Systèmes Intelligents : Raisonnement et Reconnaissance

James L. Crowley

Deuxième Année ENSIMAG Deuxième Sémestre 2007/2008

Séance 4 19 mars 2008

Systèmes de Productions :
L'algorithme RETE et les Structure de Contrôle

Architecture d'un Système de Production.............................2

L'Algorithme RETE..3
- Appariement des Faits et Règles..3
- Le réseau de décision...4
- L'arbre de jonctions : L'unification de faits..5
- Consignes pour l'éfficacité de RETE..5
- Remarque sur la Complexité :...6

L'Agenda..6
- Sélection (Résolution de Conflit) :..6
- Stratégies de Résolution de conflit :...7
- Illustration de l'Agenda :..8
- Salience :...9
- Hiérarchie des Saliences :...9

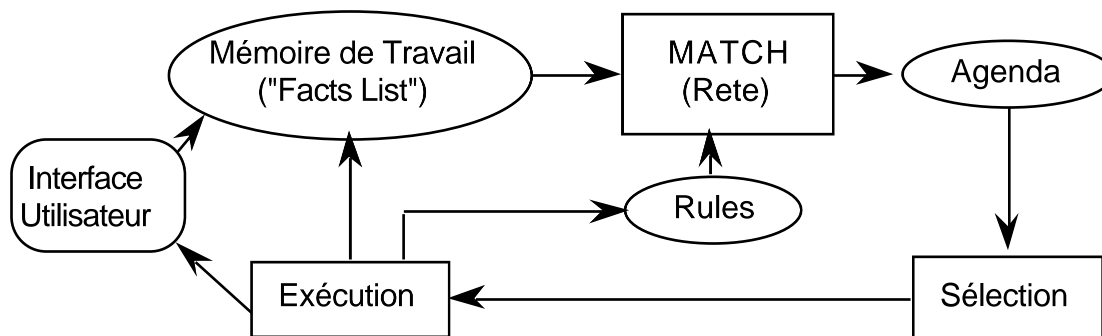
Structures de Contrôle.. 10

Phases et Éléments de Contrôle.. 10
- Expression de la structure de contrôle par règles................................11
- Expression déclarative de la structure de contrôle...............................13
- Autres formes de structure de contrôle...13

Arbres de décisions... 14

Classification des animaux 14

Architecture d'un Système de Production



Le cycle "Recognize-Act" est la base du moteur d'inférence.
Il comporte 3 phases :

- Correspondance (appariement des faits et règles)
- Sélection (Sélection d'une "activation" pour exécution)
- Exécution (L'exécution d'une règle peut engendrer la modification de la mémoire de travail).

Chaque fait est identifié par un "Indice". (ou Estampille ou "Recency")

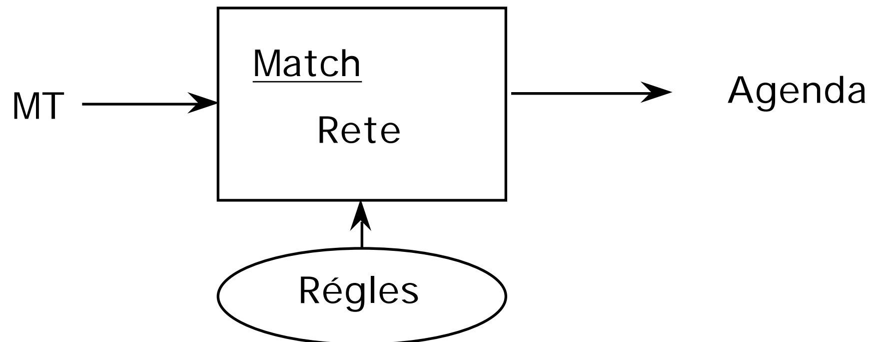
Dans chaque cycle, toutes les règles sont "mises en correspondance" avec toute la mémoire de travail.

Les associations de règles et faits sont stockées dans l'agenda.
Une des associations est choisie pour l'exécution.
Il existe plusieurs modes de triage de l'agenda.
Par défaut, l'agenda est une pile (LIFO)

L'Algorithme RETE

Appariement des Faits et Règles

Dans un système de productions, en principe, pour chaque condition de chaque règle, il faut parcourir la liste des faits.
Afin d'éviter le coût de calcul CLIPS (et OPS-5 et ART) utilise l'algorithme RETE.



RETE est une algorithme incrémentale d'unification.
De ce fait, RETE réduit fortement le temps d'exécution d'un système à règles.

L'algorithme RETE "évite" l'itération.

RETE évite l'itération. Il est incrémentale. Il fonction avec les modifications (assert, retract, modify) de la liste des faites (faits).

RETE utilise un réseau de décision fait par la compilation des règles.
L'arbre de décision contient l'information de la mémoire de travail.

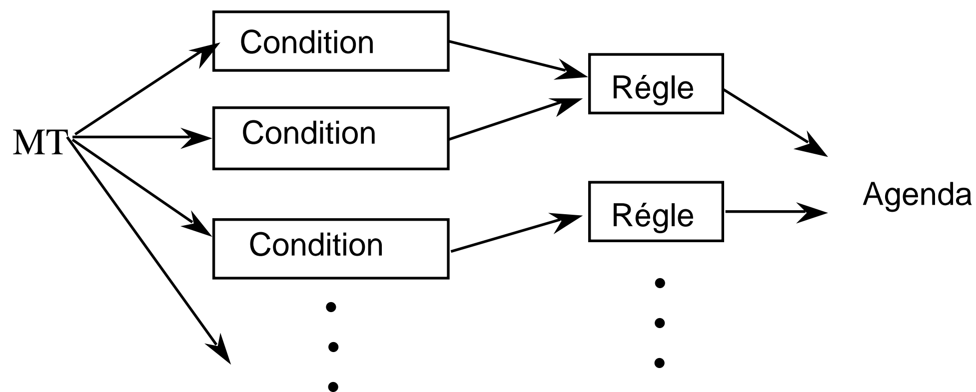
RETE est le mot latin pour Réseau.
Une modification de la Mémoire de Travail est propagée à travers un "réseau" pour engendrer des modifications dans l'agenda.
L'agenda est composé d'une liste des "activations".

Le réseau de décision

La partie condition des règles est composée de testes ("CONDITIONS").

```
(defrule    nom
        (condition-1)
        (condition-2)
=>
       (actions)
)
```

Chaque condition de chaque règle est compilé en filtre :



Les filtres de conditions sont indexés par
- Le "type" d'un template, ou
- La premier item d'une liste.

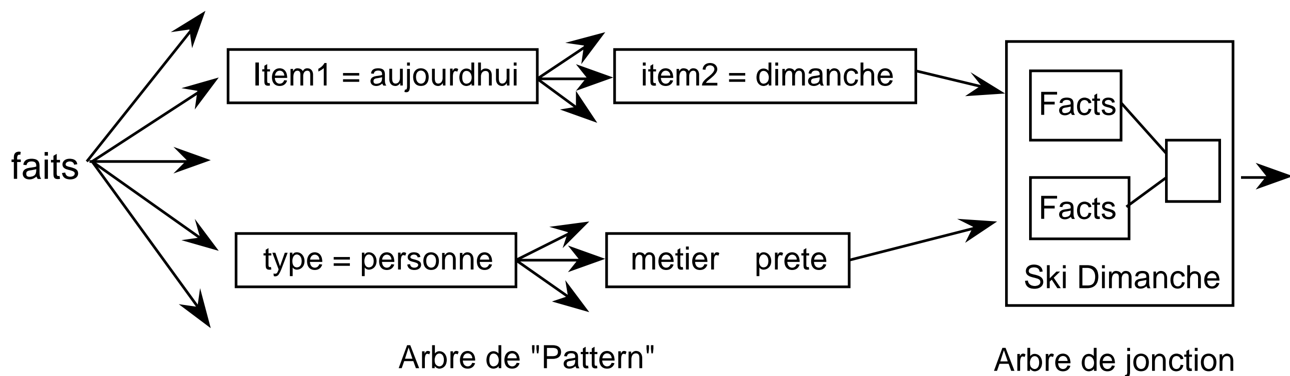
(Depuis clips 6, la première item d'une liste sert de "type" pour la liste.
D'où la nécessité de commencer les faits par un "symbole"

Ces filtres forme un arbre de patterns.

Exemple d'un arbre de décision de RETE,

```
(deftemplate personne
      (slot nom)
      (slot metier)
)

(defrule ski-dimanche
      (aujourdhui dimanche)
      (personne  (metier ?p:&~pretre))
=>
      (assert (fait du ski))
)
```



L'arbre de jonctions : L'unification de faits

Pour chaque règle il y a un arbre de jonction. Il sert à

1) maintenir une liste des objets qui satisfont chaque condition de la règle.
2) assure que les attributs variables sont les mêmes.

À l'entre de l'arbre de patterns, il y a la liste de faits ayants satisfait les conditions (passée l'arbre de patterns).

A l'intérieur, il y a un arbre de tests pour unifier les faits.
Ces testes mettres en correspondance les variables d'attributs.

Exemple : :

```
(defrule exemple
	(personne nom ?x)
	(not (père-de ?x  ?y))
	=>
```

Consignes pour l'éfficacité de RETE

Principes pour l'organisation du parti « Condition » des règles. Ces principes sont une conséquent du fait que les conditions sont évaluées de l'haut vers le bas.

1) Placer la spécifique avant générale. Les règles avec le moins de variables et jokers en avant.
2) Placer les formes rares en avant. Chaque condition concerne une classe de fact. Placez en premier les conditions concernent les classes rares.
3) Tester les facts « volatile » en derniers.

Remarque sur la Complexité :

Pour le temps d'exécution d'un cycle :

Soit :

P : Nombre de règles
W : Nombre d'éléments dans les faits
C : Nombre de conditions dans une règle.

La complexité de calcul d'un cycle "recognize-act" est

Meilleur cas $O(\log_2 P)$
plus mauvais cas $O(P W^c)$

Dans les cas normals, la croissance de temps d'exécution est pratiquement linéaire avec le nombre de faits et de règles, mais avec un taux de croissance très faible. On peut, ainsi, avoir les systèmes avec les milliers de faits et règles.

L'Agenda

AGENDA : l'agenda est une liste des instances des règles associées avec les variables. Chaque activation est une association des faits et règles.

Réfraction: Une fois qu'un fait est associe à une règle est exécuté, l'association est éliminée de l'agenda.

Sélection (Résolution de Conflit) :

Principes de Sélection :

Réfraction : Une association de règles et faits peut être exécutée qu'une fois.

Recency : Les activations sont triées utilisant le plus grand indice de leurs faits.
Variation : MEA, Le fait satisfaisant la premiere condition détermine "recency" utilisé de trier l'activation .

Spécificité : Les activations sont triées au base de nombre de tests dans les conditions.

Par exemple :

```
(defrule example
   (item ?x ?y ?x)
   (test (and (numberp ?x) (> ?x (+ 10 ?y)) (< ?x 100)))
   =>)
```

a specificity 2

Stratégies de Résolution de conflit :

CLIPS contient sept modes de "Sélection" (Stratégies de résolution de conflit)

1) "Depth Strategy" (par profondeur): Option par défaut.
L'agenda est trié par "saliency",
pour des valeurs de "saliency" égales,
l'agenda est une PILE d'activations. (LIFO)

2) "Breadth Strategy" (largeur d'abord).
L'agenda est trié par "saliency", pour une valeur de "saliency" l'agenda est une queue des activations. (FIFO)

3) LEX strategy (Lexographic). Pour compatibilité avec OPS-5.
L'agenda est une PILE, pas de saliency. Les activations sont triée par le "recency" des faits. Pour les activations le
plus récent, les activation sont triées par le nombre de conditions. (mélange de "depth" et "complexity").

4) MEA strategy (Means-Ends-Analysis) Pour compatibilité avec OPS-5 Les activations sont triées sur la base du recency de la première condition de la règle, puis par nombre de conditions

5) Complexity Strategy: Les règles avec le plus de conditions ont priorité.

6) Simplicity: Les règles avec le moins de conditions ont priorité.

7) Random: Aléatoire.

Le strategy "Depth" est recommandé.

Illustration de l'Agenda :

```
 (set-strategy depth)
 (get-strategy)

(defrule rule-A
	?f <- (a)
=>
	(printout t "Rule A fires with  " ?f  crlf)
)

(defrule rule-B
	?f <- (b)
=>
	(printout t "Rule B fires with  " ?f crlf)
)

(defrule rule-A-and-B
	?f1 <- (a)
	?f2 <- (b)
=>
	(printout t "Rule B fires with  A =" ?f1 " and B = " ?f2  crlf)
)

(assert (a))
(assert (a))
(assert (b))

(set-strategy depth)
(set-strategy breadth)
(set-strategy lex)
(set-strategy mea)
(set-strategy complexity)
(set-strategy simplicity)
(set-strategy random)
(set-strategy depth)
```

Salience :

Normalement l'agenda est une pile.
CLIPS fournie une méthode de sélection de priorité des règles. "Salience"
(sailliance)

```
(defrule exemple
        (declare (salience 99))
        (initial-fact)
    =>
        (printout "J'ai un salience de 99" crlf)
)
```

Le valeur de salience peut être entre –10 000 et 10 000.

Il y a tendance pour les débutants d'abuser le "salience".
Ce permet de commander l'ordre d'exécution des règles. Or, justement, si le système est bien conçu, on n'a pas besoins d'être concernées par l'ordre
exact des règles, mais plutôt des blocs de règles.

Un système bien fait utilisera 3 ou 4 niveaux de salience.
Plus que 7 doit jamais être nécessaire.

En place de salience, il faut structurer les règles avec les éléments de contrôle.

Hiérarchie des Saliences :

En général dans un système expert, il y a quatre niveaux de salience.
Je les marque ici par les échelles de 100, mais les chiffres exacts sont arbitraires.

Niveau	Salience	
Contraints	300	;; Règles qui permettre une réduction des hypothèses.
Expert	200	;; Connaissance du domaine (inférer et trier les hypothèses)
Query	100	;; Interroger l'utilisateur
Contrôle	0	;; transitions de phases.

Cette hiérarchie dépend du problème et du régime de Contrôle.

Structures de Contrôle

Un système expert est une machine à états fini. Ces états sont les "contextes" (ou phases). Ces contextes peuvent être organisés en cycles, en réseaux ou en arbres
Les transitions peuvent être codé par régles (procédurales) ou par faits (déclaratifs).

Le contrôle peut être autre chose qu'une séquence de phases.
Il est courant d'utiliser les arbres et les réseaux de contextes.

Arbre de phases : exemple - l'arbre de contextes de Mycin.
Réseau de phases : certains problèmes se structurent en réseau
Dans ce cas, le contrôle devient un parcours de graphe !
A chaque phase, nous avons un ensemble d'actions possibles.

Phases et Éléments de Contrôle

En général on cherche de séparer les règles en ensemble exécutable en parallèle.

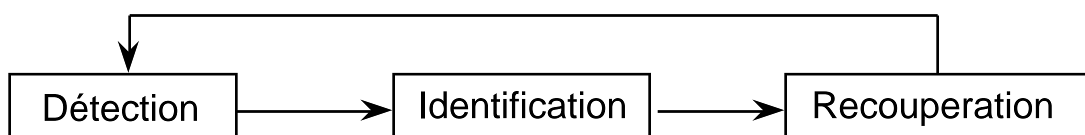
Chaque ensemble est une "phase". Les phases sont démarqués par la présence d'une élément dans la liste des faits du style :

(phase <NOM-DE-PHASE>)

Il faut chercher de séparée le problème en phases, et d'écrire un ensemble de règles qui réagir à chaque phase.

Exemple : un système de surveillance d'un appareil électronique (ex : satellite).
est composé par trois phases :

1) Détection - Détection d'existence d'une panne
2) Isolation - Détermination de l'identité du composant en panne.
3) Recouperation - Reconfiguration pour contourner la panne



On peut imaginer trois façons de coder ces règles :

1) Coder la condition de chaque phase dans les règles.

Pb: les règles deviennent complexes.
Il est difficile de déterminer quand une phase est terminée.
Pb: Le contrôle est inscrit dans les règles.

2) Utiliser le salience de définir une hiérarchie des règles.

exemple : Détection : Salience 3
Identification : Salience 2
Recouperation : Salience 1.

Pb: Il est difficile d'être sûr de la séquence d'exécution

3) Utiliser un élément "phase" de marquer chaque phase et
Utiliser les règles de exprimer les transitions.

Ce troisième solution est préférable.

Les règles de transition constitue la structure de contrôle du système
Cette structure peut être entièrement exprimé par des règle, ou
son expression peut être dans les faits avec quelque règles de interprétation.

<u>Expression de la structure de contrôle par règles</u>

Il s'agit d'une règle par transition d'état.
Par exemple :

```
(defrule detection-vers-identification
    (declare (salience -10))
    ?phase <- (phase detection)
    (panne detecte)   ;; conditions de detection du panne
=>
    (retract ?phase)
    (assert (phase identification))
    (printout t "il y a une panne!" crlf)
)

(defrule identification-vers-recouperation
    (declare (salience -10))
    ?phase <- (phase identification)
    (panne id ?p)  ;; conditions de identification  du panne
=>
```

```
    (retract ?phase)
    (assert (phase recouperation))
    (printout t "Le(a) " ?p " est en panne." crlf)
)

(defrule recouperation-vers-detection
    (declare (salience -10))
    ?phase <- (phase recouperation)
    (panne isole)  ;; conditions de identification  du panne
=>
    (retract ?phase)
    (assert (phase detection))
    (printout t "panne elimine" crlf)
)
```

À l'intérieur de chaque phase on a des règles de connaissance du domaine :

par exemple :

```
(defrule trouver-panne  ;  une triche - on demande
    (phase identification)
=>
    (printout t "eh alors? c'est quoi cette panne? ")
    (assert (panne id =(read)))
)

;; function factice pour set
(deffunction set (?a ?b)
    (printout t ?a " is set to " ?b crlf))

;; tester la fonction
    (set moteur off)

;; expertise du domaine

(defrule recouperer-du-fault-moteur
    (phase recouperation)
    (panne id moteur)
=>
    (set moteur off)  ;; fonction definit par l'utilisateur.
    (set back-up-moteur on)
    (assert (panne isole))
)
```

Expression déclarative de la structure de contrôle

Il est possible d'exprimer les phases comme une liste dans une "deffacts".
puis utiliser une seule règle d'effectuer la transition :

```
(deffacts controle-liste
    (phase detection)
    (phase-apres detection identification)
    (phase-apres identification recouperation)
    (phase-apres recouperation detection)
)

(defrule control-des-phases
    (declare (salience -10))
    ?P <- (phase ?phase)
    (phase-apres ?phase ?prochaine)
=>
    (retract ?P)
    (assert (phase ?prochaine))
)
```

Autres formes de structure de contrôle

Le contrôle peut être autre chose qu'une séquence de phases.
Il est courant d'utiliser les arbres et les réseaux de contextes.

Arbre de phases : exemple - l'arbre de contextes de Mycin.

Réseau de phases : certains problèmes se structurent en réseau
Dans ce cas, le contrôle devient un parcours de graphe !
A chaque phase, nous avons un ensemble d'actions possibles.
Certaines actions sont spécifiques au problème
(connaissance du domaine)
Certaines de ces actions nous amènent à d'autres phases (contrôle)

Arbres de décisions

Les arbres de décision sont utiles pour les problèmes de classification.
Caractéristiques d'un problème de classification :

1) L'ensemble des réponses possibles est fini et connu d'avance.
(exemples : problème de diagnoses et de taxonomie.

2) L'espace de solutions est réduit par une série de tests.

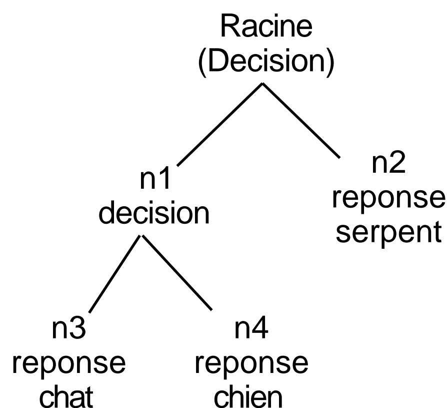
Les arbres de décisions ne fonctionnent pas bien pour les problèmes de planification, ordonnancement, ou synthèses.
Ils fonctionnent mal s'il faut construire la solution partielle à chaque étape.

Un arbre de décisions est composé de noeuds et de branches.
Les noeuds peuvent être les décisions ou les réponses.
Les ensembles de réponses possibles sont les feuilles de l'arbre.

Classification des animaux

Exemple d'un arbre de décisions étant un système pouvant "apprendre" à identifier les animaux :

L'arbre sera représenté par les noeuds du type décision ou réponse :




```
(deffacts arbre
(noeud racine décision "Est-ce un animal de sang-chaud?" n1
n2)
(noeud n1 decision "Est-ce qu’il ronronne ?" n3 n4)
(noeud n2 reponse serpent)
(noeud n3 reponse chat)
(noeud n4 reponse chien)
```

```
)

;; vielle règle d'init

(defrule init
    (initial-fact)
=>
 (assert (noeud-actuel racine))
)
```

;; règle à demander pour le noeud de décision

```
(defrule faire-décision
    ?N <- (noeud-actuel ?nom)
    (noeud ?nom decision ?q ?oui ?non)
=>
    (retract ?N)
    (format t "%s (oui ou non) " ?q)
    (bind ?réponse (read))
    (if (eq ?réponse oui)
       then (assert (noeud-actuel ?oui) )
       else (assert (noeud-actuel ?non))
    )
)
```

;; règle pour traiter les réponses

```
(defrule faire-réponse
    ?N <- (noeud-actuel ?nom)
    (noeud ?nom réponse ?r)
=>
    (printout t "Je pense que c'est un " ?r crlf)
    (printout t "c'est correct ? (oui ou non) ")
    (bind ?rep (read))
    (if (eq ?rep oui)
       then (assert (phase demande-encore))
           (retract ?N)
       else (assert (phase corrige-réponse))
    )
)
```

;; règle pour essayer encore

```
(defrule essaie-encore
    ?phase <- (phase demande-encore)
=>
    (retract ?phase)
    (printout t "essaie encore ? (oui ou non))
    (bind ?rep (read))
    (if (eq ?rep oui)
       then (assert (noeud-actuel racine))
       else (save-facts "animal.dat")
    )
)
```

```
;; règles pour apprendre une nouvelle réponse

(defrule corrige-réponse
   ?P <- (phase corrige-réponse)
   ?N <- (noeud-actuel ?nom)
   ?D <- (noeud ?nom réponse ?r)
=>
   (retract ?P ?N ?D)  ;;    demander la bonne réponse
   (printout t "quel est l'animal? ")
   (bind ?new (read))
   (printout t
      "quelle question  poser pour distinguer " crlf)
   (printout t "un " ?new " d'un " ?r "? ")
   (bind ?question (readline))
   (bind ?newnode1 (gensym*))
   (bind ?newnode2 (gensym*))
   (assert (noeud ?newnode1 reponse ?new))
   (assert (noeud ?newnode2 reponse ?r))
   (assert
      (noeud ?nom decision ?question ?newnode1 ?newnode2))
   (assert (phase demande-encore))
)


;; Règles d'ouverture et de lecture du fichier animal.dat

(defrule init
   (initial-fact)
=>
 (assert (file =(open "animal.dat" data "r")))
)
```

```
;; ouverture et fermeture du fichier animal.dat

(defrule no-file
   ?f <- (file FALSE)
=>
   (retract ?f)
   (assert (noeud-actuel racine))
)

;; lecture du fichier

(defrule init-file
   ?f <- (file TRUE)
=>
 (bind ?in (readline data))
 (printout t ?in crlf)
 (if (eq ?in EOF) then (assert (eof))
  else
    (assert-string ?in)
    (retract ?f)
    (assert (file TRUE))
   )
)

(defrule eof
   (declare (salience 30))
   ?f <- (file TRUE)
   ?eof <- (eof)
=>
   (retract ?f ?eof)
   (close data)
   (assert (noeud-actuel racine))
)
```