Publications du **La**boratoire de
Combinatoire et d'
Informatique
Mathématique

1

Edité par S. Brlek

Parallélisme: modèles et complexité

ACFAS 89

57e Congrès du 15-19 mai 1989
de l'Association Canadienne-Française pour
l'Avancement des Sciences.
UQAM, Montréal , Québec , Canada.
Actes du Colloque.

Département de mathématiques et d'informatique

**Centre de recherche
informatique de Montréal**

Université du Québec à Montréal

Edité par:

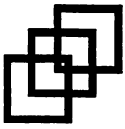
Srecko Brlek
LACIM
Université du Québec à Montréal
C.P. 8888, Succ. A
Montréal, Qc.
Canada H3C 3P8 .

Le colloque sur le thème " Parallélisme: modèles et complexité " a été tenu à Montréal du 15 au 19 mai 1989 dans le cadre du 57e Congrès de l'Association Canadienne-Française pour l'Avancement des Sciences.

Les organisateurs ont bénéficié du support généreux des organismes suivants:

Acfas
Association canadienne-française
pour l'avancement des sciences

Centre de recherche informatique de Montréal
1550, Boul. de Maisonneuve O. #1000
Montréal, Québec, H3G 1N2



FONDATION DE L' UQAM

ISBN 2-89276-077-1 LACIM Montréal

Le présent numéro a été édité dans le cadre d'une collaboration spéciale entre le CRIM et le LACIM.

Laboratoire de combinatoire et d'informatique mathématique
Département de mathématiques et d'informatique
Université du Québec à Montréal
C.P. 8888, Succ. A
Montréal, Qc.
Canada H3C 3P8

ACFAS 89

Montréal

57e Congrès
15-19 Mai 1989

Association Canadienne-Française
pour l'Avancement des Sciences

Colloque:

1. Parallélisme: modèles et complexité

16-17 Mai 1989
Université du Québec à Montréal

Avec le support de

Centre de recherche informatique de Montréal

Fondation UQAM

Organisateurs

F. Bergeron, UQAM; *S. Brlek*, UQAM;

PROGRAMME

1. Parallélisme: modèles et complexité

Mardi 16 Mai

Premiere session: Srecko Brlek, président.

9:30 **Conférencier invité:**
Robert Cori, Université de Bordeaux I, France.
Monoïdes partiellement commutatifs et réseaux.

10:30 Roman König, Université d'Erlangen, RFA.
Cross-sections for free partially commutative monoids.

11:00 Christian Choffrut, Université de Rouen [France] et Université de Waterloo [Canada]
Représentation des monoïdes de traces par des matrices de dimension 2 à coefficients entiers.

11:30 Benoit Tremblay, UQAM et INRS Telecom.
Quelques opérations sur les automates asynchrones.

12:00 **Diner** [Déjeuner pour l'Hexagone]

Deuxième session: Wilfried Probst, président.

14:00 André Arnold, Université de Bordeaux I, France.
MEC: a System for Constructing and Analyzing Transition Systems.

14:30 Halima Mounji, UQAM.
Etude comparative de la concurrence entre le langage VHDL et le langage LOTOS.

15:00 **Pause café.**

15:30 Edward Valentine, Université Concordia, Canada.
Processeur parallèle pour les systèmes experts et les algorithmes de règles.

16:00 Robert Proulx, UQAM
Nouveau modèle parallèle appliqué au oproblème de catégorisation de l'information.

16:30 Gaetan Hains, CRIM.
Mémoires partagées et réseaux systoliques: algorithmes élégants et/ou efficaces.

Mercredi 17 Mai

Troisième session: François Bergeron, président.

9:30 **Conférencier invité:**
Denis Thérien, Université McGill, Canada.
Théorie des automates et complexité.

10:30 Jean Eric Pin, LITP, France.
Logique temporelle et automates finis.

11:00 Michel Billaud, Université de Bordeaux I, France.
Réécriture de graphes et algorithmique distribuée.

11:30 Serge Dulucq, Université de Bordeaux I, France.
Complexité moyenne de l'algorithme de Naïmi-Trehel.

12:00 Dominique Gouyou-Beauchamps, Université de Paris-Sud, France
Tableaux de Havender standards.

12:30 **Diner** [re-Déjeuner pour l'Hexagone]

Lieu

Le colloque aura lieu à l'UQAM, dans les Pavillons Judith-Jasmin, 405, rue Sainte-Catherine Est et Hubert-Aquin, 1255, rue Saint-Denis, à Montréal.

Inscription

L'inscription ainsi que la distribution des documents aux personnes préinscrites se tiendra sur la "Grande Place", au niveau métro du pavillon Judith-Jasmin. Les frais d'inscription sont les suivants: 95 $ [étudiants:25 $], avant le 1er avril 1989; 120 $ [étudiants:30 $], après le 1er avril et sur place. Toutes les personnes préinscrites recevront un programme préliminaire au cours du mois d'avril 1989 et un reçu pour les frais d'inscription. Utilisez le formulaire ci-joint pour l'inscription.

Hébergement

Des prix réduits ont été négociés avec certains hotels situés à distance de marche de l'université. Une liste sera envoyée aux congressistes préinscrits, qui se chargeront eux-mêmes de leur réservation. Nous recommandons de réserver le plus tôt possible.

TABLE DES MATIERES

ESTAMPILLAGE BORNE ET AUTOMATES ASYNCHRONES

Robert CORI
LaBRI, Université Bordeaux I
Unité associée au CNRS
351, Cours de la Libération
33405 TALENCE Cedex - FRANCE -

---oOo---

Dans un système distribué des messages circulent entre les processeurs ; afin de pouvoir ordonner ces messages suivant leur moment d'émission, une estampille est généralement utilisée (voir par exemple [5]). Cette estampille, attribuée de façon centralisée et une fois pour toute à un message, consiste souvent en la valeur d'un compteur qui est incrémenté après chaque attribution. Ce procédé conduit à une infinité d'estampillages possibles. Récemment A. Israeli et Ming Li [4] se sont intéressés à un problème voisin, il s'agit de numérotation de processus. Ils ont montré que si l'on suppose le nombre de processus vivants à un instant donné borné par un entier k alors le nombre d'estampilles permettant de les ordonner deux à deux pouvait lui aussi être borné par un nombre voisin de 2^k. Ils utilisent une construction faisant intervenir des graphes de tournoi [6] et un résultat de P. Erdös [3].

Un problème de boites aux lettres et de messages a été considéré [1] dans le cadre d'une propriété d'une famille d'automates appelés automates asynchrones et introduite par W. Zielonka [7]. Dans ce qui suit nous montrons comment le résultat de A. Israeli et Ming Li peut être utilisé pour trouver une autre solution au problème posé en [1], cette nouvelle solution bien que plus simple à exposer est moins efficace en taille des messages que celle donnée en [1]. Ainsi après avoir présenté les résultats sur les numérotation de processus, nous montrons comment ceux-ci peuvent permettre de résoudre le problème des boites aux lettres. L'expression de ce

dernier problème en terme de recherche d'automate asynchrones [7] particulier est enfin donnée.

Notons que les techniques présentées ici ne permettent pas de résoudre plus simplement les questions beaucoup plus générales examinées et résolues dans [2].

1 - ESTAMPILLAGE DE PROCESSUS DANS UN SYSTEME CENTRALISE

Dans ce premier paragraphe nous rappelons et précisons des résultats, dus à A. Israeli et Ming Li [5].

Il s'agit de résoudre le problème suivant : dans un système, des processus naissent et meurent (ou bien plutôt se terminent). On suppose que deux processus ne naissent jamais au même instant et que le nombre de processus vivants à un instant donné est inférieur ou égal à une constante p. Lors de la naissance d'un processus un numéro lui est affecté, il conservera ce numéro durant toute sa durée de vie, on dit qu'il est estampillé ; et le numéro est appelé estampille. Dans toute la suite, l'estampille sert uniquement à déterminer l'ordre de naissance des processus. Plus précisément étant données deux estampilles on veut pouvoir déterminer laquelle a été attribuée avant l'autre. Une façon simple de résoudre cette question est soit de disposer d'une horloge ou d'un compteur ; avec une horloge centralisée on peut estampiller les processus par l'heure de leur naissance, avec un compteur on leur affecte la valeur de ce compteur laquelle est incrémentée une unité immédiatement après chaque nouvel estampillage. Dans ces deux cas la solution utilise un ensemble infini de possibilités, A. Israeli et Ming Li ont proposé une méthode permettant d'affecter un nombre fini d'estampilles possibles, on réutilise ainsi les estampilles des processus qui sont morts. Une solution est trouvée sans difficulté lorsque p=2. On choisit alors les étiquettes dans l'ensemble {0,1,2}, lorsqu'un processus P nait, ou bien il est le seul processus vivant et on lui attribue l'étiquette 0, ou bien il y a déjà un autre processus Q vivant et on attribue à (P) l'étiquette de (Q) augmentée de 1 modulo (3).

On obtient alors un automate (déssiné sur la figure 1) où chaque état correspond aux processus vivants avec leurs étiquettes, les arcs en trait plein indiquent la naissance d'un processus et ceux en trait pointillé leur disparition. L'état noté Ø est l'état initial ou aucun processus n'est vivant.

La solution pour un nombre de processus quelconque p-1 est bien plus complexe, afin de l'exposer quelques notions de théorie des graphes sont nécessaires :

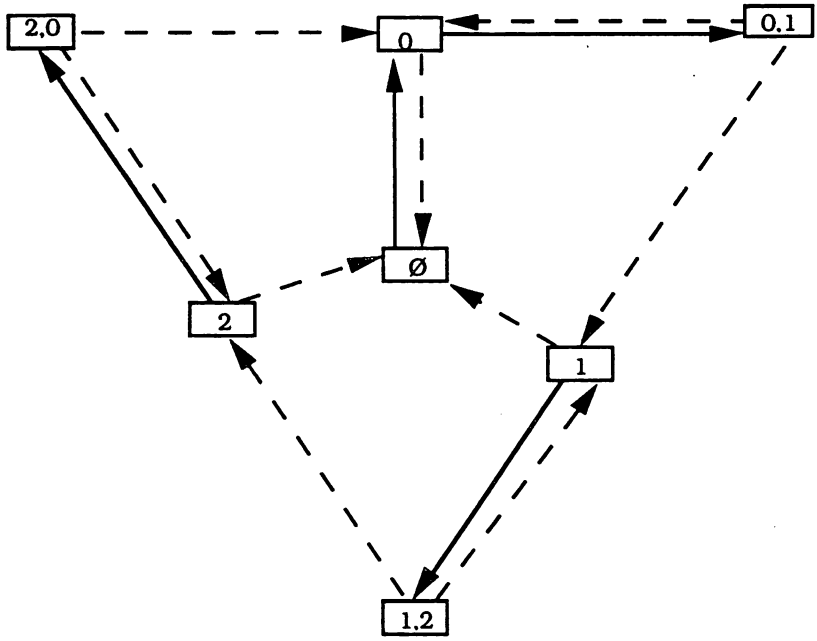


Figure 1

Un graphe orienté G est composé d'un ensemble S de sommets et d'un ensemble $E \subset S \times S$ d'arcs si $(x,y) \in E$ on dit que y est un successeur de x ; et on note $y \in \Gamma_G(x)$ les graphes considérés par la suite sont antisymétriques et sans boucle c'est-à-dire :

$$\forall x,\ (x,x) \notin E;\ (x,y) \in E \Rightarrow (y,x) \notin E.$$

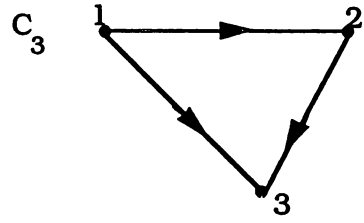
Pour tout sommet s on note $\Gamma_G(s)$ l'ensemble de ses successeurs, les <u>successeurs</u> $\Gamma_G(T)$ d'un ensemble T inclus dans S est constitué des sommets s_0 successeur de tous les sommets de T.

$$\Gamma_G(T) = \left(\bigcap_{t \in T} \Gamma_G(t) \right)$$

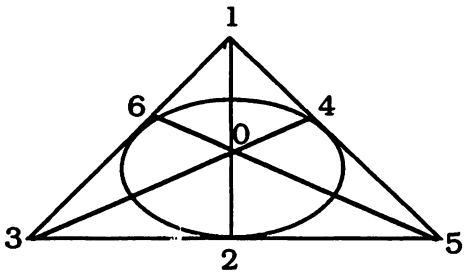
Une clique orientée ou sous ensemble transitif de G est une suite $s_1, s_2, \ldots s_k$ de S tel que pour tout i,j tels que $1 \leq i < j \leq k$ on ait $s_j \in \Gamma_G(s_i)$.

DEFINITION 1.1 - Un graphe orienté G = (X,E) est un <u>graphe d'estampillage d'ordre</u> p si toute clique orientée ayant un nombre de sommets inférieurs à p admet au moins un successeur.

Exemple 1.1 - Le graphe suivant est le plus petit système d'estampillage d'ordre 2.



Exemple 1.2 - Le graphe dont l'ensemble des sommets est {0,1,2,...6} et la fonction successeur est donnée par $\Gamma(0) = \{1,3,6\}$, $\Gamma(1)=\{2,3,5\}$, $\Gamma(2)=\{0,5,6\}$, $\Gamma(3)=\{2,46\}$, $\Gamma(4)=\{0,1,2\}$, $\Gamma(5)=\{0,3,4\}$, $\Gamma(6) = \{1,4,5\}$, constitue un système d'estampillage d'ordre 3. On peut remarquer que l'existence d'un successeur pour toute clique orientée d'ordre 2 revient à dire $\Gamma(i) \cap \Gamma(j) \neq \emptyset \ \forall i,j$, et de fait les $\Gamma(i)$ représentent les droites du plan de Fano, dont l'intersection est toujours égale à un point. De ce fait sur cet exemple tout ensemble à deux éléments constitue une clique et donc tout ensemble à deux éléments possède un successeur.



Un graphe d'estampillage G d'ordre p permet de résoudre le problème de numérotation des processus décrit plus haut en donnant aux processus des estampilles correspondant aux sommets du graphe ; la comparaison de l'ordre des naissances se fait à l'aide de l'arc joignant les sommets correspondants dans le graphe (x est plus récent que y si $x \in \Gamma_G(y)$) et l'estampillage d'un nouveau processus est fait à l'aide d'un successeur de l'ensemble des estampilles des processus vivants. On peut remarquer que cet ensemble est toujours transitif et, puisque le nombre de processus vivants est inférieur ou égal à p, la propriété du graphe d'estampillage permet alors d'assurer l'existence du successeur.

DEFINITION 1.2 - Soit deux graphes G = (S,E) et H = (T,F) le produit lexicographique G⊗H des deux graphes a pour ensemble de sommets S × T et pour ensemble d'arcs

$$G \otimes H = \{(s,t), (s',t') \mid (s,s') \in E \text{ ou } (s=s' \text{ et } (t,t') \in F)\}.$$

PROPOSITION 1.1 - Si G et H sont des graphes d'estampillage d'ordre p et q alors G⊗H est un graphe d'estampillage d'ordre p+q-1.

Preuve - Soit F $\{u_1,u_2,\ldots,u_m\}$ un sous ensemble transitif où $m<p+q-1$. Chaque u_i se met sous la forme $u_i=(s_i,t_i)$. Si le nombre des s_i distincts est inférieur à p alors le fait que G est un système d'estampillage d'ordre p permet de trouver s successeur de tous les s_i et en choisissant un $t \in T$ quelconque le sommet (s,t) de G⊗H est alors successeur des $u_1,\ldots,u_m$. Si le nombre des s_i distincts est supérieur ou égal à p alors pour chaque s_i le nombre de t tel que (s_i,t) soit dans la suite est inférieur à q c'est le cas en particulier pour s_m. Soit U l'ensemble des t tels que $(s_m,t) \in T$, U est un

ensemble transitif de H ayant moins de q élément il admet donc un successeur t* et (s_m,t^*) est successeur de tous les $u_1,\ldots,u_m$.

COROLLAIRE 1.2 - Il existe un graphe d'estampillage d'ordre 2j+1 ayant 7^j sommets et un graphe d'estampillage d'ordre 2j ayant 3.7^{j-1} sommets.

Ces graphes sont obtenus en effectuant des produits lexicographiques successifs des graphes , on peut noter que le nombre de sommets est une fonction exponentielle de l'ordre mais de fait on ne peut pas faire beaucoup mieux en raison de la proposition suivante.

PROPOSITION 1.2 - Si G=(S,E) est un graphe d'estampillage d'ordre p, s un sommet de G le graphe G_s ayant pour ensemble de sommets $\Gamma(s)$ et pour ensemble d'arcs $E \cap \Gamma(s) \times \Gamma(s)$ est un graphe d'estampillage d'ordre p-1.

Preuve - Si $s_1,s_2,\ldots,s_p$ (m < p-1) est un sous ensemble transitif de G_s alors $s,s_1,s_2,\ldots s_m$ est un sous ensemble transitif de G, il admet un successeur t du fait que G est un système d'estampillage d'ordre k. Clairement $t \in \Gamma(s)$ et est donc un successeur de $s_1,s_2,\ldots s_p$ dans G_s.

COROLLAIRE 1.4 - Le nombre de sommets d'un graphe d'estampillage d'ordre p est minoré par 2^p-1 .

Preuve - On procède par récurrence sur p, il est assez facile de voir que pour p=2 le nombre de sommets est au moins de 3. Si G est un système d'estampillage d'ordre p, la proposition 1.3 et l'hypothèse de récurrence impliquent que tout sommet admet au moins $2^{p-1}-1$ successeurs. Ainsi si N est le nombre de sommets le nombre d'arcs est au moins de $N(2^{p-1}-1)$; ceci implique

$$N(2^{p-1}-1) \leq \frac{N(N-1)}{2} \quad \text{et} \quad N \geq 2^p - 1.$$

2 - SYSTEME DE BOITES AUX LETTRES

Avec M. Latteux, Y. Roos, et E. Sopena nous avions examiné un problème concernant des processeurs partageant des mémoires communes. Il peut être présenté sous une forme imagée que l'on se propose de reprendre ici.

Un ensemble de p personnes utilisent un système de boites aux lettres pour échanger des messages. Chaque boite est accessible par deux usagers, pour tout couple d'usager il existe une boite et une seule à laquelle ils ont tous les deux accès. Chaque usager peut ainsi accéder à p-1 boites et le nombre total de boites est p(p-1)/2. En plus des messages échangés les utilisateurs doivent édicter une règle permettant à chaque usager accédant au système de connaître le nom de l'utilisateur ayant accédé immédiatement avant lui (il ne peut y avoir deux accès simultanés). Chaque usager connait pour chaque boite à laquelle il a accès le nom de la personne qui la partage avec lui. Lorsqu'il utilise le système il ouvre toutes les p-1 boites dont il possède l'accès examine leur contenu et, en fonction de ce contenu, détermine le nom de l'usager précédent et modifie les contenus en fonction du comportement à déterminer.

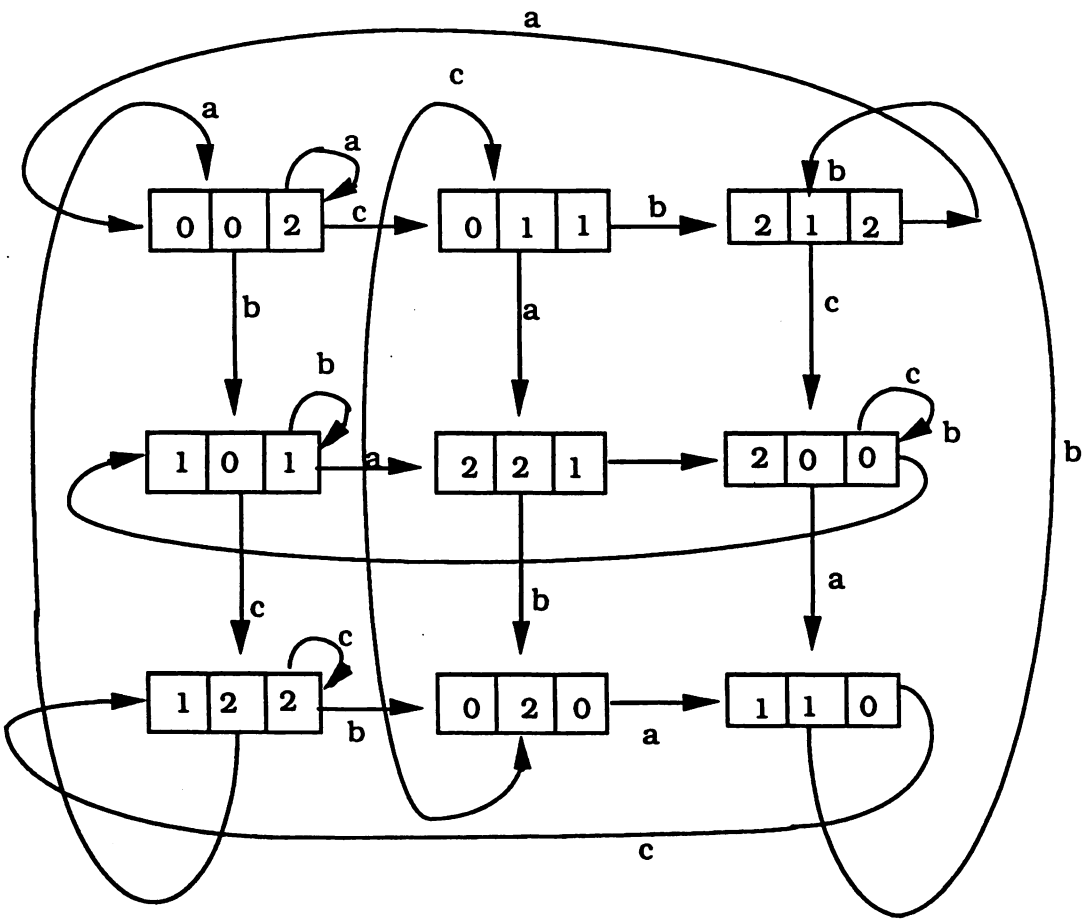
Là encore une solution consiste à indiquer son heure de passage dans toutes les boites auxquelles un usager a accès ; l'utilisateur peut déterminer son prédécesseur dans le système en déterminant la boite aux lettres qui contient l'indication de l'heure la plus récente et ce prédécesseur est lui-même dans le cas où toutes les boites contiennent la même heure.

Nous avions proposé une solution utilisant une information bornée en commençant par traiter le cas de trois utilisateurs A,B,C et de trois boites aux lettres accessibles par AB, AC et BC respectivement. Ces boites contiennent un même élément i (ce sont celles qui ont été accédées par le dernier utilisateur) et la troisième contient i-1 (mod 3). Lorsqu'un usager X ouvre ses deux boites deux situations se présentent

• les deux boites ont le même contenu, alors aucun autre usager que X n'a accédé au système depuis son dernier passage ; X laissera le contenu des boites tel quel.

• les deux boites ont pour contenu i et j ($i \neq j$) alors pour des raisons de symétrie on peut supposer $i=j+1$ (mod 3), le dernier usager est Y qui partage avec X la boite contenant i. L'action de X va consister à remplacer i et j par i+1 (mod 3).

Ce comportement peut être représenté par le diagramme suivant : c'est un automate dont chaque état est composé des 3 contenus des boites aux lettres AB, AC, et BC et où l'action des lettres a,b,c représente le comportement des usagers A,B,C lorsqu'ils accèdent au système. Il est remarquable de noter l'analogie avec le graphe C_3 système d'estampillage d'ordre 1.



Pour résoudre le problème des boites aux lettres avec k usagers, un système d'estampillage n'est pas tout à fait suffisant. Nous avons proposé un système utilisant 3^{k-2} messages possibles par boite [1].

Un graphe d'estampillage G d'ordre p permet aussi de résoudre le problème des boites aux lettres pour p+1 utilisateurs $x_1,\ldots,x_{p+1}$ donnons de façon intuitive les grandes lignes du protocole utilisé. Par la suite celui-ci sera précisé à l'aide d'automates asynchrones.

Le système se comporte comme si à chaque ouverture de boite par un usager x_i celui-ci faisait mourir un processus qu'il avait créé et comme s' il en créait un nouveau. Ainsi chacune des boites aux lettres contient deux étiquettes correspondant aux deux estampilles détenues par les derniers processus "créés" par chacun des utilisateurs de la boite. Plus précisément, lorsqu'un usager x_i ouvre ses boites il trouve dans chacune d'entre elles un couple d'estampille (u_i,u_j). L'une des composante, u_i, est la même dans toute les boites, c'est l'estampille qui lui appartient en propre ; x_i est alors capable de classer l'ensemble des usagers suivant l'ordre de leurs dernières interventions. Afin de permettre aux autres usagers de faire de même par la suite il retire l'étiquette u_i et la remplace par v_i successeur de l'ensemble des p autres estampilles $\{u_1,\ldots u_{i-1}, u_{i+1},\ldots u_{p+1}\}$ l'existence de ce successeur est assuré par la propriété des graphes d'estampillage.

3 - EXPRESSION DU PROBLEME DES BOITES AUX LETTRES EN TERMES D'AUTOMATES ASYNCHRONES

Rappelons la définition des automates asynchrones introduits par W. Zielonka [7].

Sont donnés un alphabet A, un ensemble Q d'états produit cartésien de n composantes,

$$Q = Q_1 \times Q_2 \ldots \times Q_n$$

un état initial $q^0 \in Q$ et un ensemble $F \subset Q$ d'états terminaux. A chaque lettre a est associé un sous ensemble de {1,2,...n} noté dom(a) et une application δ_a de $Q_{i_1} \times \ldots Q_{i_p}$ dans lui même où dom(a) = $\{i_1,\ldots i_p\}$. La fonction de transition globale Δ de Q dans Q est alors donnée par si $q'=\Delta(q,a)$ $\forall q \in Q$ $q'_i = q_i$ si $i \notin dom(a)$ et $q'_{i_1},\ldots q'_{i_p} = \delta_a(q_{i_1},\ldots q_{i_1})$.

Cette application s'étend classiquement en une application Δ de $Q \times A^*$ dans Q.

Pour chaque système de boites aux lettres pour les usagers $\{x_1,\ldots x_p\}$ on associe un automate asynchrone dont les composantes sont indexées par les paires d'usagers, ainsi les indices des composantes sont de la forme {i,j} avec $i \neq j$. L'alphabet $A = \{a_1,\ldots a_p\}$ correspond aux usages et l'application dom associe à chaque lettre $\underline{a}_i$ un sous ensemble composé de p-1 indices :

$$dom(a_i) = \{ \{1,i\}, \ldots, \{2,i\},\{i-1,i\}, \{i, i+1\},\ldots,\{i, p\} \}.$$

Le problème consiste à déterminer les $Q_{i,j}$ et les δ_{a_i} de façon à ce que les composantes suivant les {i,j} de $\Delta(q^0,f)$ permettent de déterminer de façon unique la dernière lettre de f. Pour cela étant donné un graphe d'estampillage G = (S,E) d'ordre p-1 on prend $Q_{i,j} = S \times S$ et la solution donnée plus haut s'exprime alors :

Pour tout $a \in A$: $\delta_a(\{v_1,v_i\}, \{v_2,v_i\} \ldots \{v_{i-1},v_i\}, \{v_i,v_{i+1}\}, \{v_i,vp\})$

$$= \{ \{v_1, v'_i\}, \{v_2, v'_i\} \quad \{v_{i-1}, v'_i\}, \{v'_i, v_{i+1}\} \quad \{v'_i, v_p\}\}$$

où $v'_i \in \Gamma_G(v_1, v_2, v_{i-1}, v_{i+1}, \ldots v_p)$.

References

[1] R. CORI, M. LATTEUX, Y. ROOS, E. SOPENA , 2-asynchronous automata. Theoret. Comp. Sci. 60 (1988) 93-102.

[2] R. CORI, Y. METIVIER , Approximation d'une trace, automates asynchrones et ordres des évènements dans un système réparti. Lecture Notes in Computer Science 317 (1988) 147-161.

[3] P. ERDÖS , On a problem of Graph theory. Math. Gazette 47 (1963) 220-223.

[4] S. ISRAELI, MING LI , Bounded Time-Stamps. Proc. 28th IEEE Symp on Foundations of Computer Science (1987) 371-382.

[5] L. LAMPORT , Time, clocks and the ordering of events in a distributed system. Comm. A.C.M. 21 (1978)558-565.

[6] J.W. MOON , Topics on tournaments, Holt, New-York, 1968.

[7] W. ZIELONKA , Notes on finite asynchronous automata, RAIRO Info. Theor. 21 (1987) 99-135.

Cross-sections for Free Partially Commutative Monoids

Extended abstract of a lecture given at the 57e Congrès
de l' ACFAS, Montréal, May 15 – May 19, 1989

Roman König

Institut für Mathematische Maschinen und Datenverarbeitung I, Universität Erlangen,
Martensstr. 3, D – 8520 Erlangen. e-mail: koenig@informatik.uni-erlangen.de

Let A be a finite set and $\binom{A}{2}$ the set of all two-element subsets of A. Every subset θ of $\binom{A}{2}$ defines a free partially commutative monoid by allowing two letters to commute if and only if $\{a,b\} \in \theta$. More precisely, the free partially commutative monoid over A with commutation relation θ is defined to be

$$M(A,\theta) = A^* / \{ab = ba \mid \{a,b\} \in \theta\},$$

where A^* is the free monoid over the set A with unit-element 1. Free partially commutative monoids have been introduced by CARTIER and FOATA [1] to represent certain combinatorial objects like flows and rearrangements. During the last ten years free partially commutative monoids also occurred in theoretical computer science as a model of concurrency and were studied by various authors [2, 5]. Since then elements of $M(A,\theta)$ usually are called traces, subsets of $M(A,\theta)$ are called trace-languages.

For graphs $G = (A,\theta)$ and $H = (B,\phi)$ we shall say that H is a *subgraph* of G, if $B \subseteq A$, and $\phi \subseteq \theta$. For every graph $G = (A,\theta)$ we have the (edge-) *complementary* graph $\overline{G} = (A,\overline{\theta})$, where $\overline{\theta} = \binom{A}{2} - \theta$. The corresponding monoid $M(\overline{G}) = M(A,\overline{\theta})$ will be denoted by $\overline{M}(A,\theta)$ as well.

We define for arbitrary graphs $G = (A,\theta)$ and $H = (B,\phi)$ their *direct sum* to be

$$G \oplus H = (A + B, \theta + \phi),$$

where "+" for sets means disjoint union, and the "*product*"

$$G \otimes H = \overline{\overline{G} \oplus \overline{H}}.$$

Then

$$M(G \oplus H) = M(G) * M(H) \qquad \text{(free product)}$$

and

$$M(G \otimes H) = M(G) \times M(H) \quad \text{(direct product)}.$$

A graph is called *indecomposable*, if it is connected and its complement is connected.

Besides $\oplus$ and $\otimes$, the following operations for graphs are convenient:

$$G \cup H = (A \cup B, \theta \cup \phi)$$

$$G \cap H = (A \cap B, \theta \cap \phi).$$

The set of all finite graphs can be considered as the free algebra of type (2,2,0) with the associative and commutative operations $\oplus$ and $\otimes$ and a common neutral element $\Phi = (\emptyset, \emptyset)$, generated by the set of indecomposable graphs over finite subsets of a countable set. On this algebra we have an involution $G \mapsto \overline{G}$ which is induced by an involution on the indecomposable graphs, and the laws $G \oplus H = \overline{\overline{G} \otimes \overline{H}}$ and $G \otimes H = \overline{\overline{G} \oplus \overline{H}}$.

There is a second normal form for every graph $G = (A, \theta)$, which we call the *canonical form* of G: Up to commutativity and associativity of $\cup$ and $\otimes$ and idempotency of $\cup$ every graph is uniquely determined by its maximal cliques, i.e. expressible as union of graphs of the form $a_1 \otimes a_2 \otimes \ldots \otimes a_m$ for some $m \in \mathbb{N}$, and $a_i \in A$ for $1 \leq i \leq m$. If $C_1, \ldots, C_n$ are the maximal cliques of G, then G can be written $G = (C_1, \binom{C_1}{2}) \cup \ldots \cup (C_n, \binom{C_n}{2})$. To simplify notation we write the canonical form of $G = C_1 \cup \ldots \cup C_n$.

The following theorem is fundamental for the studies of the combinatorial properties of free partially commutative monoids and implies the well-known representation theorems for free partially commutative monoids (see e.g. [2, 3, 5]).

Theorem

Let $G = (A, \theta)$ be a graph.

(1) If $G = G_1 \cup G_2 \cup \cdots \cup G_n$ is a representation of G as a union of some subgraphs of G, then $M(G)$ is isomorphic to the free product of the monoids $M(G_i)_{i=1,\ldots,n}$, amalgamated by the monoids $M(G_i \cap G_j)_{i,j=1,\ldots,n}$.

(2) If $\overline{G} = G_1 \cup G_2 \cup \ldots \cup G_n$ is a representation of $\overline{G}$ as a union of some subgraphs of $\overline{G}$, then $M(G)$ is isomorphic to the submonoid of $M(\overline{G_1}) \times \ldots \times M(\overline{G_n})$, generated by the elements of the form $\iota(a) = (a_1, a_2, \ldots, a_n)$, where $a \in A$ and a_i is a if $a \in G_i$ and 1 otherwise.

(3) Let $M(G)$ be a subdirect product of M_1 and M_2. Then $\overline{G} = \overline{H} \cup \overline{K}$ and $M(H) \cong M_1$ and $M(K) \cong M_2$.

(4) Let $M(G)$ be an amalgamation of $M_1 * M_2$ by some common submonoid M_3. Then $G = H \cup K$ and $M(H) \cong M_1$, $M(K) \cong M_2$, $M(H \cap K) \cong M_3$.

By a *cover by cliques* of the graph $G = (A, \theta)$ we mean a cover $A = C_1 \cup C_2 \cup \ldots \cup C_n$ of A such that $\theta = \binom{C_1}{2} \cup \binom{C_2}{2} \cup \ldots \cup \binom{C_n}{2}$. It is called *minimal*, if the number n is minimal.

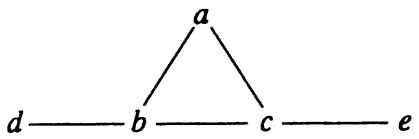
Let $M(A, \theta) \cong A^*/\rho$, where $\rho = \{ab = ba | \{a, b\} \in \theta\}$. A subset T of A^* is called a *cross-section* or a *transversal* for $M(A, \theta)$ if T contains exactly one element of every ρ-class. T is called *rational*, if T is a member of the class $Rat(A^*)$ of subsets of A^*, where for some arbitrary monoid M the class $Rat(M)$ of all rational subsets of M is defined by the following scheme:

- $\emptyset \in Rat(M), \{m\} \in Rat(M)$ for every $m \in M$
- $U, V \in Rat(M) \Rightarrow U \cdot V = \{uv \mid u \in U, v \in V\} \in Rat(M)$ and $U \cup V \in Rat(M)$
- $U \in Rat(M) \Rightarrow U^* = \bigcup_{n \geq 0} U_n \in Rat(M)$

and $U^0 = \{1\}, U^{n+1} = U \cdot U^n$, and $\emptyset^* = \{1\}$.

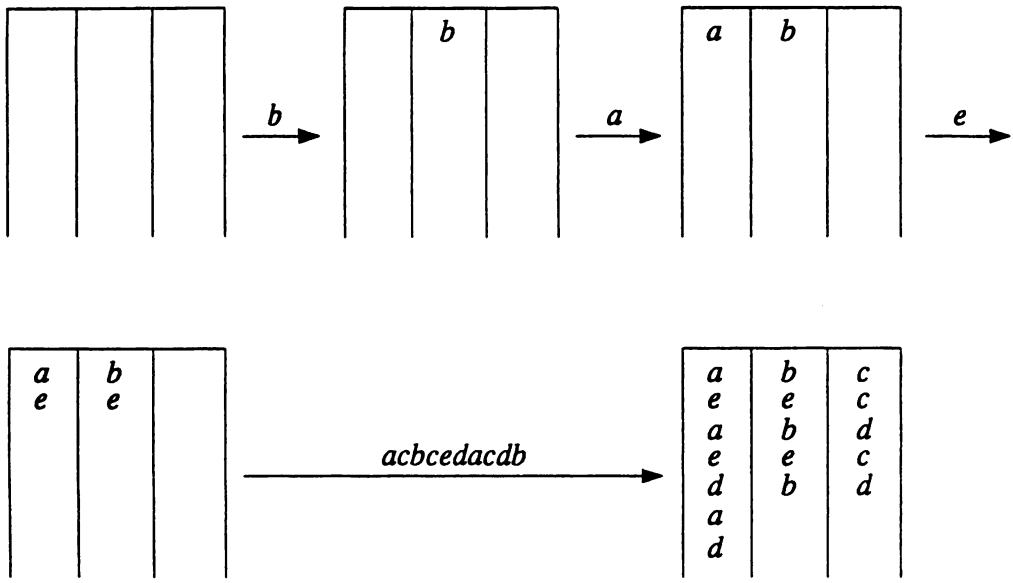
Let G be a graph with representation $\overline{G} = \overline{C_1} \cup \overline{C_2} \cup \ldots \cup \overline{C_n}$. According to the Theorem, $M(G)$ can be considered as the submonoid of the direct product of the monoids $C_1^*, C_2^*, \ldots, C_n^*$, generated by the $\iota(w)$, where $w \in A^*$ and thus is a rational submonoid of $C_1^* \times \ldots \times C_n^*$.

Let us consider the following example: Let Q be given by

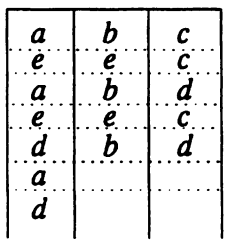


Then $\overline{Q} = (a \otimes d \otimes e) \cup (b \otimes e) \cup (c \otimes d)$ is the canonical form of $\overline{Q}$ and the word $w = baeacbcedacdb$ has $\iota(w) = (aeaedad, bebeb, ccdcd)$.

We can imagine the n-tuple $\iota(w)$ to be represented as a collection of pipelines, each representing one component. We start with empty pipes and begin reading in the first letter of w, in our example this is b, into every pipe which represents a component i such that $b \in C_i$:



Now we can read the contents of the pipeline according to the following picture. We say, a letter is visible, if it is in top position in every pipe which contains this letter.



For example, in the above picture the letters a, b, c are visible. A partial order on the vertex set of the graph with the property that every clique of the graph is a chain in the order is called an order on the graph. The rule "remove all the visible letters of the top slice and multiply them according to some given order of the graph to obtain some slice-product, and then multiply all the slice-products in the order of the slices" gives FOATA's normal form $F(w)$ of w relative to the order of the graph. Similarly, taking at every moment the visible letter which is minimal in the order of the graph gives the alphabetic normal form $L(w)$ of w. Note that this letter is unique, because all the letters visible at the same time belong to one clique. Finally, taking always the leftmost among all visible letters produces a third normal form, the so-called left-right normal form $LR(w)$ of w.

In our example, this means $F(w) = abc|ec|ab|e|bd|ac|d$ and $L(w) = abeabebccdacd$ for the oder $a < b < c, e < c, b < d$ on G, and $LR(w) = abeabeccdabcd$.

We recall that free partially commutative monoids can be considered as a model for nonsequential processes. The letters of the alphabet A are imagined as elementary actions and commutation of two letters means that the composition of the corresponding actions is independent of their order. FOATA's normal form for some given $w \in A^*$, together with its factorization $F(w) = u_1|u_2|\ldots|u_t$ contains the following information:

- t is the minimal number of steps being necessary to perform w if arbitrarily high parallelity is allowed.
- The necessary number of processes working in parallel to obtain the performance of w in t steps is the maximal length of a factor in FOATA's normal form.

It can be easily deduced that the set $T_{LR}(G)$ is rational. Let $\overline{G} = L_1 \cup L_2 \cup \ldots \cup L_n$ be a representation of $\overline{G}$ as a union of cliques, i.e. $L_1, L_2, \ldots, L_n$ is a cover of G by anticliques. For every $a \in A$ define $i(a) = \min\{k \mid a \in L_k\}$. Then the reflexive and transitive closure of the relation "$<$", defined by

$$\begin{aligned} a < b &\Leftrightarrow i(a) < i(b) \text{ and } \{a,b\} \in \theta \\ &\Leftrightarrow i(a) < i(b) \text{ and } \forall j\ (1 \le j \le n)\ \{a,b\} \notin L_j \end{aligned}$$

is a partial order $\le$ on the set A of vertices of G with the property that every clique of G is a chain in the order, because $a < b$ and $b < c$ imply $a \neq c$. Hence G is an ordered graph. Note that we also use the character "$<$" for the transitive closure of this relation. We call this partial order the *order induced by the representation* of $\overline{G}$ or induced by the representation of $M(A, \theta)$. If we take on G the order induced by the canonical form of $\overline{G}$, then the alphabetic normal form according to this order coincides with the LR normal form. This proves

Property

$$w \in T_{LR}(G) \Leftrightarrow (w = xavby,\ x, v, y \in A^*, \{a,b\} \in \theta, a > b \Rightarrow v \notin A_b^*);$$

$$T_{LR}(G) = A^* - \bigcup_{a>b,\ \{a,b\}\in\theta} A^* a A_b^* b A^*.$$

From this characterization we can again derive that $T_{LR}(G)$ is a rational subset of A^*

Every system T of normal forms of $M(A, \theta)$ can be considered as a monoid which is isomorphic to $M(A, \theta)$ if we define the product of u and v in the monoid T as the corresponding normal form of uv. Considered as monoids, the sets

T_F, T_L, and T_{LR} are thus isomorphic. Nevertheless the combinatorial structures of the sets T_F, T_L, and T_{LR} are very different. For example, every factor of some w in T_L or in T_{LR} is again in alphabetic or left-right normal form resp., whereas the set T_F is only closed under taking prefixes. The set T_{LR} has the particular property that the normal form of some word can be constructed without knowing the order on G explicitly, but only a canonical form of $\overline{G}$ together with some total order on the set of anticliques of G. The Property implies that the concatenation of two LR-normal forms u and w is again in LR-normal form iff for every letter a of u which commutes with some letter b of w the following is true:

$$u = xav_1, w = v_2by, a > b \Rightarrow v_1v_2 \notin A_b^*.$$

From this observation we can derive the following useful results.

Theorem

Let G and H be arbitrary graphs with their partial orders induced by the canonical forms of $\overline{G}$ and $\overline{H}$. Consider $G \oplus H$, $G \otimes H$ as ordered graphs. Then the following equalities hold:

$$T_{LR}(G \otimes H) = T_{LR}(G) \cdot T_{LR}(H)$$

$$T_{LR}(G \oplus H) = T_{LR}(G) \cdot ((T_{LR}(H) - 1) \cdot (T_{LR}(G) - 1))^* \cdot T_{LR}(H)$$

To construct rational expressions for $T_{LR}(G)$ and arbitrary graphs G we see that it is sufficient to know rational expressions for $T_{LR}(H)$ for all graphs H which are indecomposable. All the rational operations used in this Theorem are unambiguously rational. The Theorem remains true if T_{LR} is replaced by T_L.

The MÖBIUS-function μ_M of a locally finite monoid M is defined as the inverse of the series $\zeta_M = \sum_{m \in M} m \in \mathbf{Z}\langle\langle M \rangle\rangle : \mu_M = \zeta_M^{-1}$. For $M = M(G)$, μ_M has been determined in [1] as $\mu_M = \sum(-1)^n a_1a_2 \ldots a_n$ where the summation is over all cliques of the graph G.

It is not difficult to see that for graphs G and H the MÖBIUS-function of $G \oplus H$ and $G \otimes H$can be calculated by $\mu_{M(G \oplus H)} = \mu_{M(G)} + \mu_{M(H)} - 1$ and $\mu_{M(G \otimes H)} = \mu_{M(G)} \cdot \mu_{M(H)}$. Moreover, $\mu_{M(G)}$ is an irreducible polynomial (in $\mathbf{Z}\langle\langle M(G) \rangle\rangle$) if and only if the graph $\overline{G}$ is connected.

Let $G = (A, \theta)$ be a graph and N be a monoid generated by A having a surjective morphism f onto the monoid $M = M(G)$. We say, μ_M and ζ_M *can be lifted* or have a *lifting* to $\mathbf{Z}\langle\langle N \rangle\rangle$ if we can find cross-sections T' and T of f in N such that for $\mu = T'(\mu_M)$ $(= \sum_{m \in M} < \mu_M, m > T'(m))$ and $\zeta = T(\zeta_M)$ the relation $\mu \cdot \zeta = 1$ holds in $\mathbf{Z}\langle\langle N \rangle\rangle$. In this sense the MÖBIUS-function of $M(A, \theta)$ may be used to calculate cross-sections in N. The following theorem

shows how far μ_M and ζ_M can be lifted. It states that for the inversion of μ_M it is not always necessary to use all the commutations specified by θ, depending on the choice of the cross-section used.

Theorem

(1) Let $G = (A, \theta)$ *be an ordered graph,* $\phi = \{\{a,b\},\{b,c\} \in \theta \mid \{a,c\} \notin \theta$ and $a < b < c\}$. *Then* $\mu_{M(G)}$ *and* $\zeta_{M(G)}$ *can be lifted to* $\mathbf{Z}\langle\langle M(A,\phi)\rangle\rangle$.

(2) If T' *and* T *are cross-sections of* $M(G)$ *such that* $\mu_{M(G)}$ *and* $\zeta_{M(G)}$ *can be lifted to* $\mathbf{Z}\langle\langle N\rangle\rangle$, *then* N *is a homomorphic image of* $M(A,\phi)$, *where* $\phi = \{\{a,b\},\{b,c\} \in \theta \mid \{a,c\} \notin \theta$ and $ab, bc \in T\}$.

As a special case one can find a result obtained recently by V. DIEKERT [4], stating that the existence of liftings of $\mu_{M(G)}$ and $\zeta_{M(G)}$ to $\mathbf{Z}\langle\langle A^*\rangle\rangle$ is equivalent to the existence of transitive orientations of the underlying graph $G = (A, \theta)$.

References

[1] Cartier, P., and Foata, D. *Problèmes combinatoires de commutation et réarrangements*, vol. 85 of *Lecture Notes in Mathematics*. Springer, Berlin, 1969.

[2] Choffrut, C. Free partially commutative monoids. *LITP - report 86-20* (1986).

[3] Cori, R., and Perrin, D. Automates et commutations partielles. *RAIRO Inf. Théor. 19* (1985).

[4] Diekert, V. Transitive orientations, MöBIUS-functions, and complete semi-Thue systems for free partially commutative monoids. In *Proc. of ICALP*, T. Lepistö and A. Salomaa, Eds., vol. 317 of *LNCS*. Springer, 1988.

[5] Duboc, C. Commutations dans les monoïdes libres: Un cadre théorique pour l'étude du parallélisme. *Thèse* (1986).

CONSTRUCTIONS SUR LES AUTOMATES ASYNCHRONES

par

Benoit Tremblay

INRS-Télécommunications

Résumé

Le but de cet exposé est de présenter de nouvelles constructions sur les automates asynchrones afin de s'approcher d'une preuve alternative au théorème de Zielonka. Les constructions que nous présentons pour les automates se veulent plus directes que celles présentées par Zielonka (1987) ou Métivier (1986). En fait, plutôt que de construire l'automate à partir de classes d'équivalence, les constructions se font directement sur les automates. On verra donc ce que signifie faire le mixage, l'union, l'intersection et le produit de langages sur les automates asynchrones. Nous présenterons également une construction qui permette de calculer la co-itération d'une trace.

1 Notations et définitions

Un langage $L \subseteq M$ est dit *reconnaissable* s'il existe un monoïde fini N et un morphisme $\psi: M \to N$ tels que $L = \psi^{-1}\psi(L)$. De manière équivalente, L est reconnaissable s'il existe un congruence d'index fini (ayant un nombre fini de classes) qui sature L.

On définit récursivement les *langages rationnels* d'un monoïde comme suit:

(i) les langages finis sont rationnels;

(ii) si L_1 et L_2 sont des langages rationnels alors $L_1 \cup L_2$, L_1L_2, L_1^* sont des langages rationnels.

Théorème 1.1 (Kleene) (Lallement 1979, Eilenberg 1974) *Dans A^*, un langage L est reconnaissable si et seulement s'il est rationnel.*

Soit A un alphabet, et $\theta \subseteq A \times A$ une relation symétrique et anti-réflexive appelée *relation de commutation*, on appelle *alphabet de commutation* le couple (A, θ).

Soit $\sim_\theta$ la congruence sur A^* définie par la relation $\{(ab, ba) \mid (a, b) \in \theta\}$, le *monoïde partiellement commutatif* engendré par (A, θ), $M(A, \theta)$ est le monoïde quotient $A^* / \sim_\theta$. Les éléments de $M(A, \theta)$ sont appelés des *traces* et les sous-ensembles, des *langages traces*.

Soit t une trace de $M(A, \theta)$, $x \in A^*$ est un *représentant* de t si $[x]_{\sim_\theta} = t$. On note φ le morphisme trivial de $A^* \rightarrow M(A, \theta)$.

Le *graphe de dépendance* de (A, θ), noté $(A, \overline{\theta})$, est le graphe simple ayant pour sommets les lettres de A et pour arêtes, les paires de lettres qui ne commutent pas: $\overline{\theta} = \{\{a, b\} \mid a \neq b, (a, b) \notin \theta\}$.

Le *graphe de dépendance* de $t \in M(A, \theta)$, G(t), est le graphe $(A, \overline{\theta})$ restreint aux lettres de t. On dit qu'une trace est *connexe* si son graphe de dépendance est connexe.

Pour tout sous-ensemble B de A, on a le morphisme de *projection alphabétique* de A^* sur B^* défini par:

$$\Pi_B(a) = a \quad \text{si } a \in B$$
$$= \varepsilon \quad \text{si } a \notin B.$$

On note Π_i pour Π_{A_i} lorsque $(A_1, ..., A_p)$ est une partition en cliques du graphe de dépendance. On notera C_a l'ensemble des indices des cliques qui contiennent la lettre a.

Proposition 1.2 *Un langage trace T est reconnaissable si et seulement si $\varphi^{-1}(T)$ est reconnaissable dans A^*.*

Proposition 1.3 *Un langage trace T est rationnel si et seulement s'il existe un langage rationnel L dans A^* tel que $\varphi(L) = T$.*

Ainsi, on ne peut étendre le théorème de Kleene aux monoïdes partiellement commutatifs.

Les résultats suivants nous permettront de caractériser les langages reconnaissables de M(A, θ).

Soit t une trace de M(A, θ), la *décomposition connexe* de t, notée \t\, est l'ensemble des projections alphabétiques de t selon les composantes connexes du graphe de dépendance de t: \t\ = {v ∈ M(A, θ) | il existe une composante connexe C de G(t) et v = $\prod_C$(t)}. Si T est un langage trace de M(A, θ), la décomposition connexe de T est l'ensemble des décompositions connexes des éléments de T: $\backslash T \backslash = \bigcup_{t \in T} \backslash t \backslash$

Proposition 1.4 (Ochmanski 1984) *Si T est un langage trace reconnaissable alors sa décomposition connexe \T\ est reconnaissable.*

Soit T un langage trace de M(A, θ), la *co-itération* de T, $T^{\square}$, est l'étoile de sa décomposition connexe: $T^{\square} = (\backslash T \backslash)^*$. Voici maintenant un théorème important, à la base du travail présenté ici.

Théorème 1.5 (Ochmanski 1984) *L'ensemble des langages reconnaissables de M(A, θ) est caractérisé comme suit:*

L'ensemble des langages traces est le plus petit ensemble E de parties de M(A, θ) respectant les deux conditions suivantes:

(i) les langages traces finis sont dans E;

(ii) E est fermé pour les opérations union, concaténation et co-itération.

Un *automate asynchrone* est un quintuplet

$\mathcal{A}$ = ((Q_1, ..., Q_p), (A_1, ..., A_p), I, T ,F) où:

Q_1, ..., Q_p sont des ensembles finis définissant Q = $Q_1 \times ... \times Q_p$ l'ensemble des états;

A_1, ..., A_p sont des alphabets finis définissant A = $\bigcup_{1 \le i \le p} A_i$, l'alphabet de $\mathcal{A}$;

I ⊆ Q est l'ensemble des états initiaux;

T ⊆ Q est l'ensemble des états terminaux;

$F \subseteq Q \times A \times Q$ est l'ensemble des transitions de $\mathcal{A}$ défini à partir des ensembles de flèches d'étiquette a, noté F_a, ($a \in A$) où :

$$F_a \subseteq (\bigtimes_{i \in I_a} Q_i) \times (\bigtimes_{i \in I_a} Q_i).$$

On a $((q_1, ..., q_p), a, (q'_1, ..., q'_p)) \in F$ si et seulement si $q_j = q'_j$ pour tout $j \notin C_a$ et $((q_{i_1}, ..., q_{i_n}), (q'_{i_1}, ..., q'_{i_n})) \in F_a$, où $C_a = \{i_1, ..., i_n\}$.

On se rappelle que $C_a = \{i \mid a \in A_i\}$.

On étend F aux ensembles d'états et aux mots de A^* de la même façon que dans le cas des automates finis. Les notions de déterminisme, de complétude, d'équivalence s'appliquent également aux automates asynchrones. De même, le langage reconnu par un automate asynchrone est $|\mathcal{A}| = \{ w \in A^* \mid F(I, w) \cap T \neq \emptyset \}$.

En fait, les automates asynchrones peuvent être considérés comme une généralisation des automates finis, ceux-ci étant le cas particulier où $p = 1$.

D'un autre point de vue, les automates asynchrones peuvent également être considérés comme un cas particulier d'automates finis avec un choix particulier pour les ensembles d'états et de transitions.

Proposition 1.6 *Soit L un langage reconnu par un automate asynchrone $\mathcal{A}$, L est saturé par $\theta_{\mathcal{A}}$, c'est-à-dire $[|\mathcal{A}|] = |\mathcal{A}|$.*

On définit le langage trace reconnu par un automate asynchrone $\mathcal{A}$, $T(\mathcal{A})$, comme étant $\varphi(|\mathcal{A}|) \subseteq M(A, \theta_{\mathcal{A}})$. De plus, un langage trace $T \subseteq M(A, \theta)$ est dit reconnu par un automate asynchrone s'il existe un automate asynchrone $\mathcal{A}$ tel que $\theta = \theta_{\mathcal{A}}$ et $T = T(\mathcal{A})$.

La proposition ci-dessus montre que tout langage trace reconnu par un automate asynchrone est reconnaissable. Zielonka (1987) a démontré la réciproque:

Théorème 1.7 *Tout langage trace reconnaissable est reconnu par un automate asynchrone.*

Remarque 1.8: *Les langages reconnus par des automates asynchrones étant toujours saturés par les commutations, lorsque l'on veut vérifier que $T(\mathcal{A}) = T(\mathcal{A}')$ il suffit de vérifier*

que $|\mathcal{A}| = |\mathcal{A}'|$. *Et réciproquement si* $T(\mathcal{A}) = T(\mathcal{A}')$ *alors* $\varphi^{-1}(T(\mathcal{A})) = \varphi^{-1}(T(\mathcal{A}'))$ *c'est-à-dire* $|\mathcal{A}| = |\mathcal{A}'|$.

2 Produit de mixage

Soit $\mathcal{A}_1 = (Q_1, A_1, I_1, T_1, F_1)$ et $\mathcal{A}_2 = (Q_2, A_2, I_2, T_2, F_2)$ deux automates finis, le *mixé des automates* $\mathcal{A}_1$ et $\mathcal{A}_2$, $\mathcal{A}_1 \sqcap\!\sqcap \mathcal{A}_2 = (Q, A_1 \cup A_2, I, T, F)$ se définit en posant $Q = Q_1 \times Q_2$, $I = I_1 \times I_2$, $T = T_1 \times T_2$ et finalement,

$F = F_{A_1 \setminus A_2} \cup F_{A_2 \setminus A_1} \cup F_{A_1 \cap A_2}$ où

$F_{A_1 \setminus A_2} = \{((p, p'), a, (q, p')) \mid a \in A_1 \setminus A_2, p' \in Q_2 \text{ et } (p, a, q) \in F_1\}$,

$F_{A_2 \setminus A_1} = \{((p, p'), a, (p, q')) \mid a \in A_2 \setminus A_1, p \in Q_1 \text{ et } (p', a, q') \in F_2\}$,

$F_{A_1 \cap A_2} = \{((p, p'), a, (q, q')) \mid a \in A_1 \cap A_2, (p, a, q) \in F_1 \text{ et } (p', a, q') \in F_2\}$.

On peut vérifier facilement que le produit de mixage d'automates est associatif et qu'on peut donc définir sans ambiguïté le produit

$\mathcal{A}_1 \sqcap\!\sqcap \ldots \sqcap\!\sqcap \mathcal{A}_p$ pour $p \geq 2$.

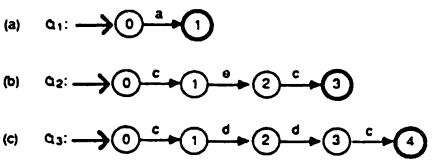


Figure 1 Automates reconnaissant les projections du mot cdeadc selon les cliques de P.

Exemple 2.1 *Soit l'alphabet de commutation* $(\{a, b, c, d, e\}, \{(a, c), (a, d), (a, e), (b, d), (c, a), (d, a), (d, b), (d, e), (e, a), (e, d)\}$ *et la partition en cliques* $P = (\{a, b\}, \{b, c, e\}, \{c, d\})$, *soit maintenant* $x = cdeadc$. *On peut facilement calculer l'automate reconnaissant* $[x]$. *On calcule d'abord les projections de x selon chacune des cliques de P:* $\Pi_1(x) = a$, $\Pi_2(x) = cec$ *et* $\Pi_3(x) = cddc$. *On construit trivialement les automates* $\mathcal{A}_1$, $\mathcal{A}_2$ *et* $\mathcal{A}_3$ *qui reconnaissent chacune des projections. On les retrouve respectivement à la figure 1 (a), (b) et*

(c). Donc, l'automate de mixage $\mathcal{A}_1 \sqcap \mathcal{A}_2 \sqcap \mathcal{A}_3$ reconnaît la classe de commutation de x. Cet automate est illustré à la figure 2.

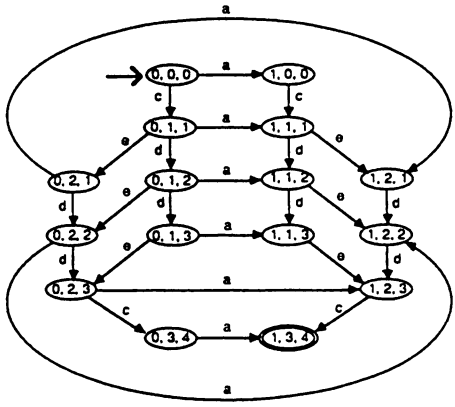


Figure 2 Automate de mixage reconnaissant [cdeadc]

3 Complétion et complémentation d'un automate asynchrone

On sait que la famille des langages traces reconnaissables est fermée pour la complémentation, on donnera ici une construction qui permettra de calculer un nouvel automate asynchrone qui reconnaîtra le complément du langage associé à un automate déterministe. La construction utilisée suppose d'avoir sous la main un automate complet, c'est pourquoi nous présenterons d'abord une méthode qui rendra possible de calculer, à partir d'un automate asynchrone quelconque, un automate asynchrone équivalent qui soit complet.

Soit $\mathcal{A} = ((Q_1, ..., Q_p), (A_1, ..., A_p), I, T, F)$ un automate asynchrone. Soit également l'automate $\mathcal{A}' = ((Q'_1, ..., Q'_p), (A_1, ..., A_p) I, T, F')$ calculé à partir de $\mathcal{A}$ en posant $Q'_i = Q_i \cup \{\perp_i\}$ ($\perp_i$ est un symbole spécial qui n'est pas dans Q_i) et F' défini à partir des ensembles de flèches F_a où

$$F'_a = F_a \cup \{(q_a, \perp_a) \mid q_a \in Q'_a \text{ et } (\{q_a\} \times Q'_a) \cap F_a = \emptyset.\}.$$

F'_a ajoute des transitions par a là où il n'y en a pas déjà. Si à partir de q_a, on a déjà une

transition par a dans F_a, il n'y en aura pas de nouvelle dans F'_a. Cette condition est suffisante pour permettre d'affirmer que:

- $\mathcal{A}'$ est un automate asynchrone complet.
- si $\mathcal{A}$ est déterministe alors $\mathcal{A}'$ l'est également.

Proposition 3.1 *La construction de complétion ne modifie pas le langage reconnu par l'automate. C'est-à-dire* $T(\mathcal{A}) = T(\mathcal{A}')$.

Exemple 3.2 *Soit l'automate de la figure 3, défini par* $\mathcal{A} = (\{\{0, 1, 2\}, \{0, 1\}\}), (\{a, b\}, \{b, c\}), \{(0, 0)\}, \{(2, 0), (0, 1)\}, F)$ *où F est défini à partir de:*

$F_a = \{((0), (1)), ((1), (2))\}$,

$F_b = \{((0, 0), (0, 1)), ((1, 0), (2, 1)), ((2, 0), (0, 0))\}$,

$F_c = \{((1), (0))\}$.

Pour obtenir l'automate complet, on doit poser $Q_1 = \{0, 1, 2, \perp\}$ *et* $Q_2 = \{0, 1, \perp\}$, *on doit modifier les transitions de la façon suivante:*

$F'_a = F_a \cup \{((2), (\perp)), ((\perp), (\perp))\}$

$F'_b = F_b \cup \{((0, 1), (\perp, \perp)), ((1, 1), (\perp, \perp)), ((2, 1), (\perp, \perp)), ((\perp, 1), (\perp, \perp)), ((\perp, 0), (\perp, \perp)), ((0, \perp), (\perp, \perp)), ((1, \perp), (\perp, \perp)), ((2, \perp), (\perp, \perp)), ((\perp, \perp), (\perp, \perp))\}$

$F'_c = F_c \cup \{((0), (\perp)), ((\perp), (\perp))\}$.

On obtient ainsi l'automate asynchrone complet de la figure 4.

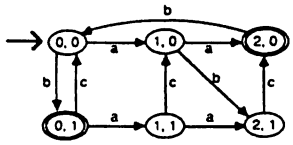


Figure 3 Automate asynchrone déterministe

On sait que la famille des langages reconnaissables d'un monoïde partiellement commutatif est fermée pour la complémentation. En voici une preuve constructive qui

permette, connaissant un automate asynchrone déterministe reconnaissant T, de calculer un automate asynchrone qui reconnaisse $M(A, \theta_{\mathcal{A}}) \setminus T$.

Proposition 3.3 *Soit* $\mathcal{A} = ((Q_1, ..., Q_p), (A_1, ..., A_p), \{i\}, T, F)$ *un automate asynchrone déterministe et complet et soit* $\mathcal{A}' = ((Q_1, ..., Q_p), (A_1, ..., A_p), \{i\}, Q \setminus T, F)$ *l'automate asynchrone déterministe et complet obtenu en changeant les états terminaux pour les états non-terminaux de* $\mathcal{A}$ *et vice-versa. On a alors que le langage reconnu par* $\mathcal{A}'$ *est le complément du langage reconnu par* $\mathcal{A}$ *dans* $M(A, \theta_{\mathcal{A}})$.

$T(\mathcal{A}) = M(A, \theta_{\mathcal{A}}) \setminus T(\mathcal{A}')$

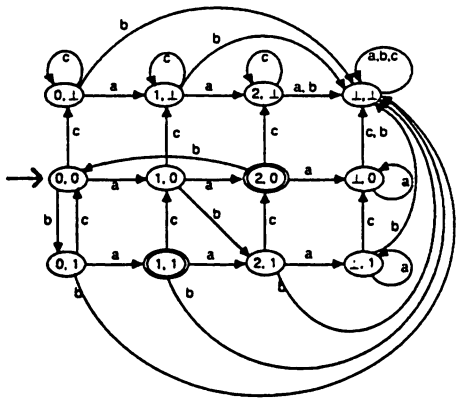


Figure 4 Automate asynchrone complet

4 Produit direct d'automates asynchrones: union et intersection de langages traces

Dans cette section, on introduit une nouvelle opération sur les automates asynchrones: le produit direct. Cette opération nous permettra de calculer un automate qui reconnaisse l'union ou l'intersection des langages reconnus par deux automates. La construction sera la même dans les deux cas, c'est le choix des états terminaux qui permettra de trouver l'union ou l'intersection des langages. Le produit direct présenté ici est fortement inspiré du produit direct d'automates finis présentés par Lallement (1979).

Une construction simple qui nous permette de construire un automate reconnaissant l'union des langages de $\mathfrak{B} = ((R_1, ..., R_p), (A_1, ..., A_p), J, U, G)$ et $\mathfrak{C} = ((S_1, ..., S_p), (A_1, ..., A_p), K, V, H)$ consiste à réunir trivialement les deux automates et à les considérer comme un seul. Cette construction qui donne un automate non-déterministe, permet de simuler les transitions sur les deux automates à la fois. Ainsi, on pose $\mathfrak{A} = ((Q_1, ..., Q_p), (A_1, ..., A_p), I, T, F)$ avec $Q_i = R_i \cup S_i$ (R_i et S_i sont supposés disjoints), $I = J \cup K$, $T = U \cup V$ et $F_a = G_a \cup H_a$. On peut voir aisément que $\mathfrak{A}$ reconnaît bien $|\mathfrak{B}| \cup |\mathfrak{C}|$.

Il sera cependant préférable d'avoir une construction qui conserve le déterminisme des automates de départ car nous ne connaissons toujours pas d'algorithme de déterminisation pour les automates asynchrones. De plus, advenant le cas où nous en trouvions un, il serait fort probablement moins coûteux d'avoir une construction qui conserve le déterminisme des automates initiaux que de déterminiser les automates. Une telle construction de l'union utilise le produit direct d'automates asynchrones. Le produit direct nous permettra également de construire un automate qui reconnaît l'intersection des langages.

Soit $r = (r_1, r_2, ..., r_p)$ et $s = (s_1, s_2, ..., s_p)$ deux états ayant le même nombre de composantes, le *produit direct* de r et s, noté $r \otimes s$, est:

$$r \otimes s = ((r_1, s_1), ..., (r_p, s_p))$$

et par extension si R et S sont deux ensembles d'états alors

$$R \otimes S = \{ r \otimes s \mid r \in R \text{ et } s \in S \}.$$

Soit $\mathfrak{B} = ((R_1, ..., R_p), (A_1, ..., A_p), J, U, G)$ et $\mathfrak{C} = ((S_1, .., S_p), (A_1, ..., A_p), K, V, H)$ deux automates asynchrones complets ayant les mêmes alphabets, le *produit direct* de $\mathfrak{B}$ et $\mathfrak{C}$, $\mathfrak{B} \otimes \mathfrak{C}$, est l'automate $\mathfrak{A} = ((Q_1, ..., Q_p), (A_1, ..., A_p), I, T, F)$ défini par:

$Q_i = R_i \times S_i$, c'est-à-dire que $Q = R \otimes S$,

$I = J \otimes K$,

F est défini à partir des ensembles F_a, $a \in A$, où

$F_a = \{((r_a \otimes s_a), (r'_a \otimes s'_a)) \mid (r_a, r'_a) \in G_a \text{ et } (s_a, s'_a) \in H_a \}$.

L'ensemble des états terminaux sera déterminé plus loin selon que l'on veut faire l'union ou l'intersection des langages.

Remarque 4.1 *L'hypothèse de complétude de $\mathcal{B}$ et $\mathcal{C}$ nous assure de toujours pouvoir trouver un chemin pour les mots de $|\mathcal{B}| \cup |\mathcal{C}|$ même si ces mots ne sont pas dans les deux langages.*

Remarque 4.2 *Pour tout mot w, $(r \otimes s) \xrightarrow{w} (r' \otimes s')$ est un chemin dans $\mathcal{A}$ si et seulement si $r \xrightarrow{w} r'$ est un chemin dans $\mathcal{B}$ et $s \xrightarrow{w} s'$ est un chemin dans $\mathcal{C}$.*

Remarque 4.3 *Si $\mathcal{B}$ et $\mathcal{C}$ sont déterministes alors $\mathcal{A}$ l'est également.*

Pour reconnaître l'union des langages, notre automate produit devra accepter les mots du langage reconnu par $\mathcal{B}$ et ceux du langage reconnu par $\mathcal{C}$. Pour réaliser une telle condition l'ensemble des états terminaux T devra être choisi de la façon suivante:

$$T = \{(r \otimes s) \in Q \mid r \in U \text{ ou } s \in V\}$$

ou de manière équivalente:

$$T = U \otimes S \cup R \otimes V.$$

La prochaine proposition vient confirmer notre intuition.

Proposition 4.4: *Soit $\mathcal{A} = ((Q_1, ..., Q_p), (A_1, ..., A_p), I, T, F)$ où $\mathcal{A} = \mathcal{B} \otimes \mathcal{C}$ et $T = U \otimes S \cup R \otimes V$ alors $T(\mathcal{A}) = T(\mathcal{B}) \cup T(\mathcal{C})$.*

On trouve une construction tout à fait semblable pour calculer l'intersection. On a encore $\mathcal{A} = \mathcal{B} \otimes \mathcal{C}$ mais cette fois, on pose T l'ensemble des états qui sont le produit direct de deux états terminaux, c'est-à-dire:

$$T = \{(r \otimes s) \in Q \mid r \in U \text{ et } s \in V\}$$

ou de manière équivalente:

$$T = U \otimes S \cap R \otimes V = U \otimes V.$$

Proposition 4.5: *Soit $\mathcal{A} = \mathcal{B} \otimes \mathcal{C}$ et $T = U \otimes V$, alors $T(\mathcal{A}) = T(\mathcal{B}) \cap T(\mathcal{C})$.*

Exemple 4.6 *Soit les deux automates asynchrones des figures 4 et 5. Pour permettre de distinguer les états de chacun des automates, nous avons notés ceux de l'automate 5 en caractères gras. L'automate de la figure 6 représente l'automate construit en utilisant le produit direct. Nous n'avons cependant conservé que les états utiles afin d'alléger sa représentation graphique.*

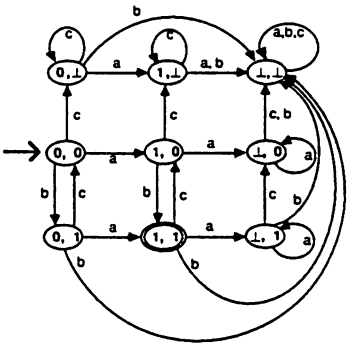


Figure 5 Automate 5 utilisé dans le produit direct

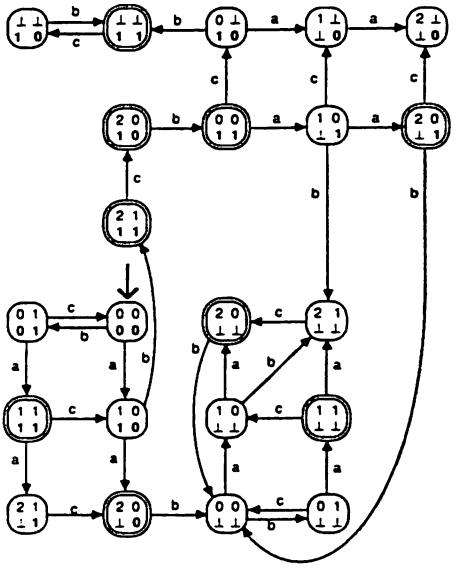
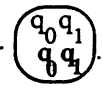


Figure 6 Automate reconnaissant l'union des langages des automates 4 et 5

Dans la figure 6, l'état $((q_0, q_0), (q_1, q_1))$ est représenté par  .

Il est à noter également que si on complète l'automate de la figure 6, on n'obtiendra pas l'automate produit comme défini ci-haut. La construction génère un nombre assez grand d'états qui ne sont pas utiles.

Si dans l'automate 6, on avait plutôt posé l'ensemble des terminaux égal à $T_4 \otimes T_5 = \{((2, 1), (0, 1)), ((1, 1), (1, 1))\}$, on aurait alors obtenu l'intersection des langages reconnus par les automates 4 et 5.

5 Concaténation d'automates asynchrones et produit de langages traces

On définit dans cette section une opération de concaténation sur les automates asynchrones qui permettra de reconnaître le produit des langages traces reconnus par deux automates asynchrones.

Remarque 5.1 *Soit $\mathcal{B} = ((R_1, ..., R_p), (A_1, ..., A_p), J, U, G)$ et $\mathcal{C} = ((S_1, ..., S_p), (A_1, ..., A_p), K, V, H)$ deux automates asynchrones, il serait facile de montrer que $|\mathcal{B}| = \bigcup_{u \in U} |\mathcal{B}_u|$ et $|\mathcal{C}| = \bigcup_{k \in K} |\mathcal{C}^k|$, où*

$\mathcal{B}_u = ((R_1, ..., R_p), (A_1, ..., A_p), J, \{u\}, G)$ *et*
$\mathcal{C}^k = ((S_1, ..., S_p), (A_1, ..., A_p), \{k\}, V, H)$.

On trouve alors que $|\mathcal{B}||\mathcal{C}| = \bigcup_{\substack{u \in U \\ k \in K}} |\mathcal{B}_u| \, |\mathcal{C}^k|$. Comme on sait faire la construction pour l'union finie de langages traces à partir d'automates asynchrones, on saura donc construire le produit des langages traces de deux automates si on sait le faire dans le cas où $|U| = |K| = 1$.

On définira donc une concaténation d'automate qui supposera que $\mathcal{B}$ n'a qu'un seul état terminal $u = (u_1, ..., u_p)$ et $\mathcal{C}$ n'a qu'un seul état initial $k = (k_1, ..., k_p)$. On suppose aussi que pour tout i, $R_i \cap S_i = \emptyset$.

On suppose également donnée une fonction de transfert τ: $Q \rightarrow Q$ définie par:

$\tau((q_1, ..., q_p)) = (q'_1, ..., q'_p)$ où

$q'_i = q_i$ si $q_i \neq u_i$,

$q'_i = k_i$ si $q_i = u_i$.

Cette fonction permet en fait de passer du premier automate au second lorsqu'on arrive à l'état terminal. D'un façon imagée, c'est comme une transition ε qui va de u_i vers k_i. Plutôt que définir τ_a pour chacune des lettres de A, on utilise également τ comme une fonction de Q_a vers Q_a.

L'automate résultant de la *concaténation* de $\mathfrak{B}$ et de $\mathfrak{C}$ est l'automate $\mathcal{A} = \mathfrak{B} \cdot \mathfrak{C}$ défini par:

$\mathcal{A} = ((Q_1, ..., Q_p), (A_1, ..., A_p), I, T, F)$ où

$Q_i = R_i \cup S_i$,

$I = J \subseteq Q$,

$T = \{q \in Q \mid \tau(q) \in V\}$,

et F est défini à partir des F_a, $a \in A$, où

$F_a = G_a \cup H'_a$ avec $H'_a = \{(q_a, s_a) \mid (\tau(q_a), s_a) \in H_a\}$.

Remarque 5.2 *L'automate produit sera déterministe si et seulement si les deux automates sont déterministes et si, pour toute lettre a de A, il existe une transition par a à partir de l'état k, il n'existe pas de transition par a à partir de l'état u. C'est-à-dire:*

$\mathcal{A}$ *est déterministe* $\Leftrightarrow (\forall\, a \in A)[(\exists\, (k, a, s) \in H) \Rightarrow (\nexists\, (u, a, r) \in G)]$. *Ainsi, si l'automate* $\mathfrak{B}$ *est complet, on produira un automate qui n'est pas déterministe (sauf si* $H = \varnothing$*).*

Proposition 5.3: *Soit* $\mathcal{A} = \mathfrak{B} \cdot \mathfrak{C}$ *alors* $T(\mathcal{A}) = T(\mathfrak{B})\, T(\mathfrak{C})$.

Il est assez simple de montrer que $[|\mathfrak{B}||\mathfrak{C}|] \subseteq |\mathcal{A}|$. Pour montrer que $|\mathcal{A}| \subseteq [|\mathfrak{B}||\mathfrak{C}|]$, on montrera que si $w \in |\mathcal{A}|$, alors il existe un mot w' congru à w qui est élément de $|\mathfrak{B}||\mathfrak{C}|$. Cette démonsatration se fait en étudiant le chemin d'étiquette w dans l'automate $\mathcal{A}$.

Exemple 5.4 *Soit* $\mathfrak{B} = ((R_1, R_2), (\{a, b\}, \{b, c\}), J, U, G)$ *où* $R_1 = \{0, 1\}$, $R_2 = \{0, 1\}$, $J = \{(0, 0)\}$, $U = \{(1, 1)\}$ *et G est défini par* $G_a = \{((0), (1))\}$, $G_b = \{((1, 1), (0,$

0))}, G_C = {((0), (1))}. Cet automate est représenté à la figure 7. Soit également $\mathfrak{C}$ *l'automate de l'exemple 3.2 représenté à la figure 3.*

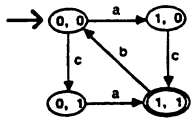


Figure 7 Automate $\mathfrak{B}$ utilisé pour la concaténation dans l'exemple 5.4

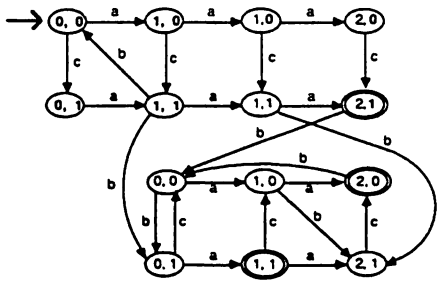


Figure 8 Automate reconnaissant le produit des langages des automates des figures 3 et 7.

Après concaténation des deux automates, on obtient l'automate $\mathfrak{A}$ *représenté à la figure 8 et défini par:*

Q_1 = {0, 1, 0, 1, 2}

Q_2 = {0, 1, 0, 1}

I = {(0, 0)}

T = {(2, 0), (1, 1), (2, 1)}

et F est défini à partir de F_a, F_b *et* F_c *avec*

F_a = {((0), (1)), ((1), (1)), ((0), (1)), ((1), (2))}

F_b = {((1, 1), (0, 0)), ((1, 1), (0, 1)), ((0, 0), (0, 1)), ((1, 0), (2, 1)), ((2, 0), (0, 0)), ((1, 1), (2, 1)), ((2, 1), (1, 1))}

F_c = {((0), (1)), ((1), (0))}

On peut observer que l'automate produit n'est pas déterministe car de l'état (1, 1) dans $\mathcal{B}$ et de l'état (0, 0) dans $\mathcal{C}$, il existe une transition par b. On obtient donc que de l'état (1, 1) dans $\mathcal{A}$, on aura les deux transitions ((1, 1), b, (0, 1)) et ((1, 1), b, (0, 0)).

On connaît maintenant une construction pour reconnaître une trace via le produit de mixage, pour l'union de langages traces via le produit direct, et pour le produit de langages traces via la concaténation d'automates asynchrones. Il reste maintenant à trouver une construction pour la co-itération.

6 Co-itération

Malheureusement, nous n'avons toujours pas trouvé de construction qui puisse nous permettre de reconnaître le co-itéré d'un langage trace. Nous donnerons cependant la construction permettant d'obtenir un automate asynchrone reconnaissant la co-itération d'une trace quelconque. Cette construction utilisera le produit de mixage. Elle est donnée pour construire l'étoile d'une trace connexe par Duboc (1986b). Nous affirmons ici que cette construction permet en fait d'obtenir un automate reconnaissant la co-itération d'une trace.

On suppose donnée une partition par cliques $P = (A_1, ..., A_p)$ du graphe de non-commutation de θ. Soit t une trace de $M(A, \theta)$, soit $t_i = \prod_i(t)$ les projections de t sur A_i, pour $i = 1, ..., p$ et finalement soit $\mathcal{A}_i = (Q_i, A_i, I_i, T_i, F_i)$ un automate fini avec $Q_i = \{0, 1, ..., |t_i| - 1\}$, $I_i = T_i = \{0\}$ et $F_i = \{(j-1, t_i(j), j) \mid 1 \leq j \leq |t_i| - 1\} \cup \{(|t_i|-1, t_i(|t_i|), 0)\}$ où $t_i(j)$ est la $j^{\text{ième}}$ lettre de t_i. Les t_i étant rigides (aucune lettre ne commute), on peut effectivement établir un ordre total sur leurs lettres.

Proposition 6.1: *Soit t une trace de M(A, θ) et soit $\mathcal{A}_1, ... \mathcal{A}_p$ les automates qui lui sont associés. Alors $t^{\square}$ est reconnu par l'automate $\mathcal{A} = \mathcal{A}_1 \sqcap ... \sqcap \mathcal{A}_p$.*

Exemple 6.2 *Soit la trace t = [cdeadc] et l'alphabet de commutation de l'exemple 2.1. On a \t\ = {[a], [cdedc]} et $t^{\square} = \{[a], [cdedc]\}^*$, on trouve également $t_1 = a$, $t_2 = cec$, $t_3 = cddc$. On peut construire les automates $\mathcal{A}_1$, $\mathcal{A}_2$ et $\mathcal{A}_3$ reconnaissant respectivement t_1, t_2 et t_3 et qui sont illustrés à la figure 9. Le co-itéré de t est reconnu par l'automate construit à l'aide du produit de mixage à partir des automates $\mathcal{A}_1$, $\mathcal{A}_2$ et $\mathcal{A}_3$ et qui est présenté à la figure 10.*

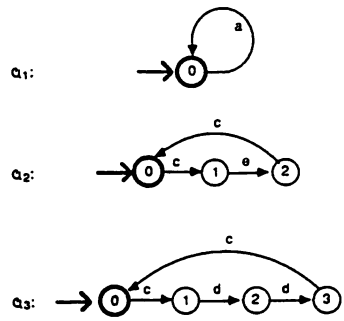


Figure 9 Automates reconnaissant respectivement a^*, $(cec)^*$ et $(cddc)^*$

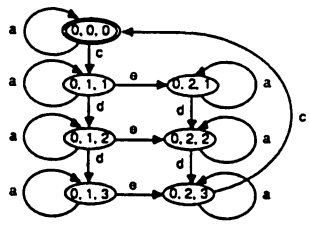


Figure 10 Automate de mixage $\mathcal{A}_1 \sqcap \mathcal{A}_2 \sqcap \mathcal{A}_3$ reconnaissant $[cdeadc]^\square$

On peut constater que le même résultat aurait été obtenu en travaillant à partir de l'automate qui reconnaît t, en effet, si sur l'automate de la figure 2, on identifiait l'état terminal à l'état initial, composante par composante, c'est-à-dire: sur la première composante l'état 1 est identifié à l'état 0, sur la seconde composante, l'état 3 à l'état 0 et enfin sur la troisième, l'état 4 à l'état 0, on obtiendrait exactement l'automate calculé ci-dessus.

Comment se fait-il qu'on ne puisse pas appliquer la même méthode sur tous les automates? Une raison fondamentale est que pour certains langages traces reconnaissables, il est parfois impossible d'obtenir un automate ayant un seul état terminal et lorsqu'il y a plus d'un état terminal, on ne peut pas utiliser la méthode d'identification des états comme le montre l'exemple suivant:

Exemple 6.3 *Soit A = {a, b, c, d}, θ = {(a, c), (c, a), (b, d), (d, b)} et soit P = ({a, b}, {b, c}, {c, d}, {d, a}) l'unique partition en cliques de (A, $\overline{\theta}$). Soit également L = {ab, ad, cb, cd} on a bien [L] = L car L est rigide et comme L est fini, on a φ(L) est un langage trace reconnaissable. L'automate asynchrone de la figure 11 reconnaît L.*

Dans l'automate de la figure 11, on ne peut identifier tous les états terminaux à l'état initial. De cette façon, on trouverait sur chacune des composantes 0 ≡ 1 ≡ 2 ≡ 3, après simplification, on obtient l'automate à un seul état de la figure 12. Cet automate reconnaît en fait A^* qui est différent de L^{π}.

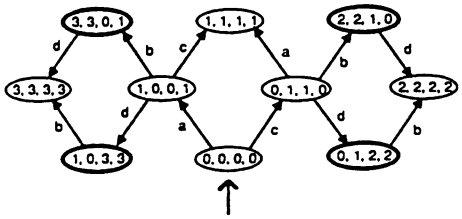


Figure 11 Automate asynchrone reconnaissant {[ab], [ad], [cb], [cd]}

Figure 12 Automate obtenu en identifiant les états terminaux à l'état initial de l'automate de la figure 11

Le fait qu'un automate $\mathcal{A}$ ait un seul état initial et un seul état terminal n'est pas un gage de réussite, il se peut que l'automate obtenu en identifiant l'état initial et l'état terminal de $\mathcal{A}$ ne reconnaisse toujours pas le co-itéré de $|\mathcal{A}|$. L'exemple suivant illustre bien ce fait.

MEC : a system for constructing and analysing transition systems

André Arnold
Laboratoire d'Informatique *
Université Bordeaux I

—

Abstract

MEC is a tool for constructing and analysing transition systems modelizing processes and systems of communicating processes.

From representations of processes by transition systems and from a representation of the interactions between the processes of a system by the set of all allowed global actions, MEC builds a transition system representing the global system of processes as the *synchronized product* of the component processes.

Such transition systems can be checked by computing sets of states and sets of transitions verifying properties given by the user of MEC. These properties are expressed in a language allowing definitions of new logical operators as least fixed points of systems of equations; thus all properties expressed in most of the branching-time temporal logic can be expressed in this language too.

MEC can handle transition systems with some hundred thousands states and transitions. Constructions of transition systems by synchronized products and computations of sets of states and transitions are performed in time linear with respect to the size of the transition system.

Introduction

The notion of *transition system* plays an important role for describing and studying processes and systems of communicating processes. A simple way to represent processes, introduced for instance in [10] and widely used in many works on semantics and verification of processes, is to consider that a process is a set of *states* and that an *action* or an *event* makes the current state of the process to change; thus the possible elementary behaviours of the process are represented by *transitions*: each transition contains the current state of the process, the new state it enters and the name of the action or event which caused this change. Transition systems are also used to describe systems of communicating processes, and not only individual processes; the states of the system are the tuples of states of its components and the transitions of the systems are tuples

*Unité de Recherche associée au Centre National de la Recherche Scientifique n° 726

†Work supported by the French Research Project C^3

of transitions of the components, provided these transitions are allowed – or obliged – to be executed simultaneously. Arnold and Nivat [12,3,1] have named this construction, which is implemented in MEC, *synchronization product*.

Once a system of processes is represented as a transition system, one can extract, from this transition system, some informations about the behaviour of the system of processes it represents. It is what we call *analysis* of a transition system and it amounts to computing the set of states or the set of transitions which satisfy some property of interest when looking at the behaviour of the system. For instance it is easy to check if the transition system has "deadlocks", i.e. states in which no transition is executable, or states in which every executable transition leads to a deadlock; a very simple algorithm can give the set of all these states. Thus an analyser is simply a tool which computes the set of all states or of all transitions of a given transition system satisfying some given property. The main feature of such an analyser is obviously the family of properties of states and transitions it can deal with.

In the systems **Cesar** [14] and **emc** [6], properties of states are expressed by formulas of branching time temporal logics. Given a formula F and a transition system $\mathcal{A}$, these systems compute the set $F_{\mathcal{A}}$ of states of $\mathcal{A}$ satisfying F (or, at least, decide if the "initial state" of $\mathcal{A}$ belongs to $F_{\mathcal{A}}$). In MEC we adopt a slightly different point of view, in some sense more algebraic than logical [7]. Let ω be some logical operator and let $F = \omega(F_1, \ldots, F_n)$ be a formula. For a transition system $\mathcal{A}$, the set $F_{\mathcal{A}}$ of states satisfying $\mathcal{A}$ depends on the sets $(F_i)_{\mathcal{A}}$ of states satisfying F_i, thus $F_{\mathcal{A}} = \omega_{\mathcal{A}}((F_1)_{\mathcal{A}}, \ldots, (F_n)_{\mathcal{A}})$, where $\omega_{\mathcal{A}}$ is an operator defined on the cartesian product of the powerset of states with the powerset of states as a range. Then formulas can be considered as expressions which have to be evaluated, in a way very similar to what happens in programming languages with arithmetic or boolean expressions. Thus the language used in MEC to express properties consists in variables and constants ranging over the powerset of states and on the powerset of transitions, and of sorted operators; the basic mechanism implemented in MEC is the execution of assignments *variable := expression*, exactly like in programming languages.

It remains to define the basic operators which can be used to build expressions. We can take the operators of branching time temporal logics (which are computable in linear time with respect to the size of the transition system) but, also, any other kind of operator which is easily computable. For instance in MEC we use the operator which associates with a set of states the union of the strongly connected components intersecting this set; although this operator is not really a logical operator, it is as easy to compute as the other ones, because of the Tarjan's algorithm [16] which is linear too.

Another feature of MEC is that the set of basic operators used to build expressions can be extended, in the same way that the set of operators in arithmetic expressions can be extended, in some programming languages, by defining new functions (especially recursive functions). It is well known that temporal logic operators can be characterized as least fixed points of equations [15,8,6], and this observation has led to the definition of the μ-calculus as an extension of branching time temporal logics [13,11]. MEC provides for the definition of new operators characterized as least fixed points of systems of equations [7] and then its expressive power is at least as powerful as the expressive power of alternation-depth-one μ-calculus defined by Emerson and Lei [9]. Indeed these

new operators defined by systems of equations are still computable in linear time, like the basic ones, because of the Arnold-Crubillé's algorithm [2] to solve fixed point equations on transition systems in linear time.

1 Transition systems

1.1 Labelled transition systems

A *labelled transition system* over an alphabet A of *actions* or *events* is a tuple $\mathcal{A} = < S, T, \alpha, \beta, \lambda >$ where

- S is a finite set of *states*,
- T is a finite set of *transitions*,
- $\alpha, \beta : T \longrightarrow S$ are the mappings which associate with every transition t its *source state* $\alpha(t)$ and its *target state* $\beta(t)$,
- $\lambda : T \longrightarrow A$ labels a transition t by the action or event $\lambda(t)$ which causes this transition.

We assume that there never exist two different transitions with the same label between the same two states, i.e. the mapping $< \alpha, \lambda, \beta >: T \longrightarrow S \times A \times S$ is injective.

1.2 Parametrized transition systems

A parametrized transition system is a labelled transition system given with some sets of designed states and some sets of designed transistions, called parameters. The role of these parameters is to give some additional informations on the transition system; it is the case when some states play a special role or when some transitions play a special role which is not specified by the label of the transition. Some example of such situations will be given below.

Example 1. Let us consider a boolean variable. It has two states, denoted by **0** and **1**, according to the current value (0 or 1) of the variable. The set A of actions performed by such a boolean variable contains

to0 which means that the variable is set to 0,

to1 which means that the variable is set to 1,

is0 which tests whether the value of the variable is 0,

is1 which tests whether the value of the variable is 1,

e which does nothing.

The first two actions modify the value of the variable, i.e. its state, in an obvious way. The two tests can be executed only if the variable has the tested value, and this value is not modified. The last action, when executed, does not change the value of the variable. As we shall see later on, (example 3, this null action is a way to express the possibility of occurrence of events which does not modify the state of the variable.

Therefore the transition system has eight transitions: for each one of these transitions we give, in the following table, its source state, its target state, and its label.

transitions	source	target	label
t_1	0	0	e
t_2	0	0	to0
t_3	0	1	to1
t_4	0	0	is0
t_5	1	1	e
t_6	1	0	to0
t_7	1	1	to1
t_8	1	1	is1

Such transition systems often have a graphical representation, more readable when their size are little, as shown in figure 1

For the system MEC, transition systems are given in the form shown in figure 2. On the figure 2 one can notice the last line

```
< initial = { 0 }  >.
```

which defines a parameter, named "initial", reduced to a single state, **0**. This parameter is used to say that the state **0** has some special property, indeed it is the initial state of the transition system: the initial value of the variable, before any action is performed, is **0**.

Example 2. Let us now consider the Peterson's algorithm for mutual exclusion of two processes. This algorithm uses three shared boolean variables, `flag[0]`, `flag[1]`, and `turn`, all three initialised to **0**. Each one of the two processes executes the program given in figure 3 where `me` is equal to `0` and `other` is equal to `1` for the first process, and `me` is equal to `1` and `other` is equal to `0` for the second one. This program can also be represented by a transition system, states of which are the locations in the program and transitions between locations are labelled by elementary actions performed in the execution of the program. We also consider a special action **e** which does not change the state, which means that a process can stay idle at every moment. The MEC description of this transition system is given in figure 4.

In this transition system there are three state parameters:

initial which indicates the starting location,

cs which indicates the location where the process is in its critical section,

ncs which indicates the location where the process is not in its critical section.

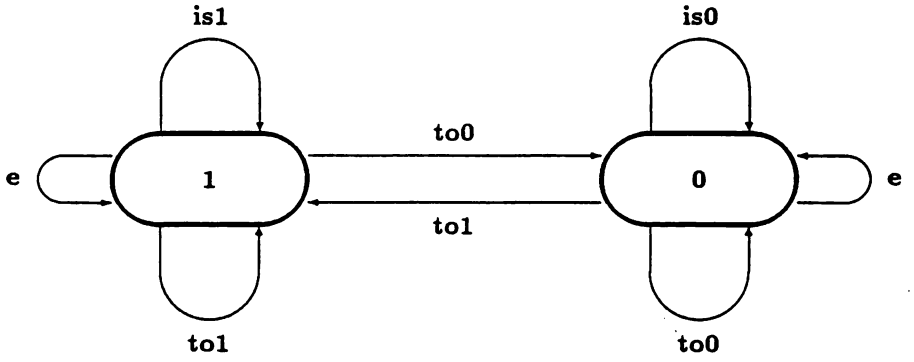


Figure 1: The graphical representation of the transition system for a boolean variable

```
transition_system  b  < width = 0 >;

0 |- e      -> 0 ,
     to0    -> 0 ,
     to1    -> 1 ,
     is0    -> 0 ;

1 |- e      -> 1 ,
     to0    -> 0 ,
     to1    -> 1 ,
     is1    -> 1 ;

< initial = { 0 }  >.
```

Figure 2: The transition system for a boolean variable in MEC

```
proc( me , other ) =

while true do
begin
{NCS}        0: ... ;
{mutexbegin}    flag[me] := 1 ;
             1: turn = me ;
             2: WAIT (flag[other] = 0  OR turn = other) ;
{CS}         3: ... ;
{mutexend}      flag[me] := 0 ;
end
```

Figure 3: The Peterson algorithm

```
transition_system  proc  < width = 0 >;

0  |- e                -> 0 ,
      my_flag_to_1     -> 1 <property=(mb)>;

1  |- e                -> 1 <property=(mb)>,
      turn_to_me       -> 2 <property=(mb)>;

2  |- e                -> 2 <property=(mb)>,
      is_other_flag_0  -> 3 <property=(mb)>,
      is_turn_other    -> 3 <property=(mb)>;

3  |- e                -> 3 ,
      my_flag_to_0     -> 0 ;

< initial = { 0 } ; cs = { 3 } ; ncs = { 0 } >.
```

Figure 4: The transition system for a process in MEC

There is also a transition parameter: it is the set of all transitions marked as having the property mb. These transitions are those executed by the processes when it tries to enter its critical section.

Indeed this transition system is obtained by interpreting the command

`WAIT( ... OR ... )`

in the following way: this command can be executed only if one of the two conditions is satisfied; in this case the execution of the process reaches the next location. It looks like idle waiting. Another interpretation of this command (busy waiting) could be: the process tests the first condition; if it is true it reaches the next location, otherwise it tests the second condition; if it is true it reaches the next location, otherwise it executes WAIT again. The transition system representing this interpretation is given in figure 5.

```
transition_system  procbis  < width = 0 >;

0     |- e                -> 0 ,
         my_flag_to_1     -> 1    <property=(mb)>;

1     |- e                -> 1    <property=(mb)>,
         turn_to_me       -> 2    <property=(mb)>;

2     |- e                -> 2    <property=(mb)>,
         is_other_flag_0  -> 3    <property=(mb)>,
         is_other_flag_1  -> 2bis <property=(mb)>;

2bis  |- e                -> 2bis <property=(mb)>,
         is_turn_other    -> 3    <property=(mb)>,
         is_turn_me       -> 2    <property=(mb)>;

3     |- e                -> 3 ,
         my_flag_to_0     -> 0 ;

< initial = { 0 } ; cs = { 3 } ; ncs = { 0 } >.
```

Figure 5: Another transition system for a process in MEC

In the sequel we will deal only with the first of these two transition systems.

1.3 Paths

A *path* of length $n > 0$ in a transition system $\mathcal{A} =< S, T, \alpha, \beta, \lambda >$ is a sequence $p = t_1, \ldots, t_n$ of transitions such that for all $i = 1, \ldots, n-1$, $\beta(t_i) = \alpha(t_{i+1})$. The *source* of this path, denoted by $\alpha(p)$, and its *target*, denoted by $\beta(p)$ are respectively $\alpha(t_1)$ and $\beta(t_n)$. The *label* of this path, denoted by $\lambda(p)$, is the word $\lambda(t_1) \cdots \lambda(t_n)$.

2 Synchronized systems

Let us consider n transition systems $\mathcal{A}_i$ over the alphabets A_i of actions, for $i = 1, \ldots, n$. Let us assume that these transition systems represent processes and shared objects constituting a system of interacting processes.(For simplicity, from now on, we shall also call processes the shared objects since they are also represented by transition systems). That means that some action in some process can be executed only simultaneously with some other action in some other process, or, on the opposite, cannot be executed simultaneously with some other action of some other process. Let us call *global action* a vector $< a_1, \ldots, a_n >$ where a_i belongs to A_i. Such a global action is executed when the actions a_i are simultaneously executed by the n processes. Thus the interactions between the processes of a system can be represented by the set of all global actions which are allowed to be executed and in [3], it is advocated that this kind of specification of the interactions between the processes of the processes of a system is general enough to formalise most of the concurrent systems of processes.

2.1 Synchronization constraints

As explained above, the interactions between the processes $\mathcal{A}_i$ of a system are represented by a subset I of $A_1 \times \cdots \times A_n$, called a *synchronization constraint*.

Example 3. Let us consider again Peterson's mutual exclusion algorithm for two processes. It is a represented by a system containing two transition systems `proc` described in figure 4 and three boolean variables `b` described in figure 2. The second line of figure 6 gives the list of the transition systems of this system. The other lines are the elements of the synchronization constraint we are going to comment.

These elements are just those we get when obeying the following rules. (Here we temporarily come back to the distinction between processes and variables).

1. We assume this system runs on a single processor; therefore the two processes cannot execute simultaneously a non null action; moreover the two processes cannot be idle simultaneously.

2. Each action performed by a process consists in setting or testing a variable. When a process executes such an action, the corresponding variable executes the corresponding action and the other variables execute the null action.

As to rule 1, consider the first two columns of the array of figure 6; the first (resp. second) column contains all the actions performed by the first (resp. second) process. In each line of this subarray there is one and only one null action (denoted by **e**). As to rule 2, consider the first line of the array: when the first process sets its flag (i.e. `flag[0]`) to **0**, then the first variable `b`, which represents `flag[0]`, executes the action "set to 0". In the third line it is the other process which sets its flag to **0** and it is the second variable, representing `flag[1]`, which executes the action "set to 0". In the same vein, the actions `turn_to_me` are executed simultaneously with an action by the third variable representing `turn`, the value to which this variable is set depending on the process which sets the variable. The test `is_other_flag_0` is executed simultaneously

with the action is0, the boolean variable executing this action beeing dependent on the process which performs this test. Also the action is0 or is1 is executed by the third boolean variable simultaneously with the test is_turn_other executed by the second or first process.

```
synchronization_system  peterson

< width = 5 ; list = (proc,proc,b,b,b) > ;

(my_flag_to_0     .e                .to0 .e   .e   ) ;
(my_flag_to_1     .e                .to1 .e   .e   ) ;
(e                .my_flag_to_0     .e   .to0 .e   ) ;
(e                .my_flag_to_1     .e   .to1 .e   ) ;
(turn_to_me       .e                .e   .e   .to0 ) ;
(e                .turn_to_me       .e   .e   .to1 ) ;
(is_other_flag_0  .e                .e   .is0 .e   ) ;
(e                .is_other_flag_0  .is0 .e   .e   ) ;
(is_turn_other    .e                .e   .e   .is1 ) ;
(e                .is_turn_other    .e   .e   .is0 ) .
```

Figure 6: The system representing Peterson's algorithm

2.2 Synchronized product

Given a vector $< \mathcal{A}_1, \ldots, \mathcal{A}_n >$ of transition systems, each $\mathcal{A}_i = < S_i, T_i, \alpha_i, \beta_i, \lambda_i >$ over the alphabet A_i, the *free product* of $< \mathcal{A}_1, \ldots, \mathcal{A}_n >$ is the transition system $< S, T, \alpha, \beta, \lambda >$ over $A_1 \times \cdots \times A_n$ defined by

$$\begin{aligned} S &= S_1 \times \cdots \times S_n, \\ T &= T_1 \times \cdots \times T_n, \\ \alpha(t_1, \ldots, t_n) &= < \alpha_1(t_1), \ldots, \alpha_n(t_n) >, \\ \beta(t_1, \ldots, t_n) &= < \beta_1(t_1), \ldots, \beta_n(t_n) >, \\ \lambda(t_1, \ldots, t_n) &= < \lambda_1(t_1), \ldots, \lambda_n(t_n) > . \end{aligned}$$

In some sense, the free product represents the evolution of the vector of transition systems when no constraint is set on the actions which can be performed simultaneously. In case of a synchronization constraint some transitions of this free product will never appear : those which are labelled by a vector of actions not allowed by the synchronization constraint. Hence we have the following definition.

Given a vector $< \mathcal{A}_1, \ldots, \mathcal{A}_n >$ of transition systems, each $\mathcal{A}_i$ over the alphabet A_i, and a synchronization constraint I included in $A_1 \times \cdots \times A_n$, the *synchronized product* of $< \mathcal{A}_1, \ldots, \mathcal{A}_n >$ with respect to I is the transition system $< S, T_I, \alpha, \beta, \lambda >$ over $A_1 \times \cdots \times A_n$ where

- $< S, T, \alpha, \beta, \lambda >$ is the free product of $\mathcal{A}_1, \ldots, \mathcal{A}_n$;
- T_I is the set of transitions $t =< t_1, \ldots, t_n >$ of T having their label $\lambda(t) = < \lambda_1(t_1), \ldots, \lambda_n(t_n) >$ in I.

Indeed the synchronized product computed by MEC is only a sub-transition system of the synchronized system defined above. Each transition system $\mathcal{A}_i = < S_i, T_i, \alpha_i, \beta_i, \lambda_i >$ which is a component of a synchronization system is assumed to have a parameter `initial`, as it is the case for the transition systems described in examples 1 and 2 (if this parameter is not mentioned, it is considered empty). This parameter defines a subset $initial_i$ of S_i and the parameter `initial` of the product is defined as the subset $initial = initial_1 \times \cdots \times initial_n$. The set of global states of the synchronized product of MEC is the set *Reach(initial)* of global states which can be reached from *initial*, i.e. the states of *initial* and the targets of paths having their sources in *initial*. The set of global transitions of the synchronized product of MEC is the set of global transitions having both their sources and their targets in *Reach(initial)*.

Example 4. The synchronized product, computed by MEC, of the synchronization system `peterson` given in figure 6 is given in figure 7. It was obtained by executing the MEC command `sync(peterson,res);` where `res` is the name given to the product.

3 Elementary computations

The general form of a computation command in MEC is

variable := expression;

the *expression* is evaluated and its value is assigned to the *variable*. The value of an expression is either a set of states or a set of transitions.

3.1 Set variables

Variables used by MEC are of two different sorts according to the kind of set they can be assigned. A variable is implicitely declared when it appears for the first time in the left hand part of an assignment command and its sort is the sort of the expression (if the sort of the expresssion can be unambiguously determined, otherwise the assignment is rejected). Every declared variable is displayed on the terminal with the number of objects (states or transitions) of its value. Parameters are considered as variables with an initial value.

Let us remark also that variables are local to a transition system. If several transition systems are under examination, the same variable (properly speaking, the same variable name) can be used (like `initial` in the previous examples) but it will have differents values according to the transition system with which it is associated (in other words a variable is a pair consisting in a variable name and a transition system).

```
transition_system res          < width =          5> ;
e(0.0.0.0.0) |- (e.my_flag_to_1.e.to1.e) -> e(0.1.0.1.0) ,
                (my_flag_to_1.e.to1.e.e) -> e(1.0.1.0.0) ;
e(1.0.1.0.0) |- (e.my_flag_to_1.e.to1.e) -> e(1.1.1.1.0) ,
                (turn_to_me.e.e.e.to0) ->   e(2.0.1.0.0) ;
e(2.0.1.0.0) |- (e.my_flag_to_1.e.to1.e) -> e(2.1.1.1.0) ,
                (is_other_fla.e.e.is0.e) -> e(3.0.1.0.0) ;
e(3.0.1.0.0) |- (e.my_flag_to_1.e.to1.e) -> e(3.1.1.1.0) ,
                (my_flag_to_0.e.to0.e.e) -> e(0.0.0.0.0) ;
e(0.1.0.1.0) |- (e.turn_to_me.e.e.to1) ->   e(0.2.0.1.1) ,
                (my_flag_to_1.e.to1.e.e) -> e(1.1.1.1.0) ;
e(1.1.1.1.0) |- (e.turn_to_me.e.e.to1) ->   e(1.2.1.1.1) ,
                (turn_to_me.e.e.e.to0) ->   e(2.1.1.1.0) ;
e(2.1.1.1.0) |- (e.turn_to_me.e.e.to1) ->   e(2.2.1.1.1) ;
e(3.1.1.1.0) |- (e.turn_to_me.e.e.to1) ->   e(3.2.1.1.1) ,
                (my_flag_to_0.e.to0.e.e) -> e(0.1.0.1.0) ;
e(2.2.1.1.0) |- (e.is_turn_othe.e.e.is0) -> e(2.3.1.1.0) ;
e(2.3.1.1.0) |- (e.my_flag_to_0.e.to0.e) -> e(2.0.1.0.0) ;
e(0.0.0.0.1) |- (e.my_flag_to_1.e.to1.e) -> e(0.1.0.1.1) ,
                (my_flag_to_1.e.to1.e.e) -> e(1.0.1.0.1) ;
e(1.0.1.0.1) |- (e.my_flag_to_1.e.to1.e) -> e(1.1.1.1.1) ,
                (turn_to_me.e.e.e.to0) ->   e(2.0.1.0.0) ;
e(0.1.0.1.1) |- (e.turn_to_me.e.e.to1) ->   e(0.2.0.1.1) ,
                (my_flag_to_1.e.to1.e.e) -> e(1.1.1.1.1) ;
e(0.2.0.1.1) |- (e.is_other_fla.is0.e.e) -> e(0.3.0.1.1) ,
                (my_flag_to_1.e.to1.e.e) -> e(1.2.1.1.1) ;
e(0.3.0.1.1) |- (e.my_flag_to_0.e.to0.e) -> e(0.0.0.0.1) ,
                (my_flag_to_1.e.to1.e.e) -> e(1.3.1.1.1) ;
e(1.1.1.1.1) |- (e.turn_to_me.e.e.to1) ->   e(1.2.1.1.1) ,
                (turn_to_me.e.e.e.to0) ->   e(2.1.1.1.0) ;
e(1.2.1.1.1) |- (turn_to_me.e.e.e.to0) ->   e(2.2.1.1.0) ;
e(2.2.1.1.1) |- (is_turn_othe.e.e.e.is1) -> e(3.2.1.1.1) ;
e(3.2.1.1.1) |- (my_flag_to_0.e.to0.e.e) -> e(0.2.0.1.1) ;
e(1.3.1.1.1) |- (e.my_flag_to_0.e.to0.e) -> e(1.0.1.0.1) ,
                (turn_to_me.e.e.e.to0) ->   e(2.3.1.1.0);
< initial = { e(0.0.0.0.0) } >.
```

Figure 7: A synchronized product

3.2 Expressions

Expressions are built up from variables and operators. Among these operators are set-theoretical (or boolean) operators union, intersection, and difference as well as the constants "empty", denoted by {}, and "all", denoted by *. Of course the use of these operators in expressions are submitted to sort restrictions.

Some others operators are primitive and will be described below. New operators can be defined by the users and this will be explained in the next section.

Finally there are some other ways to define sets of states and sets of transitions. We will not list all these ways here and we refer to the user manual [4].

Example 5. Let us consider the transition system `res` of figure 4 constructed by MEC, which will be our running example from now on.

First of all we want to know if the *mutual exclusion* property is verified, i.e. if the two processes can be or not both together in their critical section. Let us remind that the set of states in which a process is in its critical section is defined by the parameter `cs` as shown in figure 2. Therefore we have just to know whether there are (global) states in which

(*i*) the first component is in the value of `cs` in the first transition system of the synchronization system `peterson`,

(*ii*) the second component is in the value of `cs` in the second transition system of the synchronization system `peterson`.

The sets of states satisfying (*i*) and (*ii*) are respectively denoted by `cs[1]` and `cs[2]`; hence the set of states not satisfying the mutual exclusion property is defined and/or computed by the assignment `nok := cs[1]/\cs[2];`

After execution of this command, it appears on the screen that the value of `nok` is a set of states which has 0 element. □

3.3 Primitive operators

Let us denote by σ and τ the sorts "set of states" and "set of transitions". We can use the following operators `src` of sort $\tau \rightarrow \sigma$, `tgt` of sort $\tau \rightarrow \sigma$, `rsrc` of sort $\sigma \rightarrow \tau$, `rtgt` of sort $\sigma \rightarrow \tau$. The interpretation of these operators is as follows, for a given transition system $< S, T, \alpha, \beta, \lambda >$: If Q is a set of states included in S and R a set of transitions included in T, then

$$\begin{aligned} \texttt{src}(R) &= \{\alpha(t) \mid t \in R\}, \\ \texttt{tgt}(R) &= \{\beta(t) \mid t \in R\}, \\ \texttt{rsrc}(Q) &= \{t \mid \alpha(t) \in Q\}, \\ \texttt{rtgt}(Q) &= \{t \mid \beta(t) \in Q\}. \end{aligned}$$

In other words $\texttt{src}(R)$ and $\texttt{tgt}(R)$ are respectively the sets of sources and targets of transitions in R and $\texttt{rsrc}(Q)$ and $\texttt{rtgt}(Q)$ are their reciprocals : the sets of transitions having their source and their target in Q.

Example 6. If Q is a set of states, the set *Pred(Q)* is the set of all states which are the source of a transition whose target is in Q. If `Q` is a variable whose value is Q, *Pred(Q)* is the value of the expression `src(rtgt(Q))`.

In a similar way *Succ(Q)* is the value of the expression `tgt(rsrc(Q))`.

If `T` denotes a set T of transitions, the expression `src(T/\rtgt(Q))` evaluates to the set of sources of transitions in T having their target in Q and `tgt(T/\rsrc(Q))` to the set of targets of transitions in T having their source in Q. □

We also use the binary operator `loop` of sort $\tau\tau \rightarrow \tau$ defined by $t \in$ `loop`(R, R') if and only if t belongs to a path p such that

(i) the source of p is equal to its target,

(ii) every transition of p is in R',

(iii) some transition of p is in R,

In other words, a transition t is in `loop`(R, R') if it belongs to some loop in R' containing some transition in R.

Example 7. Let us now look for a *livelock* in the transition system `res`.

Roughly speaking there is a livelock if there is an infinite execution where

(i) both processes are always in their "mutexbegin", and

(ii) none of the processes remains "inactive" forever.

Condition (i) means that both processes are trying to enter their critical section and never succeed; condition (ii) means that both processes are really trying to enter their critical section : if one of the processes stays idle forever surely it will not enter its critical section and could even prevent the other process to enter.

Since the only waiting action is denoted by `e`, a process is active during a transition t if the corresponding component of the label of this transition is not equal to `e`. Therefore the sets of transitions in which the first and the second processes are active are computed by

```
active1 := !label[1] # "e"; and
active2 := !label[2] # "e";
```

The set of transitions in which both processes are in their "mutexbegin" is computed by

```
ll := mb[1]/\mb[2];
```

If there is a livelock, there is an infinite path p such that

(ia) all its transitions are in `ll`;

(iia) it has an infinite number of transitions in `active1`;

($iiia$) it has an infinite number of transitions in `active2`.

Since there is a finite number of states this infinite path p contains a loop p' which satisfies

(ib) all its transitions are in `l1`;

(iib) it has at least one transition in `active1`;

($iiib$) it has at least one transition in `active2`.

Conversely if there is a loop satisfying (ib,iib,$iiib$), there exists an infinite path satisfying (ia,iia,$iiia$). Hence there is a livelock if and only if there is a loop satisfying (ib,iib,$iiib$).

The set `l10` of transitions belonging to a loop satisfying (ib) is computed by

```
l10 := loop(*,l1);
```

The set `l11` of transitions belonging to a loop satisfying (ib) and (iib) is computed by

```
l11 := loop(active1,l10);
```

(but also by `l11 := loop(active1,l1);`

Finally the set `l12` of transitions belonging to a loop satisfying (ib),(iib) and ($iiib$) is computed by

```
l12 := loop(active2,l11);
```

Here again this set is empty. □

4 The definition of new operators

4.1 Preliminary examples

Let us consider some given transition system $\mathcal{A}$.

1. Let us consider the set *Reach(initial)* which was used in the definition of the synchronized product (cf. 2.2). Indeed for every set Q of states one can define the set *Reach(Q)* containing Q and the targets of paths having their source in Q. Thus *Reach* can be considered as an operator of sort $\sigma \to \sigma$ and one can think of adding it to the primitive operators.

But we can also remark that this operator can be formally defined in the following way.

Let us consider the operator *Succ* defined in example 6. We have the following equality:

$$Reach(Q) = Q \cup Succ(Reach(Q)) \tag{1}$$

By definition $Q \subset Reach(Q))$, and if there is a path from a state s to a state s', there is also a path from s to every state of $Succ(\{s'\})$, thus $Succ(Reach(Q)) \subset Reach(Q)$. Conversely let $Q_0 = Q$ and Q_n, for $n > 0$, be the set of targets of paths of length n having their source in Q; then it is clear that $Reach(Q) = \bigcup_{n\geq 0} Q_n$ and that $Q_{n+1} = Succ(Q_n)$, hence $Reach(Q) = Q_0 \cup \bigcup_{n\geq 0} Succ(Q_n) \subset Q \cup Succ(Reach(Q))$.

We have even more: not only *Reach(Q)* is a solution of the equation

$$X = Q \cup Succ(X) \tag{2}$$

but it is the least set of states (for inclusion) satisfying this equation: if X is a set of states satisfying the equation 2 then $Q_0 = Q \subset X$ and if $Q_n \subset X$ then $Q_{n+1} = Succ(Q_n) \subset Succ(X) \subset X$.

Therefore we can give *Reach* the following definition: for every set Q of states, *Reach(Q)* is the least solution of 2. This definition is expressed in MEC by

```
function reach(Q:state) return X:state;
begin
X = Q \/ tgt(rsrc(X))
end.
```

Once this function is defined, the operator **reach** can be used in expressions exactly like primitive operators; its sort is $\sigma \rightarrow \sigma$, as expressed by the first line of the definition.

2. Let us now consider the two sets *ReachEven(Q)* and *ReachOdd(Q)* of states reachable from a state of Q by a path of even or of odd length. We have

$$\begin{aligned} ReachEven(Q) &= \bigcup_{n \geq 0} Q_{2n} \\ ReachOdd(Q) &= \bigcup_{n \geq 0} Q_{2n+1} \end{aligned}$$

where the sets Q_n are defined above. Then

$$\begin{aligned} ReachEven(Q) &= Q \cup Succ(ReachOdd(Q)) \\ ReachOdd(Q) &= Succ(ReachEven(Q)) \end{aligned}$$

and the pair $< ReachEven(Q), ReachOdd(Q) >$ is the least pair $< X, Y >$ of sets of states satisfying

$$\begin{aligned} X &= Q \cup Succ(Y) \\ Y &= Succ(X) \end{aligned}$$

Thus we can define the operator reacheven of sort $\sigma \rightarrow \sigma$ by

```
function reacheven(Q:state) return X:state;
var Y:state
begin
X = Q \/ tgt(rsrc(Y));
Y = tgt(src(X))
end.
```

It could be noticed that the auxiliary variable Y is not really useful since this definition is equivalent to

```
function reacheven(Q:state) return X:state;
begin
X = Q \/ tgt(rsrc(tgt(src(X))));
end.
```

Here is an example where the definition cannot be simplified this way.

3. Let us consider some set R of transitions; we denote by $F(R,Q)$ (resp. $G(R,Q)$) the set of states reachable from Q by a path containing an even (resp. odd) number of transitions in R. Let us remark that for any set Z_t of transitions and any set Z_s of states, the set $tgt(Z_t \cap rsrc(Z_s))$, denoted by $Succ(Z_t, Z_s)$, is the set of states reachable from Z_s by one transition in Z_t. Thus $< F(R,Q), G(R,Q) >$ is the least pair $< X, Y >$ satisfying

$$\begin{aligned} X &= Q \cup Succ(* - R, X) \cup Succ(R, Y) \\ Y &= Succ(* - R, Y) \cup Succ(R, X) \end{aligned}$$

where $* - R$ is the complement of R. This definition is expressed in MEC by

```
function F(R:trans , Q:state) return X:state;
var Y :state
begin
X = Q \/ tgt((* - R)/\rsrc(X)) \/ tgt((R/\rsrc(Y));
Y = tgt((* - R)/\rsrc(Y)) \/ tgt((R/\rsrc(X))
end.
```

4.2 Systems of equations

We are now going to formally define the systems of equations which can be used to define new operators in MEC.

Basic operators Let D be the heterogenous algebraic signature with two sorts, σ and τ, containing the following operators [7] :

0_σ, 1_σ, 0_τ, 1_τ : constants of sorts σ and τ;

$\cup_\sigma$, $\cap_\sigma$, $-_\sigma$: binary operators of sort $\sigma\sigma \longrightarrow \sigma$;

$\cup_\tau$, $\cap_\tau$, $-_\tau$: binary operators of sort $\tau\tau \longrightarrow \tau$;

src, *tgt* : binary operators of sort $\tau \longrightarrow \sigma$;

rsrc, *rtgt* : binary operators of sort $\sigma \longrightarrow \tau$.

If $\mathcal{A} =< S, T, \alpha, \beta, \lambda >$ is a transition system, it is given a D-structure in the following way :

- The set of elements of sort σ (resp. τ) is $\wp(S)$ (resp. $\wp(T)$).

- The interpretation of the operators is

$$\begin{aligned}
(0_\sigma)_{\mathcal{A}} &= \emptyset, \\
(1_\sigma)_{\mathcal{A}} &= S, \\
(0_\tau)_{\mathcal{A}} &= \emptyset, \\
(1_\tau)_{\mathcal{A}} &= T, \\
P(\cup_\sigma)_{\mathcal{A}}P' &= P \cup P', \\
P(\cap_\sigma)_{\mathcal{A}}P' &= P \cap P', \\
P(-_\sigma)_{\mathcal{A}}P' &= P - P', \\
R(\cup_\tau)_{\mathcal{A}}R' &= R \cup R', \\
R(\cap_\tau)_{\mathcal{A}}R' &= R \cap R', \\
R(-_\tau)_{\mathcal{A}}R' &= R - R', \\
src_{\mathcal{A}}(R) &= \{\alpha(t) | t \in R\}, \\
tgt_{\mathcal{A}}(R) &= \{\beta(t) | t \in R\}, \\
rsrc_{\mathcal{A}}(P) &= \{t | \alpha(t) \in P\}, \\
rtgt_{\mathcal{A}}(P) &= \{t | \beta(t) \in P\}.
\end{aligned}$$

All these operators, but $-_\sigma$ and $-_\tau$, have monotonic interpretations with respect to set inclusion.

Signed terms Let $X_n = \{x_1, \ldots, x_n\}$ and $Y_m = \{y_1, \ldots, y_m\}$ be two sets of variables of sort σ and τ. We can build terms with these variables and the operators of D. If t is such a term, its interpretation $t_{\mathcal{A}}$ will be a mapping from $\wp(S)^n \times \wp(T)^m$ into $\wp(S)$ or $\wp(T)$ according to the sort of this term.

Since the interpretation of a difference operator is not monotonic, the interpretation of a term is not necessarily monotonic. However we can consider that the interpretation of a difference becomes monotonic if its first argument is ordered by inclusion and its second argument is ordered by the inverse relation : containment.

This led us to consider two kinds of order on $\wp(S)$ and $\wp(T)$, inclusion and containment. We shall denote these powersets by $\wp^+(S)$ and $\wp^+(T)$ (resp. $\wp^-(S)$ and $\wp^-(T)$) when we wish to make clear that they are ordered by inclusion (resp. containment). For each sort, we consider two kinds of variables : positive variables ranging over a powerset ordered by inclusion and negative variables ranging over a powerset ordered by containment. Let X^+ and X^- be two sets of positive and negative variables of sort σ, Y^+ and Y^- two sets of positive and negative variables of sort τ and Z_σ et Z_τ two set of "parameters", which will be interpreted as arbitrary sets.

We inductively define the sets of positive and negative terms of sort σ, $\mathcal{T}_\sigma^+$ et $\mathcal{T}_\sigma^-$, and of positive and negative terms of τ, $\mathcal{T}_\tau^+$ and $\mathcal{T}_\tau^-$ by

- $X^+ \subset \mathcal{T}_\sigma^+$, $X^- \subset \mathcal{T}_\sigma^-$, $Y^+ \subset \mathcal{T}_\tau^+$, $Y^- \subset \mathcal{T}_\tau^-$;
- $\{0_\rho, 1_\rho\} \cup Z_\rho \subset \mathcal{T}_\rho^+ \cap \mathcal{T}_\rho^-$; $(\rho = \sigma, \tau)$;

- if t_1 and t_2 belong to $\mathcal{T}_\rho^\varsigma$ then $t_1 \cup_\rho t_2$, $t_1 \cap_\rho t_2$ belong to $\mathcal{T}_\rho^\varsigma$; $(\rho = \sigma, \tau;\ \varsigma = +, -)$;
- if t belongs to $\mathcal{T}_\tau^\varsigma$ then $src(t)$ and $tgt(t)$ belong to $\mathcal{T}_\sigma^\varsigma$; $(\varsigma = +, -)$;
- if t belongs to $\mathcal{T}_\sigma^\varsigma$ then $rsrc(t)$ and $rtgt(t)$ belong to $\mathcal{T}_\tau^\varsigma$; $(\varsigma = +, -)$;
- if t_1 belongs to $\mathcal{T}_\rho^\varsigma$ and t_2 belongs to $\mathcal{T}_\rho^{\varsigma'}$ then $t_1 -_\rho t_2$ belongs to $\mathcal{T}_\rho^\varsigma$; $(\rho = \sigma, \tau;$ $< \varsigma, \varsigma' >=< +, - >,\ < -, + >)$;

If t is a term of $\mathcal{T}_\rho^\varsigma$ its interpretation $t_\mathcal{A}$ is then monotonic or antimonotonic (according to the value of ς) when the values of variables in Z are fixed and values of other variables are ordered according to their sign.

Example 8. If Z_τ is a parameter of sort τ, Z_σ a parameter of sort σ, X_+ a positive variable of sort σ and Y_- a negative variable of sort τ, then $Z_\tau \cap_\tau rtgt(1_\sigma -_\sigma X_+)$ is a negative term of sort τ and $Z_\sigma \cup_\sigma (1_\tau -_\tau src(Y_-))$ is a positive term of sort σ. □

Systems of equations Let us consider the following sets of variables :

$$\begin{aligned}
\mathbf{X}^+ &= \{X_1^+, \ldots, X_n^+\},\\
\mathbf{X}^- &= \{X_1^-, \ldots, X_{n'}^-\},\\
\mathbf{Y}^+ &= \{Y_1^+, \ldots, Y_m^+\},\\
\mathbf{Y}^- &= \{Y_1^-, \ldots, Y_{m'}^-\},\\
\mathbf{Z}_\sigma &= \{Z_1, \ldots, Z_p\},\\
\mathbf{Z}_\tau &= \{Z'_1, \ldots, Z'_{p'}\},
\end{aligned}$$

We consider the sets of terms $\mathcal{T}_\sigma^+$, $\mathcal{T}_\sigma^-$, $\mathcal{T}_\tau^+$, $\mathcal{T}_\tau^-$ built with these variables.

A system of equations Σ is

$$\begin{aligned}
\{X_i^+ &= u_i^+ | 1 \leq i \leq n\} \ \cup\\
\{X_i^- &= u_i^- | 1 \leq i \leq n'\} \ \cup\\
\{Y_i^+ &= v_i^+ | 1 \leq i \leq m\} \ \cup\\
\{Y_i^- &= v_i^- | 1 \leq i \leq m'\}.
\end{aligned}$$

where $u_i^+ \in \mathcal{T}_\sigma^+$, $u_i^- \in \mathcal{T}_\sigma^-$, $v_i^+ \in \mathcal{T}_\tau^+$, $v_i^- \in \mathcal{T}_\tau^-$.

With a system of equations Σ and a transition system $\mathcal{A}$ we associate the ordered set

$$D_1 = \wp^+(S)^n \times \wp^-(S)^{n'} \times \wp^+(T)^m \times \wp^-(T)^{m'}$$

and the set

$$D_2 = \wp(S)^p \times \wp(T)^{p'}$$

ordered by the empty order. Then Σ defines a mapping

$$\Sigma_{\mathcal{A}} : D_1 \times D_2 \longrightarrow D_1$$

This mapping is monotonic with respect to the order defined componentwise on D_1 and $D_1 \times D_2$. Thus it has a least fixed point $\mu\Sigma_{\mathcal{A}} : D_2 \longrightarrow D_1$.

Now if we choose one of the signed variables, by composing $\mu\Sigma_{\mathcal{A}}$ with the projection of D_1 on its component associated with this variable, we get a mapping from D_2 in $\wp(S)$ or $\wp(T)$, according to the sort of the variable. Thus we can assume that a new operator is defined by :

- the list of sorted parameters,
- the list of sorted and signed variables,
- the selected variable defining the result,
- the list of equations, one for each variables.

The interpretation of such an operator in any transition system will be the mapping defined above.

Example 9. Let us consider the two terms of the example 8. The parameters they contains are Z_τ and Z_σ, and the variables are X_+ and Y_-. Let us consider the two equations

$$\begin{aligned} X_+ &= Z_\sigma \cup_\sigma (1_\tau -_\tau src(Y_-)) \\ Y_- &= Z_\tau \cap_\tau rtgt(1_\sigma -_\sigma X_+) \end{aligned}$$

For every transition system $\mathcal{A}$ this defines a mapping from $\wp(T)\times\wp(S)$ in $\wp(S)\times\wp(T)$; if we choose X_+ as principal variable we get a mapping from $\wp(T) \times \wp(S)$ in $\wp(S)$.

Let us call `unavoidable` this operator for some reasons which will be explained below. In MEC its definition will be written

```
function unavoidable(Zt:trans ; Zs:state) return X:state;
var Y:_trans
begin
X = Zs \/ (* - src(Y));
Y = Zt /\ rtgt(* - X)
end.
```

Let us remark that negative variables are specified by an "underscore" preceding their sort.

The name "unavoidable" given to this operator comes from the following property.

Let $\mathcal{A}$ be any transition system, Q some set of states and R some set of transitions. Then a state s belongs to unavoidable(R, Q) if and only if every maximal path in $\mathcal{A}$ (i.e. an infinite path or a finite path whose last state has no successor in $\mathcal{A}$) originated in s and containing only transitions in R contains a state in Q.

To prove this assertion let us remark that unavoidable(R, Q) is also the least solution of $X = Q \cup (* - Pred(R, * - X))$ where $Pred(A, B) = src(A \cap rtgt(B))$ is the set of origins of transition of A having their target in B. Thus $s \in X$ if and only if $s \in Q$ or all transitions of R of source s have their target in X.

In particular unavoidable$(T, \emptyset)$ is the set of states s such that every maximal path originated in s is finite. This can be considered as a definition of "deadlocking" states, since once in such a state it is impossible to start an infinite computation.

Least and greatest fixed points The use of signed variables allows to define new operators as greatest fixed point of system of equations as well as least fixed point just by playing on the signs : the least element for inclusion is the greatest one for containment and vice-versa!

Computation of least fixed points If an expression contains an operator defined by a system of equation it remains to evaluate this expression. This amounts to computing the value of $op(Z_1, \ldots Z_n)$ when the values of $Z_1, \ldots, Z_n$ are known and thus to computing the least fixed point of the equations defining op when the parameters occuring in these equations are given. This can be done in time linear with respect to the size of the transition system (i.e. number of states and number of transitions, using an algorithm described in [2]). Thus all computations performed by MEC are done in a time linear with the size of the transition system.

5 An example of use

MEC has been used to check some mutual exclusion algorithms. It allowed to discover that Burns's algorithms [5] contained livelocks in the case of four processes.

The transition system obtained by synchronizing four processes, four boolean flags and a "turn" variable, representing the symmetrical Burns's algorithm with four processes has 65 016 states, 260 064 transitions stored in about 13 Mbytes of memory. On a Sun 3/60, it takes 20 minutes of CPU to construct this product and 11 minutes to compute unavoidable using the linear algorithm of Arnold and Crubillé.

References

[1] A. Arnold. Transition systems and concurrent processes. In *Mathematical problems in Computation theory (Banach Center Publications,vol. 21)*, 1987.

[2] A. Arnold and P. Crubillé. A linear algorithm to solve fixed point equations on transition systems. *Inf. Process. Lett.*, 29:57–66, 1988.

[3] A. Arnold and M. Nivat. Comportements de processus. In *Colloque AFCET "Les Mathématiques de l'Informatique"*, pages 35–68, 1982.

[4] D. Bégay. *Mode d'emploi MEC.* Technical Report I-8915, Université Bordeaux I, 1989.

[5] J. E. Burns. Symmetry in systems of asynchronous processes. In *Proc. 22nd Annual Symp. on Foundations of Computer Science*, pages 169–174, 1981.

[6] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent systems using temporal logics specifications. *ACM Trans. Prog. Lang. Syst.*, 8:244–263, 1986.

[7] A. Dicky. An algebraic and algorithmic method for analysing transition systems. *Theoretical Comput. Sci.*, 46:285–303, 1986.

[8] E. A. Emerson and E. C. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In J. de Bakker and J. van Leeuwen, editors, *7th Int. Coll. on Automata, Languages and Programming*, pages 169–181, Lect. Notes. Comput. Sci. 85, 1980.

[9] E. A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional μ-calculus. In *Symp. on Logic in Comput. Sci.*, pages 267–278, 1986.

[10] R. M. Keller. Formal verification of parallel programs. *Commun. ACM*, 19:371–384, 1976.

[11] D. Kozen. Results on the propositional μ-calculus. *Theoretical Comput. Sci.*, 27:333–354, 1983.

[12] M. Nivat. Sur la synchronisation des processus. *Revue Technique Thomson-CSF*, 11:899–919, 1979.

[13] V. Pratt. A decidable μ-calculus. In *Proc. 22nd Symp. on Foundations of Comput. Sci.*, pages 421–427, 1981.

[14] J.-P. Queille. *Le système CESAR: Description, spécification et analyse des applications réparties.* PhD thesis, I.N.P., Grenoble, 1982.

[15] J. Sifakis. *Global and local invariants in transition systems.* Technical Report 274, IMAG, Grenoble, 1981.

[16] R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM J. Comput.*, 1:146–160, 1972.

Processeur parallèle pour les systèmes experts et les algorithmes de règles

Edward C. Valentine

Concordia University,Montréal,Qc,Canada,
Collège Montmorency, Laval.

Résumé: Lors de l'exécution d'un système expert de règles, la plus grande partie du temps est utilisé pour fouiller la base de connaissance dans le but de trouver les règles susceptibles d'être déclenchées. Gupta [G86] a montré que l'utilisation du parallélisme classique ne permet guère d'espérer mieux qu'une accélération par un facteur de l'ordre de 30. Nous proposons un nouveau type de processeur permettant d'une part d'éliminer cette recherche et d'autre part de déclencher plusieurs règles en parallèle.

Les systèmes experts sont, en général, développés comme des ensembles de règles ayant la forme (1). Ces ensembles de règles sont parfois appelés "systèmes de productions" (production systems), sans doute à cause d'une certaine analogie de présentation avec les "productions" dans les grammaires des langages formels.

$$A \wedge B \wedge \sim F \Rightarrow G \wedge \sim C \qquad (1)$$

Si on restreint la notion de système expert aux systèmes qui utilisent la connaissance d'experts humains, les autres applications où l'utilisation de systèmes de règles permet des mises en œuvre efficaces (systèmes de contrôle [AAA],[SL88], traitement d'images) les termes "algorithme de règles" (rule-based algorithm) semblent préférables pour désigner ces ensembles de règles.

Un algorithme de règles (voir [CE] pour une définition plus formelle et [CF] pour un exposé détaillé) est constitué d'un ensemble de règles de la forme (1), c'est à dire: une conjonction de

faits (ou leur négation) appelée la prémisse (condition, hypothèse, partie gauche, LHS, antécédant, assumption...), suivie d'un symbole " -> " , suivi d'une conjonction de faits (ou leur négation) appelée la conclusion (conséquent, déduction, partie droite, RHS ...). Les faits prennent une valeur de vérité dans l'ensemble {vrai, faux, inconnu}. Certaines de ces valeurs sont introduites comme données et forment l'ensemble des valeurs en mémoire de travail.

La figure 1 donne le paradigme de fonctionnement d'un tel algorithme. La première étape du cycle, appelée MATCH, consiste à comparer les valeurs en mémoire de travail avec les faits composant les prémisses de règles. Lorsque tous les faits de la prémisse d'une règle correspondent a des valeurs en mémoire de travail, cette règle est placée dans un ensemble appelé "ensemble de conflit" qui est l'ensemble des règles "déclenchables" de ce cycle. La deuxième étape du cycle, appelée CHOIX (conflict resolution) consiste à décider laquelle(lesquelles) des règle de l'ensemble de conflit déclencher (un ensemble vide entraîne l'arrêt). La troisième étape, appelée ACTION (act), consiste à utiliser les conclusions de la (des) règle(s) déclenchée(s) pour modifier les valeurs correspondantes de la mémoire de travail.

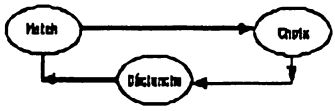


figure1

Dès les premières mises en œuvre de systèmes de règles, il fut évident que la complexité des calculs rendaient les 'gros systèmes' inéficaces. Le traitement parallèle s'imposa donc comme une solution possible.Les exemples classiques d'algorithmes de règles, les systèmes experts en particulier, ne permettent pas en général le déclenchement de plusieurs règles en parallèle. Le parallélisme proposé consiste donc à faire la première étape du cycle en parallèle en utilisant le paradigme de la figure 2. En effet Forgy a montré [Forgy79] [Forgy 82] que cette étape occupe plus de 90% du temps utilisé par le processeur et on s'attend que toute amélioration par un facteur important du temps processeur à cette étape entraînera une amélioration comparable de la performance du système entier. Cette attente n'est pas comblée dans la pratique comme le montre Gupta [G86] par des simulations. La conclusion qu'il tire de ses expériences est que le déclenchement d'une règle entrainant très peu de changements dans la mémoire de travail, l'amélioration de performance obtenue par cette forme de parallélisme plafonne rapidement. On trouve également dans cette thèse une comparaison de plusieurs architectures parallèles pour

l'exécution de systèmes de règles (DADO, NON_VON, etc...).

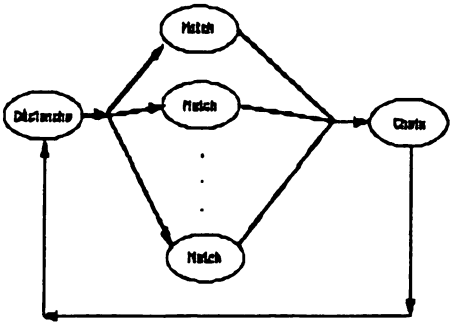


figure2

Dans [TS84] et [TS85] on trouve l'idée de l'exécution parallèle (déclenchement parallèle) des règles. Naturellement on ne saurait passer de l'exécution séquentielle à l'exécution parallèle sans apporter certaines modifications aux algorithmes. Dans [MJV86] on présente une façon de représenter les règles d'un système expert afin d'améliorer, d'une part, la performance du moteur d'inférence, et d'autre part la lisibilité et l'interface usager. Tirée d'une méthodologie de développement de système plus générale utilisant des tableaux pour représenter des relations [JFO], mais déjà présente d'une façon plus primitive dans le système IRIS [Trigoboff] [TK], cette approche suggère une architecture du type "massivement parallèle" pour le traitement des règles. La figure 3 montre un ensemble de règles du type (1) qui utilise cette représentation.

	{REGLES}
v	R1 A -> B ^ C
. v	R2 C -> E
. . v	R3 B ^ D -> F
. . . v . . .	R4 E -> H
. . . . v . .	R5 B ^ F -> ~G
. . . . . v .	R6 C -> D
. v	R7 G -> B ^ D
	{FAITS}
t	C1 A
T . t . t . T .	C2 B
T t . . . t . .	C3 C
. . t . . T T .	C4 D
. T . t . . .	C5 E
. . T . t . . .	C6 F
. . . . F . t .	C7 G
. . . T	C8 H

figure 3

La partie supérieure du tableau permet d'écrire les règles et d'établir une correspondance entre celle-ci et les colonnes de la partie inférieure. La partie inférieure permet d'établir les relations entre les règles et les faits. Un 't' minuscule (resp. 'f') à l'intersection de la ligne i et de la colonne j indique que le fait i est une prémisse vraie (resp. fausse) de la règle j. Un 'T' majuscule (resp 'F') à l'intersection de la ligne i et de la colonne j indique que le fait i est une conclusion vraie (resp. fausse) de la règle j. On voit qu'il n'est pas nécessaire comme dans [] de séparer les prémisses et les conclusions en deux tableaux, puisqu'un fait n'apparait pas en prémisse et en conclusion d'une même règle.

La figure 4 montre comment la performance du système peut être améliorée en permettant au moteur d'inférence de tirer des conclusions des règles déjà entrées (traitement à l'avance, pre-processing) pendant la phase de développement du système. On voit (colonne 1) que si la valeur 'vrai' est attribuée au fait 'A' alors les valeurs de tous les autres faits se trouvent déterminés immédiatement. On remarque aussi (colonne 7) que l'attribution de la valeur 'vrai' au fait 'G' entraine l'obligation de déduire la valeur 'faux' pour ce même fait et donc que ce traitement à l'avance permet de déterminer que l'ensemble de règles est inconsistant.

	{REGLES}	
v	R1	A -> B ^ C
. v	R2	C -> E
. . v	R3	B ^ D -> F
. . . v . . .	R4	E -> H
. . . . v . . .	R5	B ^ F -> ~G
. . . . . v . .	R6	C -> D
. . . . . v .	R7	G -> B ^ D
	{FAITS}	
t	C1	A
T . t . t . T .	C2	B
T t . . . t . .	C3	C
+T . t . . T T .	C4	D
+T T . t	C5	E
+T . T . t . +T .	C6	F
+F . +F . F . t+F.	C7	G
+T +T . T	C8	H

figure 4

Cette représentation permet de concevoir une architecture massivement parallèle pour l'exécution d'un tel système de règle. La figure 5 représente un réseau de portes logiques interconnectées où nous supposerons que les lignes représentent les faits et les colonnes, les règles. Chaque porte peut être programmée de 5 façons (ou prendre l'un des 5 états) soit:

prémisse vraie, prémisse fausse, conclusion vraie, conclusion fausse, ou non active.

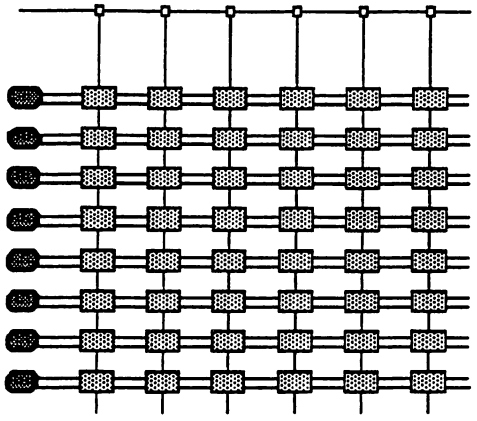


figure 5

Le processeur fonctionne de la façon suivante. Lorsqu'un signal circule dans une colonne, on dira que la règle qui y est programmée déclenche; un signal (vrai ou faux) dans une ligne établit la valeur de vérité de ce fait. Ce signal peut être reçu de l'extérieur, c'est à dire que certaines ou toutes les lignes de faits d'un processeur peuvent accepter des valeur d'entrée. De la même façon certains ou tous les faits peuvent émettre leur signal vers l'extérieur. Les faits faisant partie de la prémisse d'une règle seront programmés en prémisse vraie ou prémisse fausse; ceux de la conclusion en conclusion vraie ou conclusion fausse. Puisqu'on désire qu'une règle ne déclenche que lorsque que ses prémisses sont vérifiées, les portes logiques auront la fonction d'empêcher ce déclenchement lorsque le signal sur une ligne est différent de la valeur programmée en prémisse d'une règle. Autrement dit, une porte logique qui se trouve à l'intersection du fait A et de la règle R1 et programmée en prémisse vraie (resp. fausse) coupera le signal dans R1 à moins que le signal sur A ne soit vrai (resp faux). D'autre part, il faut que lorsqu'une règle déclenche, tous les faits faisant partie de sa conclusion prennent les valeurs de vérité appropriées.Pour cela les portes logiques programmées à conclusion vraie (resp. fausse) doivent détecter le déclenchement de leur règle (passage d'un signal) et émettre un signal vrai (resp. faux) sur leur ligne de fait. Un exemple permettra d'illustrer: prenons l'ensemble de règles (2)

Règle 1 : $F_1 \wedge F_3 \rightarrow F_2 \wedge F_4$
Règle 2 : $F_2 \rightarrow F_5$
Règle 3 : $F_4 \rightarrow \sim F_6$
Règle 4 : $F_5 \wedge \sim F_6 \rightarrow F_7$
Règle 5 : $F_2 \wedge \sim F_5 \rightarrow F_1$ (2)

La figure 6 illustre le processeur lorsque ces règles y sont programmées. Supposons que des

signaux externes soient reçus: vrai pour F1 et vrai pour F3. Aussitôt que ces deux signaux sont reçus, les portes à l'intersection de la colonne 1 du processeur qui correspond à la règle 1 et des lignes 1 et 3, qui correspondent aux faits F1 et F3, qui inhibaient le signal sur la colonne, le laissent passer déclenchant ainsi la règle. Les portes à l'intersection de la colonne 1 et des lignes 2 et 4 détectent ce signal et initient des signaux vrai sur les lignes de fait F2 et F4.

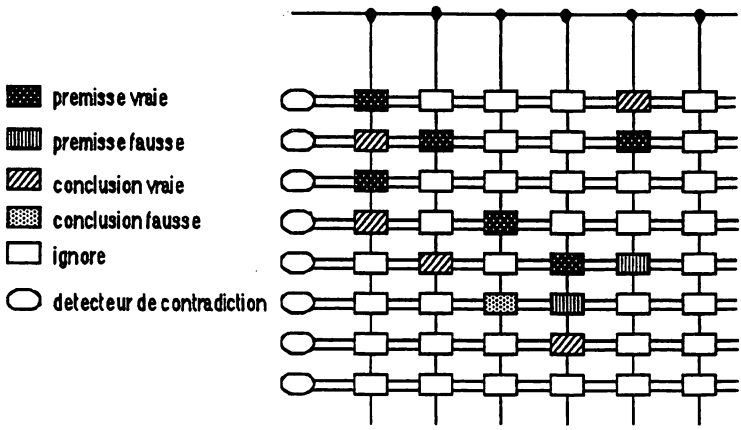


figure 6

Ces signaux correspondent aux prémisses des règles 2 et 3 (colonnes 2 et 3) qui déclenchent et initient des signaux vrai pour F5 et faux pour F6. Ces signaux permettent à leur tour le déclenchement de la règle 4 et entraîne la valeur vrai pour F7. 'F7= vrai' pourrait être considéré comme le résultat final du raisonnement; cependant n'importe laquelle des valeurs des faits peut être considérée comme tel et peut donc correspondre à une valeur de sortie du processeur.

On observe que le déclenchement des règles se fait en parallèle et que la partie MATCH du paradigme de la figure 1 se fait aussi en parallèle sur toutes les prémisses de toutes les règles dont la valeur de vérité est connue. Le processeur programmé devient un circuit analogique massivement parallèle pour cet ensemble de règles.

La technologie actuelle [Mead 87] [Mead 89] est adéquate pour réaliser ce genre de processeur que l'on peut placer dans la classe des architectures de type neuronal [Alexander 89] [Grossberg].

références:

[Alexander 89] **Alexander, I., et al.,** Neural computing architectures. MIT Press, 1989, I. Alexander editor.

[ÅAÅ86] **Åstrom, K.J., Anton, J.J., Årzén K.E.** "Expert Control", Automatica vol. 22, no 3, 276-286,1986.

[CE] **Cauhape, D., Ermine, J-L.,** Validation of a Knowledge Base in an Expert System,

[CF] **Cohen, P. R., Feigenbaum, E. A.,** "The Handbook of Artificial Intelligence vol 1-3", William Kaufman Inc., Los Altos, California, 1982.

[Forgy79] **Forgy, C.L.,** On the efficient implementation of Production Systems, Dept of Computer Science, Carnegie Mellon University, Ph.D. Thesis, 1979.

[Forgy82] **Forgy, C.L.,** RETE:A Fast Algorithm for the Many Pattern Many Object Pattern Match Problem, Artificial Intelligence 19 (1982) 17-37.

[G86] **Gupta, Anoop.** "Parallelism in Production Systems." Ph.D. dissertation. Carnegie-Mellon University, Pittsburgh,PA, March 1986.

[Grossberg] **Grossberg, S.,** Neural Networks and Natural Intelligence,The MIT Press,1988.

[JFO] **Jaworski, W.M., Ficocelli, L., O'Mara,K.S.,**The ABL/W4 Methodology for System Modelling, System Research, Vol 4, No 1,pp23-37,1987.

[MJV86] **Masse, S., Jaworski, W.M., Valentine, E.** An Algorithm for Knowledge Base Testing, Internal Report, Knowledge Engineering Technology Inc., 1986.

[Mead 87] **Mead, C.** "Silicon models for neural computation," presented at the 1st Int. Conf. Neural Networks, San Diego, CA, June 1987.

[Mead 89] **Mead, C.** "Analog VLSI and Neural Computing" ,Addison Wesley , March 1989.

[SL88] **Shakley, D.J., Lamont,G.L.** Real-Time Expert Systems for Control Using Parallel Processors, Proceedings of the 27th Conference on Decision and Control,Austin ,Texas, December 1988.

[TS84] **Toru, Ishida, Stolfo Salvatore J.** "Simultaneous firing of Production Rules on Tree Structured Machines".Dept of Computer Science, Columbia University, Technical report CUCS-109-84, March84.

[TS85] **Toru, Ishida, Stolfo Salvatore J.** "Towards the Parallel Execution of Rules in Production System Programs" Proceedings of the International Conference on Parallel Processing, 568-575 (1985).

[Trigoboff] **Trigoboff,M.,** IRIS: A Framework for the Construction of Clinical Consultation Systems, Doctoral dissertation, Computer science dept., Rutgers University,1978.

[TK] **Trigoboff,M., Kulikowski, C.,** IRIS: A System for the Propagation of Inferences in a Semantic Net, IJCAI, 5, 274-280,1977.

Mémoires partagées et réseaux systoliques: algorithmes élégants et/ou efficaces

Gaétan Hains
Centre de recherche informatique de Montréal
1550, boul. de Maisonneuve ouest, Montréal, Québec, H3G 1N2

Résumé

Réflexion sur les liens entre l'algorithmique, les architectures parallèles et les coûts de communication. D'un côté, la théorie des mémoires partagées (PRAM), bien développée, est basée sur un modèle où les accès-mémoire ont un coût unitaire. De l'autre côté, les algorithmes systoliques et distribués font un usage optimal des architectures parallèles. Les algorithmes PRAM peuvent être plus coûteux que nécéssaire lorsqu'implantés sous forme distribuée. Les algorithmes distribués, et systoliques en particulier, sont plus difficiles à concevoir et s'insèrent mal dans l'algorithmique PRAM ou séquentielle. Peut-on réconcilier ces deux approches?

Modèles de complexité et communications

On considère ici la conception d'algorithmes parallèles efficaces dans le but de minimiser le temps de calcul asymptotique. La tâche du concepteur est de relier un problème de calcul bien défini à sa solution sur une architecture-machine donnée. Pour quantifier l'efficacité d'un algorithme, on doit s'appuyer sur un modèle mathématique de l'architecture qui soit juste assez détaillé pour mesurer la complexité réelle des calculs sans trop alourdir leur description. On fait habituellement abstraction des détails qui n'affectent le temps d'exécution que par un facteur constant. Ce choix entre détails et facteurs pertinents peut être délicat comme l'illustre le cas de la taille des 'mots-machine' et le concept associé d'algorithme pseudo-polynomial [2]. Dans ce contexte, la principale question traitée ici est celle de l'opportunité de faire abstraction du temps nécéssaire aux communications à l'intérieur d'un ordinateur parallèle.

Le modèle le plus courant pour la description d'algorithmes séquentiels est celui de la **machine à accès aléatoire** (*Random Access Machine* ou RAM) [1]. Sous certaines conditions, toute implantation d'un algorithme RAM aura un temps d'exécution proportionnel à sa complexité telle que mesurée par le nombre d'instructions

RAM consécutives. C'est cette propriété qui fait l'universalité et la simplicité du modèle séquentiel.

La généralisation à plusieurs processeurs donne lieu à la **machine parallèle à accès aléatoire** (*Parallel Random Access Machine* ou PRAM) où des RAM multiples se partagent une mémoire centrale [3]. Pour autant qu'un mot-mémoire ne soit lu ou écrit par plus d'un processeur à la fois, une opération PRAM donne à tous les processeurs l'accès à la mémoire partagée. On réalise ainsi une communication globale en un temps unitaire et ce, indépendamment du nombre de processeurs impliqués. Pour les besoins de cet exposé, on peut représenter les algorithmes PRAM par des circuits acycliques où chaque porte (sommet) réalise une opération RAM et où le temps de calcul est mesuré par la profondeur du circuit. Un circuit fait donc aussi abstraction des coûts de communication en ignorant la longueur physique des arcs reliant les portes.

La conception et l'implantation d'algorithmes **systoliques** [9] est un mode de programmation mieux adapté aux contraintes physiques des architectures parallèles. De façon simplifiée, un algorithme systolique est un algorithme parallèle dont la structure des communications et le flot des données sont décrits par des équations récurrentes uniformes. La technologie VLSI permet l'implantation d'un tel algorithme sur une surface rectangulaire où la distance entre des processeurs adjacents est indépendante de la taille du réseau. On réalise ainsi des communications dont le coût 'réel' est unitaire.

Les algorithmes de type systolique donnent donc des estimés réalistes de complexité mais sont plus difficiles à concevoir, le programmeur devant restreindre la spécification du problème à une récurrence uniforme. Les algorithmes PRAM, centralisés, facilitent la définition et la conception mais font abstraction des communications, un facteur toujours déterminant pour les temps d'exécution sur machines parallèles.

On illustrera cette problématique par des algorithmes pour la solution du problème du chemin algébrique.

Algorithmes PRAM

Un problème d'importance fondamentale en algorithmique parallèle est le **problème des préfixes** (PP) utile dans un grand nombre de calculs parallèles. La solution à ce problème sert entre autres à la construction de circuits d'addition et de multiplication sans retenues. Étant donnée une opération binaire associative $\circ$, une procédure pour son calcul et une liste d'éléments $x_0, \ldots, x_{n-1}$ du semigroupe de $\circ$, le problème consiste à calculer les préfixes $x_0 \circ \ldots \circ x_i$ pour $i : (0..n-1)$. On peut immédiatement concevoir un circuit pour PP de profondeur $O(\log n)$ et taille $O(n^2)$. Fischer et Ladner ont donné une solution plus efficace qui a toujours profondeur $O(\log n)$ mais ne requiert qu'un nombre linéaire de portes [6]. Ce circuit est illustré pour $n = 8$ dans

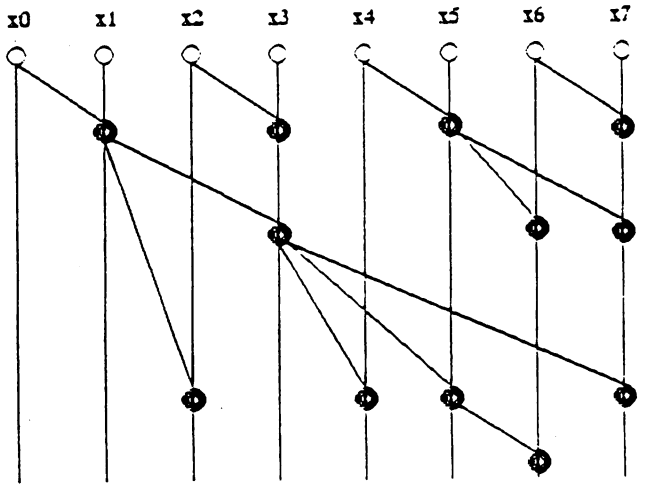


Figure 1: Circuit PP de Fischer et Ladner

la figure 1 où chaque cercle foncé réalise une opération o et les arcs sont orientés vers le bas.

Un autre problème important est le **problème du chemin algébrique** (*algebraic path problem* ou APP) [9] qui généralise les calculs de fermeture transitive, inversion de matrice, plus courts chemins et autres problèmes sur des graphes ou matrices. Dans certains cas particuliers comme celui des plus courts chemins, le problème APP est équivalent à calculer la somme algébrique finie $\sum_{i=0}^{n-1} M^i$ pour une matrice $n \times n$ M donnée. Cette équivalence suggère une solution PRAM pour le problème APP comme suit. Il est facile de construire des circuits de multiplication de matrices et d'addition de matrices de profondeur $O(\log n)$ et de taille $O(n^3)$. On peut ensuite composer ces circuits en un circuit PP de multiplication et une pyramide additive pour obtenir un circuit de profondeur $O(\log n)^2$ et de taille $O(n^4)$ soit quadratique dans la taille de M. Ce circuit est illustré par la figure 2 où: $n = 8$, un cercle foncé représente un sous-circuit de multiplication de matrices, un carré représente un sous-circuit d'addition de matrices et chaque ligne représente un 'bus' de n^2 arcs parallèles transportant une matrice $n \times n$ vers le bas.

Algorithmes distribués

L'algorithme PRAM ci-dessus a l'avantage de la simplicité et de la modularité: il réalise la composition d'opérateurs par la substitution des circuits correspondants. Le

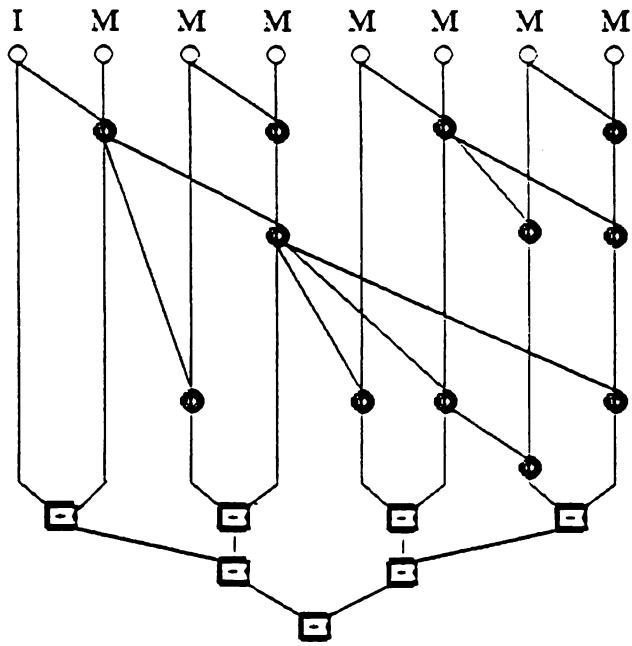


Figure 2: Circuit APP de profondeur $O(\log n)^2$ et taille $O(n^4)$

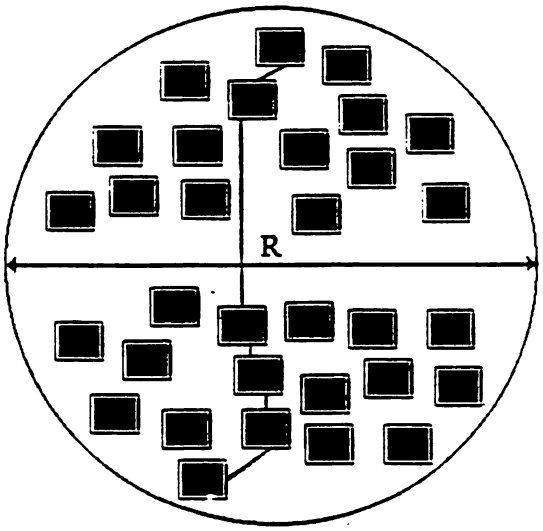


Figure 3: Distance physique dans un réseau

résultat est cependant loin d'être efficace car il nécéssite trop de ressources matérielles. De plus, comme l'algorithme utilise $\Theta(n^4)$ processeurs, on peut montrer que son temps d'exécution *réel* est de l'ordre d'au moins $n^{4/3}$.

En effet, la profondeur $O(\log n)^2$ du circuit ne peut représenter fidèlement le temps d'exécution car les fils, lasers ou autres signaux qui réalisent ses arcs devront parcourir une distance de l'ordre de $\sqrt[3]{P}$, où P est le nombre de processeurs utilisés (figure 3). Cette borne inférieure s'applique au temps moyen d'exécution de tout algorithme parallèle, tel que démontré par Vitányi [11]. Ce résultat n'a aucun effet négatif sur des algorithmes distribués ou systoliques dont le réseau de communication est une maille à deux ou trois dimensions. Dans ces réseaux, le nombre d'arcs le long d'un chemin quelconque est une borne supérieure (à une constante près) pour la longueur physique du chemin puisque tous les arcs relient des processeurs voisins. Le nombre de communications consécutives d'un calcul ou la profondeur d'un circuit se confond alors avec le temps d'exécution.

Il existe un algorithme systolique pour le problème APP [10] qui n'utilise qu'un nombre linéaire $O(n^2)$ de processeurs et dont le temps d'exécution est $O(n)$. Cet algorithme est une version systolique et généralisée de l'algorithme séquentiel $O(n^3)$ de Gauss-Jordan pour l'inversion de matrices. Son existence montre donc une première faiblesse de la méthode ad-hoc de conception d'algorithme PRAM: la définition de

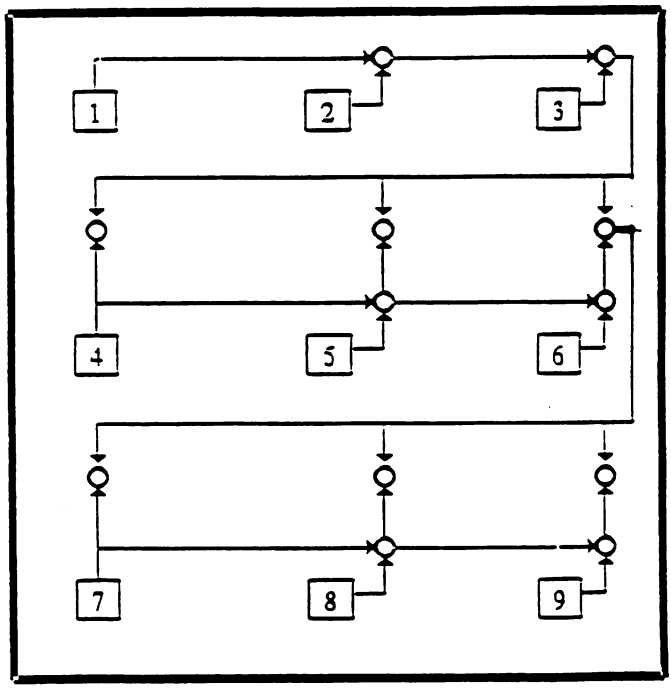


Figure 4: Circuit PP à deux dimensions

communications trop coûteuses. On pourrait tenter d'y remédier en utilisant un algorithme distribué pour le problème des préfixes (temps $O(\sqrt{n})$, taille $O(n)$, figure 4), et y insérer un algorithme systolique pour la multiplication de matrices (temps $O(n)$, taille $O(n^2)$). Le résultat comprendrait alors $O(n^3)$ processeurs mais nécéssiterait un temps $\Theta(n\sqrt{n})$, toujours plus que l'algorithme systolique de [10]. On voit donc que la seconde faiblesse de la méthode PRAM, un trop grand nombre de processeurs, se répercute aussi sur le temps de calcul. Tel qu'exprimé par le résultat de Vitányi, les deux faiblesses sont directement liées: un bon algorithme parallèle doit utiliser peu de ressources matérielles s'il veut être compact physiquement donc rapide.

Classe NC et P-complétude

Soit **P** la classe des problèmes calculables en temps polynomial. La classe de complexité **NC** est la sous-classe de P dont les problèmes sont calculables par un algorithme

PRAM utilisant un nombre polynomial de processeurs et de complexité polynomiale en $\log n$. À cause des algorithmes présentés plus haut, on peut conclure que les problèmes PP et APP font partie de NC.

Une question d'importance capitale en théorie de la complexité parallèle est de savoir si NC = P ou non [3] [8]. On conjecture présentement que NC $\neq$ P en se basant sur une notion de sous-classe des problèmes les plus ardus dans P. Les problèmes contenus dans cette sous-classe sont dits logespace-complets pour P ou simplement **P-complets**. Les problèmes P-complets sont inclus dans P par définition. Leur caractéristique est que si un seul problème P-complet était calculable en NC, alors ils le seraient tous et on aurait NC = P. Or l'état présent des recherches suggère que ce n'est pas le cas.

L'hypothèse très plausible NC $\neq$ P semble classifier les problèmes polynomiaux en 'problèmes parallélisables' NC et 'problèmes séquentiels' P-complets. Or le résultat de Vitányi nous force à nuancer cette classification. Comme on a vu dans le cas du problème APP, un algorithme PRAM ne se traduit pas nécéssairement en algorithme pratique efficace. Toutefois, si l'algorithme PRAM utilise relativement peu de processeurs il est possible de l'implanter efficacement sur une architecture tridimensionelle. Cette implantation consiste en une simulation de chaque étape synchrone de la mémoire partagée par un tri global. On réalise ainsi un algorithme PRAM de complexité $O(\log^k n)$ et utilisant p processeurs en un temps de l'ordre de $p^{1/3}\log^k n$ [7]. Pour des problèmes bien décomposables, on peut remplacer cette stratégie par une méthode de diviser-pour-régner et obtenir une simulation en temps $O(p^{1/3}\log^{k-1} n)$ [4]. On a donc là des complexités proches de la borne inférieure de Vitányi, obtenues par des méthodes générales. Malheureusement, les facteurs logarithmiques et constants associés à ces simulations d'algorithmes PRAM peuvent les rendre non-compétitifs face aux algorithmes systoliques conçus indépendamment. Il reste à découvrir des méthodes générales *et optimales* de simulation d'algorithmes PRAM sur des architectures distribuées.

A l'inverse, un problème P-complet peut être accéléré par des méthodes parallèles, même si le résultat n'est pas optimal. Par exemple, le problème P-complet du compactage des termes finis (minimisation d'automates acycliques) possède une solution systolique $O(n)$ alors que le meilleur algorithme séquentiel connu a complexité $\Omega(n \log n)$ [5]. Il existe aussi des problèmes P-complets comme l'unification [12] pour lesquels un algorithme séquentiel linéaire existe et aucune autre méthode connue n'a de complexité inférieure. Pour clarifier la signification pratique de la P-complétude, il faudrait découvrir des bornes inférieures au temps d'exécution de problèmes P-complets sur des architectures distribuées.

Des réponses à ces questions serviraient à rapprocher la théorie PRAM et la programmation distribuée. On aurait ainsi une base théorique solide et complémentaire des théories de la correction des programmes parallèles pour donner des fondements stables à la programmation parallèle.

Références

[1] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] M. R. Garey and D. S. Johnson. *Computers and Intractability.* W. H. Freeman, 1979.

[3] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms.* Cambridge University Press, 1988.

[4] A. M. Gibbons. Dynamic expression evaluation is one of a class of problems which are efficiently solvable on mesh-connected computers. Technical report, University of Warwick, Coventry, England, 1988.

[5] G. Hains. The compaction of acyclic terms. In *PARLE-89*, number 366 in Lecture Notes in Computer Science, pages 288–303, Eindhoven, Netherlands, 1989. Springer.

[6] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27:831–838, 1980.

[7] D. Nassimi and S. Sahni. Data broadcasting in SIMD computers. *IEEE transactions on computers*, c-30(2):101–107, 1981.

[8] I. Parberry. *Parallel Complexity Theory.* Pitman and John Wiley, 1987.

[9] P. Quinton and Y. Robert. *Algorithmes et architectures systoliques.* Masson, 1989.

[10] Y. Robert and D. Trystram. An orthogonal systolic array for the algebraic path problem. *Computing*, 39:187–199, 1987.

[11] P. M. B. Vitányi. Locality, communication and interconnect length in multicomputers. Technical Report CS-8708, CWI, Amsterdam, 1987.

[12] H. Yasuura. On parallel computational complexity of unification. In *Conference on Fifth Generation Computer Systems (FGCS)*, Tokyo, 1984. North-Holland.

Une approche algébrique à la théorie de la complexité

Denis Thérien

Ecole d'Informatique
Université McGill
3480 Université
Montréal, Québec
H3A 2A7
Mars 1990

0. Introduction

Dans cet article, je me propose d'exposer, de façon assez informelle, quelques résultats de la théorie de la complexité. Le fil directeur de ma présentation veut suggérer que l'approche algébrique, telle qu'utilisée avec grand succès dans l'analyse des automates finis, peut être adaptée à l'étude de classes beaucoup plus générales.

Dans un premier temps je résumerai quelques idées qui m'apparaissent être fondamentales en théorie algébrique des automates. Ensuite, je décrirai des résultats récents indiquant comment ces idées ont été adaptées à l'étude de la classe NC^1, i.e. celle des calculs réalisables en temps parallèle logarithmique. Finalement je proposerai quelques avenues qui me semblent prometteuses pour généraliser davantage l'utilisation des outils algébriques en théorie de la complexité.

La discussion se tiendra en terme de langages. Soit A un alphabet fini, A^n l'ensemble des mots de longueur n sur cet alphabet, et $A^* = \bigcup_{n \geq 0} A^n$: on appelle langage un sous-ensemble quelconque de A^*, et on s'intéresse au problème d'appartenance, c'est à dire au calcul des fonctions $M: A^* \rightarrow \{0,1\}$. De façon générale, la structure d'un modèle de calcul se caractérise par l'ensemble des opérations considérées comme elémentaires, et par les règles permettant de combiner ces opérations. Deux exemples typiques sont les machines de Turing [HU], modélisant les calculs

séquentiels, et les circuits booléens [C], adaptés à l'étude des calculs parallèles. Dans les deux cas, le modèle général se paramétrise en limitant l'utilisation des ressources en fonction de la longueur du mot d'entrée. Le plus souvent, on s'intéresse au temps de calcul et à l'espace de mémoire pour les calculs séquentiels, au temps de calcul et au nombre de processeurs dans le cas parallèle.

Malgré le développement remarquable de notre compréhension de la théorie du calcul, plusieurs questions de base demeurent ouvertes. Par exemple, notons L et P les classes de machines séquentielles opérant respectivement en espace logarithmique et en temps polynomial; notons NC la famille des machines parallèles utilisant un nombre polynomial de processeurs et un temps de calcul polylogarithmique. Les inclusions $L \subseteq NC \subseteq P$ sont assez faciles à montrer, mais on s'interroge toujours à savoir si ces inclusions sont strictes.

On peut résumer la problématique de la théorie de la complexité comme la classification des modèles de calcul selon les langages qui y sont reconnaissables, ou inversement comme la classification des langages selon les ressources de calcul nécessaires pour résoudre la question de l'appartenance. On cherche donc à relier les caractéristiques combinatoires des langages aux propriétés structurelles des machines, incluant sous cette rubrique une analyse quantitative des ressources qu'elles utilisent. Pour ce qui est des automates finis, ces relations sont beaucoup mieux comprises que dans le cas général. Les idées soustendant cette compréhension me semblent naturelles et adaptables à l'étude de classes plus importantes. La section 2 donnera un exemple de ce qu'on peut entrevoir comme généralisation.

1. Les automates finis

Les automates finis constituent un modèle de calcul assez restreint, puisqu'ils formalisent les machines séquentielles à espace de mémoire borné. Un théorème célèbre de Kleene [K] donne une caractérisation combinatoire des langages qu'ils peuvent définir: $L \subseteq A^*$ est reconnaissable par un automate fini ssi L est rationnel, i.e. L peut s'exprimer à partir de l'ensemble vide et des lettres de

A en utilisant les opérations d'union, de concaténation et d'étoile. Cette famille est aussi fermée sous de nombreuses autres opérations, entre autre sous celle de la complémentation.

A partir d'un point de vue algébrique, on a développé une théorie très riche pour classifier automates et langages rationnels [P]. Cette approche remplace la notion d'automate par celle de monoide. Rappelons qu'un monoide fini M est un ensemble fini (aussi noté M) equipé d'une opération binaire associative et admettant un élément neutre. Cette opération permet de voir le monoide comme un automate calculant sur l'alphabet M: à tout mot $m_1 \cdots m_n$ de M^*, le monoide associe l'élément $m = M(m_1 \cdots m_n)$ qui est le produit des m_i. On peut utiliser M pour reconnaitre des langages sur un alphabet A arbitraire de la façon suivante: $L \subseteq A^*$ est reconnu par M ssi il existe une fonction $\theta: A \rightarrow M$ et un sous-ensemble F de M tels que $x = a_1 \cdots a_n$ appartient à L ssi $M(\theta(a_1) \cdots \theta(a_n))$ appartient à F.

Fait 1.1: L est rationnel ssi L est reconnu par un monoide fini.

La puissance de calcul d'un monoide conduit à la relation suivante. Notons $\Theta(A^*,S)$ la famille des sous-ensembles de A^* reconnus par le monoide S: on définit S divise T, noté $S < T$, ssi, pour tout A, $\Theta(A^*,S) \subseteq \Theta(A^*,T)$, i.e. ssi tout langage reconnu par S l'est aussi par T. Cette relation est réflexive et transitive. On peut caractériser algébriquement la relation de division.

Fait 1.2: $S < T$ ssi S est une image homomorphe d'un sous-monoide de T. Si on ne distingue pas entre deux monoides qui sont isomorphes, la relation de division forme donc un ordre partiel.

Une propriété remarquable des langages rationnels est l'existence pour chaque L d'un unique monoide $M(L)$ reconnaissant L et tel que $M(L)$ divise tout autre S qui reconnait aussi L. Cette propriété conduit à son tour à un ordre partiel sur les langages rationnels: posons $L < K$ ssi $M(L) < M(K)$, i.e. ssi L est reconnu par $M(K)$.

La classification des monoides finis et des langages rationnels s'exprime en terme de variétés. Une M-variété $\mathbf{V}$ est une famille de monoides finis fermée sous la division et le produit direct. Une

L-variété V est une famille de langages rationnels fermée sous la division et les opérations booléennes.

Fait 1.3: Il existe une correspondance biunivoque entre variétés de monoides et de langages. Cette correspondance associe à une M-variété $\mathbf{V}$ la L-variété $\Theta(\mathbf{V})$ des langages reconnaissables par les monoides de $\mathbf{V}$. Son inverse associe à une L-variété V la M-variété $\Theta^{-1}(V)$ des monoides $M(L_1)\times\cdots\times M(L_k)$ où chacun des L_i appartient à V.

Il sera commode d'utiliser les notations suivantes: soit $\mathbf{V}$ et V des familles quelconques de monoides finis et de langages rationnels; on écrira $M < \mathbf{V}$ si M appartient à la variété engendrée par les monoides de $\mathbf{V}$, et $L < V$ si L appartient à la variété engendrée par les langages de V.

Les structures algébriques les plus simples sont celles engendrées par un seul élément, soit les compteurs finis. Il existe deux types, fondamentalement différents l'un de l'autre, de compteurs réalisables avec un automate fini, les compteurs à seuil et les compteurs modulaires. Définissons les sous-ensembles suivants de $\{a\}^*$:

pour $t \geq 0$, $SEUIL_t = \{ a^t, a^{t+1}, \cdots \}$

pour $q \geq 1$, $MOD_q = \{ a^0, a^q, \cdots \}$.

Fait 1.4: Soit $L \subseteq \{a\}^*$: L est rationnel ssi $L < \{ SEUIL_t, MOD_q : t \geq 0, q \geq 1 \}$.

Ces opérations se généralisent au comptage des factorisations d'un type donné que possède un mot x. Soit $m \geq 0$, $L_0, \cdots, L_m \subseteq A^*$, $a_1, \cdots, a_m \in A$: on définit les langages

$SEUIL_t(L_0, a_1, L_1, \cdots, a_m, L_m) = \{ x : x$ possède $c+t$ factorisations de la forme $x = x_0 a_1 x_1 \cdots a_m x_m$ avec $x_i \in L_i$ pour $i = 0, \cdots, m \}$,

$MOD_q(L_0, a_1, L_1, \cdots, a_m, L_m) = \{ x : x$ possède cq factorisations de la forme $x = x_0 a_1 x_1 \cdots a_m x_m$ avec $x_i \in L_i$ pour $i = 0, \cdots, m \}$.

Fait 1.5: La classe des langages rationnels est fermée sous les opérations $SEUIL_t$ et MOD_q.

Ces opérations induisent les hiérarchies suivantes. Posons $Aper_0(t) = Gres_0(q) = Mres_0(t,q)$

comme étant la variété contenant, pour chaque alphabet A, l'ensemble vide et le langage A^*.

Pour $k > 0$, posons

$$Aper_k(t) = \bigcup_A \{ L \subseteq A^* : L < \{SEUIL_t(L_0,a_1,L_1,\cdots,a_m,L_m) :$$

$$L_i \text{ dans } Aper_{k-1}(t) \text{ pour tout } i \} \}$$

$$Gres_k(q) = \bigcup_A \{ L \subseteq A^* : L < \{MOD_q(L_0,a_1,L_1,\cdots,a_m,L_m) :$$

$$L_i \text{ dans } Gres_{k-1}(q) \text{ pour tout } i \} \}$$

$$Mres_k(t,q) = \bigcup_A \{ L \subseteq A^* : L < \{SEUIL_t(L_0,a_1,L_1,\cdots,a_m,L_m),$$

$$MOD_q(L_0,a_1,L_1,\cdots,a_m,L_m) : L_i \text{ dans } Mres_{k-1}(t,q) \text{ pour tout } i \} \}$$

Fait 1.6: a) Chacune de ces familles forment une L-variété.

b) Si $t \neq 0$ ou $q \neq 1$, chacune des hiérarchies est stricte (e.g. pour tout k $Aper_k(t) \subseteq Aper_{k+1}(t)$).

c) Pour tout $k > 0$ $Aper_k(1) = Aper_k(t)$ ssi $t > 0$.

d) Pour tout $k > 0$ $Gres_k(q) = Gres_k(r)$ ssi q et r ont les mêmes diviseurs premiers.

e) Pour tout $k > 0$, $q = p^\alpha$ avec p premier, $Gres_k(q) = Gres_1(p)$.

f) $L \in \bigcup_{k,t} Aper_k(t)$ ssi $M(L)$ est un monoide apériodique (i.e. si $F \subseteq M(L)$ et F est un groupe alors $|F| = 1$).

g) $L \in \bigcup_{k,q} Gres_k(q)$ ssi $M(L)$ est un groupe résoluble; de plus si q est fixé $M(L)$ est de cardinalité q^α.

h) $L \in \bigcup_{k,t,q} Mres_k(t,q)$ ssi $M(L)$ est un monoide résoluble (i.e. si $F \subseteq M(L)$ et F est un groupe alors F est résoluble).

De façon informelle, l'énoncé précédent se ramène aux points suivants:

- les opérateurs *SEUIL* et *MOD* sont irréductibles l'un à l'autre;
- la puissance de l'opérateur $SEUIL_t$ ne dépend que de la condition $1 < t$;

- la puissance de l'opérateur MOD_q ne dépend que des conditions $p \mid q$ avec p premier;
- k niveaux de récursion ne sont pas simulables en $k-1$ niveaux, même en augmentant la puissance des compteurs.

Finalement notons que les seuls langages rationnels échappant à cette classification, les langages non résolubles, sont ceux dont le monoide minimal contient un groupe simple non commutatif.

2. La classe NC^1

Les calculs parallèles peuvent être étudiés dans le modèle des circuits booléens. Soit $C = (C_n)_{n \in N}$ une suite de graphes dirigés sans cycle[1]. On note $s(n)$ et $d(n)$ le nombre de noeuds et la profondeur de C_n; on suppose aussi un unique noeud de profondeur $d(n)$, la sortie du graphe, et que chaque noeud du graphe est relié à celui-ci. Les noeuds d'entrance 0 sont étiquetés $\xi_{i,a}$, avec $1 \leq i \leq n$ et $a \in A$, et chaque noeud d'entrance k est étiqueté par une fonction $f: \{0,1\}^k \rightarrow \{0,1\}$. Chacun des graphes de la suite détermine donc naturellement une fonction $C_n: A^n \rightarrow \{0,1\}$, en posant $\xi_{i,a}(x) =$ si $x_i = a$ alors 1 sinon 0,

$f(g_1, \cdots g_k)(x) = f(g_1(x), \cdots, g_k(x))$, et en identifiant le résultat de C_n à la valeur produite au noeud de sortie. Les paramètres du modèle sont alors la classe de fonctions admissibles pour étiqueter les noeuds d'entrance > 0, la profondeur et le nombre de noeuds de chaque graphe de la suite. La classe NC^1 est obtenue en considérant les graphes de profondeur $O(\log n)$, construits à l'aide des fonctions ET et OU d'entrance 2: il suit que la taille de ces circuits est d'ordre polynomial.

On parlera aussi de certaines sous-classes de NC^1, obtenues en utilisant des compteurs. Pour $t \geq 0$ et $q \geq 1$ on définit les fonctions suivantes de $\{0,1\}^*$ dans $\{0,1\}$:

[1] Puisqu'on ne pose pas de conditions sur la constructibilité des suites (C_n), le modèle permet de définir des langages non récursifs. Divers critères d'uniformité peuvent être imposés pour rendre le modèle algorithmique au sens traditionnel. Les résultats que je cite plus bas restent valides sous ce genre de restrictions.

$SEUIL_t(x) =$ si $t \leq \Sigma x_i$ alors 1 sinon 0

$MOD_q(x) =$ si $q \mid \Sigma x_i$ alors 1 sinon 0.

On pose $AC^0(t)$ comme étant la classe de circuits de profondeur constante et de taille polynomiale, construits avec les fonctions $SEUIL_t$ sans restriction sur l'entrance. Les classes $CC^0(q)$ et $ACC^0(t,q)$ sont de meme obtenues à partir des MOD_q et en combinant les deux types de compteurs respectivement. Il est facile de vérifier que chacune de ces trois familles est contenue dans NC^1.

La relation entre NC^1 et les langages rationnels est basée sur une généralisation de l'utilisation des monoides finis comme reconnaisseurs. Soit A, B des alphabets: un programme de A dans B est une suite $\Pi = (\pi_n)_{n \in N}$, où le n^{ieme} programme est un mot de longueur n^c sur l'alphabet $I_n = \{ (i,f) : 1 \leq i \leq n, f : A \rightarrow B \}$. Ce mot π_n détermine une fonction de A^n dans B^{n^c}, en posant $(i,f)(x) = f(x_i)$, et pour $\pi_n = e_1 \cdots e_{n^c}$, $\pi_n(x) = e_1(x) \cdots e_{n^c}(x)$. Définissons aussi, pour $L \subseteq A^*$ et $K \subseteq B^*$, $L <_\Pi K$ lorsqu'il existe un tel programme tel que, pour tout mot x dans A^n, $x \in L$ ssi $\pi_n(x) \in K$. Un langage L est Π-reconnu par M ssi $L <_\Pi K$ pour un langage rationnel K reconnaissable par M au sens de la section 1.

Un théorème magnifique de Barrington [B] caractérise la classe NC^1 par la Π-reconnaissance.

Fait 2.1: L est dans NC^1 ssi L est Π-reconnu par un monoide fini. De plus, le résultat reste vrai si on fixe M comme étant n'importe quel monoide non résoluble.

Ce théorème suggère de relier la classification des sous-classes de NC^1 à celle des langages rationnels. L'adéquation s'est avérée remarquable, et les caractérisations suivantes ont été obtenues.

Fait 2.2: a) L est dans $AC^0(t)$ ssi $L <_\Pi K$ où K est un langage rationnel appartenant à $Aper(t)$.

b) L est dans $CC^0(q)$ ssi $L <_\Pi K$ où K est un langage rationnel appartenant à $Gres(q)$.

c) L est dans $ACC^0(t,q)$ ssi $L <_\Pi K$ où K est un langage rationnel appartenant à $Mres(t,q)$.

De plus, en définissant convenablement la profondeur exacte des circuits, on obtient une correspondance avec le niveau de récursion paramétrisant les langages résolubles [McT]. La structure interne de NC^1 se décrit donc de façon précise à l'aide du formalisme des variétés.

Cette description ne se traduit néammoins pas immédiatement en résultats de séparation pour les classes de calcul parallèles. En effet, il reste à comprendre algébriquement la notion de Π-division. Cependant tous les théorèmes connus indiquent que les idées à la base des propriétés énoncées au Fait 1.6 restent valables dans cette situation plus générale. Par exemple, on sait que le comptage $SEUIL_t$ se simule avec $SEUIL_1$ et le comptage MOD_{p^α} avec MOD_p lorsque p est premier. On sait aussi que la hiérarchie des classes $AC^0_k(t)$ est stricte, et que celle des $CC^0_k(p)$ s'effondre au premier niveau. Il reste donc à analyser la puissance des MOD_q lorsque q n'est pas une puissance d'un nombre premier mais on conjecture que les phénomènes vérifiés dans le cas rationnel resteront vrais.

3. Conclusion

La classification des langages rationnels résolubles, décrite dans la section 1, est basée sur l'application récursive d'une opération définie en terme de compteurs: les sous-familles apparaissant dans cette classification sont caractérisées par le type de compteurs utilisés et par la profondeur de la récursion. Les résultats et les conjectures de la section 2 décrivant la structure interne de la classe NC^1 font apparaitre les mêmes paramètres. Peut-on espérer étendre ces idées à d'autres modèles de calcul?

Deux voies me semblent mériter considération, puisque le passage des morphismes aux programmes a préservé deux caractéristiques importantes. D'abord, le monoide utilisé pour traiter les mots de longueur n est fixé: ensuite la transformation de x en $\pi_n(x)$ a la propriété que chaque

lettre du résultat ne dépend que d'une seule lettre de l'entrée. On peut abandonner l'une ou l'autre de ces restrictions.

Pour le premier cas, on peut admettre que les mots de A^n soient traités par un monoide M_n. Par exemple, on peut prendre $M_n = M^{\log(n)}$ où M est fixe: ceci revient à considérer les fonctions booléennes de $\log(n)$ langages Π-reconnus par M. On peut aussi demander que M_n soit le monoide minimal d'un langage rationnel K_n appartenant à une variété définie par k_n, t_n, q_n. On cherche alors à comprendre comment la variation de ces paramètres affecte la puissance de calcul.

La deuxième idée veut considérer des transductions plus générales que les programmes pour prétraiter une entrée avant d'effectuer le calcul dans le monoide M. On peut par exemple penser à allouer la composition de $\log(n)$ programmes, avec ou sans la restriction que le résultat final soit de longueur polynomiale.

Pour ces deux suggestions, le but visé est évidemment de capturer des classes de complexité plus vastes que NC^1; les classes L, NC et P, mentionnées en introduction fournissent des candidats naturels.

4. Bibliographie

[B] Barrington, D.A.,
Bounded-width polynomial-size programs recognize exactly those languages in NC^1,
J. Comput. Syst. Sci. 38:1, 1989, 150-164.

[C] Cook, S.A.,
The taxonomy of problems with fast parallel algorithms,
Information and Control 64, 1985, 2-22.

[HU] Hopcroft, J.E. and Ullman, J.D.,
Introduction Automata Theory, Languages, and Computation,
Addison-Wesley, 1979.

[K] Kleene, S.C.,
Representation of events in nerve nets and finite automata,
Automata Studies (C.E. Shannon and J. McCarthy, ed.), Princeton University Press, Princeton, New-Jersey, 1956, 39-42.

[McT] McKenzie, P. et Thérien, D.,
Automata Theory Meets Complexity Theory,

Lect. Notes in Comp. Sci. 372, Springer-Verlag, 1989, 589-602.

[P] Pin, J.E.,
Varietes de langages formels,
Masson, 1984.

On the expressive power of temporal logic.

D. Perrin and J.E. Pin
LITP, Paris, FRANCE

Temporal logic has been introduced by Pnueli [15] in view of applications to specification, development and verification of possibly parallel or non-deterministic processes.

In this paper we are interested in the descriptive power of propositional temporal logic (PTL). Indeed it is known that the class of PTL-definable languages is exactly the class of star-free languages. These languages can also be defined in the first-order theory of linear order [9] and are characterized by a deep theorem of Schützenberger : a rational (or regular) language is star-free iff its syntactic semigroup is group-free. Since the syntactic semigroup of a given rational language can be effectively computed, this provides an algorithm to know whether a rational language is PTL-definable.

Various proofs of the equivalence between "first-order", "star-free" and "PTL-definable" have been announced or given in the litterature [4,5,10,11] but all these proofs are rather involved. In this paper, we give a short and simple proof of this result, based on a weak version of the decomposition theorem for finite semigroups.

Next we study the descriptive power of a restriction of temporal logic (RTL) obtained by considering only the operators "next" and "eventually". It was known [4,6] that RTL is strictly less expressive than PTL but an effective characterization of PTL-definable languages was still to be found. We show here that RTL-definable languages admits a syntactic characterization analoguous to Schutzenberger's theorem : a rational language is RTL-definable iff its syntactic semigroup is "locally $\mathcal{L}$-trivial" (in the jargon of semigroup theory, a finite semigroup S is *locally "something"* if for every idempotent e of S, the subsemigroup eSe is "something" and is *$\mathcal{L}$-trivial* if for every $x,y,u,v \in S^1$, $xu = v$ and $yv = u$ imply $u = v$). Of course this provides a decision process to know whether a language is RTL-definable. There is another (less effective) description of RTL-definable languages : these languages form the smallest boolean algebra of languages closed under the operations $L \to aL$ (for every letter a) and $L \to A^*L$.

Similar results can be obtained when temporal logic is interpreted on infinite words. For instance, the notions of star-free, first-order definable and PTL-definable languages can be extended to ω-languages and are still equivalent [4,5,17] . One can also extend the notion of syntactic semigroup and generalize Schützenberger's theorem to ω-languages [12]. These results can also be proved by automata-theoretic methods [13,14], but have been here omitted in order to keep a reasonable size to this extended abstract.

In conclusion, the results of this paper appear as a new justification of the point of view argued in [18] : the "automata-theoretic perspective" is entlightening for temporal logic.

1. Languages and semigroups.

A *semigroup* is a set S together with an associative multiplication. A *monoid* M is a semigroup that has an identity element, always denoted by 1. A semigroup S *divides* a semigroup T if S is a quotient of a subsemigroup of T. A semigroup S is *aperiodic* if it is finite and if for every $s \in S$, there exists an $n > 0$ such that $s^n = s^{n+1}$.

Given two monoids M and N, the *wreath product* M o N of M and N is the monoid of all pairs (f,s) - where $s \in N$ and f is a map from N to M - under multiplication $(f,s)(g,t) = (h,st)$ where $h : N \to M$ is defined by

$$h(n) = f(n)g(sn) .$$

Let A be a finite alphabet. We denote by A^+ the free semigroup over A. A subset of A^+ is called a *language*. *Rational languages* form the smallest class of languages containing letters and closed under union, concatenation and plus-operation ($L^+ = \bigcup_{n>0} L^n$). *Star-free languages* form the smallest class of languages containing letters and closed under boolean operations (union, intersection and complementation) and concatenation product.

The length of a word w is denoted by $|w|$.

A semigroup morphism $\eta : A^+ \to S$ *recognizes* a language L of A^+ if there exists a subset P of S such that $L = \eta^{-1}(P)$. By extension, we say that a semigroup S recognizes L if there is a morphism $\eta : A^+ \to S$ that recognizes L. Let, for instance, U be the monoid {1, x, y} under multiplication

$$\text{for every } u,v \in U,\ u.1 = 1.u = u \text{ and } u.v = v \text{ if } v \neq 1 .$$

One could easily characterize the languages of A^+ recognized by U but we shall only need the following description.

Lemma 1.1. Every language of A^+ recognized by U is a boolean combination of languages of the form A^*aB^* where $a \in A$ and $B \subset A$.

The *syntactic semigroup* of a language $L \subset A^+$ is the quotient of A^+ by the congruence L defined by

$$u \sim_L v \text{ if and only if, for every } x,y \in A^*, xuy \in L \Leftrightarrow xvy \in L .$$

The syntactic semigroup of a language L is the smallest semigroup that recognizes L. It is also the transition semigroup of the minimal automaton of L. As it is well-known, a language is rational iff it can be recognized by a finite automaton. This provides an algorithm to compute the syntactic semigroup of a given rational language.

For star-free languages, we have the following result.

Theorem 1.2. (Schützenberger [16]) Let L be a language of A^+. The following conditions are equivalent :

(1) L is star-free.

(2) L is recognized by an aperiodic semigroup.

(3) The syntactic semigroup of L is aperiodic.

Aperiodic semigroups admit a simple wreath-product decomposition

Theorem 1.3. (Krohn-Rhodes) Every aperiodic semigroup divides a wreath product of the form $(U \circ \ldots \circ (U \circ (U \circ U))\ldots)$

It is easy to see that if a language L is recognized by a semigroup S, and if S divides a semigroup T, then T also recognizes L. Therefore we have

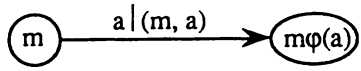
Corollary 1.4. Every star-free language is recognized by a wreath-product of the form $(U \circ \ldots \circ (U \circ (U \circ U))\ldots)$.

In view of this corollary we need a description of the languages recognized by a monoid of the form $U \circ M$. Let $\eta : A^+ \to U \circ M$ be a morphism, and let $\pi : U \circ M \to M$ be the morphism defined by $\pi(f,m) = m$. Finally, let $\varphi : A^+ \to M$

be the composition of η and π. Let $B = M \times A$. We define a sequential function $\sigma : A^+ \to B^+$ by setting

$$\sigma(a_1 \dots a_n) = (1, a_1)(\varphi(a_1), a_2) \dots (\varphi(a_1 \dots a_{n-1}), a_n).$$

In fact σ is realized by a transducer (= deterministic automaton with output) with M as set of states and transition given by



Then one can state the following result, due to H. Straubing

Proposition 1.5 (Wreath product principle) Every language recognized by U o M is a boolean combination of languages of the form $X \cap \sigma^{-1}(Y)$ where $X \subset A^+$ is recognized by M and $Y \subset B^+$ is recognized by U.

Proposition 1.5 can be used to prove Schutzenberger's theorem [1, 2, 8]. We shall see that it can also be used to study the expressive power of temporal logic.

2. Propositional Temporal Logic.

Propositional temporal logic (PTL for short) on an alphabet A is defined as follows. The vocabulary consists of

(1) An atomic proposition p_a for each letter $a \in A$,

(2) Connectives $\vee$, $\wedge$ and $\neg$.

(3) Temporal operators o ("next"), $\Diamond$ ("eventually") and U ("until").

and the formation rules are

(1) For every $a \in A$, p_a is a formula,

(2) If φ and ψ are formulas, so are $\varphi \vee \psi$, $\varphi \wedge \psi$, $\neg\varphi$, $o\varphi$, $\Diamond\varphi$, φ U ψ.

Semantics are defined by induction on the formation rules. Given a word $w \in A^+$, and $n \in \{1, 2, \dots, |w|\}$, we define the expression "w satisfies φ at the instant n" (denoted $(w,n) \models \varphi$) as follows :

(1) $(w,n) \models p_a$ if the n^{th} letter of w is an a.

(2) $(w,n) \models \varphi \vee \psi$ (resp. $\varphi \wedge \psi$, $\neg\varphi$) if $(w,n) \models \varphi$ or $(w,n) \models \psi$ (resp. if $(w,n) \models \varphi$ and $(w,n) \models \psi$, if (w,n) does not satisfy φ).

(3) $(w,n) \models o\varphi$ if $(w,n+1)$ satisfies φ.

(4) $(w,n) \models \Diamond\varphi$ if there exists $m \geq n$ such that $(w,m) \models \varphi$.

(5) $(w,n) \not\models \varphi \, U \, \psi$ if there exists $m \geq n$ such that $(w,m) \not\models \psi$ and for every k such that $n \leq k \leq m$, $(w,k) \not\models \varphi$.

We just have defined "future" temporal formulas but one can define in the same way "past" temporal formulas by reversing time: it suffices to replace "next" by "previous", "eventually" by "sometime" and "until" by "since". The corresponding semantics is obtained by exchanging < and > (resp. n+1 and n-1) in the definition.

If φ is a future (past) temporal formula, we say that w satisfies φ if $(w,1) \not\models \varphi$ (resp. $(w,|w|) \not\models \varphi$). The language defined by φ is the set $L(\varphi)$ of all words of A^+ that satisfy φ.

3. PTL-definable languages.

In this section, we give a short proof of the following result

Theorem 3.1. A language of A^+ is PTL-definable if and only if it is star-free.

We first prove that every PTL-definable language is star-free. This is done by induction on the formation rules. Indeed

$L(p_a) = aA^*$ (for every letter a) is star-free.

$L(o\varphi) = AL(\varphi)$. Thus if $L(\varphi)$ is star-free, so is $L(o\varphi)$.

$L(\Diamond\varphi) = A^*L(\varphi)$. Thus if $L(\varphi)$ is star-free, so is $L(\Diamond\varphi)$.

Finally assume that $L = L(\varphi)$ and $K = L(\psi)$ are star-free. Then there exists in particular an aperiodic semigroup S that recognizes L. That is, there is a semigroup morphism $\eta : A^+ \to S$ and a subset P of S such that $L = \eta^{-1}(P)$. Now a little computation shows that

$$L(\varphi \, U \, \psi) = \bigcup_{s \in S} \left(A^* \setminus A^*(A^* \setminus \eta^{-1}(Ps^{-1}))\right)\left(\eta^{-1}(s) \cap K\right)$$

where $Ps^{-1} = \{t \in S \mid ts \in P\}$. But any language of the form $\eta^{-1}(Q)$ for some $Q \subset S$ is recognized by S, and thus is star-free by the theorem of Schutzenberger. Therefore $L(\varphi \, U \, \psi)$ is star-free and this concludes the first part of the proof.

We now show that every star-free language is PTL-definable. Since star-free languages are closed under reversal we may use past temporal formulas as well. We first observe that languages definable by past temporal formulas are closed under boolean operations and under the operation $L \to La$, for every letter $a \in A$. (This is the reverse version of the formula $aL(\varphi) = L(o\varphi \wedge p_a)$.

We shall denote by S the operator on languages naturally associated to "since":

$L\ S\ K = \{w \in A^+ \mid w = uv, u \in K$ and for every left factor $v' \neq 1$ of v, $uv' \in L\}$.

First of all, every language recognized by the trivial monoid $\{1\}$ is PTL-definable, since $\{1\}$ recognizes the empty set and A^+ only. Now, by corollary 1.5 it suffices to show that if every language recognized by a finite monoid M is PTL-definable, then every language recognized by U o M is also PTL-definable. By the wreath-product principle, every language recognized by U o M is a boolean combination of languages of the form $X \cap \sigma^{-1}(Y)$ where $X \subset A^+$ is recognized by M and $Y \subset B^+$ is recognized by U. Furthermore, by lemma 1.1, every such Y is a boolean combination of languages of the form B^*bC^*, where $b \in B$ and $C \subset B$. Now, by assumption, every language recognized by M is PTL-definable and PTL-definable languages are closed under boolean operations. Thus it remains to show that languages of the form $\sigma^{-1}(B^*bC^*)$ are PTL-definable. We claim that

$$\sigma^{-1}(B^*bC^*) = \sigma^{-1}(B^*C)\ S\ \sigma^{-1}(B^*b) \quad (1)$$

Indeed, let $u = a_1 \ldots a_n$ be a word of A^+ and let $\sigma(a_1 \ldots a_n) = b_1 \ldots b_n$. Then $\sigma(u) \in B^*bC^*$ if and only if there exists an i such that $b_i = b$ and, for every $j > i$, $b_j \in C$. This is equivalent to say that $\sigma(a_1 \ldots a_i) \in B^*b$ and for every $j > i$, $\sigma(a_1 \ldots a_j) \in B^*C$, and this proves (1). Now

$$\sigma^{-1}(B^*C) = \bigcup_{b \in C} \sigma^{-1}(B^*b)$$

and therefore it suffices to show that languages of the form $\sigma^{-1}(B^*b)$ are PTL-definable. Set $b = (m,a)$ (recall that $B = M \times A$) and let $\varphi : A^+ \to M$ be the morphism that is used in the definition of σ (cf. proposition 1.5). Then we have

$$\sigma(a_1 \ldots a_n) = (1, a_1)(\varphi(a_1),a_2) \ldots (\varphi(a_1 \ldots a_{n-1}),a_n)$$

It follows that $\sigma(a_1 \ldots a_n) \in B^*b$ if and only if $\varphi(a_1 \ldots a_{n-1}) = m$ and $a_n = a$.Therefore $\sigma^{-1}(B^*b) = \varphi^{-1}(m)a$. Now $\varphi^{-1}(m)$ is a language recognized by M and thus is PTL-definable. It follows that $\sigma^{-1}(B^*b)$ is PTL-definable and this concludes the proof. □

4. A restriction of temporal logic.

If we omit the "until" operator, we get a restriction of temporal logic (RTL) that was considered in [4,5]. Similar arguments using wreath product decompositions of semigroups lead to the following result:

Theorem 4.1. Let L be a language of A^+. The following conditions are equivalent:

(1) L is RTL-definable,

(2) L belongs to the smallest boolean algebra of languages closed under the operations $L \to A^*L$ and $L \to aL$ for every letter $a \in A$,

(3) The syntactic semigroup of L is locally $\mathcal{L}$-trivial.

References.

[1] R.S. Cohen and J.A. Brzozowski, On star-free events, Proc. Hawaii Internat. Conf. Syst. Sci., Honolulu, (1968), 1-4.

[2] S. Eilenberg, *Automata, Languages and Machines*, Academic Press, New York, Vol A, (1974); Vol B, (1976).

[3] E.H. Emerson, J.Y. Halpern, "Sometimes" and "Not" revisited: On Branching vs. Linear Time", Proc. 10th ACM Symp. on Principles of Programming Languages, (1983).

[4] D. Gabbay, A. Pnueli, S. Shelah, J. Stavi, On the temporal analysis of fairness, Proc. 12th ACM Symp. on Principles of Programming Languages, Las Vegas, (1980), 163-173.

[5] J.A. Kamp, Tense logic and the theory of linear order, Ph. D. Thesis, University of California, Los Angeles, (1968).

[6] O. Katai, Completeness and the expressive power of nexttime temporal logical system by semantic tableau method, INRIA report 109, (1981).

[7] G. Lallement, Semigroups and combinatorial applications, Wiley, New-York, (1979).

[8] A.R. Meyer, A note on star-free events, J.ACM 16, (1969), 220-225.

[9] R. McNaughton and S. Papert, Counter-free automata, MIT Press, Cambridge, Mass, (1971).

[10] M. Parigot, Automates, réseaux, formules. Actes des Journées "Informatique et Mathématiques", Luminy (1984), 74-89.

[11] R. Peikert, ω-regular languages and propositional temporal logic, preprint.

[12] D. Perrin, Recent results on automata and infinite words., *Math. Found. of Comp. Sci.* (M.P. Chytil, V. Koubek, Eds.), Lecture Notes in Computer Science **176**, 1984, 134-148.

[13] D. Perrin and J.E. Pin, First order logic and star-free sets, *J. Comput. System Sci.*. **32**, 1986, 393-406.

[14] J.E. Pin, Star-free ω-languages and first order logic, Automata on Infinite Words, (edited by M. Nivat and D. Perrin) LNCS 192, (1984), 56-67.

[15] A. Pnueli, The temporal logic of programs, Proc. 18th FOCS, Providence, RI, (1977), 46-57.

[16] M.P. Schützenberger, On finite monoids having only trivial subgroups, Inform. and Control 48, (1965), 190-194.

[17] W. Thomas, Star-free regular sets of ω-sequences, Inform. and Control, 42, (1979) 148-156.

[18] M.Y. Vardi and P. Wolper, Applications of temporal logic : an automata-theoretic perspective, preprint, (1985).

[19] P. Wolper, M.Y. Vardi, A.P. Sistla, Reasoning about infinite computation paths, Proc. 24th IEEE Symp. on Foundations of Computer Science, Tucson, (1983), 185-194.

[20] P.Wolper, Temporal logic can be more expressive, Information and Control 56, (1983), 72-99.

EXPRESSING DISTRIBUTED ALGORITHMS WITH GRAPH REWRITING SYSTEMS WITH PRIORITIES

Michel BILLAUD, Pierre LAFON
Yves METIVIER, Eric SOPENA

LABORATOIRE BORDELAIS DE RECHERCHE EN INFORMATIQUE
Université de Bordeaux
U.A. 726 du C.N.R.S.
351, Cours de la Libération
33405 TALENCE - FRANCE

Abstract : *In this paper we develop a new theory of attribute graph rewriting systems with priorities. This theory provides a very general tool for describing algorithms from classical graph theory, and for algorithms implemented on networks of communicating processors and distributed systems. Moreover, this theory gives an algebraic model which allows us to mathematically prove properties of distributed algorithms.*

Mailing Address : *Yves METIVIER, ENSERB, 351 cours de la Libération 33405 TALENCE (FRANCE)*

Electronic Mail : *metivier@geocub-prog.fr*

SECTION 0. INTRODUCTION

This paper presents a new formalism for the description of distributed algorithms. It relies on a special kind of graph rewriting systems working on networks of communicating processors. This work is an attempt to provide a general (and convenient) mathematical tool for proof of properties in distributed algorithms, such as termination and correctness [6,11]. We also use it for some classical algorithms of graph theory [1,3].

As usual (see e.g. [9,10]), a network of communicating processors is represented by a graph, whose vertices stand for processors and whose edges stand for communication channels. By labeling the graph components (vertices and edges) we can simulate processor states, channels contents, etc. A distributed algorithm will be described by an initial graph (the initial state of the network) and a set of "calculation rules" expressed as graph rewriting rules which modify the labeling of the components.

As a consequence of our initial aims, the graph rewriting systems we use are not very powerful, in the sense that they never modify the underlying structure of the graph, and thus can not be used as a graph generating devicet : our graph rewriting systems are not related to the field of graph grammar theory.

In order to increase the expressive power of our model, we use the classical concept of rewriting with priorities [2,8], which allows us to simulate the different kinds of control statements required for the expression of distributed algorithms. This priority concept is intended to solve strictly local rewriting conflicts and is used whenever we need to translate statements such as *"for each neighbour of processor x ..."*, *"if processor x has no neighbour in state s ..."*, ...

In a network each processor does some computation on its local variables; so we introduce the concept of attributed rewriting which only provides a simplification of the expression of graph rewriting systems whenever the attribute domains are finite (which is obviously always the case when we work on computers).

Our paper is organized as follows : in the first section, we recall some basic definitions concerning binary relations, graphs and labeled graphs. The second section is devoted to the definition of graph rewriting systems with priorities (PGRSs for short). In the third section we give some examples of PGRSs devised to solve some classical graph problems. The last section is introduces the concept of attribute graph rewriting systems with priorities (APGRSs), for which we give formal definitions and an illustrating example.

The reader will find all proofs and more examples in [4,5].

SECTION 1. BASIC DEFINITIONS

In this section, we review some basic definitions and general notations concerning binary relations, graphs and labeled graphs that we shall use throughout this paper.

1.1 Binary relations

(1) Let X be a set. $\mathcal{P}(X)$ is the powerset of X; $\#X$ is the cardinality of X. Let $\longrightarrow$ be a binary relation on X; $\overset{*}{\longrightarrow}$ denotes the reflexive-transitive closure of $\longrightarrow$. For a given binary relation $\longrightarrow$ we let $\Delta(x) = \{ y \mid x \longrightarrow y \}$. Let $(x_i)_{0 \le i \le n}$ be a sequence of elements of X such that :

$$x = x_0 \longrightarrow x_1 \longrightarrow \ldots \longrightarrow x_n = y,$$

we say that n is the *length* of this sequence, and we write $x \overset{n}{\longrightarrow} y$.

(2) An element x of X is said to be *irreducible* with respect to relation $\longrightarrow$ if $\Delta(x)$ is empty ; we say that x is a $\longrightarrow$*-normal form*. We let :

$$Irred(x) = \{ y \mid x \overset{*}{\longrightarrow} y \text{ and } y \text{ is irreducible } \}.$$

(3) We say that the relation $\longrightarrow$ is :

i) noetherian if there is no infinite sequence of the form

$$x_1 \longrightarrow x_2 \longrightarrow \ldots \longrightarrow x_n \longrightarrow \ldots, \quad \text{then } \overset{*}{\longrightarrow} \text{ is } \textit{well-founded},$$

ii) compatible with an order $>$ if for any elements x and y, we have :

$$x \longrightarrow y \implies x > y.$$

1.2 Graphs and labeled graphs

We recall here some (classical) graph theoretical definitions (see e.g. [1,3]).

Definition 1.1 : A *graph* G consists of a finite set of *vertices* V, a finite set of *edges* E, and a mapping *Ends* from E to $V \times V$, assigning to each edge e two, not necessarily distinct, vertices of V (the extremities of edge e). The graph G will be denoted by $G = (V, E, Ends)$.

Definition 1.2 : A *directed graph* G (or simply *d-graph*) consists of a finite set of vertices V, a finite set of (directed) *edges* E, and two mappings s and t from E to $V \times V$, assigning to each edge e its source and target nodes respectively. The graph G will be denoted by $G = (V, E, s, t)$.

Definition 1.3 : Let $G = (V, E, s, t)$ be a d-graph ; we define the *underlying* (undirected) *graph of* G, as $Und(G) = (V, E, Ends)$ with :

$$\forall\ e \in E,\quad Ends(e) = \{\ s(e),\ t(e)\ \}.$$

As our rewriting systems are based on edge- or node-labels, we now introduce the concept of labeled graph. Let $C = (C_E, C_V)$ be a pair of disjoint sets of labels ; C_V (resp. C_E) stands for node (resp. edge) labels.

Definition 1.4 : A *labeled graph over* C, or simply a C-graph, consists in a graph $G = (V, E, Ends)$ and a labeling function which is a pair of mappings $l = (l_V, l_E)$ such that :

$l_V : V \longrightarrow C_V$ is the node-labeling function,

$l_E : E \longrightarrow C_E$ is the edge-labeling function.

Similarly, a labeled directed graph, or simply a C-d-graph, is defined as a directed graph with a labeling function l defined as above.

If x stands for a component of a graph G (i.e. a vertex or an edge) we will denote by $l(x)$ its label (which stands for $l_V(x)$ if x is a vertex and $l_E(x)$ otherwise). If c is a node- (resp. an edge-) label, $|G|_c$ is the number of c-labeled nodes (resp. edges) in the graph G. For any graph (or d-, C-, C-d-graph) G, we will denote its components by V_G, E_G, $Ends_G$, s_G, t_G or l_G.

We now introduce the concepts of subgraph, partial labeling functions and graph morphisms we shall use in the next section. All these definitions will be given for C-d-graphs as they can easily be transposed to the other kinds of graphs we deal with.

Definition 1.5 : Let G and H be two C-d-graphs, we say that G is a *subgraph* of H, denoted by $G < H$, if :

i) $V_G \subset V_H$ and $E_G \subset E_H$,

ii) s_G and t_G are respectively the restrictions of s_H and t_H to E_G,

iii) l_{V_G} (resp. l_{E_G}) is the restriction of l_{V_H} (resp. l_{E_H}) to V_G (resp. E_G).

Definition 1.6 : Let G be a C-d-graph and $\lambda = (\lambda_V, \lambda_E)$ a partial labeling function of G (i.e. λ_V is a mapping from $X \subseteq V$ to C_V and λ_E a mapping from $Y \subseteq E$ to C_E) ; we define the C-d-graph $H = \lambda G$ as follows :

i) $V_H = V_G$, $E_H = E_G$, $s_H = s_G$, $t_H = t_G$,

ii) $\forall\, v \in V_H$, if $v \in X$ then $l_{V_H}(v) = \lambda_V(v)$ else $l_{V_H}(v) = l_{V_G}(v)$,

iii) $\forall\, e \in E_H$, if $e \in Y$ then $l_{E_H}(e) = \lambda_E(e)$ else $l_{E_H}(e) = l_{E_G}(e)$.

Definition 1.7 : Let G and H be two C-d-graphs ; a *graph morphism* from G to H is a pair $m = (m_V, m_E)$ of mappings $m_V : V_G \longrightarrow V_H$, $m_E : E_G \longrightarrow E_H$, such that :

i) $\forall\, e \in E_G$, $s_H(m_E(e)) = m_V(s_G(e))$,

ii) $\forall\, e \in E_G$, $t_H(m_E(e)) = m_V(t_G(e))$,

iii) $l_{V_G} = l_{V_H} \circ m_V$ and $l_{E_G} = l_{E_H} \circ m_E$.

If m_V and m_E are injective (resp. surjective) mappings, we say that m is injective (resp. surjective). If m is injective and surjective we say that m is an isomorphism and we shall write $H \cong G$.

mG is the subgraph of H defined by $mG = (m_V V, m_E E, s', t', l')$ where s', t' and l' stand for the adhoc restrictions of s_H, t_H and l_H respectively.

SECTION 2 : GRAPH REWRITING SYSTEMS WITH PRIORITIES (PGRSs)

In this section, we introduce the main concepts of our graph rewriting model. We deal with a special kind of graph rewriting systems in which the rewriting rules do not modify the underlying structure of the rewritten graphs but only the coloring of their components (vertices and edges).

Let $C = (C_V, C_E)$ be a color alphabet as defined in the previous section.

Definition 2.1 : A *rewriting rule* over C consists in a connected C-d-graph D and a partial labeling function of D denoted by λ. We shall write $r = (D, \lambda)$.

Note that such a rewriting rule can be more classically viewed as a pair of graphs (the left-hand-side and the right-hand-side of the rule) given by $(D, \lambda D)$.

Definition 2.2 : Let G be a C-d-graph and $r = (D,\lambda)$ a rewriting rule. We say that the graph G is *reducible by using rule* r if there exists an injective morphism μ from D to G. We call the subgraph μD of G an *occurrence* of D in G.

Definition 2.3 : A *rewriting step* is defined by a 4-tuple (G,r,μ,H) where :

i) $r = (D,\lambda)$ is a rewriting rule,

ii) G is a C-d-graph reducible by using rule r (with corresponding injective morphism μ),

iii) H is a C-d-graph obtained from graph G by relabeling vertices and edges of occurrence μD by function λ.

We will say that G is rewritten into H and denote it by $G \xrightarrow[r,\mu]{} H$, or simply $G \xrightarrow[r]{} H$.

Example 2.4 : Figure 2.1 shows a rewriting rule $r = (D,\lambda)$ and two graphs G and H such that $G \xrightarrow[r]{} H$. The color alphabet is given by $C_V = \{a,b,c\}$ and $C_E = \{x,y,z\}$.

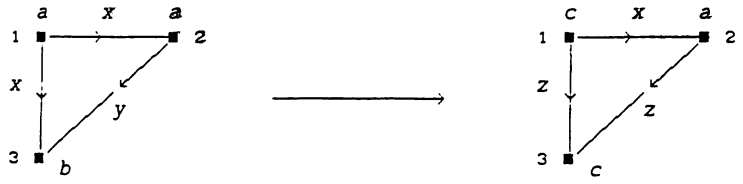
Definition 2.5 : A *rewriting chain* is a sequence $(G_1,r_1,\mu_1,\ldots,G_{n-1},r_{n-1},\mu_n,G_n)$ such that : $\forall\ i,\ 1 \le i < n,\ (G_i,r_i,\mu_i,G_{i+1})$ is a rewriting step.

Definition 2.6 : A *graph rewriting system with priorities* (a PGRS for short) is defined by a triple $(C,P,>)$ where :

i) $C = (C_V,C_E)$ is the color alphabet,

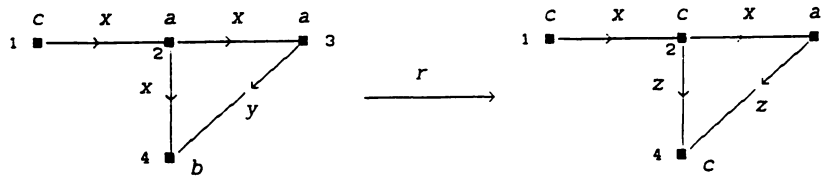
ii) P is a finite set of rewriting rules over C,

iii) $>$ is a partial order on the rules of P.



the graph D the graph λD

(a) the rewriting rule $r = (D,\lambda)$



(b) a rewriting step $G \xrightarrow[r]{} H$

- figure 2.1 -

The concept of correctness, introduced below, enables us to precise the effect of priorities within a PGRS.

Definition 2.7 : Let $\mathbb{R} = (C, P, >)$ be a PGRS, $r = (D, \lambda)$ a rule of P and G a C-d-graph. A rewriting step (G, r, μ, H) is *correct* with respect to $\mathbb{R}$ if and only if :

for any rule $r' = (D', \lambda')$ of P such that $r' > r$, there exists no occurrence $\mu' D'$ of r' which intersects the occurrence μD of r.

For such a rewriting step, we will write $G \xrightarrow[\mathbb{R}]{r} H$.

We can easily extend this notion of correctness to rewriting chains in the following way :

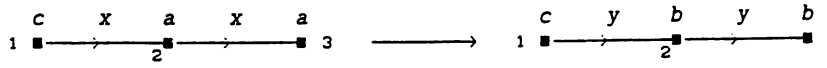
a rewriting chain is said to be correct with respect to $\mathbb{R}$ if and only if all of its rewriting steps are correct. We shall then write $G_1 \xrightarrow[\mathbb{R}]{*} G_n$.

Example 2.8 : Let r and G be the rewriting rule and the C-d-graph of example 2.4, and $r' = (D', \lambda')$ the rewriting rule shown by figure 2.2.
Let us consider the PGRS $\mathbb{R} = (C, \{r, r'\}, >)$ defined by $r' > r$ (rule r' has a stronger priority than rule r) ; then :

$G \xrightarrow{r} H$ is not correct with respect to $\mathbb{R}$,

$G \xrightarrow{r'} H'$ is correct with respect to $\mathbb{R}$,

where H' denotes the graph obtained from graph G by applying rule r'.



- figure 2.2 -

Notice that the effect of the priorities is strictly local (when there is a conflict between two intersecting occurrences); this means that if two rules r and r' such that $r' > r$ can be applied in two *distinct* parts of a graph G, one can freely choose to apply r or r'. This a la Church-Rosser property is precised by the following lemma :

Lemma 2.9 : *Let* $\mathbb{R} = (C,P,>)$ *be a PGRS, and* G *a C-d-graph ; let* (G,r_1,μ_1,H_1) *and* (G,r_2,μ_2,H_2) *be two rewriting steps correct with respect to* $\mathbb{R}$ *and such that occurrences* $\mu_1 D_1$ *and* $\mu_2 D_2$ *does not intersect. Then :*

i) (H_1,r_2,μ_2,H_{12}) *and* (H_2,r_1,μ_1,H_{21}) *are both correct with respect to* $\mathbb{R}$,

ii) moreover, we have $H_{12} = H_{21}$.

Hence, one can view the behaviour of a PGRS on a given graph G in the following way :

1. *Sequential behaviour :*
   Consider all the occurrences in graph G which can be *correctly* rewritten, choose one of them and apply the corresponding rewriting rule. Repeat this process while possible. If no more rule can be applied, then the computation is terminated.

2. *Distributed behaviour :*
   Consider all the occurrences in graph G which can be *correctly* rewritten, choose one or several not intersecting ones of them and apply the corresponding rewriting rule(s). In this case, two (or more) rewriting steps *may* be applied at the same time. Repeat this process while possible. If no more rule can be applied, then the computation is terminated.
   This behaviour is clearly equivalent to one (or more) sequential ones.

SECTION 3. EXAMPLES

In this section we use PGRSs to describe some graph algorithms. We first present two techniques for the computation of a directed spanning tree in a connected graph (or a network of processors [10]). The first one is rather similar to the (sequential) Tremaux algorithm, whereas the second one is a distributed computation.

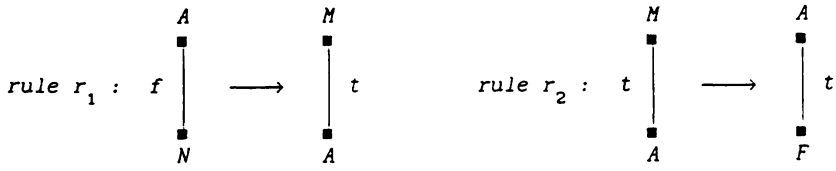
3.1 Computing a Spanning Tree in a Connected Graph

This example demonstrates the use of a PGRS for the computation of a spanning tree in a non-directed connected graph. This PGRS acts in the very same way as the classical Tremaux algorithm [3].

Example 3.1 : To solve this problem, we use a PGRS $\mathbb{R}_1 = (C,>,P)$ defined by :

1. $C = (C_V, C_E)$ with :
   - $C_V = \{ A, M, N, F \}$ for vertices, where A stands for *active*, M for *marked*, N for *not yet visited* and F for *finished*.
   - $C_E = \{ t, f \}$ for edges, where t indicates that the edge belongs to the computed tree.

2. $P = \{ r_1, r_2 \}$, with :



3. $>$ is defined by $r_1 > r_2$.

Proposition 3.2 : *i) relation* $\xrightarrow[\mathbb{R}_1]{}$ *is noetherian,*

ii) Let G be a connected graph with n vertices such that :

- *each edge is labeled with f,*
- *exactly one vertex r (the root) is labeled with A, the other ones with N,*

then:

a) every graph G' in Irred(G) satisfies the following properties :

- *the root r is labeled with A, all other vertices with F,*
- *the t-edges of G' constitute a spanning tree for the underlying graph.*

b) $G' \in Irred(G)$ *iff* $G \xrightarrow[\mathbb{R}_1]{2n-2} G'$.

Sketch of the proof : One can easily check that the quantity $(|G|_N, |G|_M)$ induces a noetherian order compatible with $\mathbb{R}_1$. As this quantity is always positive, we obtain a termination argument.

The correctness of $\mathbb{R}_1$ is deduced from the following invariants :

(I1) the *t*-labeled edges constitute a tree ; a vertex is in the tree iff it is not labeled with *N*,

(I2) there is one and only one *A*-labeled vertex ; the *M*-labeled vertices constitute a chain in the tree from the root to this vertex,

(I3) a *F*-labeled vertex has no neighbour with label *N*. □

3.2 Computing a Directed Spanning Tree

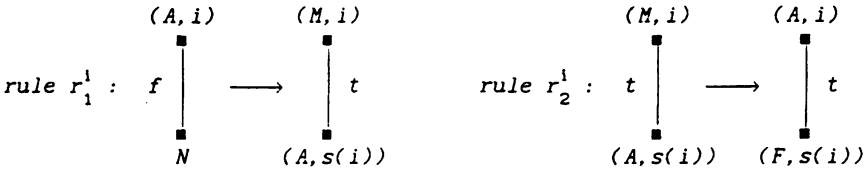
A slight modification in the preceding PGRS allows the computation of a directed spanning tree. The trick is the following : we use a modulo-3 numbering for successive layers in the tree.

For any *i* in *{0,1,2}*, let $s(i) = (i+1) \bmod 3$.

Example 3.3 : We consider the new PGRS $\mathbb{R}_2 = (C, >, P)$ defined by :

1. the colour set $C = (C_V, C_E)$ is given by :
   - $C_V = \{ A, F, M \} \times \{ 0, 1, 2 \} \cup \{ N \}$,
   - $C_E = \{ f, t \}$,

2. $P = \{ r_1^1, r_2^1 \}$ for any $i \in \{0,1,2\}$ with :



3. relation $<$ is defined by : $\forall\ i,j \in \{0,1,2\}\quad r_1^i > r_2^i$.

Proposition 3.4 : *i) relation* $\xrightarrow[\mathbb{R}_2]{}$ *is noetherian,*

ii) Let G be a connected graph with n vertices such that :

- *each edge is labeled with f,*
- *exactly one vertex r (the root) is labeled with (A,0), the other ones with N,*

then :

a) every graph G' in Irred(G) satisfies the following properties :

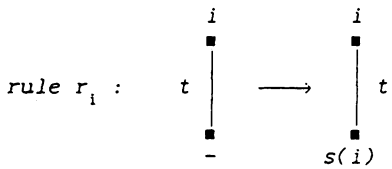
- *the root r is labeled with (A,0), and all other vertices with (F,i), i ∈ {0,1,2},*
- *the t-edges of G' constitute a spanning tree for the underlying graph, directed from the root to leaves by successive indexes.*

b) $G' \in Irred(G)$ *iff* $G \xrightarrow[\mathbb{R}_2]{2n-2} G'$.

Remark 3.5 : We can also present this technique under the form of a "cartesian product" between the Spanning Tree Construction PGRS above and the simple Tree Directing PGRS below :

1. $C = (C_V, C_E)$ with :
   - $C_V = \{ -, 0, 1, 2 \}$ (- stands for *not numbered*),
   - $C_E = \{ t \}$.

2. $P = \{ r_i,\ i = 0,1,2 \}$ is given by :



which obviously provides an orientation on a tree (here *t*-edges) where :

i) the root is initially numbered *0*,

ii) all other vertices are initially not numbered (labeled with -).

Nota : In the rest of the paper we will omit indices and edge labels, and we simply draw arrows on those edges which belong to the directed tree.

Remark 3.6 : an important property of both these PGRSs is the local termination detection property (LDTP) : one easily notices that the computation stops as soon as the root has label *A* and all of its neighbours have label *F*. This property will be fundamental when we attempt to solve problems on networks (see e.g. [9,10,11]) : the node (i.e. the processor) which initiates the computation often has to know whether the computation has terminated or not.

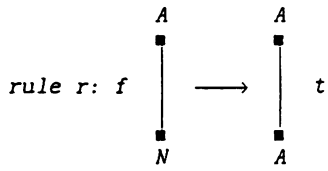
3.3 Distributed Computation of a Spanning Tree

In this section we first present a very simple PGRS for the distributed computation of a directed spanning tree in a connected graph. As this PGRS does not satisfy the LTDP, we also present a more sophisticated PGRS which fulfills that requirement.

Example 3.7 : Let $\mathbb{R}_3 = (C, >, P)$ be the PGRS defined by :

1. $C = (C_V, C_E)$ is given by $C_V = \{ A, N \}$ and $C_E = \{ t, f \}$ with the same meaning as in example 3.1.

2. $P = \{ r \}$ is given below :



Proposition 3.8 : *i) relation* $\xrightarrow[\mathbb{R}_3]{}$ *is noetherian,*

ii) Let G be a connected graph with n vertices such that :

- *each edge is labeled with f,*
- *exactly one vertex r (the root) is labeled with A, the other ones with N,*

then :

a) every graph G' in Irred(G) satisfies the following properties :

- *all vertices are labeled with A,*
- *the t-edges of G' constitute a spanning tree for the underlying graph,*

b) $G' \in Irred(G)$ *iff* $G \xrightarrow[\mathbb{R}_3]{n-1} G'$.

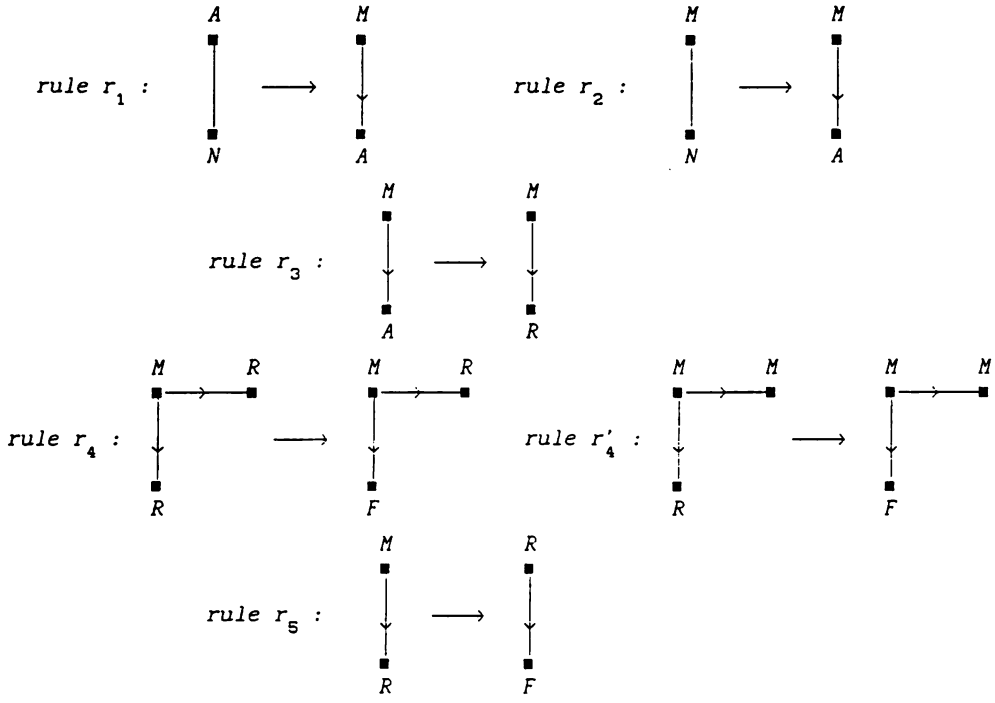
Example 3.9 : PGRS with local termination detection

Here we achieve the LTD requirement by means of "termination messages" emitted by vertices when a subtree has been explored.

Let us consider the PGRS $\mathbb{R}_4 = (C, >, P)$ defined by :

1. $C = (C_V, C_E)$ is given by :
   - $C_V = \{ A, N, M, R, F \}$ where A stands for *active*, N for *not_reached*, M for *marked*, R for *ready_to_finish* and F for *finished*,
   - $C_E = \{ \text{—} , \longrightarrow \}$ (as we are building a directed tree we use the same trick as in example 3.3 and we use arrows instead of t-labeled edges and modulo 3 numbering of successive layers).

2. $P = \{ r_1, r_2, r_3, r_4, r'_4, r_5 \}$ is given below :



3. relation > is defined by : $\{ r_1, r_2 \} > \{ r_3, r_4, r'_4 \} > \{ r_5 \}$.

Proposition 3.10 : *i) relation* $\xrightarrow[\mathbb{R}_4]{}$ *is noetherian,*

ii) Let G be a connected graph with n (n>1) vertices such that :
- *each edge is labeled with f,*
- *exactly one vertex r (the root) is labeled with A, the other ones with N,*

then every graph G' in Irred(G) satisfies the following properties :

a) the root r is labeled with R, all other vertices with F,

b) the t-edges of G' constitute a spanning tree for the underlying graph.

SECTION 4. ATTRIBUTE GRAPH REWRITING SYSTEMS

In this section, we introduce the concept of attribute graph rewriting systems (APGRSs for short) which allows us to do some computations by means of semantic rules associated with each rewriting rule of a given PGRS.

Definitions will be given for C-graphs as they can easily be transposed in a natural way to the other kinds of graphs.

Definition 4.1 : An attribute graph rewriting system (with priorities) is given by $(C, P, >, A, Att, Dom, Sem)$ where :

i) $(C, P, >)$ is a PGRS,

ii) A is a finite set of symbols called *attribute names* or simply *attributes*,

iii) Att is a function from C into $\mathcal{P}(A)$,

iv) Dom is a mapping from A into *domains* which associates with each attribute a its domain of values $Dom(a)$,

v) for each rewriting rule $r = (D, \lambda)$ of P, $Sem(r)$ is the set of semantic rules of r, which is defined as follows :
Let $D = \langle V, E, Ends, l \rangle$; an *attribute of* r is a pair (a, x) with $x \in V \cup E$, and $a \in Att(\lambda(x)) \cup Att(l(x))$.
A *semantic rule* α in $Sem(r)$ has the following form :

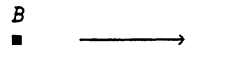
$$(a_0, x) = f\ (\ (a_1, x_1),\ \ldots,\ (a_n, x_n)\ )$$

with $x \in V \cup E$, $a_0 \in Att(\lambda(x))$, and for i, $1 \le i \le k$, $x_i \in V \cup E$, $a_i \in Att(l(x))$ and f is a mapping from $Dom(a_1) \times \ldots \times Dom(a_n)$ into $Dom(a_0)$.

Example 4.2 : In this example we adapt the PGRS of 3.9 for the well-known problem [10] of assigning a unique name (here *Dewey identifiers*) to each vertex in a graph. To each label A or M, we associate an integer attribute a, which denotes the first free value to be assigned to a son of the labeled vertex. We use the following integer attributes : i(A), i(M), i(R), i(F), a(A) and a(M). Attribute i stands for *identifier* of a vertex ; a for the first free value to be assigned to a son of the labeled vertex.

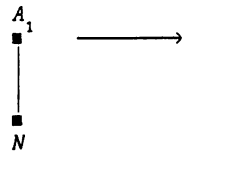
We obtain the following APGRS :

rule r_0 :

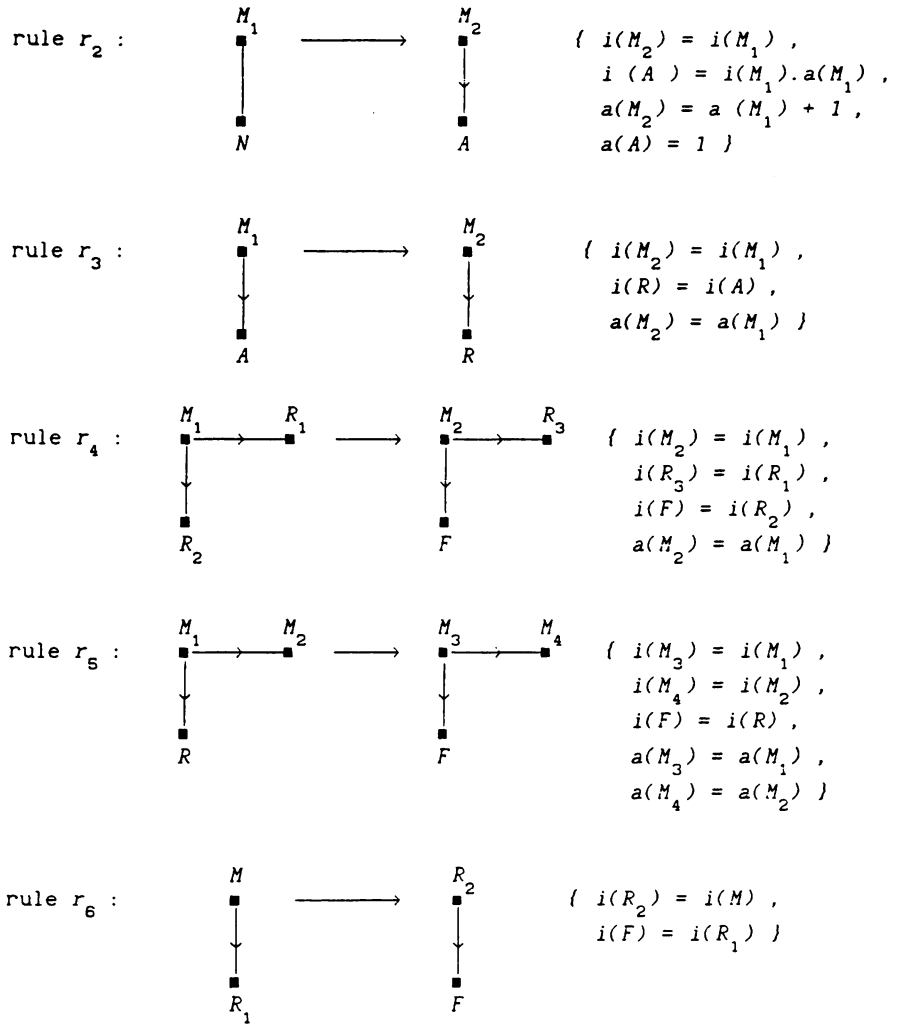


$\{\ a(A) = 1\ ,\ i(A) = 1\ \}$

rule r_1 :



$\{\ i(M) = i(A_1)\ ,\ i(A_2) = i(A_1).a(A_1)\ ,\ a(M) = a(A_1) + 1\ ,\ a(A_2) = 1\ \}$



Lemma 4.3 : *Let G be a connected graph (with at least 2 vertices) with each vertex labeled N except one of them which has label B.*

Then any graph G' in Irred(G) satisfies the following property :

let x and y be two any distinct vertices of G', then their associated identifiers (given by attribute i) are distinct.

Acknowledgements : We are very indebted to R. Cori and B. Vauquelin who have inspired this work.

REFERENCES

[1] Aho, A.O., Hopcroft, J.E., and Ullman, J.D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.

[2] Baeten, J.C.M., Bergstra, J.A., Klop, J.W., *Term Rewriting Systems with Priorities*, Rewriting Techniques and Applications, Lecture Notes in Comp. Science 256, pp 83-94, 1987.

[3] Berge, C., *Graphes et Hypergraphes*, Dunod, Paris, 1970.

[4] Billaud, M., *Using PGRS to Express some Backtracking Algorithms on Graphs*, Technical Report n°8989, LABRI, Université Bordeaux I, 1989.

[5] Billaud, M., Lafon, P., Metivier, Y., Sopena, E., *Graph Rewriting Systems with Priorities : Definitions and Applications*, Technical Report n°8909, LABRI, Université Bordeaux I, 1989.

[6] Dijkstra, E.W., and Scholten, C.S, *Termination Detection for Diffusing Computations*, Information Processing Letters. 11, pp 1-4, 1980.

[7] Hoare, C.A.R., *Communicating Sequential Processes*, Comm. ACM, Vol 21 n°8 pp 666-676, Aug. 1978.

[8] Mohan, C.K., *Priority Rewriting : Semantics, Confluence and Conditionnals*, Rewriting Techniques and Applications, Lecture Notes in Comp. Science 256, pp 278-291, 1987.

[9] Raynal, M., *Algorithmes Distribués et Protocoles*, Eyrolles 1985.

[10] Raynal, M., *Systèmes Répartis et Réseaux : Concepts Outils et Algorithmes*, Eyrolles 1987.

[11] Topor, R.W., *Termination Detection for Distributed Computations*, Information Processing Letters 18 (1984) pp 33-36.

LaBRI
Laboratoire Bordelais de Recherche en Informatique
Unité de Recherche associée au C.N.R.S. n° 726
Université de Bordeaux I

COMPLEXITE MOYENNE DE L'ALGORITHME D'EXCLUSION MUTUELLE DE NAIMI ET TREHEL

A.ARNOLD
M.DELEST
S.DULUCQ

Résumé.

Nous montrons que la complexité moyenne de l'algorithme réparti d'exclusion mutuelle proposé par Naïmi et Tréhel pour un réseau comportant n+1 processus est exactement le $n^{ième}$ nombre harmonique. L'utilisation de séries formelles en variables non commutatives et d'opérateurs sur des séries algébriques permet d'obtenir ce résultat.

Abstract.

We prove that the average complexity for Naïmi-Tréhel's algorithm concerning the mutual exclusion of n+1 process in a distributed network is exactly the n^{th} harmonic number. The proof is made using formal power series and some operators over algebraic languages.

INTRODUCTION

Cet article a pour but de montrer que la complexité moyenne de l'algorithme d'exclusion mutuelle dans un réseau distribué proposé par Naïmi et Trehel [11] est exactement H_n le $n^{ième}$ nombre harmonique, ceci pour n+1 processus.

L'algorithme consiste en une transformation sur les arborescences. Elle reflète les opérations qu'effectue un processus lorsqu'il demande l'entrée en section critique.

Des opérations analogues sur les arborescences apparaissent naturellement dans d'autres problèmes. Il s'agit des compressions de chemins utilisées dans des procédures de tri [10,14] et dans le calcul de l'enveloppe d'un ensemble de fonctions [9].

Nous montrons que la complexité moyenne de l'algorithme de Naïmi-Trehel (nombre moyen de messages échangés par les processus) n'est autre que le nombre moyen de modifications qu'implique la transformation des arborescences sur la structure de donnée les représentant. Ceci est l'objet du second paragraphe où nous rappelons quelques résultats obtenus par Trehel [15] (le premier étant consacré au rappel de définitions élémentaires).

Au paragraphe 3, nous étendons cette transformation aux arbres dessinés, objets codés simplement par les mots du langage de Dyck [3]. Ceci nous permet d'étudier la transformation de manière plus algébrique. Ainsi dans un premier temps, au paragraphe 4, nous donnons une expression exacte des probabilités limites d'obtention de chaque arbre dessiné au bout d'un nombre infini de transformations de ces arbres. Pour calculer cette expression nous introduisons des opérateurs sur le langage de Dyck. L'utilisation d'opérateurs a été introduite par Cori, Richard [5] à propos de graphes planaires. Des opérateurs plus généraux ont été étudiés successivement par Chottin [4], puis Dulucq [6]. Ce dernier s'est intéressé à eux en tant qu'outils combinatoires pour prouver l'algébricité de séries solutions de systèmes d'équations les comportant.

Cette étude nous permet au paragraphe 5 de déterminer exactement le coût moyen de cette transformation et de montrer qu'il vaut H_n lorsque l'on considère des arbres ayant n+1 sommets. Un raisonnement simple nous assure alors qu'il en est de même dans le cas des arborescences, d'où le résultat annoncé.

Dans la dernière partie, nous montrons que l'expression obtenue des probabilités limites conduit naturellement à étudier un problème purement combinatoire concernant l'énumération d'une classe particulière d'involutions, étude que nous développerons dans un autre article [7].

1 - DEFINITIONS ET NOTATIONS

Une *arborescence* $\mathcal{A}=(A,r)$ est un graphe A connexe sans cycle [1] où r est un sommet distingué appelé *racine* . La hauteur d'un sommet x est le nombre de sommets sur l'unique chemin menant de la racine à ce sommet. En particulier la hauteur de la racine est 1. On note $N^i(\mathcal{A})$ le nombre de sommets à hauteur i dans l'arborescence $\mathcal{A}$. Ainsi $N^2(\mathcal{A})$ représente le *degré* de l'arborescence (degré de la racine).

Une arborescence (A,r) à n sommets numérotés de 1 à n peut être représentée par un tableau PERE[1..n] tel que, pour tout sommet i différent de la racine, PERE[i] est l'unique sommet j précédant i sur le chemin joignant r à i (on dit que i est le *fils* de j), et PERE[r]=0. Les sommets appartenant au chemin joignant r à i, différents de i, sont les *ancêtres* de i.

L'ensemble de ce travail utilise des notions sur les langages algébriques, les grammaires qui leurs sont associées et les séries formelles. Le lecteur trouvera les définitions de ces objets dans [2] et [3].

2 - TRANSFORMATION D'ARBORESCENCES ET PREMIERS RESULATS

Nous allons rappeler la transformation d'arborescences considérée par Naïmi et Trehel [11] dans leur algorithme d'exclusion mutuelle.

Soient $\mathcal{A}=(A,r)$ une arborescence à n sommets représentée par son tableau PERE et i un sommet quelconque de cette arborescence. L'arborescence transformée $\varphi_i.\mathcal{A}$ est l'arborescence obtenue en effectuant les modifications suivantes sur le tableau PERE:

- pour tout sommet j situé sur le chemin de r à i dans A autre que i (tout sommet j *ancêtre* de i), faire PERE[j]←i,
- PERE[i]←0.

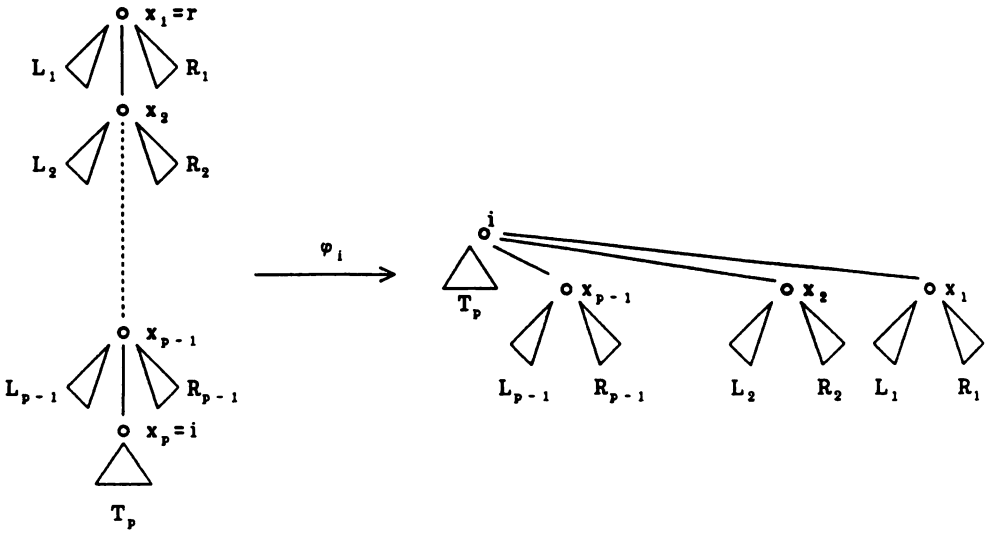


Figure 1. La transformation φ_i.

Dans le cas où i est la racine, on a

$$\varphi_i . \mathcal{A} = \mathcal{A}.$$

Cette transformation φ_i est présentée figure 1.

On appelle *coût* de cette transformation φ_i sur $\mathcal{A}$ le nombre de modifications effectuées sur le tableau PERE. On le note $C(\varphi_i . \mathcal{A})$. On a alors:

$$C(\varphi_i . \mathcal{A}) = \begin{cases} 0 \text{ si } i \text{ est la racine de } \mathcal{A} , \\ \text{hauteur de } i \text{ dans } \mathcal{A} \text{ sinon .} \end{cases}$$

Cette transformation d'arborescences est celle considérée par Naïmi et Trehel dans leur algorithme d'exclusion mutuelle dans un réseau distribué [11]. De plus, le coût de la transformation φ_i considérée ici représente le nombre de messages envoyés lorsque le processus situé sur le site i demande l'accès en section critique.

Etant donnée une arborescence $\mathcal{A}$ ayant n sommets, considérons le coût moyen $M(\mathcal{A})$ de la transformation φ sur $\mathcal{A}$ défini par

$$M(\mathcal{A}) = \frac{1}{n} \sum_{i:\ \text{sommet}} C(\varphi_i . \mathcal{A}) \ .$$

On a

$$M(\mathcal{A}) = \frac{1}{n} \sum_{j \geq 2} j\ N^j(\mathcal{A}) .$$

Soit $\mathfrak{B}_n$ l'ensemble des arborescences à n sommets, nous notons $|\mathfrak{B}_n|$ le cardinal de cet ensemble

$$\mathfrak{B}_n = \{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_{|\mathfrak{B}_n|}\} .$$

Soit T la matrice $|\mathfrak{B}_n| \times |\mathfrak{B}_n|$ de transition définie par

$$T_{i,j} = \frac{1}{n} \text{ Card } \{x. / \varphi_x . \mathcal{A}_j = \mathcal{A}_i\} \qquad \text{pour } i,j \in [1, |\mathfrak{B}_n|] .$$

Cette matrice représente la transformation φ sur l'ensemble $\mathfrak{B}_n$ des arborescences à n sommets.

Trehel a montré [15] que T est une matrice stochastique et que de ce fait, partant d'une arborescence $\mathcal{A}_j$ de $\mathfrak{B}_n$ quelconque (disposition initiale des processus sous forme arborescente), la probabilité $\pi^p_{\mathcal{A}}$ d'obtention de l'arborescence $\mathcal{A}$ au bout de p transformations φ converge vers une limite $\pi_{\mathcal{A}}$ quand p tend vers l'infini.

De plus, le vecteur des probabilités limites

$$\Pi = (\pi_{\mathcal{A}_1}, \ldots, \pi_{\mathcal{A}_{|\mathfrak{B}_n|}})$$

est le vecteur propre de la matrice T associée à la valeur propre 1, c'est à dire le vecteur Π tel que $T\,\Pi = \Pi$.

Par ailleurs, la complexité de l'algorithme d'exclusion mutuelle de Naïmi et Trehel est donnée par

$$M_n = \sum_{\mathcal{A} \in \mathcal{B}_n} \pi_{\mathcal{A}} \, M(\mathcal{A}).$$

Dans [15], Trehel a montré que

$$M_n = \sum_{\mathcal{A} \in \mathcal{B}_n} \pi_{\mathcal{A}} \, N^2(\mathcal{A}) \qquad (1)$$

et a conjecturé que M_n est le (n-1) ième nombre harmonique soit

$$M_n = H_{n-1} = \sum_{i=1}^{n-1} \frac{1}{i}.$$

Pour démontrer ce résultat, nous allons étendre la transformation φ sur l'ensemble des arbres dessinés, objets plus aisés à manipuler de façon algébrique.

3 - EXTENSION DE LA TRANSFORMATION φ AUX ARBRES DESSINES

Un *arbre dessiné* est une arborescence telle que, pour tout sommet i, l'ensemble de ses fils est totalement ordonné.

Etant donné un arbre dessiné $\mathcal{A}$ et un sommet i de cet arbre, nous considérons la transformation φ_i sur cet arbre comme celle exactement décrite par la figure 1. Cette transformation conserve l'ordre des fils de tous les sommets autres que la racine (l'ordre est le même dans $\varphi_i.\mathcal{A}$ et dans $\mathcal{A}$), et l'ordre des fils de la nouvelle racine, le sommet i, est celui décrit par la figure 1.

On pourrait montrer que l'on obtient les mêmes résultats d'existence des probabilités limites $\pi_{\mathcal{A}}$ pour les arbres dessinés que pour les arborescences. Dans la suite de cet article, nous nommerons arbres les arbres dessinés.

Soit $\mathcal{T}_n$ l'ensemble des arbres à n sommets. Nous allons, dans un premier temps, donner une expression des probabilités $\pi_{\mathcal{A}}$ pour tout arbre $\mathcal{A}$ de $\mathcal{T}_n$, et, dans un second temps, montrer que

$$\sum_{\mathcal{A}\in\mathcal{T}_n} \pi_{\mathcal{A}}\, N^2(\mathcal{A}) = H_{n-1} \quad . \tag{2}$$

Ainsi, nous en déduirons que la complexité moyenne de l'algorithme de Naïmi-Trehel est

$$M_n = H_{n-1} \quad . \tag{3}$$

En effet, $\mathcal{B}_n$ étant l'ensemble des arborescences à n sommets, considérons l'application surjective (et non injective) suivante

$$\Theta : \mathcal{T}_n \longrightarrow \mathcal{B}_n$$

qui a tout arbre $\mathcal{A}\in\mathcal{T}_n$ associe l'arborescence $\Theta(\mathcal{A})$ en ignorant l'ordre total sur les fils de chaque sommet de $\mathcal{A}$. D'après la définition de la transformation φ, on a

$$\forall \mathcal{A}\in\mathcal{B}_n,\ \pi_{\mathcal{A}} = \sum_{b\in\Theta^{-1}(\mathcal{A})} \pi_b$$

avec

$$\Theta^{-1}(\mathcal{A}) = \{ b\in\mathcal{T}_n \ /\ \Theta(b)=\mathcal{A} \}.$$

Or,

$$\forall \mathcal{A}\in\mathcal{B}_n,\ \forall b\in\Theta^{-1}(\mathcal{A}),\ N^2(\mathcal{A}) = N^2(b) \quad .$$

On déduit donc aisément des égalités (1) et (2) l'égalité (3).

4 - LES PROBABILITES $\Pi_{\mathcal{A}}$ POUR LES ARBRES

Pour obtenir une expression de ces probabilités, nous allons considérer des séries formelles en variables non commutatives x et $\bar{x}$ dans l'algèbre $\mathbb{Q}\langle\langle x,\bar{x}\rangle\rangle$.

Soient X l'alphabet $\{x,\bar{x}\}$ et D le langage de Dyck

restreint défini par les règles de grammaire

$$D \longrightarrow \varepsilon \quad \text{(le mot vide)},$$
$$D \longrightarrow DxD\bar{x} \ .$$

Ces règles signifient que tout mot w de D, s'il n'est pas vide, s'écrit de manière unique sous la forme $w=uxv\bar{x}$ où u et v sont deux mots de D. Cette unique factorisation correspond à l'unique décomposition d'un arbre, et au codage décrit par la figure 2.

Notation. Dans toute la suite, pour des raisons de clarté de l'exposé, nous noterons

- $\mathcal{A}[u]$ l'arbre codé par le mot $u \in D$,
- $u[\mathcal{A}]$ le mot codant l'arbre $\mathcal{A} \in \mathcal{T}_n$.

Considérons $\{d_n\}_{n>0}$ la famille de polynômes de $\mathbb{Q}\langle\langle x,\bar{x}\rangle\rangle$ codant les arbres à n+1 sommets (ou n arêtes),

$$d_n = \sum_{\mathcal{A} \in \mathcal{T}_{n+1}} u[\mathcal{A}] \ .$$

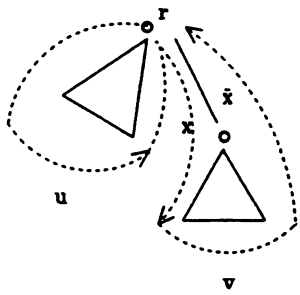


Figure 2. Codage d'un arbre par un mot de Dyck.

Ces polynômes vérifient l'équation récurrente suivante:

$$\begin{cases} d_0 = 1 , \\ d_{n+1} = \sum_{k=0}^{n} d_k \; x \; d_{n-k} \; \bar{x} . \end{cases}$$

Nous allons définir une deuxième famille de polynômes de $\mathbb{Q}\langle\langle x,\bar{x}\rangle\rangle$ dont les coefficients seront exactement les probabilités limites $\pi_{\mathcal{A}}$.

Définition 1. *$(p_n)_{n\geq 0}$ est la famille de polynômes définie par*

$$\begin{cases} p_0 = 1 , \\ p_{n+1} = \dfrac{1}{n+1} \displaystyle\sum_{k=0}^{n} p_k \; x \; p_{n-k} \; \bar{x} . \end{cases} \tag{4}$$

Nous pouvons énoncer le théorème suivant:

Théorème 2. *Pour tout* $n\geq 0$,

$$p_n = \sum_{\mathcal{A}\in\mathcal{T}_{n+1}} \pi_{\mathcal{A}} \; u[\mathcal{A}] . \tag{5}$$

Soit $\mathcal{P}$ l'ensemble des séries de $\mathbb{Q}\langle\langle x,\bar{x}\rangle\rangle$ ayant pour support le langage de Dyck. La démonstration de ce théorème nécessite la définition d'un opérateur μ qui agit sur les séries de $\mathcal{P}$ et qui représente exactement la transformation φ sur les arbres. Cet opérateur utilise une opération $\square$ définie comme suit:

Définition 3. *L'opération* $\square$ *est définie de* $\mathcal{P}\times\mathcal{P}$ *dans* $\mathcal{P}$ *par*

$0 \square s = 0$ *pour tout* $s\in\mathcal{P}$,

$(rxs\bar{x}) \square t = rxst\bar{x}$ *pour tout* r,s,t *de* $\mathcal{P}$.

La figure 3 montre l'action de l'opération $\square$ sur un mot de D.

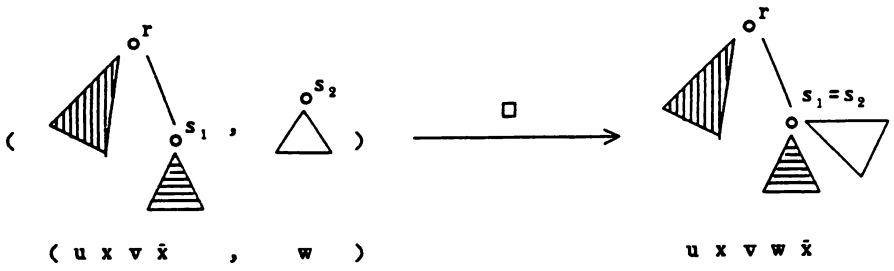


Figure 3. L'opération □ sur les mots de Dyck.

Définition 4.

Soit μ *l'opérateur défini sur* $\mathcal{P}$ *par* $\mu=\nu+\mathrm{Id}$ *avec*

$\forall r\in\mathcal{P}$ $\mathrm{Id}(r)=r$,

$\nu(1)=0$,

$\forall r,s\in\mathcal{P}$ $\nu(rxs\hat{x}) = \nu(r) \square (xs\hat{x}) + \mu(s)xr\hat{x}$.

Notations.

Etant donné un arbre $\mathcal{A}$,

- si i est un sommet de $\mathcal{A}$, nous le noterons par $i\in\mathcal{A}$,
- si i est un sommet de $\mathcal{A}$ différent de la racine, nous le noterons par $i\in\in\mathcal{A}$.

Nous montrons le lemme suivant qui justifie le fait que μ représente la transformation φ.

Lemme 5. *Pour tout mot* $u\in D$, *on a*

$$\mu(u)= \sum_{i\in\mathcal{A}[u]} u[\varphi_i.\mathcal{A}[u]] \ .$$

En effet, pour tout mot $u\in D$, nous avons

$$\sum_{i\in\mathcal{A}[u]} u[\varphi_i.\mathcal{A}[u]] = u + \sum_{i\in\in\mathcal{A}[u]} u[\varphi_i.\mathcal{A}[u]] \ .$$

Montrons par récurrence sur la longueur de u, notée |u|, que

$$\nu(u) = \sum_{i \in \mathcal{A}[u]} u[\varphi_i . \mathcal{A}[u]] \quad . \qquad (6)$$

- Si |u|=0, alors u est le mot vide et $\mathcal{A}[u]$ l'arbre réduit à un seul sommet qui est donc sa racine. Ainsi l'égalité (6) est exacte car $\nu(1)=0$.

- Supposons l'égalité (6) exacte jusqu'au rang n. Soit u un mot de D, |u|=2n+2. Alors $u=vxw\tilde{x}$ avec v et w deux mots de D de longueur inférieure à 2n. On a

$$\sum_{i \in \mathcal{A}[u]} u[\varphi_i . \mathcal{A}[u]] = A + B \ ,$$

avec

$$A = \sum_{i \in \mathcal{A}[v]} u[\varphi_i . \mathcal{A}[u]]$$

et

$$B = \sum_{i \in \mathcal{A}[w]} u[\varphi_i . \mathcal{A}[u]] \quad .$$

Par définition de l'opération $\square$ et de l'application φ_i,

$$A = \sum_{i \in \mathcal{A}[v]} u[\varphi_i . \mathcal{A}[v]] \square xw\tilde{x} \quad ,$$

et donc d'après l'hypothèse de récurrence

$$A = \nu(v) \square xw\tilde{x} \quad .$$

De plus

$$B = u[A[w]]xv\tilde{x} + \sum_{i \in \mathcal{A}[w]} u[\varphi_i . \mathcal{A}[w]]xv\tilde{x} \ ,$$

et donc

$$B = (w+\nu(w))x v \check{x} = \mu(w)x v \check{x} \ .$$

Ainsi, on obtient

$$\sum_{i \in \mathcal{A}[u]} u[\varphi_i . \mathcal{A}[u]] = \mu(w)x v \check{x} + \nu(v) \square x w \check{x}$$

$$= \nu(v x w \check{x}) \ ,$$

ce qui achève la preuve du lemme 5.

En utilisant l'opérateur μ, nous allons démontrer le théorème 2 par récurrence sur n. Nous allons prouver que le polynôme p_n (associé aux arbres de $\mathcal{T}_{n+1}$) est point fixe de l'opérateur $\mu/(n+1)$ ce qui équivaut à montrer que le vecteur des probabilités limites est vecteur propre associé à la valeur propre 1 de la matrice de transition correspondant à φ.

Montrons donc que

$$\frac{1}{n+1} \mu(p_n) = p_n \ .$$

Il suffit de prouver par récurrence que

$$\nu(p_n) = n \, p_n \ . \qquad (7)$$

- L'égalité (7) est évidemment vraie pour n=0.

- Supposons cette égalité vraie jusqu'à l'ordre n. En utilisant la formule de récurrence (4) donnant p_{n+1}, on a

$$\nu(p_{n+1}) = \alpha+\beta$$

avec

$$\alpha = \frac{1}{n+1} \sum_{k=0}^{n} \nu(p_k) \ \square \ (x p_{n-k} \check{x}) \ ,$$

et

$$\beta = \frac{1}{n+1} \sum_{k=0}^{n} \mu(p_{n-k}) x p_k \tilde{x} \ .$$

Nous obtenons successivement pour α:

$$\begin{aligned}
\alpha &= \frac{1}{n+1} \sum_{k=0}^{n} k\, p_k \,\square\, (x p_{n-k} \tilde{x}) \\
&= \frac{1}{n+1} \sum_{k=0}^{n} \sum_{i=0}^{k-1} p_i x p_{k-i-1} x p_{n-k} \tilde{x}\ \tilde{x} \\
&= \frac{1}{n+1} \sum_{i=0}^{n-1} p_i x \Big(\sum_{k=i+1}^{n} p_{k-i-1} x p_{n-k} \tilde{x} \Big) \tilde{x} \\
&= \frac{1}{n+1} \sum_{i=0}^{n-1} (n-i)\, p_i x p_{n-i} \tilde{x} \\
&= \frac{1}{n+1} \sum_{k=0}^{n} k\, p_{n-k} x p_k \tilde{x} \ .
\end{aligned}$$

On a également

$$\beta = \frac{1}{n+1} \sum_{k=0}^{n} (n-k+1)\, p_{n-k} x p_k \tilde{x} \ ,$$

et en sommant

$$\nu(p_{n+1}) = \sum_{k=0}^{n} p_{n-k} x p_k \tilde{x} \ .$$

L'égalité (7) est donc démontrée et par suite le théorème 2. Nous en déduisons immédiatement le corollaire suivant

Corollaire 6. *Soit* w *un mot de Dyck de longueur* 2n *se factorisant de manière unique en* w=uxvx̆, *et soit* $\mathcal{A}[w]$ *l'arbre ayant* n+1 *sommets associé à* w. *Alors*

$$\pi_{\mathcal{A}[w]} = \frac{1}{n}\, \pi_{\mathcal{A}[u]}\, \pi_{\mathcal{A}[v]} \cdot$$

5 - COMPLEXITE MOYENNE DE L'ALGORITHME DE NAIMI ET TREHEL

Nous prouvons dans ce paragraphe que la complexité moyenne de l'algorithme de Naïmi-Trehel est donnée par le théorème suivant.

Théorème 7. *Pour tout* n≥1, *on a*

$$M_n = \sum_{\mathcal{A}\in\mathcal{T}_n} \pi_{\mathcal{A}}\, N^2(\mathcal{A}) = H_{n-1} \cdot$$

D'après le théorème 2, les probabilités $\pi_{\mathcal{A}}$ des arbres $\mathcal{A}$ de $\mathcal{T}_{n+1}$ sont données par la suite de polynômes $(p_n)_{n\geq 0}$ de $\mathbb{Q}\langle\langle x,\breve{x}\rangle\rangle$ définie au paragraphe 3 définition 1. Considérons maintenant la suite de polynômes r_n de $\mathbb{Q}\langle\langle x,\breve{x},y\rangle\rangle$ définie par

$$\begin{cases} r_0 = 1 \,, \\ r_{n+1} = \dfrac{1}{n+1} \displaystyle\sum_{k=0}^{n} y\, r_k x p_{n-k} \breve{x} \cdot \end{cases} \tag{8}$$

Compte-tenu des propriétés du codage des arbres par les mots de Dyck et du fait que

$$p_n = \sum_{\mathcal{A}\in\mathcal{T}_{n+1}} \pi_{\mathcal{A}}\, u[\mathcal{A}] \,,$$

le polynôme r_n n'est autre que le polynôme

$$r_n = \sum_{\alpha \in \mathcal{T}_{n+1}} \pi_\alpha \, y^{N^2(\alpha)} \, u[\alpha] \quad . \tag{9}$$

Ainsi

$$M_n = \left. \frac{\partial r_{n-1}}{\partial y} \right|_{x=\bar{x}=y=1} .$$

Nous allons utiliser ce dernier fait pour montrer le théorème 7 par récurrence.

- pour n=1, on a $r_0=1$ et donc $M_1=0$.

- pour n=2, $r_1=yx\bar{x}$ et donc $M_2=1=H_1$.

- Supposons le résultat du thèorème 7 vérifié jusqu'à l'ordre n. En utilisant l'égalité (8), nous calculons

$$R_n = \frac{\partial r_n}{\partial y} .$$

On a

$$R_n = \frac{1}{n} \sum_{k=0}^{n-1} r_k x p_{n-1-k} \bar{x} + \frac{1}{n} \sum_{k=0}^{n-1} y \frac{\partial r_k}{\partial y} x p_{n-1-k} \bar{x} .$$

De plus, d'après les égalités (5) et (9), il est clair que pour tout $k \geq 0$,

$$r_k \Big|_{x=\bar{x}=y=1} = p_k \Big|_{x=\bar{x}=1} = 1 .$$

Ainsi nous obtenons

$$M_{n+1} = 1 + \frac{1}{n} \sum_{k=0}^{n-1} H_k .$$

On utilise alors le lemme suivant dont la preuve est immédiate.

Lemme 8. *Pour tout* $n \geq 1$, *la suite des nombres harmoniques vérifie*

$$\sum_{k=0}^{n} H_k = (n+1)(H_{n+1}-1) \ .$$

On obtient donc le résutat attendu, à savoir

$$M_{n+1} = H_n \ .$$

6 - REMARQUES FINALES

La famille de polynômes $\{p_n\}_{n\geq 0}$ définie précédement, nous a conduit à étudier la famille de polynômes $\{s_n\}_{n\geq 0}$ définie par

$$\begin{cases} s_0 = 1 \ , \\ s_{n+1} = (n+1) \sum_{k=0}^{n} s_k \ x \ s_{n-k} \ \bar{x} \ . \end{cases} \tag{10}$$

Ces polynômes représentent "l'inverse" des polynômes p_n définis paragraphe 3 définition 1. Plus précisément

Proposition 9. *La famille de polynômes* $\{s_n\}_{n\geq 0}$ *vérifie*

$$s_n = \sum_{\mathcal{A} \in \mathcal{T}_{n+1}} \frac{1}{\pi_{\mathcal{A}}} u[\mathcal{A}] \ ,$$

et $s_n \in \mathbb{N}\langle\langle x, \bar{x} \rangle\rangle$.

Cette propriété résulte immédiatement de la définition des polynômes $\{p_n\}_{n\geq 0}$ et $\{s_n\}_{n\geq 0}$ et du corollaire 6.

Dans un autre article [7], nous montrons que les polynômes $\{s_n\}_{n\geq 0}$ sont liés à un problème combinatoire concernant

l'énumération d'une classe particulière d'involutions sans point fixe. En particulier, nous montrons que le nombre

$$s_n\Big|_{x=\bar{x}=1}$$

énumère les involutions sans point fixe sur $[1,2(n+1)]$ qui constituent un système propre [13]. Ainsi, nous obtenons une récurrence simple sur ces nombres. Cette récurrence nous permet de montrer que les tables de nombres données dans [13] et les suites n°783 et n°1468 de [12] sont erronées. De plus, nous montrons que les polynômes $(s_n)_{n\geq 0}$ donnent une distribution nouvelle de ces objets, distincte de celle considérée par Touchard [13] qui énumère ces involutions suivant le nombre de points doubles.

D'autre part, il existe une démonstration totalement bijective reposant sur la combinatoire des permutations et les arbres binaires croissants. Cette combinatoire a été abondamment décrite par Françon, Viennot, Vuillemin [8] à propos des *pagodes*. Dans un article en cours de rédaction nous donnons la démonstration bijective du théorème 7 concernant l'algorithme de Naïmi et Trehel.

BIBLIOGRAPHIE

[1] C. BERGE, *Graphes et hypergraphes* , Dunod, Paris (1970).

[2] J. BERSTEL, *Séries formelles en variables non commutatives et applications*, actes de la 5ième école de printemps d'Informatique théorique, Vieux-Boucau les Bains, 1977, LITP et ENSTA, Paris (1978).

[3] J. BERSTEL, Transductions and context-free languages, Teubner, Stuttgart (1979).

[4] L. CHOTTIN, Etude syntaxique de certains langages solutions d'équations avec opérateurs, Theor. Comp. Sc., vol. 5, 1977, 51-84.

[5] R. CORI, J. RICHARD, Enumération des graphes planaires à l'aide des séries formelles en variables non commutatives, Discrete Mathematics, Vol. 2, 1972.

[6] S. DULUCQ, Equations avec opérateurs: un outil combinatoire, Thèse de 3ème cycle, Bordeaux, 1981.

[7] S. DULUCQ, J.G. PENAUD, Enumération des arbres et graphes de cordes connexes, preprint 1987.

[8] J. FRANCON, G. VIENNOT, J. VUILLEMIN, Description and analysis of an efficient priority queue representation, Proc. 19th FOCS, Ann Harbor, Michigan, 1978, Article complet: L.R.I. n°12, (1978), Université d'Orsay.

[9] S. HART, M. SHARIR, Nonlinearity of Davenport-Schinzel sequences and of generalized path compression schemes, Combinatorica 6(2) (1986) 151-177.

[10] K. MELHORN, *Data structures and algorithms 1: Sorting and searching*, Springer Verlag, New-York/Berlin, 1984.

[11] M. NAIMI, M. TREHEL, Un algorithme distribué d'exclusion mutuelle en Log(n), Technique et Science Informatiques 6(1987), 141-150.

[12] N.J. SLOANE, *A handbook of integer sequences*, Academic Press, New-york, 1979.

[13] J. TOUCHARD, Sur un problème de configurations et sur les fractions continues, Can. J. Math., Vol. 4 (1952) 2-25.

[14] R.E. TARJAN, Efficiency of a good but not linear set union algorithm, Journal of the Association for Computing Machinery, Vol. 22, N°2, April 1975, 215-225.

[15] M. TREHEL, Propriétés des nombres harmoniques H_n liés à des transformations d'arborescences, Preprint Septembre 1986.

Tableaux de Havender standards

D. Gouyou-Beauchamps
Laboratoire de Recherche en Informatique
Bât. 490, Université de Paris Sud
91405 Orsay Cedex, France

Abstract

We give the generating function for standard Havender tableaux of height 2 and exact formulas for these tableaux according the number of empty cells. The problem is related to the comparison of algorithms controlling concurrent access to a database.

1 Introduction

Un problème important en pratique comme en théorie concernant les bases de données est de comparer les performances des algorithmes de maintien de leur cohérence en particulier du point de vue du parallélisme qu'ils autorisent. J.Françon [8] propose de mesurer la fréquence des exécutions correctes dans les exécutions à priori possibles. ce qui est une mesure de la fréquence des modifications qu'un algorithme de contrôle de concurrence est amené à apporter à l'exécution des transactions telles qu'elles se présentent, en quelque sorte une mesure de la quantité de contrôle à effectuer.

Dans ce contexte une base de données est vue comme une collection d'objets appelés entités dont les valeurs doivent, à chaque instant, vérifier des relations caractèristiques de la situation modélisée par la base de données. Ces relations sont appelées les contraintes d'intégrité de la base [7][1][2][16]. Une transaction est une suite ordonnée d'opérations atomiques sur les entités dont l'effet global préserve l'intégrité de la base. Par conséquent toute composition séquentielle de transactions, appelée exécution sérielle, transforme un état cohérent en un état cohérent.

Dans la pratique, plusieurs transactions peuvent accéder en paralléle à une même base de données. Le problème de la sérialisabilité (dit aussi du maintien de la cohérence ou du contrôle de la concurrence d'accès) est de synchroniser l'exécution des transactions de façon à n'autoriser que les exécutions qui sont équivalentes à une exécution sérielle. La notion d'équivalence qui est retenue est que les opérations conflictuelles ont le même ordre relatif dans les exécutions équivalentes, les actions conflictuelles étant les accès à une même entité par des transactions différentes. La synchronisation voulue est réalisée par un algorithme dit de contrôle de la concurrence des accès ou encore de maintien de cohérence.

Dans [7], J.Françon propose de modéliser l'algorithme, les transactions et les entités par des shuffles de mots sur un alphabet donné. Le dénombrement d'ensembles d'exécutions revient alors à énumérer des classes de commutation partielle. Ces classes peuvent être caractérisées en terme de tableaux de Young. Pour traiter plus finement l'analyse du contrôle de l'accès à une base données, J.Françon a défini une classe de tableaux, généralisant ceux de Young, qu'il a appelé *Tableaux de Havender*, en raison de leur liens avec l'algorithme de Havender [12] qui permet d'éviter les interblocages dans les systèmes d'exploitation. En effet, le modèle utilisant les tableaux de Havender impose que, dans toutes les transactions, les actions atomiques soient effectuées dans le même ordre.

Par contre, avec ce modèle, les actions atomiques ne sont pas obligées de toutes apparaître dans une transaction.

Le but de cet article est d'étudier ces tableaux et de donner des formules d'énumération exactes pour la hauteur 2. Les résultats sont à rapprocher de ceux obtenus dans le cas des tableaux de Young où on ne connait de formules exactes que jusqu'à la hauteur 5 [11][13] ou bien des formules asymptotiques pour une hauteur quelconque [17].

Les méthodes utilisées sont la combinatoire bijective et le codage par des mots de langages algébriques. Le principal résultat obtenu est que le nombre de tableaux de Havender ayant 2 lignes, p colonnes et q cases vides ($p > 0$, $0 \leq q \leq 2p$) est égal à:

$$\sum_{j=max(0,p-q)}^{\lfloor \frac{2p-q}{2} \rfloor} \frac{p!(4p-2q-2j)!}{(2p-q-2j)!(j+1)!(2p-q-j)!(2p-q-j)!(q-p+j)!} .$$

Un corollaire de ce résultat est une preuve des identites combinatoires suivantes:

$$\sum_{i=0}^{p} \binom{p}{i}(2^{i+1}-1)C_i = 1 + \sum_{n=1}^{p} \sum_{i=0}^{\lfloor \frac{n-1}{2} \rfloor} \binom{n-1}{2i} C_i (4^{i+1}5^{n-2i-1} - 3^{n-2i-1})$$
$$= \sum_{q=0}^{2p} \sum_{j=max(0,p-q)}^{\lfloor \frac{2p-q}{2} \rfloor} \frac{p!(4p-2q-2j)!}{(2p-q-2j)!(j+1)!(2p-q-j)!(2p-q-j)!(q-p+j)!} .$$

2 Définitions

Soit $\lambda = (\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_k)$ avec $\lambda_k \neq 0$ une partition d'un entier positif n. Le *diagramme de Ferrers* F_λ de forme λ est le tableau constitué de k lignes justifiées à gauche, la $i^{ème}$ ligne ayant λ_i cases si on prend la convention d'écrire les lignes de bas en haut. Un *tableau de Young standard* de forme λ est un remplissage des n cases de F_λ avec les nombres $1, 2, \ldots, n$ de façon que les nombres forment une suite croissante dans chaque ligne et chaque colonne, les lignes étant lues de gauche à droite et les colonnes de bas en haut. La figure 1 donne un exemple d'un tableau de Young standard de forme $\lambda = (5, 3, 3, 2)$.

9	12			
4	10	13		
3	5	8		
1	2	6	7	11

Figure 1

Appelons *tableau de Havender standard* un tableau de forme rectangulaire ayant p lignes et r colonnes dont certaines cases contiennent un entier pris entre 1 et n, entier donné tel que $n \leq pr$, de façon que tous les entiers entre 1 et n apparaissent une seule fois et que. pour toute ligne et pour toute colonne du tableau, la suite des entiers qui y apparaissent soit ou bien vide, ou bien croissante. La figure 2 présente un exemple pour $p = 3$, $r = 4$ et $n = 7$.

5			7
2	3		6
	1		4

Figure 2

Pour $n = rp$, un tableau de Havender standard est évidemment un tableau de Young standard.

Nous utiliserons des ensembles finis appelés *alphabets*, leurs éléments étant des *lettres*. Un *mot* est une suite finie de lettres. La suite vide (ou mot vide) est notée 1. L'ensemble A^* des mots sur un alphabet A (le monoïde libre généré par A) est muni de l'opération binaire de concaténation et ainsi un mot peut être considéré comme la concaténation de ses lettres. Bien entendu, 1 est l'élément neutre de cette opération.

La longueur d'un mot f, notée $|f|$, est le nombre de lettres qui composent f. Pour une lettre x de l'alphabet, $|f|_x$ note le nombre de lettres de f qui sont égales à x. Un mot f' est un *facteur* d'un mot f s'il existe deux mots f_1 et f_2 tels que $f = f_1 f' f_2$. Si f_1 est le mot vide, alors f' est un *facteur gauche* de f.

Soit l'alphabet $X = \{x, \overline{x}\}$. Le morphisme δ de X^* dans N est défini par $\delta(x) = 1$ et $\delta(\overline{x}) = -1$. Le langage de Dyck D est l'ensemble des mots f de X^* tels que $\delta(f) = 0$ et pour tout facteur gauche f' de f, $\delta(f') \geq 0$. Il est bien connu que $card(D \cap X^{2n}) = \frac{1}{n+1}\binom{2n}{n} = C_n$ (le nombre de Catalan [3][10]).

Soit l'alphabet $Y = \{x, \overline{x}, y\}$. Le morphisme δ' de Y^* dans N est défini par $\delta'(x) = 1$, $\delta'(\overline{x}) = -1$ et $\delta'(y) = 0$. Le langage de Motzkin M est l'ensemble des mots f de Y^* tels que $\delta'(f) = 0$ et que, pour tout facteur gauche f' de f, $\delta'(f') \geq 0$. Il est aussi bien connu que $card(M \cap Y^n) = \sum_{i=0}^{\lfloor n/2 \rfloor} \binom{n}{2i} C_i = M_n$ (le nombre de Motzkin [4][15]).

Les langages D et M (vus comme des séries formelles non commutatives) vérifient les équations:

$$D = 1 + xD\overline{x}D \tag{1}$$

$$M = 1 + yM + xM\overline{x}M \tag{2}$$

L'équation (1) signifie qu'un mot de D est soit le mot vide, soit un mot qui peut être décomposé de façon unique en $xf_1\overline{x}f_2$ où f_1 et f_2 sont eux-mêmes des mots de Dyck.

L'équation (2) signifie qu'un mot de M est soit le mot vide, soit un mot commençant par un y suivi d'un mot de M, soit un mot qui peut être décomposé de façon unique en $xf_1\overline{x}f_2$ où f_1 et f_2 sont aussi des mots de M.

Enfin, on dit qu'un mot f du langage de Dyck D est *premier* s'il est de la forme $f = xf'\overline{x}$ où f' est lui-même un mot de Dyck.

3 Séries génératrices

Dans tout ce paragraphe, tableau signifiera tableau de Havender standard de hauteur 2 ($p = 2$) ayant au moins une colonne ($r \geq 1$). Soient deux tableaux P et P', P ayant les paramètres $p = 2$.

r et n et P' les paramètres $p' = 2$, r' et n'. On définit la concaténation de P et P' (notée PP') comme étant le tableau ayant les paramètres 2, $r + r'$ et $n + n'$ et formé par la juxtaposition des tableaux P et P' (P à gauche et P' à droite) où les nombres $1, 2, \ldots, r'$ dans P' sont remplacés par $r + 1, r + 2, \ldots, r + r'$. La figure 3 donne un exemple de la concaténation de deux tableaux.

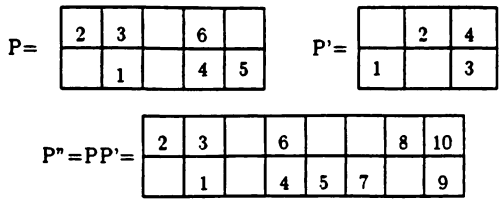


Figure 3

Un tableau est dit *premier* s'il ne peut être décomposé en la concaténation de deux tableaux. Ainsi un tableau est soit premier, soit décomposable de façon unique en la concaténation d'au moins deux tableaux premiers. La figure 4 donne un exemple de décomposition d'un tableau non premier en la concaténation de tableaux premiers.

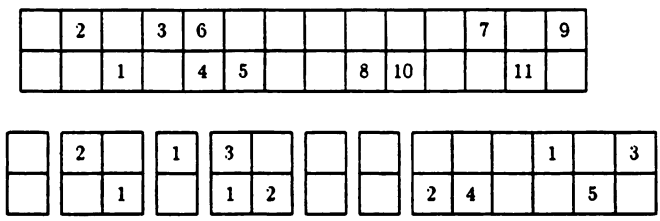


Figure 4

Nous allons diviser les tableaux premiers en deux classes:

- la classe α comprenant d'une part le tableau formé d'une colonne de deux cases vides et d'autre part les tableaux où le nombre 1 apparaît sur la première ligne ou sur la première case de la deuxième ligne,
- la classe β comprenant les tableaux où le nombre 1 apparaît sur une case de la deuxième ligne qui n'est pas la première.

Nous dirons qu'un tableau est de *type* α (resp. β) s'il est décomposable en la concaténation de tableaux appartenant tous à la classe α (resp. β).

Soit l la série énumératrice des tableaux de type α, série en deux variables commutatives x et y où x compte toutes les cases du tableau et y les cases vides. Ainsi $l = \sum_{p,q} l_{2p,q} x^{2p} y^q$ où $l_{2p,q}$ est le nombre de tableaux de Havender de type α ayant 2 lignes, p colonnes et q cases vides.

Théorème 1 $l = \frac{1}{2x^2}(1 - 2x^2 - 2x^2y - x^2y^2 - \sqrt{(1 - x^2(y+2)^2)(1 - x^2y^2)})$

PREUVE:

Nous allons coder les tableaux de type α par des mots sur l'alphabet $\{x, \overline{x}, y, \overline{y}\}$. Pour cela nous allons parcourir le tableau case par case, et à chaque case recontrée nous écrirons une lettre:

- un x pour une case pleine de la première ligne,
- un $\overline{x}$ pour une case pleine de la deuxième ligne,
- un y pour une case vide de la première ligne,
- un $\overline{y}$ pour une case vide de la deuxième ligne.

Les régles définissant l'ordre de parcours sont les suivantes:

- on parcourt les cases dans l'ordre croissant des nombres qu'elles contiennent,
- si la première ligne commence par des cases vides, on parcourt ces cases vides jusqu'à la première case non vide, puis on va à la case numérotée 1,
- sur la première ligne, on parcourt les cases vides qui suivent une case pleine, s'il y en a, immédiatement après avoir lu cette case pleine,
- sur la deuxième ligne, on parcourt les cases vides qui précèdent une case pleine, s'il y en a, juste avant de lire cette case pleine,
- on finit le parcours par les cases vides qui terminent la deuxième ligne, s'il y en a.

La figure 5 donne un exemple de codage.

1		4	5	8	9	10		11		
		2	3		6		7		12	13

$yy\overline{x}xxy\overline{y}\overline{x}\overline{x}xyxy\overline{x}\overline{x}\overline{x}y\overline{x}xx\overline{y}\overline{y}$

Figure 5

Soit φ le morphisme de $\{x, \overline{x}, y, \overline{y}\}$ sur $\{x, \overline{x}\}$ défini par $\varphi(x) = \varphi(y) = x$ et $\varphi(\overline{x}) = \varphi(\overline{y}) = \overline{x}$. Le langage L des mots qui codent les tableaux de type α est caractérisé de la façon suivante: f appartient à L si et seulememt si il vérifie les deux propriétés:

i) $\varphi(f)$ appartient à D, le langage de Dyck,

ii) $f = f_1\overline{f}_1\overline{x}xf_2\overline{f}_2\overline{x}xf_3\overline{f}_3\overline{x}\ldots xf_k\overline{f}_k$ où:

- f_i est un mot sur l'alphabet $\{x, y\}$ pour $i = 2, 3, \ldots, k$,
- $\overline{f}_j$ est un mot sur l'alphabet $\{\overline{x}, \overline{y}\}$ pour $j = 1, 2, \ldots, k-1$,
- f_1 est un mot non vide sur l'alphabet $\{x, y\}$,
- $\overline{f}_k$ est un mot non vide sur l'alphabet $\{\overline{x}, \overline{y}\}$.

La condition i) indique que f est le codage d'un tableau de type α, c'est à dire qu'à tout instant de la lecture, le nombres de cases lues sur la première ligne est supérieur ou égal au nombre de cases lues sur la deuxième ligne. La condition ii) reflète les régles de lecture des cases vides: lorsqu'on passe de la deuxième ligne à la première ligne, on vient de lire une case pleine sur la deuxième ligne et on va lire une case pleine sur la première ligne.

Nous noterons L' le langage constitué par les mots de L commençant par x. Ainsi tout mot f de L est:

- soit $x\overline{x}$, $x\overline{y}$, $y\overline{x}$ ou $y\overline{y}$,
- soit de la forme af_1b avec $a \in \{x, y\}$, $b \in \{\overline{x}, \overline{y}\}$ et $f_1 \in L'$,
- soit de la forme $a\overline{x}f_1$ avec $a \in \{x, y\}$ et $f_1 \in L'$,
- soit de la forme $af_1\overline{x}f_2$ avec $a \in \{x, y\}$, $f_1 \in L$ et $f_2 \in L'$.

L et L' vérifient donc l'équation de langages:

$$L = x\overline{x} + x\overline{y} + y\overline{x} + y\overline{y} + xL\overline{x} + xL\overline{y} + yL\overline{x} + yL\overline{y} + x\overline{x}L' + y\overline{x}L' + xL\overline{x}L' + yL\overline{x}L' \qquad (3)$$

Comme les décompositions reflétées par l'équation (3) sont uniques, on peut passer en commutatif pour obtenir une équation pour l et l', à condition d'appliquer en même temps le morphisme qui transforme x et $\overline{x}$ en x, et y et $\overline{y}$ en xy de façon à obtenir exactement la série énumératrice l recherchée. On trouve ainsi l'équation:

$$l = x^2 + 2x^2y + x^2y^2 + (x^2 + 2x^2y - x^2y^2)l + (x^2 - x^2y)l' + (x^2 + x^2y)ll'$$

Si on remarque que $l = (x + xy)l' + x$, on obtient alors, pour l, l'équation:

$$l = x^2 + 2x^2y + x^2y^2 + (2x^2 + 2x^2y + x^2y^2)l + x^2l^2 \qquad (4)$$

Lorsqu'on résoud (4) et qu'on ne garde que la solution qui donne des coefficients positifs pour la série, on obtient le résultat annoncé. ■

Nous allons maintenant considérer les tableaux de la classe β. On peut vérifier facilement qu'ils peuvent être caractérisés par les propriétés suivantes:

- Ce sont des tableaux premiers.
- La case inférieure de la première colonne et la case supérieure de la dernière colonne ne sont pas vides.
- Dans chaque colonne, au moins une case est vide.
- Si la case supérieure d'une colonne i est occupée par le nombre r, alors il existe $j < i$ et $s > r$ tels que la case inférieure de la colonne j soit occupée par le nombre s.
- Si la case inférieure d'une colonne i est occupée par le nombre r, alors il existe $j > i$ et $s < r$ tels que la case supérieure de la colonne j soit occupée par le nombre s.

Soit r la série énumératrice des tableaux appartenant à la classe β, série de deux variables commutatives x et y où x compte toutes les cases du tableau et y les cases vides. Ainsi $r = \sum_{p,q} r_{2p,q} x^{2p} y^q$ où $r_{2p,q}$ est le nombre de tableaux de la classe β ayant 2 lignes, p colonnes et q cases vides.

Théorème 2 $r = \frac{1}{2}(1 - 2x^2y - x^2y^2 - \sqrt{(1 - x^2y(y+4))(1 - x^2y^2)})$

PREUVE:

Nous allons coder les tableaux de la classe β par des mots sur l'alphabet $\{x, \overline{x}, a, b\}$. Pour cela nous allons parcourir le tableau dans l'ordre suivant:

- on parcourt les cases dans l'ordre croissant des nombres qu'elles contiennent en respectant pour les cases vides les régles suivantes:
- sur la deuxième ligne, on parcourt les cases vides qui précédent une case pleine, s'il y en a, juste avant d'accéder à cette case pleine,
- sur la première ligne, on parcourt, jusqu'à la prochaine case non vide, les cases vides qui suivent une case pleine, immédiatement après avoir lu cette case pleine.

On code alors le tableau de la façon suivante:

- chaque fois qu'on rencontre pour la première fois une colonne où la première ligne est vide et où la deuxième ligne est numérotée, on écrit la lettre a, et lorsqu'on rencontre pour la deuxième fois cette colonne, on n'écrit rien,
- chaque fois qu'on rencontre pour la première fois une colonne où la deuxième ligne est vide et où la première ligne est numérotée, on écrit la lettre x, et lorsqu'on rencontre pour la deuxième fois cette colonne, on écrit une lettre $\overline{x}$,
- chaque fois qu'on rencontre pour la première fois une colonne où la première et la deuxième ligne sont vides, on écrit la lettre b et lorsqu'on rencontre pour la deuxième fois cette colonne, on n'écrit rien.

On peut noter que, dans ce codage, la lettre a représente une case vide et une case pleine, la lettre x une case vide, la lettre $\overline{x}$ une case pleine et la lettre b deux cases vides.

La figure 6 donne un exemple de tableau de la classe β et de son codage.

					1	2		5		8	9
3	4		6	7			10				

$xxbxxaa\overline{x}\overline{x}xa\overline{x}\overline{x}baa\overline{x}$

Figure 6

Soit ψ le morphisme de $\{x, \overline{x}, a, b\}$ dans $\{x, \overline{x}\}$ défini par $\psi(x) = x$, $\psi(\overline{x}) = \overline{x}$ et $\psi(a) = \psi(b) = 1$. On peut alors caractériser un mot f qui code un tableau de la classe β de la façon suivante:

i) $\psi(f)$ est un mot de Dyck premier,

ii) toute suite de $\overline{x}$ consécutifs doit être précédée d'un a,

iii) f commence par un x et finit par un $\overline{x}$.

La condition i) indique que f est le codage d'un tableau de la classe β, c'est à dire qu'à tout instant du parcourt le nombre de cases lues sur la deuxième ligne est supérieur au nombre de cases lues sur la première ligne et il n'y a égalité qu'à la fin de la lecture.

La condition ii) reflète les régles de lecture des cases vides: lorsqu'on passe de la deuxième ligne à la première ligne, on vient de lire une case pleine sur la deuxième ligne et on va lire une case pleine sur la première ligne.

La condition iii) reflète le fait que le tableau est un tableau premier dans la classe β: la deuxième ligne débute forcément par une case vide au-dessus d'une case pleine et le nombre le plus grand est forcément sur la première ligne au-dessous d'une case vide.

Le langage ainsi défini est bien en bijection avec les tableaux de la classe β. Cela se vérifie facilement en considérant l'algorithme de décodage qui suit.

On lit le mot de gauche à droite et on remplit les lignes de gauche à droite de la façon suivante (le numéro courant démarant a 1):

- lorsqu'on lit un x, on met une croix sur la case vide la plus à gauche de la deuxième ligne,
- lorsqu'on lit un $\overline{x}$, on met le numéro courant dans la case vide la plus à gauche de la première ligne, puis on incrémente le numéro courant,
- lorsqu'on lit un a, on met le numéro courant dans la case vide la plus à gauche de la deuxième ligne et une croix dans la même colonne sur la première ligne, puis on incrémente le numéro courant,
- lorsqu'on lit un b, on met une croix sur la case vide la plus à gauche de la deuxième ligne et une croix dans la même colonne sur la première ligne.

Les cases avec des croix représentent donc les cases vides du tableau. La figure 7 donne un exemple de décodage.

Soit R le langage des mots de $\{x, \overline{x}, a, b\}^*$ codant les tableaux de la classe β. Si f appartient à R, il est de la forme $f = xf'\overline{x}$ avec f' vérifiant les trois conditions suivantes:

a) $\psi(f')$ est un mot de Dyck,

b) toute suite de $\overline{x}$ consécutifs doit être précédée d'un a,

c) f' finit par un $\overline{x}$ ou un a.

Appelons N le langage des mots de $\{x, \overline{x}, a, b\}^*$ vérifiant les trois conditions précédentes. Si g appartient a N alors g est:

- soit le mot a,
- soit de la forme ag_1 ou bg' avec $g_1 \in N$,
- soit de la forme $xg_1\overline{x}$ avec $g_1 \in N$,
- soit de la forme $xg_1\overline{x}g_2$avec g_1 et g_2 dans N.

On obtient alors pour N et R les équations non commutatives :

$$N = a + aN + bN + xN\overline{x} + xN\overline{x}N \tag{5}$$

$$R = xN\overline{x} \tag{6}$$

Par passage en commutatif, ce qui est licite puisque les décompositions utilisées dans les équations (5)et (6) sont uniques, et en utilisant un morphisme qui envoie x sur xy, $\overline{x}$ sur x, a sur x^2y et b sur x^2y^2, on obtient pour r l'équation:

$$r = x^4y^2 + x^2y(y-2)r + r^2 \tag{7}$$

En résolvant et en ne gardant que la solution qui donne des coefficients positifs, on obtient le résultat annoncé. ■

Théorème 3 *La série $s = \sum_{p,q} s_{2p,q} x^{2p} y^q$ qui énumère les tableaux de type β (où $s_{2p,q}$ est le nombre de tableaux de type β ayant 2 lignes, p colonnes et q cases vides) est égale à:*

$$s = \frac{1-2x^2y-x^2y^2-2x^4y^2-\sqrt{(1-x^2y(y+4))(1-x^2y^2)}}{2x^2y(2+y+x^2y)}$$

PREUVE:

Puisqu'un tableau de type β est une suite non vide de tableaux de la classe β, la relation $s = r/(1-r)$ relie les séries s et r (ou encore $r = s/(1+s)$). En portant la valeur de r dans l'équation (7) et en résolvant, on obtient le résultat annoncé. ■

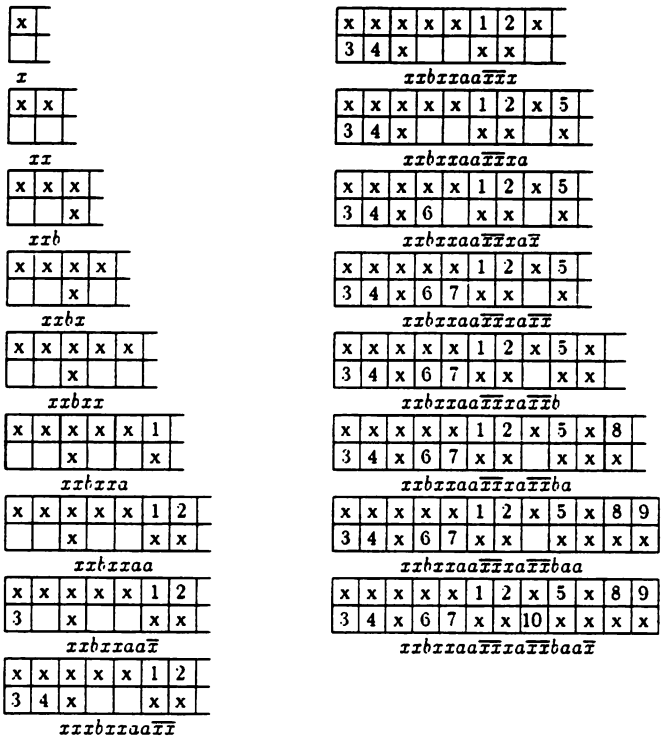


Figure 7

Théorème 4 *La série $t = \sum_{p,q} t_{2p,q} x^{2p} y^q$ qui énumère les tableaux de Havender de hauteur 2 (où $t_{2p,q}$ est le nombre de tableaux Havender ayant 2 lignes, p colonnes et q cases vides) est égale à:*

$$t = \frac{\sqrt{(1-x^2y(y-4))(1-x^2y^2)}-\sqrt{(1-x^2(y+2)^2)(1-x^2y^2)}}{2x^2(1-x^2y^2)} - 1$$

PREUVE:

Il suffit de remarquer qu'un tableau est une suite alternée non vide de tableaux de type α et de tableaux de type β. Il y a quatre possibilités:

- les suites ayant un nombre impair d'éléments premiers et commençant par un tableau de type α (énumérées par $l/(1-sl)$),
- les suites ayant un nombre impair d'éléments premiers et commençant par un tableau de type β (énumérées par $s/(1-ls)$),
- les suites ayant un nombre pair d'éléments premiers et commençant par un tableau de type α (énumérées par $ls/(1-ls)$),
- les suites ayant un nombre pair d'éléments premiers et commençant par un tableau de type β (énumérées par $sl/(1-sl)$).

La série t est donc égale à $(s+l+2sl)/(1-sl)$. En remplaçant s et l par leurs valeurs, on obtient le résultat annoncé. ■

Si on on ne s'intéresse plus seulement aux tableaux rectangulaires mais aux tableaux dont la deuxième ligne est de longueur inférieure ou égale à celle de la première, on peut calculer par la même méthode que la série $\sum_{p,q} u_{p,q}x^p y^q$ (où $u_{p,q}$ est le nombre de tels tableaux ayant au plus deux lignes, p cases dont q vides) est égale à:

$$\frac{4x^3y-4x^2-xy-2x-1}{4x^2(1-xy)} + \frac{\sqrt{(1-x^2y(y+4))(1-x^2y^2)}-\sqrt{(1-x^2(y-2)^2)(1-x^2y^2)}}{4x^2(1-x^2y^2)} + \frac{\sqrt{(1-x^2y(y+4))(1-x^2(y+2)^2)}}{4x^2(1-xy)(1-x(y+2))}$$

4 Enumérations

Théorème 5 *$l_p = \sum_{q=0}^{2p} l_{2p,q}$, le coefficient de x^{2p} dans l, est égal à (pour $p \geq 1$):*

- $\sum_{i=0}^{\lfloor \frac{p-1}{2} \rfloor} \binom{p-1}{2i} C_i (1+(y+1)^2)^{p-2i-1}(y+1)^{2i+2}$
- $\sum_{i=1}^{p} \frac{1}{p}\binom{p}{i}\binom{p}{i-1}(1+y)^{2p-2i+2}$

PREUVE:
L'équation(4) peut sécrire:

$$l = x^2(1+y)^2 + x^2(1+(1+y)^2)l + x^2l^2 \tag{8}$$

En faisant le changement de variable $l' = \frac{l}{x^2(1+y)^2}$, on obtient à partir de (8) l'équation suivante pour l':

$$l' = 1 + x^2(1+(1+y)^2)l' + x^4(1+y)^2l'^2 \tag{9}$$

Or le coefficient de x^n dans la série b solution de l'équation:

$$b = 1 + xub + x^2vb^2 \tag{10}$$

est égal à $\sum_{i=0}^{\lfloor n/2 \rfloor} \binom{n}{2i} C_i u^{n-2i}v^i$. En effet b est la série énumératrice des mots de Motzkin où les $\overline{x}$ sont valués par v, les y sont valués par u et les x sont valués par 1 (cf. [5] et [18]).

Le coefficient de x^{2p} dans la série l' est donc $\sum_{i=0}^{\lfloor p/2 \rfloor} \binom{p}{2i} C_i (1+(1+y)^2)^{p-2i}(1+y)^{2i}$.

Et ainsi le coefficient de x^{2p} dans la série l est égal à:

$$\sum_{i=0}^{\lfloor \frac{p-1}{2} \rfloor} \binom{p-1}{2i} C_i (1+(y+1)^2)^{p-2i-1}(y-1)^{2i+2} .$$

D'autre part, un mot de Dyck non vide est:

- soit $x\overline{x}$,
- soit de la forme $x\overline{x}f$ où f est un mot de Dyck non vide,
- soit de la forme $xf\overline{x}$ où f est un mot de Dyck non vide,
- soit de la forme $xf\overline{x}g$ où f et g sont des mots de Dyck non vides.

Si on remplace dans chaque mot de Dyck non vide chaque facteur $x\overline{x}$ par $xy\overline{x}$, le langage A obtenu vérifie l'équation non commutative:

$$A = xy\overline{x} + xy\overline{x}a + xA\overline{x} + xA\overline{x}A \tag{11}$$

La série énumératrice $a = \sum_{p,q} a_{2p,q} x^{2p} y^q$, où $a_{2p,q}$ compte les mots de A ayant q lettres y, p lettres x et p lettres $\overline{x}$, vérifie donc l'équation:

$$a = x^2y + x^2(1+y)a + x^2a^2 \tag{12}$$

Or $a_{2p,q}$ compte aussi les arbres ayant p arêtes et q feuilles. Il est bien connu que $a_{2p,q}$ est le *nombre de Narayana* $p^{-1}\binom{p}{q}\binom{p}{q-1}$ que Kreweras [14] appelle aussi distribution β et qui est parfois appelé *Runyon number*. En rapprochant les équations (12) et (8), on s'apperçoit que $l_{2p,q}$ compte les arbres ayant p arêtes et q feuilles, chaque feuille étant valuée par $(1+y)^2$.
Donc $l_p = \sum_{i=1}^{p} \frac{1}{p}\binom{p}{i}\binom{p}{i-1}(1+y)^{2i} = \sum_{i=1}^{p} \frac{1}{p}\binom{p}{i}\binom{p}{i-1}(1+y)^{2p-2i+2}$. ■

Théorème 6 $r_p = \sum_{q=0}^{2p} r_{2p,q}$, *le coefficient de* x^{2p} *dans la série* r, *est égal à*

$$\sum_{i=0}^{\lfloor \frac{p-2}{2} \rfloor} \binom{p-2}{2i} C_i (2+y)^{p-2-2i} y^n \ .$$

PREUVE:
En faisant le changement de variable $r' = \frac{r}{x^4y^2}$, on obtient, à partir de l'équation (7), l'équation suivante pour r':

$$r' = 1 + yx^2(2+y)r' + x^4y^2r'^2 \tag{13}$$

L'équation (13) est de la même forme que l'équation (10), le coefficient de x^{2p} dans r' est donc $\sum_{i=0}^{\lfloor \frac{p}{2} \rfloor} \binom{p}{2i} C_i (2+y)^{p-2i} y^n$. En reportant cette valeur dans la série r, on obtient le résultat annoncé. ■

Théorème 7 $t_p = \sum_{q=0}^{2p} t_{2p,q}$, *le coefficient de* x^{2p} *dans* t *est égal à:*

$$y^{2p} + \sum_{n=1}^{p} \sum_{i=0}^{\lfloor \frac{n-1}{2} \rfloor} \binom{n-1}{2i} C_i y^{2p-2n} ((1+y)^{2i+2}(1+(1+y)^2)^{n-2i-1} - y^{n+1}(2+y)^{n-2i-1}) \ .$$

PREUVE:
En rapprochant les théorème 1 et 2 et le théorème 4 , on peut établir la relation:

$$t = \left(l - \frac{r}{x^2} + 1\right)\left(\frac{1}{1-x^2y^2}\right) \tag{14}$$

D'autre part, considérons la série $w(x,y) = \sum_{p>0} w_p x^{2p}$. Le coefficient de x^{2p} de la série $\frac{w(x,y)}{1-x^2y^2}$ est égal à $\sum_{n=1}^{p} w_n y^{2p-2n}$.
En portant dans (14) la valeur des coefficients de x^{2p} dans $\frac{l}{1-x^2y^2}$, $\frac{r}{x^2(1-x^2y^2)}$ et $\frac{1}{1-x^2y^2}$, on obtient le résultat annoncé. ■

Théorème 8 *Le coefficient de y^q dans*
$y^{2p} + \sum_{n=1}^{p} \sum_{i=0}^{\lfloor \frac{n-1}{2} \rfloor} \binom{n-1}{2i} C_i y^{2p-2n}((1+y)^{2i+2}(1+(1+y)^2)^{n-2i-1} - y^{n+1}(2+y)^{n-2i-1})$,
c'est à dire le nombre de tableaux de Havender ayant 2 lignes, p colonnes et q cases vides ($p > 0$, $0 \leq q \leq 2p$) est égal à: $\sum_{j=max(0,p-q)}^{\lfloor \frac{2p-q}{2} \rfloor} \frac{p!(4p-2q-2j)!}{(2p-q-2j)!(j+1)!(2p-q-j)!(2p-q-j)!(q-p+j)!}$.

PREUVE:

Soit $v(x,y) = \sum_{p,q} v_{2p,q} x^{2p} y^q$ la série définie par $v(x,y) = t(xy, \frac{1}{y})$. Le coefficient de $x^{2p}q$ dans $v(x,y)$ est le nombre de tableaux de Havender ayant 2 lignes, p colonnes et $2p-q$ cases vides.

$$v(x,y) = \frac{\sqrt{1-x^2-4x^2y}}{2x^2y^2\sqrt{1-x^2}} - \frac{\sqrt{1-x^2-4x^2y-4x^2y^2}}{2x^2y^2\sqrt{1-x^2}} - 1$$

Soit $w(x,y) = \sum_{p,q} w_{2p,q} x^{2p} y^q$ la série définie par $w(x,y) = \frac{1}{1-x^2} v(\frac{x}{\sqrt{1+x^2}}, y)$.

Les coefficients $v_{2p,q}$ et $w_{2i,j}$ sont liés par la relation: $v_{2p,q} = \sum_{i=\lceil q/2 \rceil}^{p} \binom{p}{i} w_{2i,q}$.

$w(x,y)$ est égal à $\frac{\sqrt{1-4x^2y}}{2x^2y^2} - \frac{\sqrt{1-4x^2y-4x^2y^2}}{2x^2y} - \frac{1}{1+x^2}$.

Or le développement de Taylor de $\frac{\sqrt{1-4x^2y}}{2x^2y^2}$ est égal à: $\frac{1}{2x^2y^2} - \sum_{p \geq 0} C_p x^{2p} y^{p-1}$

et celui de $-\frac{\sqrt{1-4x^2(y+y^2)}}{2x^2y^2}$ à: $\frac{-1}{2x^2y^2} + \sum_{p \geq 0} x^{2p} \sum_{k=0}^{p+1} \binom{p+1}{k} C_p y^{p-1+k}$.

Les coefficients $w_{2p,q}$ sont alors donnés par les formules:

- $w_{2p,q} = \binom{p-1}{2q-p} C_p$ pour $0 < p \leq q \leq 2p$
- $w_{2p,0} = (-1)^{p-1}$ pour $p \geq 1$
- $w_{0,0} = 0$

On obtient ainsi pour $v_{2p,q}$ les valeurs suivantes:

$$\begin{aligned} v_{2p,q} &= \sum_{i=\lceil q/2 \rceil}^{min(p,q)} \binom{p}{i} \binom{i+1}{2i-q} C_i \\ &= \sum_{i=\lceil q/2 \rceil}^{min(p,q)} \frac{p!(2i)!}{(p-i)!(2i-q)!(q-i+1)!i!} \\ &= \sum_{j=max(0,q-p)}^{\lfloor q/2 \rfloor} \frac{(2q-2j)!p!}{(p-q+j)!(q-2j)!(j+1)!(q-j)!(q-j)!} \end{aligned}$$

Notons que cette formule est vraie pour $q = 0$ ($v_{2p,0} = 1$) puisque $\sum_{i=1}^{p} \binom{p}{i} (-1)^{i+1} = 1$. Il suffit de constater que $t_{2p,q}$ est égal à $v_{2p,2p-q}$ pour obtenir le résultat annoncé. La figure 8 donne les premières valeurs de $t_{2p,q}$. ■

p/q	0	1	2	3	4	5	6	7	8	9	10	11	12
1	1	2	1										
2	2	6	8	4	1								
3	5	20	36	38	21	6	1						
4	14	70	160	220	202	116	40	8	1				
5	42	252	700	1190	1380	1152	670	260	65	10	1		
6	132	924	3024	6132	8610	8862	6904	4012	1680	490	96	12	1

Figure 8

Soit Y_1 l'alphabet $\{x, \overline{x}, y, a\}$ et ϕ_1 le morphisme de Y_1^* dans $\{x, \overline{x}\}^*$ défini par $\phi_1(x) = \phi_1(y) = x$, $\phi_1(\overline{x}) = \overline{x}$ et $\phi_1(a) = 1$.

Le langage $U \subset Y_1^*$ est défini de la façon suivante: $f \in Y_1^*$ appartient à U si et seulement si il vérifie les 2 conditions suivantes:

i) $\phi_1(f) \in D$, le langage de Dyck,

ii) une suite de a est soit au début de f, soit précédée par un $\overline{x}$.

U_a forme le langage des mots de U commençant par le lettre a et U_x le langage des mots de U ne commençant pas par une lettre a.

On a donc les équations de langages:

$$U = U_a + U_x$$
$$U_a = aU_a + aU_x$$

D'autre part, un mot f de U_x est:

- soit le mot vide 1,
- soit un mot de la forme $f = xf_1\overline{x}f_2$ où $f_1 \in U_x$ et $f_2 \in U$,
- soit un mot de la forme $f = yf_1\overline{x}f_2$ où $f_1 \in U_x$ et $f_2 \in U$.

Donc U_x vérifie l'équation: $U_x = 1 + xU_x\overline{x}U + yU_x\overline{x}U$ et ainsi U est donné par le système d'équations:

$$\begin{cases} U = U_a + U_x \\ U_a = aU_a + aU_x \\ U_x = 1 + xU_x\overline{x}U + yU_x\overline{x}U \end{cases} \tag{15}$$

Théorème 9 *La série énumératrice $u = \sum_{n\geq 0} u_n x^n$ des mots de U (où u_n compte les mots de U ayant n lettres dans l'alphabet $\{x, y, a\}$) est égale à $u = \frac{1-x-\sqrt{(1-9x)(1-x)}}{4x(1-x)}$ et u_n est égal à $\sum_{i=0}^{n} C_i 2^i \binom{n}{i}$.*

PREUVE:

A partir du systéme d'équations (15), en passant en commutatif après avoir appliqué le morphisme qui transforme les lettres x, y et a en x et la lettre $\overline{x}$ en 1, on obtient le système:

$$\begin{cases} u = u_a + u_x \\ u_a = xu_a + xu_x \\ u_x = 1 + 2xu_xu \end{cases}$$

Après élimination, on trouve pour u l'équation du second degré:

$$2(x - x^2)u^2 + (x - 1)u + 1 = 0 .$$

La solution qui donne des coefficients positifs pour u est $\frac{1-x-\sqrt{(1-9x)(1-x)}}{4x(1-x)}$.

D'autre part, un mot f de U ayant n lettres dans l'alphabet $\{x, y, a\}$ est un mot de Dyck de longueur $2i$ $(0 \leq i \leq n)$ où les les lettres x sont remplacées par des x ou des y (2^i possibilités) et où les $n - i$ lettres a sont mises dans les $i + 1$ intervalles délimités par les $\overline{x}$ ($\binom{n}{i}$ possibilités). u_n est donc égal à $\sum_{i=0}^{n} C_i 2^i \binom{n}{i}$. ■

Soit Y_2 l'alphabet $\{x, \overline{x}, a\}$ et ϕ_2 le morphisme de Y_2^* dans $\{x, \overline{x}\}^*$ qui efface les lettres a. Le langage $V \subset Y_2^*$ est défini de la façon suivante: $f \in Y_2^*$ appartient à V si et seulement il vérifie les deux conditions suivantes:

i) $\phi_2(f) \in D$, le langage de Dyck,

ii) une suite de a est soit au début de f, soit précédée par un $\overline{x}$.

V_a forme le langage des mots de V commençant par la lettre a et V_x le langage des mots de V ne commençant pas par un a .

On a donc les équations de langages:

$V + V_a + V_x$
$V_a = aV_a + aV_x$

D'autre part un mot f de V_x est:

- soit le mot vide 1,
- soit un mot de la forme $f = xf_1\overline{x}f_2$ où $f_1 \in V_x$ et $f_2 \in V$.

Donc V_x vérifie l'équation $V_x = 1 + xV_x\overline{x}V$ et ainsi V est donné par le système d'équations:

$$\begin{cases} V = V_a + V_x \\ V_a = aV_a + aV_x \\ V_x = 1 + xV_x\overline{x}V \end{cases} \tag{16}$$

Théorème 10 *La série énumératrice $v = \sum_{n\geq 0} v_n x^n$ des mots de V (où v_n compte les mots de V ayant n lettres dans l'alphabet $\{x, a\}$) est égale à $v = \frac{1-x-\sqrt{(1-5x)(1-x)}}{2x(1-x)}$ et v_n est égal à $\sum_{i=0}^{n} C_i \binom{n}{i}$.*

PREUVE:

A partir du systéme d'équations (16), en passant en commutatif après avoir appliqué le morphisme qui transforme les lettres x et a en x et la lettre $\overline{x}$ en 1, on obtient le système:

$$\begin{cases} v = v_a + v_x \\ v_a = xv_a + xv_x \\ v_x = 1 + xv_xv \end{cases}$$

Après élimination, on trouve pour v l'équation du second degré:

$$(x - x^2)v^2 + (x - 1)v + 1 = 0 .$$

La solution qui donne des coefficients positifs pour v est $\frac{1-x-\sqrt{5x^2-6x+1}}{2x(1-x)}$.

D'autre part, un mot f de V ayant n lettres dans l'alphabet $\{x, a\}$ est un mot de Dyck de longueur $2i$ $(0 \leq i \leq n)$ où les $n - i$ lettres a sont mises dans les $i + 1$ intervalles délimités par les $\overline{x}$ ($\binom{n}{i}$ possibilités). v_n est donc égal à $\sum_{i=0}^{n} C_i \binom{n}{i}$. ■

Théorème 11 $h(x) = \sum_{p\geq 0} h_{2p}x^{2p}$, *la série énumératrice des tableaux de Havender de hauteur deux (où h_{2p} est le nombre de tableaux de Havender ayant 2 lignes et p colonnes), est égale à* $\frac{\sqrt{1-5x^2}-\sqrt{1-9x^2}}{2x^2\sqrt{1-x^2}} - 1$ *et h_{2p} est égal à:*

$$\begin{aligned} h_{2p} &= \sum_{i=0}^{p} \binom{p}{i}(2^{i+1} - 1)C_i \\ &= 1 + \sum_{n=1}^{p} \sum_{i=0}^{\lfloor \frac{n-1}{2} \rfloor} \binom{n-1}{2i} C_i (4^{i+1}5^{n-2i-1} - 3^{n-2i-1}) \\ &= \sum_{q=0}^{2p} \sum_{j=max(0,p-q)}^{\lfloor \frac{2p-q}{2} \rfloor} \frac{p!(4p-2q-2j)!}{(2p-q-2j)!(j-1)!(2p-q-j)!(2p-q-j)!(q-p+j)!} . \end{aligned}$$

PREUVE:

Il suffit de voire que $h(x) = t(x, 1)$. Ainsi, en utilisant le théorème 4, on obtient $h(x) = \frac{\sqrt{1-5x^2}-\sqrt{1-9x^2}}{2x^2\sqrt{1-x^2}} - 1$. En utilisant les résultats des théorèmes 9 et 10, on trouve pour h_{2p} la formule exacte $h_{2p} = \sum_{i=0}^{p} \binom{p}{i}(2^{i+1} - 1)C_i$.

En utilisant le théorème 7, on obtient pour h_{2p} une deuxième formule exacte:

$$h_{2p} = 1 + \sum_{n=1}^{p} \sum_{i=0}^{\lfloor \frac{n-1}{2} \rfloor} \binom{n-1}{2i} C_i (4^{i+1}5^{n-2i-1} - 3^{n-2i-1}) .$$

Enfin en utilisant le théorème 8, après sommation sur q, le nombre de cases vides. on trouve la troisième formule exacte $h_{2p} = \sum_{q=0}^{2p} \sum_{j=max(0,p-q)}^{\lfloor \frac{2p-q}{2} \rfloor} \frac{p!(4p-2q-2j)!}{(2p-q-2j)!(j-1)!(2p-q-j)!(2p-q-j)!(q-p+j)!}$.

Les premières valeurs de h_{2p} sont $(1 \leq p \leq 12)$: 4, 21, 127, 831, 5722, 40879, 300440, 2258455, 17291704, 134417955, 1058279251, 8422155293. ■

Théorème 12 *Le nombre de tableaux de Havender de hauteur 2 ayant p colonnes est asymptotiquement égal à* $\frac{27}{8\sqrt{2\pi}}\frac{9^p}{p\sqrt{p}} + O(9^p p^{-\frac{5}{2}})$ *et le nombre moyen de cases vides dans un tel tableau est asymptotiquement égal à*
$\frac{2}{3}p + O(1)$.

PREUVE:

Le nombre de tableaux de Havender de hauteur deux ayant p colonnes du point de vue asymptotique est obtenu en étudiant les singularités de la série $\frac{\sqrt{1-5z^2}-\sqrt{1-9z^2}}{2z^2\sqrt{1-z^2}}$.

La somme du nombre de cases vides des tableaux de Havender ayant p colonnes est obtenue en étudiant les singularités de la série $\frac{\partial}{\partial y}\left(\frac{\sqrt{(1-z^2y(y+4))(1-z^2y^2)}-\sqrt{(1-z^2(y+2)^2)(1-z^2y^2)}}{2z^2(1-z^2y^2)}\right)\Big|_{y=1}$.

Ce qui donne, en utilisant le système de calcul formel ΛϒΩ [6]: $\frac{9}{4\sqrt{2\pi}}\frac{9^p}{\sqrt{p}} + O(9^p p^{-\frac{3}{2}})$. ■

5 Références

[1] D.Arques, J.Françon, M.T.Guichet, P.Guichet, *Comparison of algorithms controlling concurrent access to a database: a combinatorial approach*, Theoretical Computer Science, 58(1988), 3-16.

[2] D.Arques, P.Guichet, *asymptotic behaviour and comparison of algorithms controlling concurrent access to a database*, Tech. Rept. UHA No. 40, Université de Haute Alsace, 1986.

[3] L.Comtet, *Advanced Combinatorics*, D.Reidel publ. comp. Boston, 1974.

[4] R.Donaghey, L.W.Shapiro, *Motzkin Numbers*,J.C.T., Ser. A, 23(1977), 291-301.

[5] P.Flajolet, *Combinatorial Aspects of Continued Fractions*, Discrete Math., 32(1980), 125-161.

[6] P.Flajolet, B.Salvy, P.Zimmermann, *Lambda-Upsilon-Omega: an assistant algorithms analyser*, Rapport de Recherche N^o 876, INRIA Rocquencourt, 1988.

[7] J.Françon, *Sérialisabilité, communication, mélange et tableaux de Young*, Tech. Rept. UHA No. 27, Université de Haute Alsace, 1985.

[8] J.Françon, *Une approche quantitative de l'exclusion mutuelle*, RAIRO Inform. Théor., 20(1986), 275-289.

[9] J.Françon, *Combinatoire et parallélisme*, Journées Mathématiques et Informatique, Marseille-Luminy, 1987.

[10] H.W.Gould, *Research Bibliography of two Special Number Sequences* rev. ed., Combinatorial Research Institute, Morgantown, W. Va., 1977.

[11] D.Gouyou-Beauchamps, *Standard Young Tableaux of Height 4 and 5*, Europ. J. Combinatorics, 10(1989), 69-82.

[12] J.W.Havender, *Avoiding Deadlock in Multitasking Systems*, IBM Systems Journal, 7(1968), 74-84.

[13] D.E.Knuth, *The art of Computer Programming*, vol. 3, Sorting and Searching, 2nd ed., Addison-Wesley, Reading, Mass., 1973.

[14] G.Kreweras, *Sur les éventails de segments*, Cahiers du B.U.R.O., 15(1970), 3-41.

[15] T.Motzkin, *Relation between hypersurface cross ratio and a combinatorial formula for partitions of a polygon, for permanent preponderance and for non-associative products*, Bul. Amer. Soc., 54(1948), 352-360.

[16] C.H.Papadimitriou, *Serializability of Concurrent Updates*, JACM, 26(1979), 631-653.

[17] A.Regev, *Asymptotic values for degrees associated with strips of Young diagrams*, Adv. Math., 41(1981), 115-136.

[18] G.Viennot, *Une théorie combinatoire des polynômes orthogonaux généraux*, Soc. Math. France (à paraître).