état au sens de l'exécution concrète, c'est-à-dire celui qui est causalement lié à tous les autres. Soit C_i l'état concret correspondant. C_i est un état légitime. En effet, nous savons qu'il existe au moins un message portant le numéro de chaque processus. Par conséquent, comme toutes les exécutions abstraites ont convergé, chaque processus p est tel que cpt(p)=N, ce qui montre la convergence. Montrons maintenant la correction. La correction est prouvée de manière triviale par la composition des exécutions abstraites : toute action concrète correspond à une action abstraite qui ne modifie la valeur d'aucun processus. $\square$

7.1.5 Conclusion

Bien que cette technique d'abstraction ne s'applique pas à tous les cas de passage de messages, elle permet d'appliquer la méthode d'une manière relativement facile. Une fois la preuve abstraite effectuée, la preuve concrète n'est plus qu'une simple composition d'états ou d'exécutions.

7.2 Autres Topologies

Dans le chapitre 3, nous n'avons considéré que des topologies linéaires. En effet, celles-ci s'appliquent parfaitement à la méthode par la structure de mots qu'elles engendrent. Toutefois, nous allons montrer dans cette section qu'il est possible de faire des preuves par réécritures sur d'autres topologies en utilisant la technique de l'abstraction. Dans cette section, nous présentons une abstraction sur une topologie de grille.

Une grille est une généralisation d'une chaîne en deux dimensions. C'est une topologie où chaque noeud possède quatre voisins (sauf s'ils sont sur les bords). De même que l'anneau est une chaîne dont les deux extrémités sont reliées par un lien, un tore est une grille dont les noeuds qui sont situés sur les bords sont connectés aux extrémités correspondantes. La figure 7.4 donne un exemple de tore.

Le problème de ce type de topologie par rapport à notre méthode de preuves est que les systèmes de réécriture que nous devons considérer ne sont pas linéaires gauche, ce qui est une hypothèse de base de l'un des lemmes les plus importants (celui qui fait correspondre tout cycle clos à une chaîne du premier ordre). Par conséquent, nous ne pouvons pas appliquer la méthode directement. Néanmoins, il existe des cas où il est possible d'abstraire le comportement individuel des processus et de montrer la convergence de l'algorithme. Nous allons présenter cette abstraction sur un problème tiré de la littérature.

Le problème de l'appariement maximal auto-stabilisant à été décrit dans [HH92].

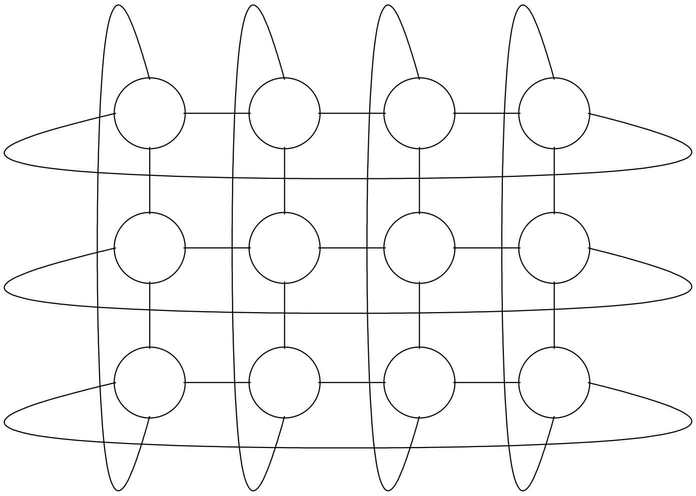

FIG. 7.4 – *Topologie de tore*

Le but de cet algorithme est d'apparier les processus deux à deux de façon maximale. Sur une grille, l'alphabet de chaque processus est constitué de cinq lettres : $\Sigma = \{\leftarrow, \rightarrow, \uparrow, \downarrow, \perp\}$. Ces lettres représentent un pointeur qui désigne le voisin avec lequel le processus croit être apparié. Nous appliquons l'algorithme sur un tore anonyme. Celui-ci est constitué de douze règles :

R1:	$\rightarrow$	$\perp$	$\Rightarrow$	$\rightarrow$	$\leftarrow$
R2:	$\perp$	$\perp$	$\Rightarrow$	$\rightarrow$	$\perp$
R3:	$\rightarrow$	$\{\uparrow\rightarrow\downarrow\}$	$\Rightarrow$	$\perp$	$\{\uparrow\rightarrow\downarrow\}$

A une rotation d'un multiple de 90 degrés près.

Les états légitimes sont tels qu'un processus doit être : soit apparié, c'est-à-dire qu'il pointe vers un voisin qui pointe sur lui; soit seul, c'est-à-dire dans l'état $\perp$ à condition que tous ses voisins soient appariés.

Pour prouver cet algorithme, nous définissons quatre schémas centrés sur un processus quelconque :

- L1: Le processus a la valeur $\leftarrow$, les valeurs des autres processus sont quelconques.
- L2 : Le processus a la valeur $\perp$, les valeurs des autres processus sont quelconques.
- L3 : Le processus a la valeur $\rightarrow$ et son voisin de droite a la valeur $\perp$, les valeurs des autres processus sont quelconques.
- L4 : Le processus a la valeur $\rightarrow$ et son voisin de droite a la valeur $\leftarrow$, les valeurs des autres processus sont quelconques.

Calculons le graphe de dérivations, nous obtenons :

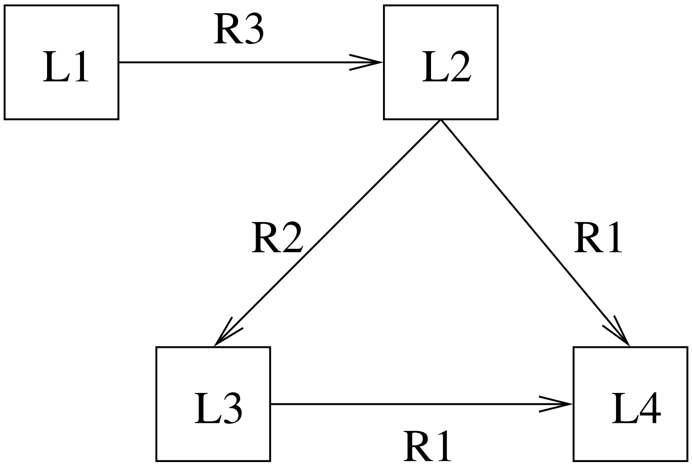

FIG. 7.5 – *Graphe de dérivation pour l'appariement maximal*

Supposons qu'il existe un cycle infini, cela implique qu'un processus doit changer infiniment de valeur. Nous pouvons supposer sans perte de généralité que le processus commence dans l'état $\leftarrow$ ou $\perp$ car le tore est non orienté. Comme le graphe ne contient pas de cycle, cela contredit le fait qu'un processus puisse changer infiniment souvent de valeur et donc l'algorithme est auto-stabilisant.

L'exemple de la grille n'influe que sur la construction de l'alphabet et du système concret. Dans ce cadre, nous aurions pu prendre toute topologie de degré fini. Pour les topologies non régulières, il suffit de considérer la topologie régulière de degré égal au plus grand degré de la topologie originale. La preuve ainsi construite est un sur-ensemble de la preuve réelle. Les particularités des topologies peuvent être utilisées comme meta-arguments dans la justification des cycles non infinis.

7.3 Les perspectives de Poulet

Le programme Poulet a été conçu dans une optique entièrement objet. En effet, les automates utilisés sont encapsulés dans une couche objet qui les abstrait vis-à-vis du programme principal. Cette propriété permet d'étendre à volonté le type d'objets manipulés. En particulier, les notions d'abstractions présentées précédemment sont facilement intégrables. Une autre perspective, tant au niveau de Poulet que de la méthode théorique de preuves est de considérer des topologies d'arbres. En effet, les arbres présentent des propriétés intéressantes au niveau des systèmes qui les manipulent. En particulier, les systèmes correspondants restent linéaires gauche, réguliers et nous espérons pouvoir exprimer des résultats similaires pour ces topologies. L'intégration des arbres dans Poulet devrait être quasiment automatique : il suffit d'écrire les fonctions de transformation des arbres en expressions régulières et d'appliquer les méthodes telles quelles. L'intérêt d'un tel travail serait d'augmenter le nombre d'algorithmes réels sur lesquels la méthode s'applique et donc de rendre Poulet plus intéressant en tant qu'outil de preuves.

Bibliographie

[ADGFT01] M. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Stable leader election. *15th International Symposium on Distributed Computing (DISC'01)*, 2001.

[AN99] A. Arora and M. Nesterenko. Stabilization-preserving atomicity refinement. *DISC'99*, 1999.

[Ang80] D. Angluin. Local and global properties in networks of processors. *Proc. Symp. on Foundation of Computer Science (FOCS'79)*, pages 218–223, 1980.

[Ang87] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation, 75*, pages 87–106, 1987.

[Arn92] A. Arnold. *Système de transitions finis et sémantique des processus communiquants.* MASSON, 1992.

[AS90] B. Awerbuch and M. Saks. A dining philosophers algorithm with polynomial response time. *31st Annual Symposium on Foundations of Computer Science*, volume I:65–74, october 1990.

[AS98] G.H. Antonoiu and P.K. Srimani. A self-stabilizing distributed algorithm for minimal spanning tree problem in a symetric graph. *Computers and Mathematics with Applications 35(10)*, pages 15–23, 1998.

[AS99] G.H. Antonoiu and P.K. Srimani. Mutual exclusion between neighboring nodes in an arbitrary system graph tree that stabilizes using read/write atomicity. *Europar'99, Parallel Processing, Proceedings LNCS 1685*, pages 823–830, 1999.

[AW98] H. Attiya and J. Welch. *Distributed Computing : Fundamentals, Simulations and Advanced Topics.* The McGraw-Hill Companies, 1998.

[BBFM01] J. Beauquier, B. Berard, L. Fribourg, and F. Magniette. Proving convergence of self-stabilizing systems using first-order rewriting and regular languages. *Distributed Computing 14(2)*, pages 83–95, 2001.

[BD95] J. Beauquier and O. Debas. An optimal self-stabilizing algorithm for mutual exclusion on uniform bidirectional rings. *Proc. 2nd Workshop on Self-Stabilizing Systems, Las Vegas*, pages 226–239, 1995.

[BDGM00] J. Beauquier, A. Datta, M. Gradinariu, and F. Magniette. Self-stabilizing local mutual exclusion and daemon refinement. *DISC'2000*, pages 233–237, 2000.

[BDGM02] J. Beauquier, A. Datta, M. Gradinariu, and F. Magniette. Self-stabilizing local mutual exclusion and daemon refinement. *Chicago Journal of Theoritical Computer Science*, page à paraître, 2002.

[BG89] V. Barbosa and E. Gafni. Concurrency in heavily loaded neighborhood-constrained systems. *Transactions on Programming Languages and Systems Vol 11 Num 4*, pages 562–584, 1989.

[BGJ99] J. Beauquier, M. Gradinariu, and C. Johnen. Randomized self-stabilizing and space optimal leader election under arbitrary scheduler on rings. *Tech. Rep. 99-1225 Universite Paris Sud*, 1999.

[BGM93] J. Burns, M. Gouda, and R. Miller. Stabilization and pseudo-stabilization. *Distributed Computing Vol 7 Num 1*, pages 35–42, 1993.

[CHT96] T. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM 43(4)*, pages 685–722, 1996.

[CM84] M. Chandy and J. Misra. The drinking philosophers problem. *ACM Transations on Programming Languages and Systems*, pages 6(4):632–646, october 1984.

[Cou02] Pierre Courtieu. Proving self-stabilization with a proof assistant. *Proceedings of FMPPTA'2002, IEEE CS Press*, page à paraître, 2002.

[Der81] N. Dershowitz. Termination of linear rewriting systems. *Proceedings of ICALP, LNCS 115*, pages 448–458, 1981.

[Dij74] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *ACM 17*, pages 643–644, 1974.

[DIM93] S. Dolev, A. Israeli, and S. Moran. Self-stabilizing of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7:3–16, 1993.

[DMV94] L. Dupont, L. Miclet, and E. Vidal. What is the search space of the regular inference? *Proceedings of the second colloquium on grammatical inference (ICGI'94)*, pages 25–37, 1994.

[Dol00] S. Dolev. *Self-Stabilization*. The MIT Press, 2000.

[Dup96] L. Dupont. Incremental regular inference. *Proceedings of the third ICGI'96, Lecture notes in artificial intelligence 1147*, pages 222–237, 1996.

[Eij] J. Eijndhoven. graphplace : programme de placement de graphe. *Eindhoven University of Technology, disponible sur ftp.es.ele.tue.nl/pub/users/jos/graphplace.tar.gz.*

[Fay79] M. Fay. First-order unification in an equational theory. *Proc. 4th Workshop on Automated Deduction, Austin, Texas*, pages 161–167, 1979.

[FLP85] M. Fischer, N. Lynch, and M. Patterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM 32(2)*, pages 374–382, 1985.

[GH91] M. Gouda and T. Herman. Adaptative programming. *IEEE TSE 17*, pages 911–921, 1991.

[GH97] M. Gouda and F. Hadix. The linear alternator. *Proceedings of the third workshop on self-stabilizing systems (WSS-97), International Informatics Series 7, Carleton University Press*, pages 31–47, 1997.

[GH99] M. Gouda and F. Hadix. The alternator. *In Proceedings of the Third Workshop on Self-Stabilizing Systems (published in association with ICDCS99 The 19th IEEE International Conference on Distributed Computing Systems)*, pages 48–53, 1999.

[Gho93] S. Ghosh. An alternative solution to a problem on self-stabilization. *ACM TOPLAS 15:4*, pages 735–742, 1993.

[GM91] M. Gouda and N. Multari. Stabilizing communication protocols. *IEEE Transactions on Computers 40(4)*, pages 448–458, 1991.

[GMC+91] C. Giles, C. Miller, D. Chen, G. Sun, and Y. Lee. Learning and extracting finite state automata with second order recurrent neural networks. *Neural Computation 4(3)*, pages 393–405, 1991.

[Gol67] E.M. Gold. Language identification in the limit. *Information and control 10(5)*, pages 447–474, 1967.

[Gou87] M. Gouda. The stabilizing philosopher: asymmetry by memory and by action. *Tech Rep TR-87-12, Univesity of Texas at Austin*, 1987.

[Gou95] M. G. Gouda. The triumph and tribulation of system stabilization. *Lecture Notes in Computer Science Vol 972, Springer-Verlag*, pages 1–18, 1995.

[Her91] T. Herman. Adaptativity through distributed convergence. *PhD thesis, University of Texas at Austin, Departement of Computer Science*, 1991.

[HH92] S.C. Hsu and S.T. Huang. A self-stabilizing algorithm for maximal matching. *Information processing letters*, pages 77–81, 1992.

[Hoe94] J-H. Hoepman. Uniform deterministic self-stabilizing ring-orientation on odd-length rings. *Proc. 8th Workshop on Distributed Algorithms, LNCS 857*, pages 265–279, 1994.

[HP89] D. Hoover and J. Poole. A distributed self-stabilizing solution for the dining philosophers problem. *Information Processing Letter 41*, pages 209–213, 1989.

[Hua00] Shing-Tsaan Huang. The fuzzy philoshophers. In *IPDPS Workshops*, pages 130–136, 2000.

[Hul80] J-M. Hullot. Canonical forms and unification. *Proceedings of 5th conf. on Automated Deduction, LNCS*, pages 318–334, 1980.

[JADT99] C. Johnen, L. O. Alima, A. K. Datta, and S. Tixeuil. Self-stabilizing neighborhood synchronizer in tree networks. In *Proceedings of the Nineteenth International Conference on Distributed Computing Systems (ICDCS'99)*, pages 487–494, 1999.

[Kes88] JLW. Kessels. An exercise in proving self-stabilization with a variant function. *Information Processing Letters,29*, pages 39–42, 1988.

[Lyn96] N Lynch. Distributed algorithms. *Morgan Kaufmann*, 1996.

[Mak77] G.S. Makanin. The problem of solvability of equations in a free semigroup. *Matematiceskij Sbornik*, pages 147–236, 1977.

[MN97] M. Mizuno and M. Nesterenko. A transformation of self-stabilizing serial model programs for asynchronous parallel computing environments. *Information Processing Letter 66*, pages 285–290, 1997.

[Par98] R. Parekh. Constructive learning : Inducing grammars and neural networks. *PhD Thesis*, 1998.

[Pet98] F. Petit. Efficacité et simplicité dans les algorithmes distribués auto-stabilisants de parcours en profondeur de jeton. *PhD Thesis, Laria, Université de Picardie Jules Verne, France*, 1998.

[PH93] R. Parekh and V.G. Honavar. Efficient learning of regular languages using teacher supplied positive examples and learner generated queries. *Proceedings of the Fifth UNB conference on AI*, pages 195–203, 1993.

[PH96] R. Parekh and V.G. Honavar. An incremental interactive algorithm for regular grammar inference. *Proceedings of the Third ICGI-96, Lecture notes in artificial intelligence 1147*, pages 238–250, 1996.

[Rus00] J. Rushby. Theorem proving with verification. *Proceedings of Movep'2k, Nantes*, pages 71–84, 2000.

[Shn93] M. Shneider. Self-stabilisation. *ACM Computing Surveys 25(1)*, pages 45–57, 1993.

[Sla74] J.R. Slagle. Automated theorem-proving for theories with simplifiers, commutativity, and associativity. *J. ACM 21:4*, pages 622–642, 1974.

[Tel94] Gerard Tel. Introduction to distributed algorithms. *Cambridge University Press*, 1994.

[Tho68] K. Thompson. Regular expression search algorithm. *Comm. ACM, 11(6)*, pages 419–422, 1968.

[Tix00] S. Tixeuil. Auto-stabilisation efficace. *PhD Thesis, LRI Université d'Orsay Paris-Sud, France*, 2000.

[Var97] G. Varghese. Compositional proofs of self-stabilizing protocols. *Proceedings of the third Workshop on Self-stabilizing Systems*, pages 80–94, 1997.

[VG00] M. Vasiliu-Gradinariu. Modélisation, vérification et raffinement des algorithmes auto-stabilisants. *PhD Thesis, LRI Université d'Orsay Paris-Sud, France*, 2000.

[Vil98] V. Villain. Mémoire en vue de l'obtention de l'habilitation à diriger des recherches en informatique. *Technical Report RR98-17,LaRIA, Université de Picardie Jules Verne*, 1998.