[BRE 88] Brézillon P., P. Fauquembergue & A. Hertz, "SEPT : an expert system approach for the monitoring of EHV substation control equipment", *First Symposium on Expert Systems Application to Power Systems*, August 22-26, 1988, Stockholm Helsinki.

[BRE 91] Brézillon P., D.Y. BAU & A. Hertz, "Use of a simulator and a diagnoser in the SEPT expert system", *International Journal of Applied Intelligence* (forthcoming).

[CAR 88] Carnegie Group Inc. "Knowledge Craft Reference manual, 1988.

[CHA 83] Chandrasekaran B. & S. Mittal, "Deep versus compiled Knowledge Approaches to Diagnostic Problem Solving", *International Journal of Man-Machine Studies*, (1983) : 425-436.

[CHA 89] Chandrasekaran B., J.W. Smith. & J. Sticklen, "Deep Models and Their Relation to Diagnosis", *Artificial Intelligence in Medicine*, (1989) : 29-40.

[CON 89] Console L., D.T. Dupré & P. Torasso, "A theory of diagnosis for incomplete causal models", *Proc. IJCAI* (1989), Detroit, pp. 1311-1317.

[DOY 89] Doyle R.J., S.M. Sellers & D.J. Atkinson, "A Focused, Context-Sensitive Approach to Monitoring", *Proc. IJCAI* (1989), Detroit, pp. 1231-1237.

[DVO 89] Dvorak D.L. & B.J. Kuipers, "Model-based Monitoring of Dynamic Systems", *Proc. IJCAI* (1989), Detroit, pp. 1238-1243.

[FAU 89] Fauquembergue P. & L. Perrot, "An expert system for the fault analysis of electric power systems", *Second Symposium on Expert Systems Application to Power Systems*, July 17-20, 1989, Seattle, USA, pp. 264-268.

[PAT 81] Patil R.S., P. Szolovits, & W.B. Schwartz, "Causal Understanding of Patient Illness in Medical Diagnosis", *Proc. IJCAI* (1981), pp. 893-899.

[REI 87] Reiter R., "A Theory of Diagnosis from First Principles", *Artificial Intelligence*, 32 (1987) : 56-95.

[SAF 90] Safar B., "Répondre à des questions de type POURQUOI-PAS?", Revue d'Intelligence Artificielle, 1990, 4(2) : 101-112.

[SEK 89] Sekine Y., H. Okamoto & T. Shibamoto, "Fault Section Estimation Using Cause-Effect Network", *Second Symposium on Expert Systems Application to Power Systems*, July 17-20, 1989, Seattle, USA, pp. 276-282.

[SEM 86] Sembugamoorthy V., and B. Chandrasekaran, "Functional Representation of Devices and Compilation of Diagnostic Problem Solving Systems", In : *Experience, Memory, and Reasoning*, edited by J. Kolodner and C. Reisbeck (Lawrence Erlbaum Associates publishers, 1986), pp. 47-73.

[WAL 89] Walther E. & A.B. Boveri, "Power Network Diagnosis using Causal and Temporal Reasoning", *Second Symposium on Expert Systems Application to Power Systems*, July 17-20, 1989, Seattle, USA, pp. 290-295.

[WEI 78] Weiss S.M., C.A. Kulikowski, S. Amarel & S. Safir, "A Model-Based Method for Computer-Aided Medical Decision-Making", *Artificial Intelligence*, August 1978, 11 : 145-172.

Patrick BRÉZILLON
Dong Yih BAU

Article soumis en avril 1991

# Sémantique opérationnelle et compilation des systèmes de production

**François Fages* — Rémi Lissajoux**

*LCR Thomson-CSF*
*Domaine de Corbeville*
*91404 Orsay*

** et également au LIENS CNRS*
*Ecole normale supérieure*
*45 rue d'Ulm*
*75005 Paris*

RÉSUMÉ. *Nous présentons dans cet article une sémantique opérationnelle des systèmes de production de type OPS5 [For 81] dans le formalisme des règles d'inférence de Plotkin [Plo 81]. Nous montrons comment la compilation de ces langages de programmation peut être décrite par raffinements successifs de la définition de leur sémantique opérationnelle. Nous montrons que l'algorithme de Rete [For 82] peut ainsi être reformulé et son implémentation, tant séquentielle que parallèle, prouvée. Nous voyons également que de simples modifications du système formel présenté permettent de dériver l'algorithme Treat [Mir 90] ainsi que différentes optimisations proposées pour Rete.*

ABSTRACT. *We present an operational semantics, in the formalism of inferences rules a la Plotkin, for production systems like OPS5 [For 81]. We show how these programming languages can be compiled from their operational semantics. We describe how sequential and parallel versions of the Rete algorithm [For 82] are formulated and their implementations proved. We show that simple modifications of the formal system described we obtain the Treat algorithm [Mir 90] and other optimisations of Rete.*

MOTS-CLÉS : *systèmes de production, compilation, sémantique opérationnelle, langage à base de règles.*

KEY WORDS : *production systems, compilation, operational semantics, rule based languages.*

## 1 Introduction

Le développement des systèmes experts a contribué à l'émergence du concept purement logiciel de *programmation déclarative*, que l'on peut aussi résumer par la formule de Robert Kowalski:

Algorithme = Logique + Contrôle

La programmation par règles est le meilleur exemple de ce style de programmation. Contrairement à un programme écrit dans un langage procédural ou fonctionnel dans lequel l'enchaînement des instructions est explicite, une base de règles reflète uniquement la partie logique des traitements, le séquencement des instructions est défini séparément et repose sur des techniques générales de recherche d'une solution. Les outils générateurs de systèmes experts sont des langages (ou des surlangages) de programmation déclaratifs. L'utilisation de ces outils permet de développer efficacement les applications et de bénéficier de techniques de compilation puissantes. Le fait que ces techniques assurent d'excellentes performances sur les *processeurs standards* est un point décisif pour leur intégration dans les systèmes informatiques conventionnels. L'échec relatif des projets de machines dédiées ou de machines massivement parallèles pour les systèmes experts montre bien que la problématique est de nature purement logicielle.

De fait, les langages à base de règles permettent aujourd'hui d'atteindre des performances (en termes de temps d'exécution) permettant l'utilisation de ces langages dans les applications en "temps réel" (voir [FL91]). Ces résultats sont dus autant à l'évolution technologique du matériel qu'à l'amélioration des techniques de compilation. Le rapport de performances entre la version originale de OPS5 (développé à *Carnegie Mellon University*, OPS5 est l'ancêtre de nombre de langages à base de règles existant à ce jour [For81]) et la dernière version réalisée [GFK+88] est de deux ordres de grandeur (à machine égale) [ML91].

Pour développer et maîtriser ces techniques de compilation de plus en plus complexes, il est nécessaire d'avoir une bonne compréhension de ces langages. C'est ce que permet d'atteindre une étude sémantique du langage, qu'il s'agisse de la *sémantique opérationnelle* qui formalise l'exécution d'un programme, ou de la *sémantique logique* qui caractérise de façon abstraite le résultat de l'exécution.

Notre effort tend à donner une analyse sémantique complète des *systèmes de production* (de la famille OPS5), et plus particulièrement du langage XRete[1] [FL91, Tho91a, Tho91b, Fag87] utilisé pour l'écriture d'applications temps réel embarquées de fusion de données ou d'interprétation de données [SF88]. XRete combine l'emploi des règles de production et des règles de déduction dont la sémantique logique est décrite par ailleurs [FL91, Fag90b, Fag90a].

La formalisation de la sémantique opérationnelle permet de décrire mathématiquement l'exécution des programmes, de définir et valider les algorithmes de compilation ou d'étudier par exemple leur parallélisation. Le but de cet article est de donner une sémantique opérationnelle complète des systèmes de production et d'en dériver l'implémentation effective via l'algorithme de *Rete* utilisé. Une telle approche fournit une preuve du compilateur qui effectue en pratique ces transformations. Nous montrons également que l'algorithme *Treat* proposé par D. Miranker comme une alternative à *Rete* [Mir90] peut être défini en utilisant la même méthode par une simple modification du système formel présenté. Cette méthode est suffisamment générale pour être utilisée aussi bien pour les implémentations parallèles que séquentielles. D'autre part, nous

[1] XRete est une marque déposée de Thomson CSF. La conception de ce logiciel a débuté au LCR en 1986



verrons que les optimisations de *Rete* généralement proposées (*hash-code*, permutation des tests mono et multi-schémas [GD84]) se définissent aisément dans ce formalisme.

Le plan est le suivant: dans une première partie nous présentons succinctement les langages à base de règles de production, et donnons leur sémantique opérationnelle. Nous dérivons ensuite par raffinements successifs à partir de cette sémantique opérationnelle un algorithme (*Rete* [For82]) qui la met en œuvre; nous donnons la preuve de correction de cet algorithme, dans ses versions séquentielles et parallèles. Nous terminons par une description des différentes implémentations existantes et possibles de ces langages.

# 2 Sémantique opérationnelle des systèmes de production

## 2.1 Les systèmes de production

Un *système de production*, noté $< \mathcal{W}, RB >$, est constitué d'une base de règles $RB$, appelée également *programme*, et d'une base de faits/objets $\mathcal{W}$, aussi appelée *mémoire de travail*. L'ensemble des faits/objets sera identifié dans cet article à la base de Herbrand $B_H$ (c'est-à-dire l'ensemble des propositions atomiques closes) formée sur un alphabet de symboles de constantes, de fonctions et de prédicats du premier ordre.

Une *règle de production* est un triplet

$$\pi \rightarrow \alpha, \rho$$

où $\pi$ exprime une condition sur la base de faits, $\alpha$ représente une suite de faits ajoutés et $\rho$ une suite de faits retirés. Plus précisément $\pi$, $\alpha$ et $\rho$ peuvent être identifiés à des suites de littéraux de la logique du premier ordre, c'est-à-dire des propositions atomiques (appelées *schémas*), ou la négation de propositions atomiques, pouvant contenir des variables. Une règle de production a donc la forme suivante:

$$A_1, ..., A_k, \neg B_1, ..., \neg B_l \rightarrow C_1, ..., C_m, \neg D_1, ..., \neg D_n$$

avec généralement la restriction supplémentaire que les variables des schémas $B_j$, $C_j$,$D_j$ apparaissent dans les $A_i$.

La *négation par défaut* consiste à prendre $\neg A$ vrai si aucune instance de $A$ n'est dans la base de faits. Une *instance de règle* est un k-uplet d'éléments de la mémoire de travail $(o_1, ..., o_k)$ satisfaisant le membre gauche de la règle, c'est-à-dire tel qu'il existe une substitution $\sigma$ des variables de la règle vérifiant $A_i\sigma = o_i$, pour $1 \leq i \leq k$, et $\neg B_j\sigma$, pour $1 \leq j \leq l$. La négation en partie droite de la règle représente l'action de suppression d'un fait $(\neg D_i\sigma)$ de la mémoire de travail.

Les règles de production permettent d'exprimer des transformations très générales de la base de faits tout en conservant un caractère déclaratif attrayant. Par exemple l'algorithme de tri le plus simple que l'on puisse imaginer: permuter itérativement les couples d'éléments mal ordonnés, s'écrit directement avec une seule règle de production:

```
rule tri
?x : (elem indice=?i val=?v)
?y : (elem indice=?j&>?i val=?w&<?v)
->
modify (?x indice=?j)
modify (?y indice=?i)
endrule
```

où les variables sont préfixées par ? (syntaxe XRete), le filtrage dans les schémas se fait par le nom des attributs, et *modify* est une action équivalente à une suppression suivie d'un ajout du fait modifié. Quand plus aucune règle n'est déclenchable, *l'état stable* obtenu (le point fixe du programme) correspond à une configuration triée. Les différents algorithmes de tri par échanges s'obtiennent en spécifiant l'ordre de déclenchement des règles par un mécanisme de priorités par exemple. Il est remarquable que l'on obtienne de cette manière des algorithmes de tri de complexité en moyenne optimale en $O(n \times \log(n))$ où $n$ est le nombre d'éléments à trier, en appliquant les techniques de compilation décrites dans cet article (cf. section 4.2.1).

Les diverses stratégies de contrôle de l'ordre de déclenchement des règles de production (stratégies *lex*, *mea* [For81], priorités statiques, dynamiques, metarègles [Tho91a], etc...) ne sont pas traitées dans cet article. Nous considérons les systèmes de production comme étant des systèmes non-déterministes.

## 2.2 Cycle d'inférence

Le contenu de la mémoire de travail représente l'*état* du système à un moment donné du calcul, comme la valeur des variables représente l'état de l'exécution d'un programme dans un langage classique. Les règles sont analogues au "programme" qui définit les modifications apportées à la mémoire de travail. La définition formelle du cycle d'inférence nécessite de définir formellement la phase de filtrage, c'est-à-dire la recherche des instances de règle.

Une condition de règle $\pi$ est constituée d'une séquence de littéraux $L$. On a donc $\pi = \pi'.L$ ou $\pi = vrai$. On définit la relation de satisfaction d'une condition $\pi$ sur $\mathcal{W}$ par une instance partielle $(o_1, ..., o_k)$, notée $(\mathcal{W}, o_1, ..., o_k) \models \pi$, par le système suivant:

$$\mathcal{W} \models vrai \tag{1}$$

$$\frac{(\mathcal{W}, o_1, ..., o_{k-1}) \models \pi \quad o_k \in \mathcal{W} \quad p(o_1, ..., o_k)}{(\mathcal{W}, o_1, ..., o_k) \models \pi.P} \tag{2}$$

$$\frac{(\mathcal{W}, o_1, ..., o_k) \models \pi \quad \neg\exists o \in \mathcal{W} \quad p(o_1, ..., o_k, o)}{(\mathcal{W}, o_1, ..., o_k) \models \pi.\neg P} \tag{3}$$

où $p(o_1, ..., o_j)$ dénote la conjonction de tests sur $(o_1, ..., o_j)$ apparaissant dans le schéma $P$.



Une *instance de règle* de $< \mathcal{W}, RB >$ est un couple $(R, (o_1, ..., o_{ar(R)}))$ avec $R \in RB$, $(o_1, ..., o_n) \in \mathcal{W}^n$ tel que $(\mathcal{W}, o_1, ..., o_n) \models \pi(R)$. Chaque instance de règle définit une transition sur la mémoire de travail.

$$\mathcal{W} \rightarrow_{(R,(o_1,...,o_n))} (\mathcal{W} \setminus \rho(R, o_1, ..., o_n)) \cup \alpha(R, o_1, ..., o_n)$$

La transition non-déterministe associée à la base de règles $RB$, dénotée par $\rightarrow$, constitue le *cycle d'inférence*. Il est défini par:

$$\frac{R \in RB \;\; (\mathcal{W}, o_1, ..., o_n) \models \pi(R) \;\; \mathcal{W} \rightarrow_{(R,(o_1,...,o_n))} \mathcal{W}'}{RB \vdash \mathcal{W} \rightarrow \mathcal{W}'} \tag{4}$$

La figure 1 représente schématiquement le cycle d'inférence.

## 2.3 Sémantique Opérationnelle

La sémantique opérationnelle d'un programme $< \mathcal{W}, RB >$ est un *ensemble d'états stables* défini comme l'ensemble des états terminaux pour la relation $\rightarrow$, c'est-à-dire:

$$\begin{aligned} \| < \mathcal{W}, RB > \| \;=\; & \{\mathcal{W}_s \in \mathcal{P}(B_H) \mid RB \vdash \mathcal{W} \rightarrow^* \mathcal{W}_s \\ & et\ RB \vdash \mathcal{W}_s \rightarrow \mathcal{W}' \Rightarrow \mathcal{W}_s = \mathcal{W}'\} \end{aligned}$$

où la relation $\rightarrow^*$ est la clôture réflexive transitive de la relation $\rightarrow$:

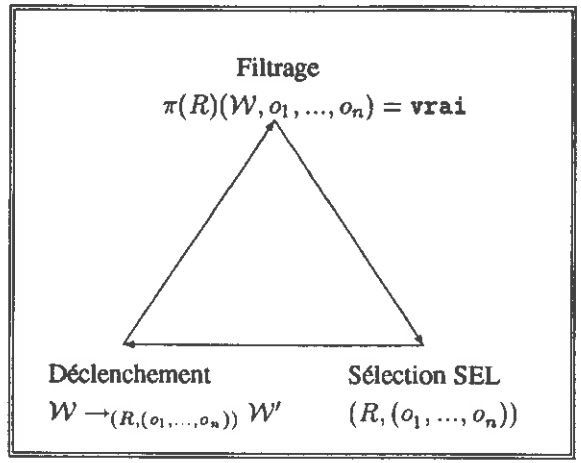



Figure 1: Le cycle d'inférence

$$RB \vdash \mathcal{W} \rightarrow^* \mathcal{W}$$

$$\frac{RB \vdash \mathcal{W} \rightarrow^* \mathcal{W}' \quad RB \vdash \mathcal{W}' \rightarrow \mathcal{W}''}{RB \vdash \mathcal{W} \rightarrow^* \mathcal{W}''}$$

# 3 Compilation

## 3.1 Algorithme *Rete*: construction du graphe *Rete*

L'algorithme *Rete* [For82] est un algorithme efficace qui calcule l'ensemble des instances de règles incrémentalement. Il a été développé pour OPS5. Il repose sur deux principes:

1. La mémorisation: on fait l'hypothèse que le nombre de modifications apportées à la mémoire de travail à chaque cycle d'inférence est faible par rapport à la taille de la mémoire de travail. Cette hypothèse est vérifiée en pratique dans de nombreuses bases de règles [GF83]. Par conséquent, l'ensemble des instances de règles au cycle $n$ est relativement peu différent de son état au cycle $n-1$. L'idée de *Rete* est donc de ne calculer que les modifications de l'ensemble des instances de règle d'un cycle à l'autre. Pour cela, les calculs intermédiaires (les instances partielles de règle) sont *mémorisés*.

2. Le partage: deux règles peuvent avoir une partie de leurs conditions commune. Le calcul relatif à cette sous-condition est factorisé entre les deux règles.

*Rete* a été largement utilisé pour l'implémentation efficace de langages à base de règles.

Nous dérivons des règles 2 et 3 une présentation formelle de cet algorithme.

Tout d'abord, le prédicat $p(o_1, ..., o_j, o)$ associé au schéma $P$ dans les règles 2 et 3 peut être décomposé en deux parties

$$p(o_1, ..., o_j, o) \equiv q(o) \wedge r(o_1, ..., o_j, o)$$

où $q(o)$ dénote la conjonction des *tests mono-schéma* sur $o$ apparaissant dans le schéma $P$, et $r(o_1, ..., o_j, o)$ dénote la conjonction de *tests multi-schéma* entre $(o_1, ..., o_j)$ et $o$ apparaissant dans le schéma $P_i$. La décomposition du test $p$ en $q$ et $r$ n'apporte rien sur le plan logique, mais elle permet de réduire la combinatoire de la recherche des n-uplets $(o_1, ..., o_j, o)$ qui satisfont $p$ car on restreint la recherche de $o$ aux objets vérifiant $q$.

Cela nous permet de transformer les règles 2 et 3 en:

$$\frac{(\mathcal{W}, o_1, ..., o_{k-1}) \models \pi \quad r(o_1, ..., o_k) \quad q(o_k)}{(\mathcal{W}, o_1, ..., o_k) \models \pi.P} \tag{5}$$



$$\frac{(\mathcal{W}, o_1, ..., o_k) \models \pi \quad \neg\exists o \in \mathcal{W} \quad q(o) \wedge r(o_1, ..., o_k, o)}{(\mathcal{W}, o_1, ..., o_k) \models \pi.\neg P} \tag{6}$$

Le caractère *incrémental* de l'algorithme *Rete* provient de la mémorisation des n-uplets (respectivement des objets) qui vérifient les tests $\pi$ (respectivement $q$) dans des *piles gauches* $lm$ (resp. *piles droites* $rm$).

Ainsi, pour une condition

$$\pi = P_1, ..., P_n$$

on définit les mémoires gauches $lm$[2] et droites $rm$[3] suivantes:

**Définition 1**

$$lm_i(E) = \{(o_1, ..., o_{k_i}) \in E^{k_i} | ((o_1, ..., o_{k_i}), E) \models \pi_i\} = E/\pi_i$$

$$rm_i(E) = \{o \in E | q_{i+1}(o)\} = E/q_{i+1}$$

Les règles 1, 5 et 6 deviennent naturellement:

$$\frac{(o_1, ..., o_{k-1}) \in lm_{i-1}(\mathcal{W}) \quad o_k \in rm_{i-1}(\mathcal{W}) \quad r_{i-1}(o_1, ..., o_k)}{(o_1, ..., o_k) \in lm_i(\mathcal{W})} \tag{7}$$

pour un schéma positif et $i \neq 1$,

$$\frac{(o_1, ..., o_k) \in lm_{i-1}(\mathcal{W}) \quad \neg\exists o \in rm_{i-1}(\mathcal{W})\ r_{i-1}(o_1, ..., o_k, o)}{(o_1, ..., o_k) \in lm_i(\mathcal{W})} \tag{8}$$

pour un schéma négatif et $i \neq 1$,

et

$$lm_1(\mathcal{W}) = rm_0(\mathcal{W}) \tag{9}$$

On définit maintenant les ensemble $rulem$ qui contiennent les instances de règle déclenchables.

$$rulem(R, \mathcal{W})) = \{(R, (o_1, ..., o_n)) \mid \ (o_1, ..., o_n) \in lm_n(\mathcal{W})\}$$

Nous pouvons maintenant écrire la règle 4 en:

$$\frac{(o_1, ..., o_k) \in rulem(R) \quad \mathcal{W} \rightarrow_{(R,(o_1,...,o_k))} \mathcal{W}' \quad R \in RB}{\mathcal{W} \rightarrow \mathcal{W}'} \tag{10}$$

L'algorithme *Rete* utilise une structure de graphe.

A chaque pile gauche $lm$ et $rm$ est associé un nœud du graphe identifié par la pile à laquelle il se rattache.

D'autre part, on associe à chacune des règles 7 et 8 un nœud appelé *jointure*. Les jointures correspondant à la règle 7 sont dites *positives*; celles correspondant à la règle 8

[2] mémoires $\beta$ dans la terminologie Ops5

sont dites négatives. A chacun de ces nœuds sont associés trois piles: la *pile gauche* $lm_{i-1}(\mathcal{W})$, la *pile droite* $rm_{i-1}(\mathcal{W})$ et la *pile résultante* $lm_i(\mathcal{W})$.

En reliant par un arc les nœuds jointure aux nœuds des piles associées, on obtient un graphe. En orientant les arcs des jointures vers les piles résultantes et des piles gauches et droites vers les jointures, on obtient un graphe orienté (figure 2).

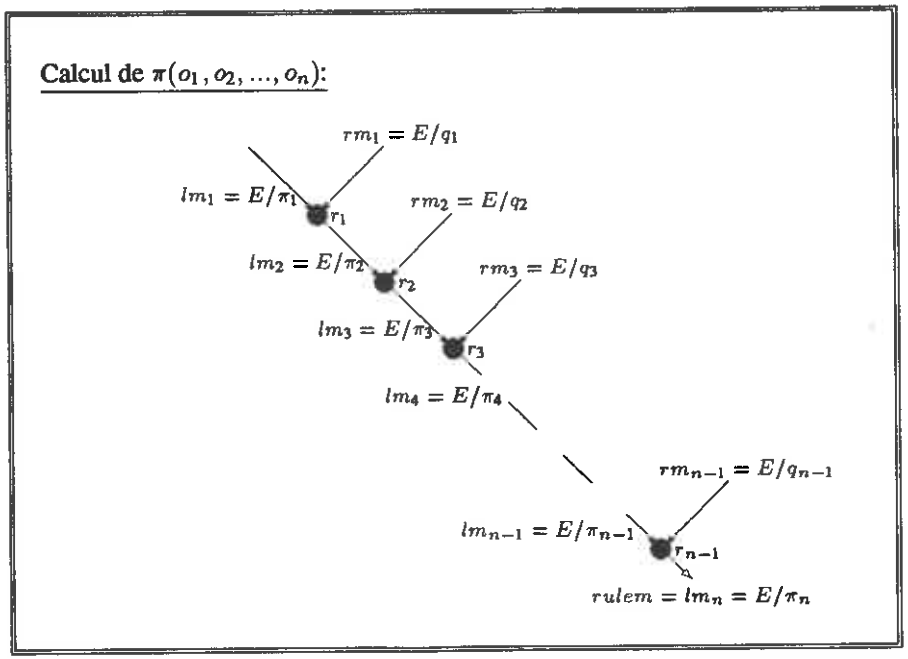



Figure 2: Graphe (orienté) de *Rete*

## 3.2 Algorithme *Rete*: exécution du graphe

Formellement, *Rete* associe un *agent* à chaque nœud de son graphe. Schématiquement, cet agent correspondant à une jointure est chargé de maintenir le contenu de la *pile résultante* $lm_i$ en fonction des évolutions des piles gauches $lm_{i-1}$ et $rm_{i-1}$.

Ces agents communiquent par l'envoi de messages, appelé jetons, de la forme $<[+-], e>$. L'*élément* $e$ est soit un objet soit un n-uplet. Le signe + ou − signifie qu'on cherche ajouter ou supprimer l'élément. Il correspond aux deux actions de base définies dans les parties droites de règles.

Les agents communiquent entre eux par des communications synchrones. Les arcs du graphe sont les canaux de communications de ces messages. Les communications se font dans le sens de l'orientation des arcs.

Nous supposons qu'il existe un agent spécial *Root* chargé de recevoir et de distribuer les messages reçus de l'extérieur. Un retrait d'un objet $o$ de la mémoire de travail envoie



Nous donnons la sémantique opérationnelle de ces agents.
Pour chaque agent, la règle est de la forme

$$m? \xrightarrow{a} ms!$$

où $m?$ est le message (jeton) reçu, $ms!$ la liste des jetons envoyés en réponse et $a$ l'éventuelle action exécutée.

### 3.2.1 Piles droites $rm$

De la définition 1 des piles droites nous dérivons les deux règles suivantes:

$$<-, o>? \xrightarrow{rm \leftarrow rm - \{o\}} \{<-, o>\}! \tag{11}$$

$$<+, o>? \xrightarrow{rm \leftarrow rm \cup \{o\}} \{<+, o>\}! \quad si \quad q(o) \tag{12}$$

$$<+, o>? \rightarrow \emptyset! \quad si \quad \neg q(o) \tag{13}$$

### 3.2.2 Piles gauches $lm$

Les règles associées aux piles gauches sont extrêmement simples:

$$<-, o>? \xrightarrow{lm \leftarrow lm - \{o\}} \{<-, o>\}!$$

$$<+, o>? \xrightarrow{lm \leftarrow lm \cup \{o\}} \{<+, o>\}!$$

### 3.2.3 Jointures positives

Nous appelons *jointure positive* un agent correspondant à la règle 7. L'action des différents messages peut être dérivée facilement de cette règle.

$$<-, o>? \rightarrow \{<-, o>\}!$$

Une jointure positive peut recevoir des jetons de deux agents, correspondant aux piles gauches et droites du nœud *Rete*. L'action du jeton $<+, o>$ dépend de l'émetteur.

<u>Gauche:</u>

$$<+, o>? \rightarrow \{<+, (o, l)> \quad \forall l \in rm \mid r(l, o)\}!$$

<u>Droite:</u>

$$<+, o>? \rightarrow \{<+, (l, o)> \quad \forall l \in lm \mid r(l, o)\}!$$

### 3.2.4 Jointures négatives

Nous appelons *jointure négative* un agent correspondant à la règle 8. Pour ces nœuds l'action des jetons $<-, o>$ dépend également de l'émetteur. Nous effectuons la même

Gauche:

$$<-,o>? \rightarrow \{<-,o>\}!$$

Droite:

$$<-,o>? \rightarrow \{<+,l> \ \forall l \in lm \mid \forall o' \in rm \ \neg r(l,o')\}!$$

Gauche:

$$<+,o>? \rightarrow \{<+,o>\}! \ si \ \forall l \in rm \ \neg r(l,o)$$

Droite:

$$<+,o>? \rightarrow \{<-,l> \ \forall l \in lm \mid r(l,o)\}!$$

### 3.2.5 Piles de règle

La sémantique opérationnelle des nœuds $rulem$ est:

$$rulem : <+,o>? \overset{lm \leftarrow lm \cup \{o\}}{\rightarrow} \{<+,o>\}!$$

$$rulem : <-,o>? \overset{lm \leftarrow lm - \{o\}}{\rightarrow} \{<-,o>\}!$$

## 3.3 Exécution séquentielle

L'exécution du graphe *Rete* décrite dans la section précédente est naturellement parallèle. C'est en partant de cette remarque que de nombreuses recherches ont été lancées afin d'implémenter des versions parallèles de *Rete* [Gup86, Lis89, GFK+88]. Le modèle précédent correspond au *node parallelism* défini par A. Gupta [Gup86]. Une description plus détaillée sera faite dans la section 4.3.

Toutefois, on peut facilement dériver du formalisme précédent un modèle d'exécution séquentielle. Supposons en effet que les messages soient transmis via un medium de communication commun que nous appelons l'*échéancier*. Les communications étant synchrones, cet échéancier est une pile LIFO. D'autre part, les messages intègrent maintenant le nom du nœud destinataire. Il sont donc de la forme $<[+,-],e,ag>$.

L'algorithme peut être décrit par des règles de réécritures sur l'échéancier $S$.

Notations:

- les nœuds de jointure positive sont appelés $Any(l,r)$ où $l$ est la pile gauche et $r$ la pile droite.
- les nœuds de jointure négative sont appelés $Not(l,r)$ avec les mêmes conventions.
- les nœuds associés aux pile $lm$ et $rm$ portent le même nom que cette pile.
- On note $n \succ n'$ pour exprimer que $n'$ est connecté à $n$.
- on note $S = t.S'$ pour exprimer que $t$ est le premier jeton de l'échéancier $S$ et



- Si $(t_i)_{i \in I}$ est un ensemble fini de jetons, $t_{i\{i \in I\}}.S$ est l'échéancier obtenu par empilage des jetons $t_i$ (indépendamment de l'ordre) dans $S$.

Ainsi, la règle 12 devient

$$<-,o,rm> .S \overset{rm \leftarrow rm - \{o\}}{\rightarrow} <-,o,n>_{n \succ rm} .S$$

De même, 11 devient: Si $q(o)$:

$$<-,o,rm> .S \overset{rm \leftarrow rm \cup \{o\}}{\rightarrow} <-,o,n>_{n \succ rm} .S$$

Si $\neg q(o)$:

$$<+,o,rm> .S \overset{rm \leftarrow rm \cup \{o\}}{\rightarrow} <+,o,n>_{n \succ rm} .S$$

Les règles obtenues sont rassemblées dans le tableau suivant:

| | |
|---|---|
| | $<-,o,rm> .S \overset{rm \leftarrow rm - \{o\}}{\rightarrow} <-,o,n>_{n \succ rm} .S$ si $q(o)$<br>$<-,o,rm> .S \rightarrow S$ si $\neg q(o)$<br>$<+,o,rm> .S \overset{rm \leftarrow rm \cup \{o\}}{\rightarrow} <+,o,n>_{n \succ rm} .S$ |
| | $<-,o,lm> .S \overset{lm \leftarrow lm - \{o\}}{\rightarrow} <-,o,n>_{n \succ lm} .S$<br>$<+,o,lm> .S \overset{lm \leftarrow lm \cup \{o\}}{\rightarrow} <+,o,n>_{n \succ lm} .S$ |
| *Left* | $<-,o,Any(rm,lm)> .S \rightarrow <-,o,n>_{n \succ Any} .S$<br>$<+,o,Any(rm,lm)> .S \rightarrow <+,(o,l),n>_{\forall l \in rm \mid r(o,l) \ \forall n \succ Any} .S$ |
| *Right* | $<-,o,Any(rm,lm)> .S \rightarrow <-,o,n>_{n \succ Any} .S$<br>$<+,o,Any(rm,lm)> .S \rightarrow <+,(o,l),n>_{\forall l \in rm \mid r(l,o) \ \forall n \succ Any} .S$ |
| *Left* | $<-,o,Not(lm,rm)> .S \rightarrow <-,o,n>_{n \succ Not} .S$<br>$<+,o,Not(rm,lm)> .S \rightarrow <+,o,n>_{n \succ Not \ si \ \forall l \in rm \ \neg r(o,l)} .S$ |
| *Left* | $<-,o,Not(rm,lm)> .S \rightarrow <+,l,n>_{n \succ Not, \forall l \in lm \mid \forall o \in rm \ \neg r(l,o)} .S$<br>$<+,o,Not(rm,lm)> .S \rightarrow <-,l,n>_{n \succ Not \ \forall l \in lm \mid r(l,o)} .S$ |
| | $<+,o,Rulem(lm)> .S \overset{lm \leftarrow lm \cup \{o\}}{\rightarrow} S$ |

## 3.4 Arbre de discrimination

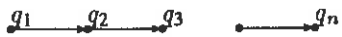
Nous n'avons pas décrit dans ce qui précède comment les tests mono-schéma (voir section 3.1) sont implémentés avec *Rete*. Nous ne nous sommes intéressé qu'à la partie "jointure" du graphe.

Les tests mono-schémas ne portent que sur un seul objet et sont de la forme $q(o)$. Ils sont constitués de la conjonction de tests élémentaires qui nous permettent d'écrire:

$$q(o) \equiv q_1(o) \wedge \cdots \wedge q_n(o)$$

A chacun de ses tests élémentaires $q_i$ est associé un nœud/agent (appelé $Anode(q_i)$) chargé d'effectuer le test. Pour tout $i$ on crée un arc $Anode(q_{i-1}) \rightarrow Anode(q_i)$. On relie le nœud *Root* à $Anode(q_1)$ et $Anode(q_n)$ à la pile $rm$ associée à $q$.

On obtient un (sous-) graphe:




La sémantique opérationnelle générale des $Anode$ est définie par:

$$<-,o>? \rightarrow \{<-,o>\}! \quad si \ q(o)$$
$$<-,o> \rightarrow \emptyset! \quad si \ \neg q(o)$$
$$<+,o>? \rightarrow \{<+,o>\}! \quad si \ q(o)$$
$$<+,o> \rightarrow \emptyset! \quad si \ \neg q(o)$$

et la sémantique opérationnelle séquentielle:

$$<-,o,Anode(q)> .S \rightarrow <-,o,n>_{n \succ Anode} .S \quad si \ q(o)$$
$$<-,o,Anode(q)> .S \rightarrow S \quad si \ \neg q(o)$$
$$<+,o,Anode(q)> .S \rightarrow <+,o,n>_{n \succ Anode} .S \quad si \ q(o)$$
$$<+,o,Anode(q)> .S \rightarrow S \quad si \ \neg q(o)$$

## 3.5 Partage

La seconde idée de *Rete* est le partage. Celui-ci peut se produire soit dans la partie jointure du graphe, soit dans la partie discrimination.

### 3.5.1 Jointures

Dans la partie jointure, le partage est possible quand deux règles ont une même sous-condition en commun. Cela correspond aux cas où deux instances de la règle opérationnelle 7 ou 8 sont identiques. Les nœuds jointure correspondant sont alors partagés entre les deux règles. On obtient ainsi un graphe comme décrit figure 3.



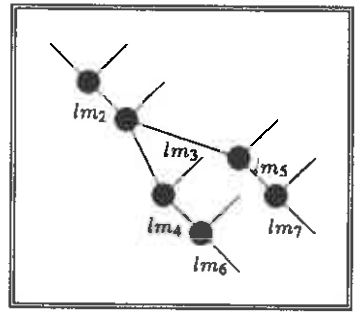
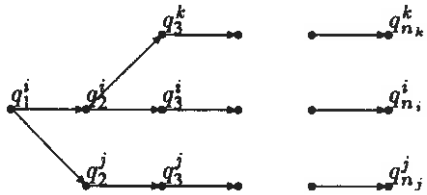



Figure 3: Réseau de jointure avec partage

### 3.5.2 Discrimination

Si deux tests mono-schéma partagent les mêmes sous-tests, ceux-ci peuvent être mis en commun, ce qui revient à partager certains nœuds du graphe de discrimination. La figure suivante illustre ce cas pour trois prédicats $q^i$, $q^j$ et $q^k$ tels $q_1^i = q_1^j = q_1^k$ et $q_2^k = q_2^j$:




### 3.5.3 Généralisation et Exemple

D'autres types de nœuds sont utilisés dans la compilation. Nous les décrivons brièvement:

- Les *nœuds Call* apparaissent dans la partie jointure du graphe et sont aux n-uplets d'objets ce que les *Anode* sont aux objets. Il leur est donc associé un test $\pi(o_1, ..., o_n)$ qui s'exécute après la jointure.
- Les *Tnode* sont des *Anode* particuliers. Les objets manipulés dans les systèmes de production sont souvent typés. Les *Tnode* effectuent le test de type.

La figure 4 donne un exemple d'un graphe de *Rete* complet.

## 3.6 Algorithme *Treat*

L'algorithme *Treat* a été proposé par D. Miranker comme une alternative à *Rete* [Mir90]

- Dans l'algorithme *Rete* classique, les tests de jointure sont (ré-)effectués lors de la phase de suppression d'un objet.
- Les piles gauches $lm$ (mémoires $\beta$) peuvent croître de manière explosive.
- *Rete* est mal adapté à un traitement parallèle:
  - On peut montrer que l'ordre de traitement des jetons ne peut être quelconque [Lis89], et est donc difficilement parallélisable.
  - Dans un environnement parallèle, le contenu des mémoires gauches doit être communiqué.

Notre propos n'est pas ici de faire une critique de l'algorithme de *Treat*, mais remarquons que la première remarque est fausse puisqu'il est possible d'améliorer *Rete* de telle sorte que la phase de retrait n'effectue pas de tests de jointure [Fag87, Tho91a]. Le seul coût de cette phase réside dans le parcours du graphe et la mise à jour des piles. Dans cette version, les retraits d'objet consomment moins de 20% du temps consacré aux ajouts d'objet. D'autre part, *Rete* a été parallélisé avec succès [GFK+88].

L'idée principale de *Treat* est d'économiser le coût de gestion des piles gauches (en les supprimant). Cela revient à utiliser les règles 1, 5 et 6, en les dérivant cette fois en:

$$\mathcal{W} \models true \tag{14}$$

$$\frac{(o_1, ..., o_{k-1}, \mathcal{W}) \models \pi \quad r(o_1, ..., o_k) \quad o_k \in rm_{i-1}(\mathcal{W})}{(o_1, ..., o_k, \mathcal{W}) \models \pi.P} \tag{15}$$

$$\frac{(o_1, ..., o_k, \mathcal{W}) \models \pi \quad \neg\exists o \in \mathcal{W} \quad o \in rm_{i-1}(\mathcal{W}) \wedge r(o_1, ..., o_k, o)}{(o_1, ..., o_k, \mathcal{W}) \models \pi.\neg P} \tag{16}$$

Comme en *Rete*, un agent est associé à chacune de ces règles, mais il n'agit qu'en fonction de l'évolution de la mémoire droite.

- Si un objet de $rm$ est retiré de la mémoire de travail, l'agent le retire de $rm$.
- Si un objet $o$ est ajouté à la mémoire de travail et qu'il vérifie $q$, le contenu de $rm$ est mis à jour. On cherche en suite les n-uplets $(o_1, ..., o_n)$ qui vérifient $(o_1, ..., o_{k-1}, \mathcal{W}) \models \pi$, et pour chacun vérifiant $(o_1, ..., o_{k-1}, o, \mathcal{W}) \models \pi.P$, le n-uplet $(o_1, ..., o_{k-1}, o)$ est envoyé vers les nœuds inférieurs.
- Si une instance arrive à gauche, le comportement est identique à celui décrit pour *Rete*.

Une règle est dite *active* si, quel que soit un schéma positif $P$ apparaissant dans sa condition, il existe un objet de la mémoire de travail qui vérifie $P$.

En *Treat*, les agents précédents ne sont actifs que si la règle à laquelle ils sont attachés est active.



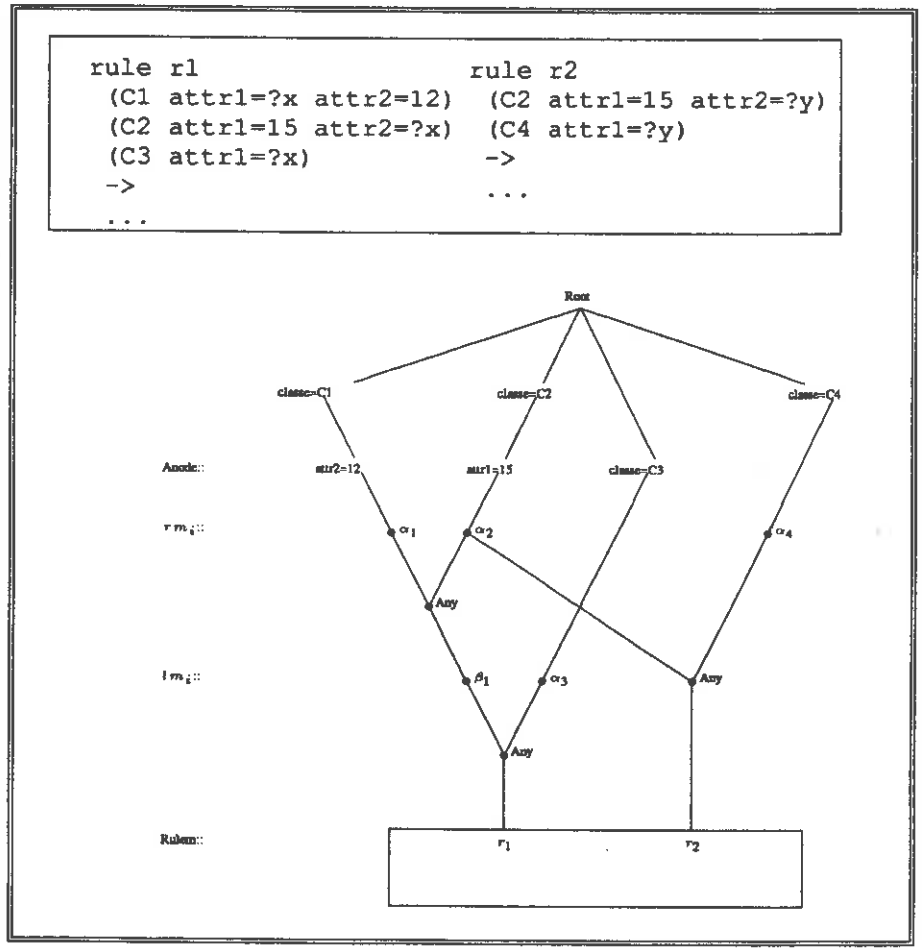



Figure 4: Graphe de *Rete*

# 4 Implémentation

Il n'y a pas de norme de fait quant à l'implémentation de *Rete*. Toutefois, nous allons essayer de dégager les caractéristiques des différentes implémentations connues.

## 4.1 Implémentation séquentielle

Les premières versions de *Rete* [For81], généralement écrites en LISP, utilisaient un graphe et des jetons explicites. Peu à peu, des versions C de *Rete* (ainsi que *Treat*) sont apparues [For84, Ard87, Tho91a, GFK+88, ML91] produisant un code plus efficace

Par le passage de LISP à C et l'utilisation de techniques de compilation plus avancées, les vitesses d'exécution des programmes ont considérablement augmenté. A titre d'exemple, le tableau suivant présente les ratios de performances obtenues avec OPS5 entre deux implémentations: l'originale (basée sur LISP) et une version récente[4] sur HP 9000/370.

| | Implémentation initiale | | Implémentation actuelle | | |
|---|---|---|---|---|---|
| | temps (sec.) | règles/sec. | temps (sec.) | règles/sec. | Accélération |
| Weaver | 1253 | 0,66 | 33,6 | 22,35 | 37,2 |
| Rubik | 2760 | 0,11 | 36,7 | 8,80 | 75,2 |
| Tourney | 1143 | 0,46 | 6,3 | 83 | 181,4 |

Quand la compilation par procéduralisation ou micro-code est également réalisée dans la version Lisp (ainsi que la gestion mémoire), le gain qui demeure en faveur de la version C est dû soit à la qualité du compilateur Lisp (particulièrement sensible sur processeurs RISC), soit aux calculs arithmétiques ou à la représentation des faits/objets, qui peuvent être moins efficaces en Lisp. En dehors de ces cas, pour des bases de règles purement symboliques le gain est négligeable.

Nous donnons à titre d'exemples les temps d'exécution de XRete version C (sans hash code, Cf. section 4.2.1) sur une Sun Sparc II:

| Base de règles | Temps (secondes) | Règles/seconde |
|---|---|---|
| *Tourney* | 4,14 | 124 |
| *Rubik* | 2,75 | 94 |

## 4.2 Optimisations

### 4.2.1 Partitionnement des mémoires locales

Les techniques de *hash-code* ont été introduites dans [For84] et [Inf85]. L'idée est d'organiser les piles droites $rm$ et gauches $lm$ de chaque jointure afin de réduire le coût de recherche des couples $<(o_1, ..., o_n), o>$ dans $lm \times rm$ tels que $\pi(o_1, ..., o_n, o)$ soit vérifié. Pour cela, $lm$ et $rm$ sont hash-codés en fonction de $p$.

Dans les réalisations pratiques, les piles gauches et droites sont regroupées dans des tables de hash uniques. L'accès d'un objet $o$ de la pile $rm$ se fait en calculant la clé fonction de l'objet et de la pile.

Cette technique est intéressante dès que le nombre d'objets dans la mémoire de travail est important, et que le nombre de tests potentiels à effectuer augmente. Elle est toutefois limitée aux tests d'égalité dans les jointures. Enfin elle présente deux inconvénients majeurs: elle présente un surcoût qui n'est pas négligeable en cas de faible combinatoire, et elle interdit le partage d'une pile entre plusieurs jointures (puisque la structure de la pile dépend de la jointure à laquelle elle est attachée), diminuant donc le taux de partage du graphe et donc les performances. A. Gupta a évalué sur quelques bases de règles l'intérêt du hash [GF83]: le gain en performances varie de 5 à 0,7 (perte).



Notons enfin que dans le même esprit les tests d'inégalité sur les attributs numériques des objets peuvent être utilisés pour partitionner les mémoires locales en *arbres de recherche* et ainsi ramener la complexité de recherche linéaire à une complexité logarithmique en moyenne.

### 4.2.2 Test mono-schéma dans les jointures

Il est possible de faire "glisser" certains tests mono-schémas dans la partie jointure du graphe ce qui permet d'améliorer le taux de partage et donc, *a priori*, l'efficacité [GD84].

Exemple Considérons la compilation des deux conditions ci-dessous:

```
Condition C1 (personne nom=?x)
------------ (personne pere=?x)

Condition C2 (personne nom=?x)
------------ (personne pere=?x age<10)
```

La condition C2 est plus forte que C1: les couples d'objets qui vérifient C2 vérifient également C1. Le graphe normal est celui à gauche de la figure 5. Il n'y a aucun partage entre les deux conditions car les schémas de gauche sont différents, celui de C1 étant plus contraint (par le test "age < 10") que celui de C2. L'idée consiste donc à "retarder" l'exécution du test "age < 10" après la jointure. On obtient pour la condition C2 le graphe de droite de la figure. Il est alors possible de le partager avec le graphe de C1.

Nous avons dit plus haut que les tests de discrimination (mono-schéma) permettent de réduire la combinatoire des jointures. Exécuter ces tests après les jointures ne semble donc pas à première vue une optimisation: la jointure du graphe "optimisé" de C2 construit plus d'instances partielles que celle du graphe originel. Toutefois, ces instances sont construites par la condition C1. Comme ce calcul est maintenant factorisé, cela représente, globalement, une économie.

## 4.3 Implémentation parallèle

La parallélisation a été une des solutions envisagées pour améliorer les performances des systèmes de production.

Dans un premier temps, tous les travaux tendaient à ne paralléliser que la phase de filtrage, représentant plus de 90% du temps de filtrage [GF83]. A. Gupta [Gup86] a défini 4 types principaux de parallélisme:

- **Production Parallelism**: Si on effectue une partition d'une base de règles $\mathcal{R}$ en $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2 \cup \cdots \cup \mathcal{R}_n$, alors, pour toute mémoire de travail $\mathcal{W}$ on a $SR_{\mathcal{R}}(\mathcal{W}) = SR_{\mathcal{R}_1}(\mathcal{W}) \cup SR_{\mathcal{R}_2}(\mathcal{W}) \cup \cdots \cup SR_{\mathcal{R}_n}(\mathcal{W})$ (on appelle $SR_{\mathcal{R}}(E)$ l'ensemble des instances de règles satisfaites pour une mémoire de travail $E$ et

découpage de la base de règles, associe chacun des sous-groupes de règles à un processeur. Chaque processeur calcule à chaque cycle l'ensemble des instances de règles déclenchables.

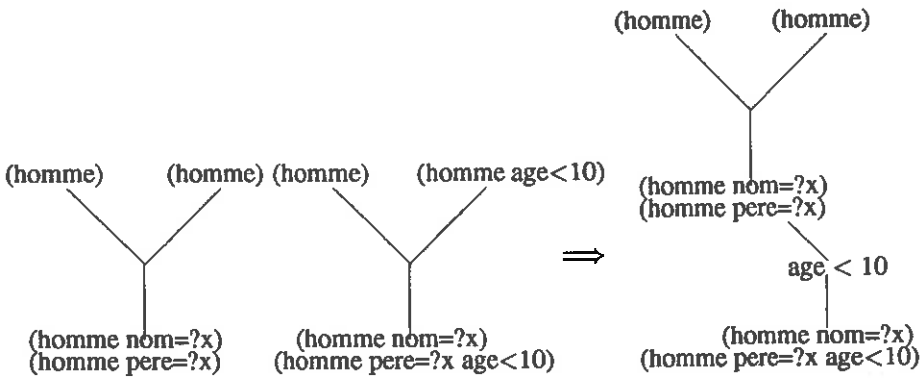



Figure 5: Test mono-schéma dans les jointures

- **Node Parallelism**: Ce parallélisme est d'un grain plus fin que le précédent. Il repose sur la structure de graphe qu'utilise *Rete*. Il consiste à considérer les nœuds du graphe comme un agent communiquant des messages (jetons). Ce parallélisme correspond à la sémantique opérationnelle définie section 3.2.

- **Token Parallelism** (ou **Intra-node Parallelism**): ce parallélisme est le dual du précédent. Les jetons circulant dans le graphe sont distribués entre les processeurs.

- **Action Parallelism**: Le déclenchement d'une règle provoque plusieurs modifications (retraits ou ajouts) de la mémoire de travail (2,44 en moyenne pour OPS5 d'après [Gup86]) qui peuvent être exécutées en parallèle. Ces actions entraînant des modifications du graphe de *Rete*, ce parallélisme n'est possible que couplé à l'une des trois formes précédentes. Gupta [Gup86] montre qu'elle peut apporter une accélération supplémentaire par rapport à l'*Intra-node Parallelism*.

Quelque soit l'approche retenue, le *Production Parallelism* [BAB88, HHW87, Ofl87] ou *Intra-Node Parallelism* [AT89, GFK+88, TAG89], les résultats montrent que les accélérations obtenues ne dépassent pas 15, quelque soit le nombre de processeurs. Il est à noter que ces études ont été réalisées en visant aussi bien des machines à mémoire partagée [Gup86, GFK+88] que distribuée[BAB88, HHW87, AT89, TAG89].

Le parallélisme massif sur des machines en forme d'arbres comme *Dado* [SM84], *Non-Von* [HS86] ou *Mapps* [OD87], de pyramide *Cupid* [KS87, KS89] ou à base de Transputers comme *Pamela* [BMB89] a également été étudié. A notre connaissance les résultats obtenus (quand ces implémentations ont vu effectivement le jour) sont loin des résultats escomptés.



La médiocrité des accélérations obtenues avec la parallélisation de *Rete* par rapport à ce que le parallélisme naturel de cet algorithme (cf. section 3.2) laissait envisager initialement s'explique par trois principales raisons:

- Si *Rete* peut être considéré naturellement parallèle relativement au découpage en nœuds du graphe, il n'en est pas de même relativement aux jetons. On peut montrer en effet que l'ordre de traitement des jetons ne peut être quelconque [Lis89]. Ce problème, qui réduit considérablement l'intérêt du *Token Parallelism*, a été informellement introduit par A. Gupta sous le nom de *paires conjuguées*.

  D'autre part, un jeton circulant dans le graphe est créé par exécution d'un autre jeton. Par conséquent, ces deux jetons sont nécessairement successifs. Dans le cas de règles dont les conditions contenant beaucoup de schémas, cela peut provoquer le phénomène des *longues chaînes* [Gup86].

- La loi d'Amdhal indique que si on ne parallèlise qu'un portion $r$ d'un programme, l'accélération maximale (quelque soit le nombre de processeurs) est de $\frac{1}{1-r}$. Suite à *Forgy & Gupta* [GF83], il a été longtemps affirmé que le filtrage représentait plus de 90% du temps de calcul. Or, les améliorations de la compilation de *Rete* aidant, la part du filtrage dans le calcul a sensiblement diminué (certains ayant même mesuré sur certaines bases de règles un ration de 20%). La barre des 90% est maintenant un maximum, ce qui limite l'accélération à 10.

- Enfin, les systèmes de production sont cycliques. A chaque cycle d'inférence il est nécessaire de synchroniser les processeurs. Cette synchronisation est d'autant plus coûteuse que les implémentations sont efficaces.

Pour dépasser ces limites une nouvelle technique a été développée par *Ishida & Stolfo*: le déclenchement en parallèle des règles [HW89, IYG90, IS85, SG90]. L'objectif est d'extraire le parallélisme "naturel" de l'application (quand des traitements similaires sont effectués sur plusieurs données par exemple) en déclenchant plusieurs instances de règles satisfaites simultanément quand cela est légitime (critère de *sérialisation*).

Malheureusement, le critère de sérialisation proposé généralement n'est pas complet. Une version exacte de ce critère a été proposée par [HW89], mais il conduit à un algorithme complexe dont on peut penser qu'il conduirait à un surcoût important.

A notre connaissance, il n'existe pas de résultat numérique suite à une implémentation effective de cette technique, mais les résultats attendus sont modestes (< 5 selon *Schmoltze & al.* [SG90]).

## 4.4 Arbre de jointures

le graphe de *Rete* est généralement construit sous la forme d'un *peigne* (figure 6). Mais il est possible de le construire un *arbre* de jointures (figure 7).

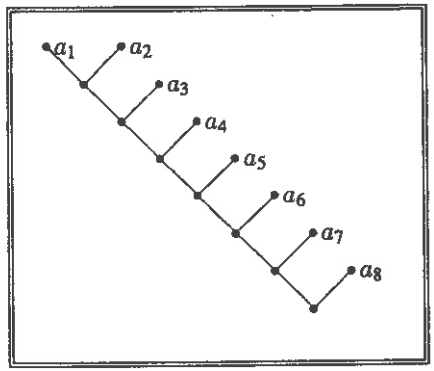



Figure 6: Jointures en peigne

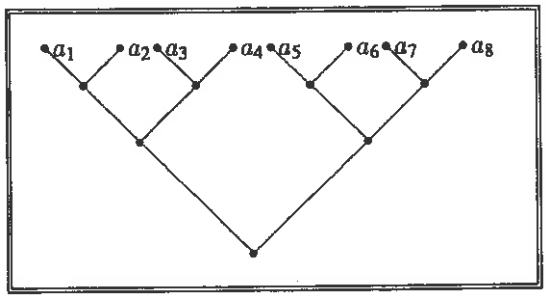



Figure 7: Jointures en arbre

Les conditions de règles $\pi$ sont définies comme:

$$\begin{aligned} \pi ::= & \ \pi_1.\pi_2 \\ | & \ P \end{aligned}$$

où $P$ est un schéma. La structure en arbre est alors obtenue en transformant les règles opérationnelles 1 et 2 (le traitement des schémas négatifs est identique) en:

$$\mathcal{W} \models true \tag{17}$$

$$\frac{(o_1, ..., o_i, \mathcal{W}) \models \pi_i \quad (o'_1, ..., o'_j, \mathcal{W}) \models \pi_j}{(o_1, ..., o_i, o'_1, ..., o'_j, \mathcal{W}) \models \pi_i.\pi_j} \tag{18}$$

Les deux prédicats $\pi_i$ et $\pi_j$ définissent les deux branches supérieures du nœud associé à $\pi_i.\pi_j$ dans l'arbre de *Rete*.

L'avantage de l'organisation en arbre des jointures est qu'elle permet d'accroître le partage des noeuds du graphe *Rete*. L'inconvénient est qu'elle a pour effet d'augmenter



le nombre d'instances partielles calculées éventuellement inutilement (aggravation du défaut principal de *Rete* vis à vis de Treat).

## 4.5 *Rete* paresseux

D. Miranker a défini une version *paresseuse* de l'algorithme *Treat* [MBLG90]. L'idée principale est de limiter le filtrage afin:

1. De ne pas construire puis détruire une instance de règle au cours d'une même phase de filtrage,

2. de ne pas construire des instances de règle qui ne seront jamais déclenchées.

D. Miranker estime à un facteur 2 le gain avec *Treat* en vitesse d'exécution pour certaines bases de règles.

Cette approche est d'autant plus intéressante qu'on ne cherche à calculer les instances de règles que une à une. Ceci est le cas quand la stratégie permet de savoir avant l'opération de filtrage l'ordre de priorité dans le déclenchement des règles. En associant systématiquement *Rete* à OPS5 (pour lequel il a été développé) dont les stratégies ne répondent pas au critère énoncé , certains ont estimé qu'une approche paresseuse de *Rete* n'était pas envisageable. En outre, la lourde structure de *Rete* qui mémorise beaucoup d'information semble moins bien adaptée à l'évaluation paresseuse que celle de *Treat*.

Notre approche du filtrage (de *Rete*) dans cet article est totalement indépendante de l'usage qui en est fait. Avec cette démarche nous avons pu montrer par ailleurs [Lis92] que le formalisme décrit permet de définir une évaluation paresseuse de *Rete* dont nous avons pu en dériver une implémentation pour XRete. Nous avons pour cela restreint temps de calcul est inférieur à un facteur 2 quelle que soit la base de règles testée. Sur des exemples très combinatoires telle que *Tourney*, le gain peut atteindre un facteur 7.

Pour obtenir un modèle paresseux de *Rete*, il suffit de modifier le comportement des *agents* qui exécutent le graphe: leur activation ne vient pas du "haut" du graphe, à l'arrivée d'un nouveau jeton, mais du "bas". Chaque agent (nœud) est sollicité par un nœud fils. En réponse, il essaie de construire une (et une seule) nouvelle instance partielle et la lui transmet. S'il ne peut pas construire une telle instance avec le contenu de ses piles gauches et droites, il sollicite à son tour son nœud père (s'il en a un) afin de recevoir une nouvelle instance partielle qu'il stocke dans sa pile gauche, et ainsi de suite.

Le modèle d'exécution séquentiel décrit peut ainsi être modifié pour reproduire le modèle paresseux.

### 4.5.1 Autres approches du filtrage

D'autres approches considèrent le filtrage dans les langages à base de règles comme un processus de satisfaction de contraintes [Dor89, Via85]. L'alogrithme *Treat* lui-même se démarquent de *Rete* par certaines de ses optimisations qui en sont inspirées. Le modèle d'exécution que nous avons donné ne permet pas bien entendu de rendre

compte de ces approches. Mais on peut montrer que la démarche exposée peut être reprise et adaptée.

Considérons (pour prendre un exemple simple que nous généraliserons par la suite) l'optimisation de *Treat* consistant à ne chercher à construire des instances de règles que pour celles pour lesquelles il existe au moins un objet satisfaisant chaque prémisse. Dans les autres cas, il est clair qu'il ne peut pas exister d'instances satisfaisant la règle. Nous modifions les formules 2 et 3 de façon à tenir compte de cette optimisation. Soit $R = r_1 ... r_n$ une condition de règle dont les $r_i$ sont les prémisses. Les formules deviennent:

$$\frac{(\mathcal{W}, o_1, ..., o_{k-1}) \models \pi \quad o_k \in \mathcal{W} \quad p(o_1, ..., o_k) \ \forall i \in [1..n] \ \exists o \in \mathcal{W} \ r_i(o)}{(\mathcal{W}, o_1, ..., o_k) \models \pi.P} \tag{19}$$

$$\frac{(\mathcal{W}, o_1, ..., o_k) \models \pi \quad \neg\exists o \in \mathcal{W} \quad p(o_1, ..., o_k, o) \ \forall i \in [1..n] \ \exists o \in \mathcal{W} \ r_i(o)}{(\mathcal{W}, o_1, ..., o_k) \models \pi.\neg P} \tag{20}$$

De façon plus générale, le filtrage vu comme un problème de satisfaction de contraintes n'est fondamentalement pas très différent de l'approche présentée ici, même si, du point de vue de l'implémentation, cela suppose des structures de données et des algorithmes radicalement différents. Quel que soit le point de vue choisi, on ne peut éviter la décomposition des conditions de règles en prémisses élémentaires. L'optimisation de *Treat* présentée plus haut est un cas particulier de filtrage vu comme un problème de satisfaction de contraintes qui utilise intensément le *look-ahead*, bien connu en satisfaction de contraintes, là ou nous n'avons dans notre formalisme qu'une vision locale des contraintes à vérifier. Il est possible, comme nous l'avons fait pour l'optimisation de *Treat*, d'étendre notre système d'inférence, à savoir les règles 1, 2 et 3 en utilisant cette technique.

## 5 Conclusions

Le formalisme que nous avons présenté permet d'étudier et de prouver facilement chacune des variantes de *Rete* (et de *Treat*). Il est suffisamment général pour couvrir la plupart des optimisations connues de ce langage.

Toutefois, il nous semble intéressant de le généraliser à certaines extensions de *Rete/Treat* apparues:

- On peut montrer qu'il est possible de doter les systèmes de production de primitives temporelles et de les intégrer efficacement dans *Rete* [Dur90].
- L'algorithme *Rete* peut être aussi facilement étendu pour implémenter efficacement un système de maintien des justifications [FL91]
- Il est possible d'étendre les systèmes de production afin de suivre plusieurs solutions en parallèle. On peut ainsi construire un système à mondes multiples [Cav91, MC91, Sau90], ou intégrer un ATMS à la de Kleer [dK85]



Le formalisme nous semble également fournir un bon cadre d'étude de la *compilation de l'enchaînement des règles* (et non pas uniquement de la phase de filtrage) dans les parties des systèmes de production où le résultat est indépendant de la stratégie de contrôle.

## Références


[Ard87] Ardouin (Claire). – *Spock-C V1: le Manuel de Référence.* – Rapport technique n° SE/87-12, Thomson-CSF, LCR, septembre 1987.

[AT89] Acharya (Anurag) et Tambe (Milind). – Production systems on message passing computers: Simulation results and analysis. *In : International Conference on Parallel Processing*. IEEE. – Saint-Charles, Ill. Usa, août 1989.

[BAB88] Butler (P. L.), Allen (J. D.) et Bouldin (D. W.). – Parallel architecture for Ops5. *In : 15th IEEE Symposium on Computer Architecture.* – Hawaii, Usa, 1988.

[BMB89] Barachini (Franz), Mistelberger (Hanz) et Bahr (Ernest). – A parallel architecture for real-time expert systems. *In : Conférence Internationale sur les Systèmes Experts et leurs Applications.* – Avignon, France, 1989.

[Cav91] Cavalcanti (Marcos). – *Les mondes multiples en* XRETE. – Rapport technique n° LACS-91-1, Thomson-CSF, LCR, avril 1991.

[dK85] de Kleer (Johan). – *An assumption-based TMS.* – Rapport technique, Palo Alto, Ca. Usa, Xerox PARC, 1985.

[Dor89] Dormoy (J-L). – Amélioration de l'efficacité du pattern-matching dans le langage à base de règles Boojum. *In : Actes de la convention IA.* – 1989.

[Dur90] Duroyon (Olivier). – *Primitives temporelles dans les générateurs de systèmes experts.* – Rapport technique n° LACS-90-4, Thomson-CSF, LCR, juillet 1990.

[Fag87] Fages (François). – On the proceduralization of rules in expert systems. *In : Programming of Future Generation Computers*, éd. par Nivat (M.) et Fuchi (K.). – North-Holland, juin 1987.

[Fag90a] Fages (François). – Consistency of Clark's completion and existence of stable models. *In : NACLP Workshop on Non-Monotonic Reasoning and Logic Programming.* – Austin, Tx, novembre 1990.

[Fag90b] Fages (François). – A new fixpoint semantics for general logic programs compared with the well-founded and the stable model semantics. *In : In-*

Revised version in Journal of New Generation Computing, 9 (3+4), October 1991.

[FL91] Fages (François) et Lissajoux (Rémi). – Systèmes experts temps-réel: une introduction au langage XRETE. *Revue Technique Thomson-CSF*, vol. 23, n° 3, septembre 1991.

[For81] Forgy (Charles L.). – *Ops5 User's manual*. – Rapport technique n° CMU-CS-81-135, Pittsburgh, Penn. Usa, Carnegie Mellon University, juillet 1981.

[For82] Forgy (Charles L.). – Rete : A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, vol. 19, septembre 1982.

[For84] Forgy (Charles L.). – *The Ops83 report*. – Rapport technique n° CMU-CS-84-133, Pittsburgh, Penn. Usa, Carnegie Mellon University, mai 1984.

[GD84] Ghallab (Malik) et Dufresne (Pierre). – Moteurs d'inférence pour systèmes de production: techniques de compilation et d'interprétation. *In : Colloque International d'Intelligence Artificielle*. – Marseille, France, octobre 1984.

[GF83] Gupta (Anoop) et Forgy (Charles L.). – *Measurements on Production Systems*. – Rapport technique n° CMU-CS-83-167, Pittsburgh, Penn. Usa, Carnegie Mellon University, décembre 1983.

[GFK+88] Gupta (Anoop), Forgy (Charles. L.), Kalp (Dirk), Newell (Allen) et Tambe (Milind). – Parallel implementation of Ops5 on the Encore multiprocessor: Results and analysis. *International Journal of Parallel Programming*, vol. 17, n° 2, 1988.

[Gup86] Gupta (Anoop). – *Parallelism in Production Systems*. – Morgan Kaufmann Publishers, 1986.

[HHW87] Hsu (Ching-Chi), Hsu (Shao-Ming) et Wu (Jan-Jan). – A distributed approach for inferring production systems. *In : International Joint Conference on Artificial Intelligence*. – Milano, Italy, août 1987.

[HS86] Hillyer (Bruce K.) et Shaw (David E.). – Execution of Ops5 production systems on a massively parallel machine. *Journal of Parallel and Distributed Computing*, vol. 3, 1986, pp. 236–268.

[HW89] Hsu (Ching-Chi) et Wang (Feng-Hsu). – The search ahead conflict resolution for parallel firing of production systems. *In : International Joint Conference on Artificial Intelligence*, pp. 91–96. – Detroit, Mn. Usa, août 1989.

[Ger85] Gerber (Richard). – *APT reference manual*, avril 1985.



[IS85] Ishida (Toru) et Stolfo (Salvatore J.). – Towards the parallel execution of rules in production systems programs. *In : International Conference on Parallel Processing*. IEEE. – 1985.

[IYG90] Ishida (Toru), Yokoo (Makoto) et Gasser (Les). – A organizationnal approach to adaptative production systems. *In : National Conference on Artificial Intelligence*. AAAI, pp. 52–58. – 1990.

[KS87] Kelly (Michael A.) et Saviora (Rudolph E.). – A multiprocessor architecture for production system matching. *In : National Conference on Artificial Intelligence*. AAAI. – Seattle, Wa. Usa, août 1987.

[KS89] Kelly (Michael A.) et Saviora (Rudolph E.). – An evaluation of DRete on CUPID for Ops5 matching. *In : International Joint Conference on Artificial Intelligence*, pp. 84–90. – Detroit, Mn. Usa, août 1989.

[Lis89] Lissajoux (Rémi). – *Spock parallèle*. – Rapport technique n° SE/89-8, Thomson-CSF, LCR, décembre 1989.

[Lis92] Lissajoux (Rémi). – *Rete paresseux*. – Rapport technique n° LACS-92-4, Thomson-CSF, LCR, août 1992.

[MBLG90] Miranker (Daniel P.), Brant (David A.), Lofaso (Bernard J.) et Gadbois (David). – On the performance of lazy pattern-matching in production systems. *In : National Conference on Artificial Intelligence*. AAAI, pp. 685–692. – 1990.

[MC91] Morgue (Geneviève) et Chéhiré (Thomas). – Performances du couplage entre un ATMS et un générateur de systèmes experts. *In : Conférence Internationale sur les Systèmes Experts et leurs Applications*. – Avignon, France, 1991.

[Mir90] Miranker (Daniel P.). – *Treat: a New and efficient match algorithm for AI production systems*. – Morgan Kaufmann Publishers, 1990.

[ML91] Miranker (Daniel P.) et Lofaso (Bernard J.). – The organization and performance of a Treat-based production system compiler. *IEEE Transactions on Knowledge and Data Engineering*, vol. 3, n° 1, mars 1991, pp. 3–10.

[OD87] Oshisanwo (A. O.) et Dasiewicz (P. P.). – A parallel model and architecture for production systems. *In : International Conference on Parallel Processing*. IEEE, pp. 147–153. – 1987.

[Ofl87] Oflazer (Kemal). – *Partitionning in Parallel Processing of Production Systems*. – Rapport technique n° CMU-CS-87-114, Pittsburgh, Penn. Usa, Carnegie Mellon University, 1987. *(Ph.D Thesis)*.

[Plo81] Plotkin (Gordon). – *A structural approach for operationnal semantics*. – Rapport technique n° Daimi FN-10, Aarhus University, 1981.

[Sav90] Savéant (Pierre). – *Raisonnement hypothétique et temps multiforme discret dans les systèmes de production: étude et implémentation.* – Thèse de PhD, Université Paris VI, décembre 1990.

[SF88] Schang (Thierry) et Fages (François). – Real-time expert system for on-board radar identification. *In : 55th Symposium AVP-AGARD on Software Engineering and its Applications to Avionics.* – avril 1988.

[SG90] Schmoltze (James G.) et Goel (Suraj). – A parallel asynchronous distributed production system. *In : National Conference on Artificial Intelligence.* AAAI, pp. 65–71. – 1990.

[SM84] Stolfo (Salvatore J.) et Miranker (Daniel P.). – DADO: a parallel processing for expert systems. *In : International Conference on Parallel Processing.* IEEE. – août 1984.

[TAG89] Tambe (Milind), Acharya (Anurag) et Gupta (Anoop). – *Implementation of production systems on Message Passing Computers: techniques, simulation results and analysis.* – Rapport technique n° CMU-CS-89-129, Pittsburgh, Penn. Usa, Carnegie Mellon University, avril 1989.

[Tho91a] Thomson-CSF/RCC. – *XIA_C Version 1.0, Manuel Utilisateur*, février 1991.

[Tho91b] Thomson-CSF/RCC. – *XIA_LISP Version 1.0, Manuel Utilisateur*, février 1991.

[Via85] Vialatte (Michèle). – *Description et applications du moteur d'inférence SNARK.* – Thèse de PhD, Université Paris VI, mai 1985.

François FAGES
Rémi LISSAJOUX

Article soumis en octobre 1991
Publication recommandée par J.-L. Dormoy

# TREE : une stratégie de jointure heuristique pour un algorithme de filtrage*

**Jacques Bouaud**

*Département intelligence artificielle et médecine*
*INSERM U194*
*Service d'informatique médicale de l'AP-HP*
*91 boulevard de l'Hôpital*
*75634 Paris cedex 13*

*Travail réalisé dans le cadre du PRC-GDR IA.

RÉSUMÉ. *TREE est la stratégie de jointure d'un algorithme de filtrage de type RETE. Une propagation des contraintes dues aux liaisons de variables sur les motifs-conditions suivi d'un relâchement fondé sur des heuristiques de choix d'index permet une réduction des espaces de recherche explorés lors de l'étape de jointure. Cette réduction s'applique à la construction des instanciations partielles et à leur actualisation. Adapté à des faits représentés par des triplets, TREE nécessite un schéma d'indexation qui n'est pas fondée sur les motifs-conditions. Une comparaison expérimentale entre TREE et une stratégie n'effectuant pas de propagation de contraintes sur les motifs-conditions montre une réduction effective des espaces de recherches utilisés.*

ABSTRACT. *This paper describes the join strategy of TREE, a RETE-like match algorithm. Constraints due to variable sharing are propagated in the condition part of a rule and then relaxed according to a heuristic choice of an index in order to reduce the search spaces during the join step. This reduction applies to partial instanciation building as well as to their updating. TREE is relevant when facts are triples and the supporting indexing scheme does not rely on memory support. In order to evaluate the benefits of TREE, we have compared it with a more classical one where no constraint propagation is performed on the condition elements. Experimental results show an actual search space reduction.*

MOTS-CLÉS : *systèmes de production, algorithme de jointure, RETE, propagation et relâchement de contraintes.*

KEY WORDS : *production systems, join algorithm, RETE, constraint propagation and relaxation.*

## 1. Introduction

Les systèmes de production, ou "systèmes à base de règles", jouent un rôle important dans le domaine de l'Intelligence Artificielle [WHR 78]. Ils sont utili-