

Programmation orientée agents #3

Architectures réactives

v2.6 – Octobre 2013

M1 S2 - Université de Montpellier II

FMIN207 - Master d'informatique

Jacques Ferber

Resp du module: J. Ferber

Notion d'architecture d'agent

◆ Question: comment programmer un agent?

◆ 1^{ère} solution:

- Programmer en faisant au mieux..
 - ☞ Difficile de mettre au point (car tout est mélangé)
 - ☞ Pas d'abstraction

◆ 2^{ème} solution:

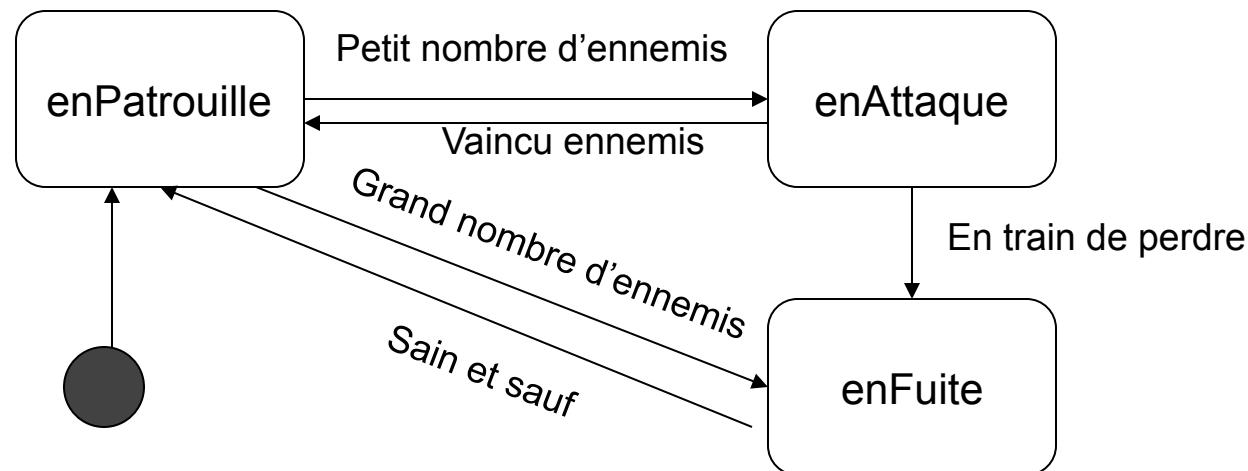
- Identifier des patterns de programmation caractéristiques
 - ☞ Permet d'avoir une approche plus méthodique

Architectures réactives

- ◆ **Architectures d'agent qui ne prennent pas en compte la notion de représentation**
 - (Ou de manière très simplifiée)
- ◆ **Deux approches:**
 - Partir d'une notion d'état interne:
 - ☞ FSM (Finite State Machine)
 - Partir d'une relation aux perceptions
 - ☞ Architecture de subsumption
 - ☞ Architecture neuronale
 - Prise en compte de la satisfaction de l'agent
 - ☞ Eco-résolution
 - ☞ Satisfaction / altruisme
 - Adaptation:
 - ☞ EMF
 - ☞ Etc...

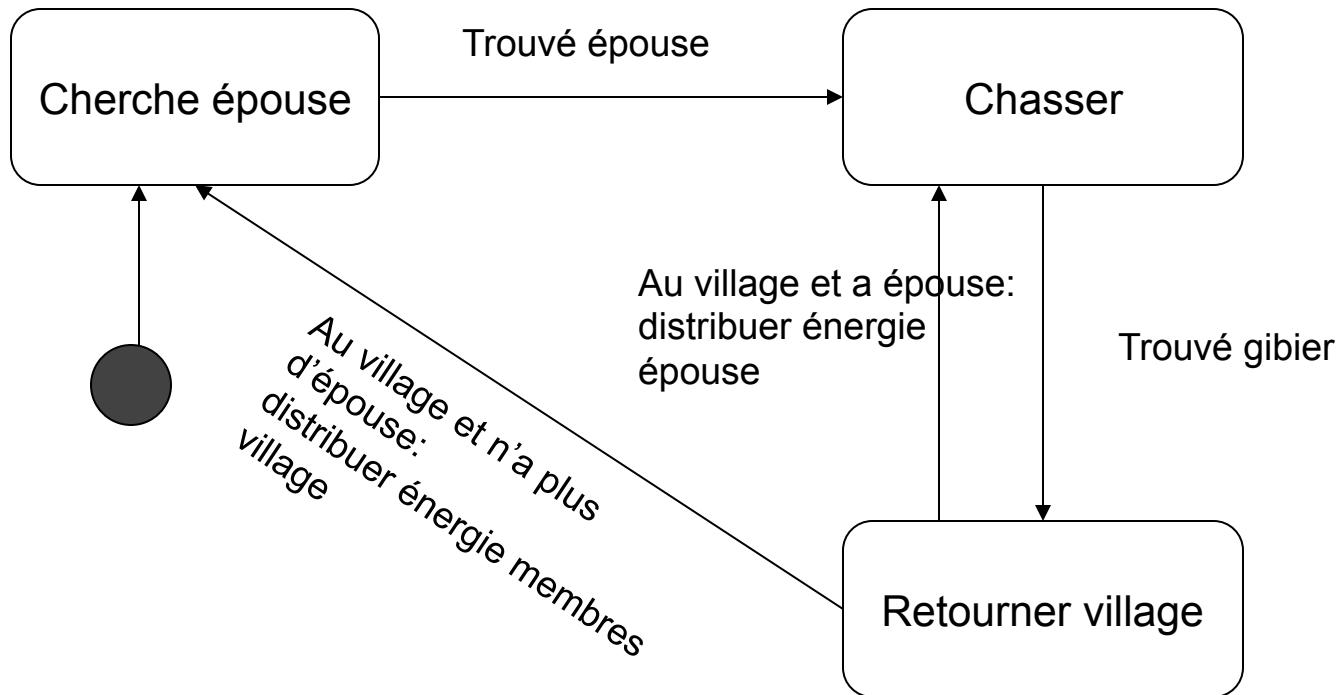
Machines à états finis (FSM, automates)

- ◆ **Etats:** un état ou une activité de l'agent
- ◆ **Événement:** quelque chose qui se passe dans le monde extérieur (ou intérieur à l'agent) et qui peut déclencher une action
- ◆ **Action:** quelque chose que l'agent fait et aura pour conséquence de modifier la situation du monde et de produire d'autres événements
 - Souvent l'action est impliquée par l'état



Jeu d'évolution du monde

◆ Un chasseur dans une société de chasseur-cueilleur



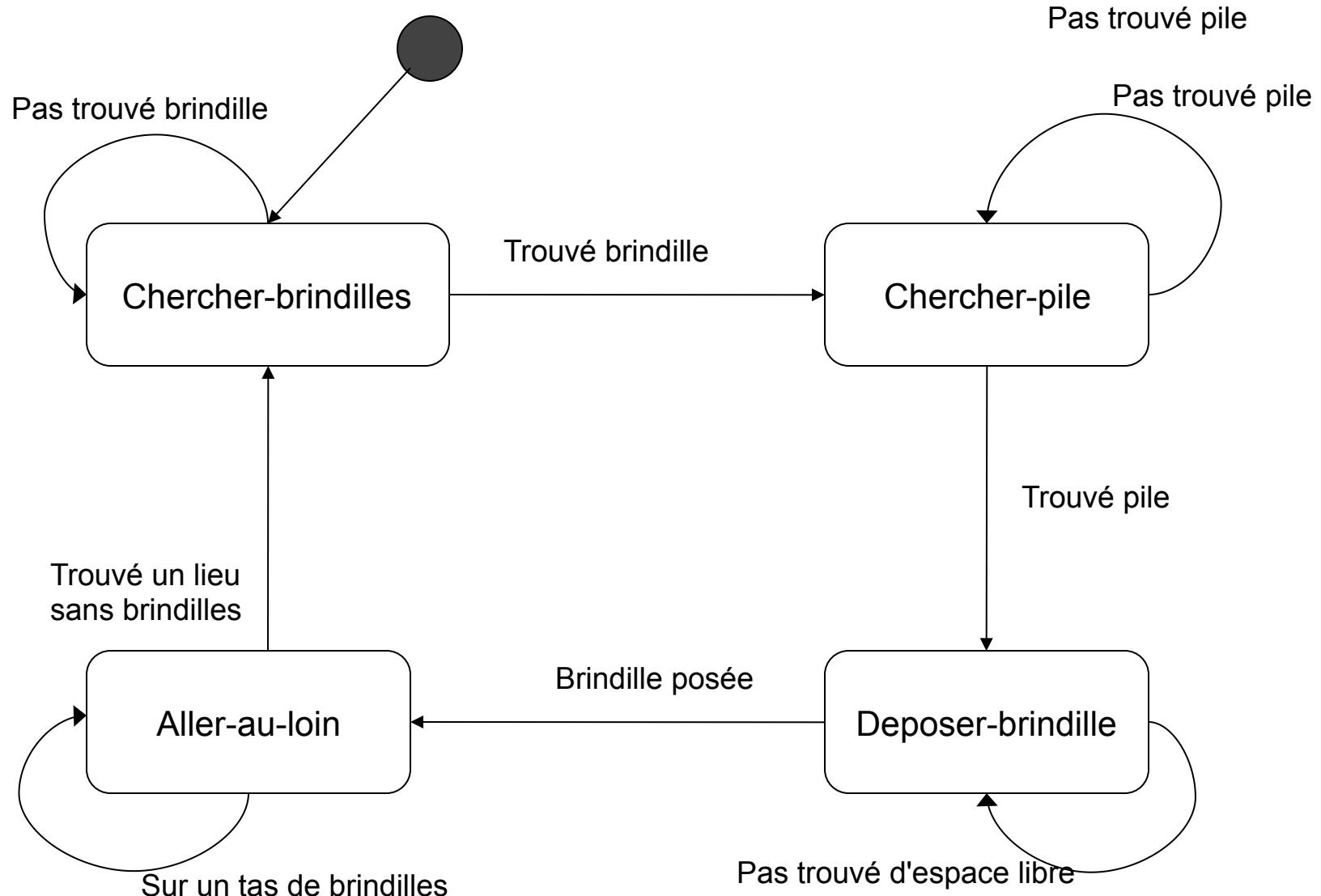
Implémentation de FSM #1

```
void Garde::FSM(TypeEtat etat){  
    switch(etat){  
        case etat_enFuite:  
            echapperEnnemis();  
            if (sauf())  
                changerEtat(etat_Patrouille);  
            break;  
        case etat_Patrouille :  
            patrouiller();  
            if (menace()) {  
                if (plusFortQueEnnemis())  
                    changerEtat(etat_attaque);  
                else  
                    changerEtat(etat_enFuite);  
            } break;  
        case etat_attaque :  
            seBattre();  
            if (ennemisVaincus())  
                changerEtat(etat_enPatrouille);  
            else if (plusFaibleQueEnnemis())  
                changerEtat(etat_enFuite);  
            break;  
    }//end switch  
}
```

◆ Coder l'automate directement dans le code

- grand ‘switch’ en fonction des états

Les termites en FSM



Le code

◆ Implémentation en utilisant une variable "task"

```
turtles-own [ctask]

to go
  ask turtles
    [run ctask]
end

to chercher-brindilles
  ifelse pcolor = yellow
    [ set pcolor black
      set color orange
      fd 20
      set ctask "chercher-pile"
      [wiggle fd 1]
    ]
  end

  to chercher-pile
    ifelse pcolor = yellow
      [set ctask "deposer-brindille"]
      [wiggle fd 1]
    end

  to deposer-brindille
    ifelse pcolor = black
      [ set pcolor yellow
        set color white
        fd 20
        set ctask "aller-au-loin" ]
      [wiggle fd 1 ]
    end

  end

  to get-away
    ifelse pcolor = black
      [ set task "chercher-brindilles" ]
      [wiggle fd 1]
    end
```

Implémentation à l'aide de tables

◆ Construit une table correspondant au système des états

- Un interprète va sélectionner l'état courant dans la table et déclencher la chose à faire ensuite
- Si pas de condition vérifiée on demeure dans le même état

Etat courant	Action	Condition	Etat suivant
enFuite	fuirEnnemis	Sauf	Patrouille
Patrouille	Patrouiller	Menace ET ennemisPlusfort	enFuite
--		Menace Et ennemisMoinsfort	enAttaque
enAttaque	seBattre	ennemisVaincus	Patrouille
--		ennemisPlusFort	enFuite

Approche Objet

◆ Classe Etat

- Chaque état est implémenté sous la forme d'une classe
- Création ou recherche dans un dictionnaire

```
class Etat {  
    abstract void exec(Agent ag);  
}  
  
class EtatEnFuite extends Etat {  
    void exec(Agent ag){  
        ag.echapperEnnemis();  
        if (ag.sauf())  
            ag.changerEtat(  
                Etat.look("EtatPatrouille"));  
    }  
}  
  
Class Agent {  
    Etat etatCourant;  
  
    void do(){  
        etatCourant.exec(this);  
    }  
    void changerEtat(Etat etat){  
        etatCourant = etat;  
    }  
    boolean sauf(){... }  
    void echapperEnnemis() { ... }  
}
```

Améliorer le FSM

Avoir une machine totalement générique

- Tout est codé dans les états et les transitions..

```
class Agent {  
    Etat etatCourant;  
  
    void go(){  
        while (!etatCourant.etatFinal())  
            etatCourant.exec(this);  
    }  
    void changerEtat(Etat etat){  
        etatCourant = etat;  
    }  
}
```

```
class Etat {  
    Agent ag;  
    List transitions;  
    void exec(Agent ag){  
        boolean tvalide=false;  
        for (Transition x : transitions)  
            if x.valide(){  
                x.execTransition()  
                tvalide=true; break;  
            }  
        if !tvalide  
            this.activite();  
    }  
    abstract void activite();  
  
    void etatFinal(){  
        return false;  
    }  
}  
  
class Transition  
{  
    Agent ag;  
    Etat etat;  
    void execTransition(){  
        ag.changerEtat(etat);  
        etat.activite();  
    }  
}  
abstract boolean valide();  
}
```

Application

◆ Application à la patrouille

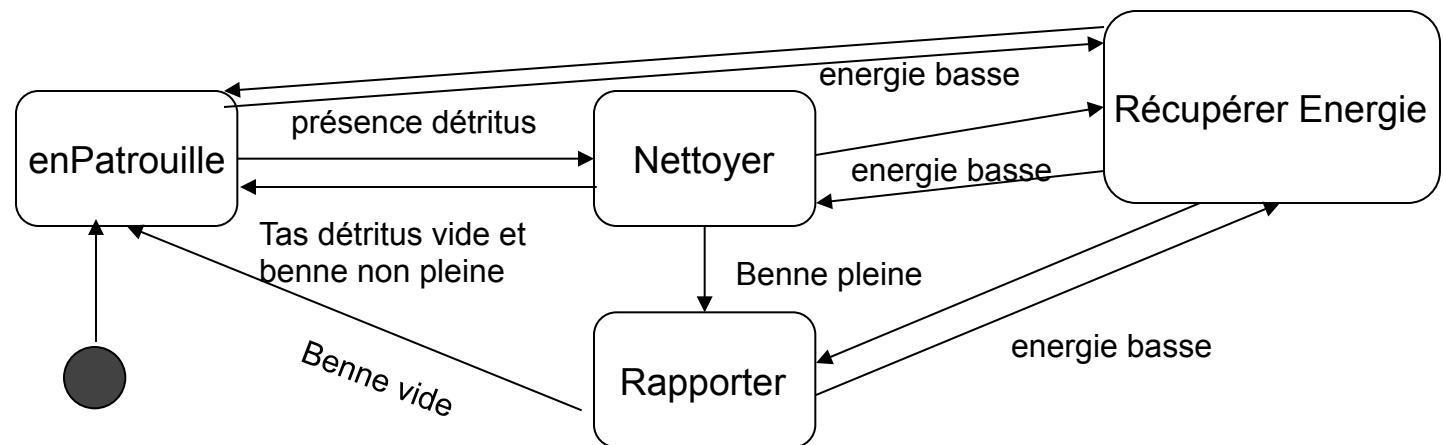
```
class Etat—EnFuite extends Etat{
    void exec(){
        // fuir..
    }
}

class Etat—EnPatrouille extends Etat{
    void exec(){
        // patrouiller..
    }
}
```

Machine à états finis hiérarchique

- ◆ Si l'on a besoin de caractériser des "modes" comportementaux différents, exprimés de manière hiérarchique
 - Ex:

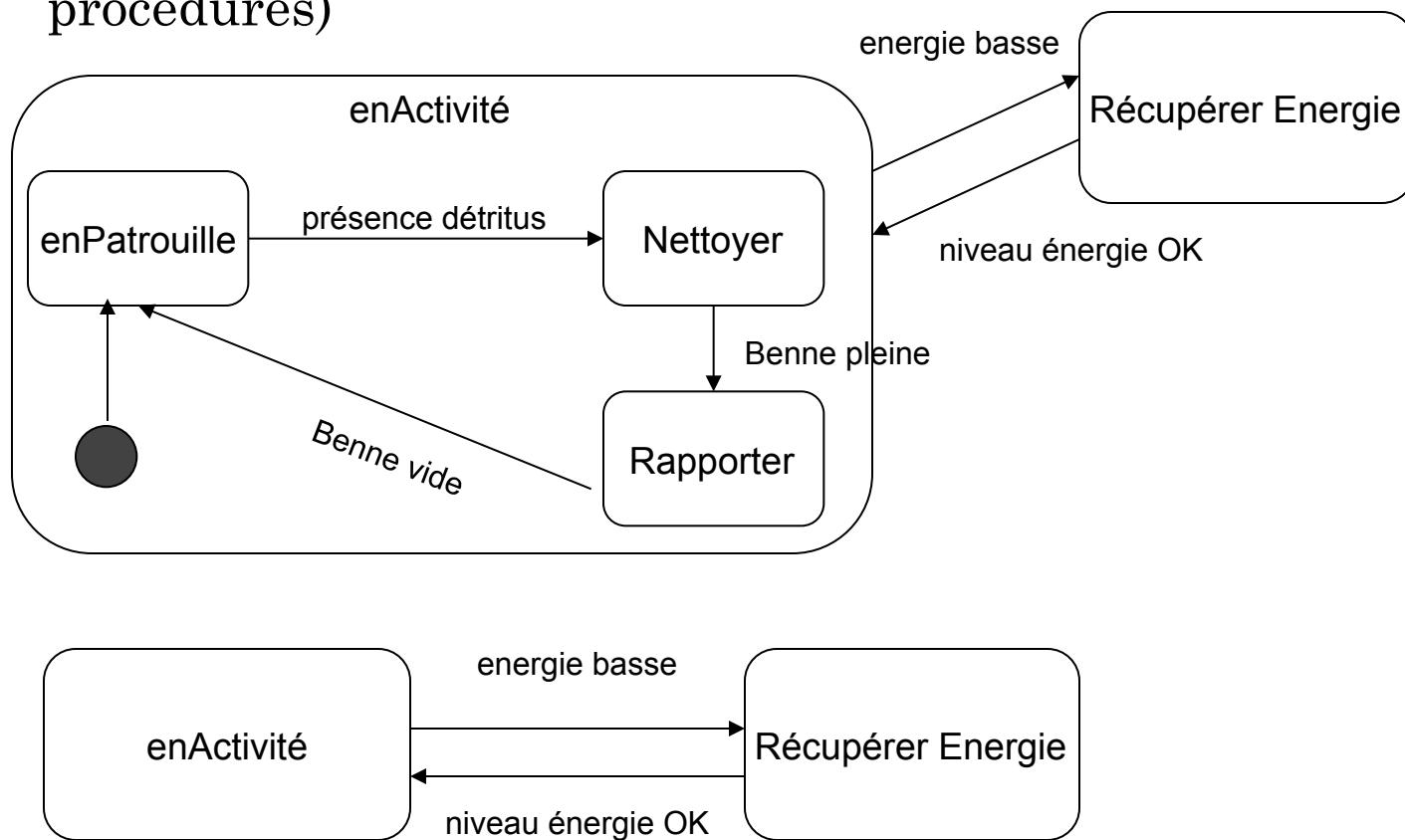
☞ Un robot doit récupérer de l'énergie quoi qu'il fasse par ailleurs



FSM hiérarchique

◆ Construction d'activités correspondant à un automate

- Nécessite d'implémenter une pile (fonctionnement des procédures)



FMS Hiérarchique

- ◆ **Nécessite d'implémenter une pile d'états**
 - Pour conserver l'état d'où l'on est parti
 - Fonctionnement standard des appels de procédures
- ◆ **Nécessite d'implémenter des mécanismes d'exception**
 - Quel que soit l'état situé dans le FSM de "enActivité", il faut dépiler le FMS courant

Noyau d'une FSM hiérarchique

```
turtles-own [task-pile ctask]

;; penser à initialiser l'agent
set task-pile []
set ctask "<proc initiale>"

to go-mode [mode] ;; en fait la proc initiale du mode
  set tasks-pile fput ctask task-pile
  set ctask mode
end

to return-mode
  set ctask first task-pile
  set task-pile bf task-pile
end

to go
  ask turtles [run ctask]
end
```

Différence réflexe - mode

- ◆ **Les réflexes correspondent à une action élémentaire qui peut être faite à chaque tick et qui se termine à chaque tick**
 - Ce n'est pas une activité
- ◆ **Implémentation des modes sous la forme de FMS hiérachique**
 - Retour à l'endroit de l'appel
- ◆ **Implémentation des réflexes au début du 'go', du comportement de l'agent**

Implémentation des réflexes

```
turtles-own [tctask]  
;; penser à initialiser l'agentset ctask "<proc  
initiale>"  
  
to go  
  ask turtles [  
    do-reflexes  
    run ctask  
  ]  
end  
  
to do-reflexes  
  .... ;; les réflexes  
end
```

Architectures à partir de perceptions

- ◆ Considérer que pratiquement toute l'info se trouve dans l'environnement
 - Action situées
- ◆ Dirigée par les événements (percepts)
- ◆ Donner une grande importance à l'environnement
 - Buts et obstacles sont dans l'environnement
 - Communication par dépôt de marques (indices et chemins)

Actions situées

◆ Comportement est lié aux événements (percepts)

- Pas de mémorisation de l'environnement
 - ☞ Pur: perception immédiate (pas de mémoire)
 - ☞ Impur (avec apprentissage): mémorisation uniquement des états internes passés..

◆ Règles d'action

- Si <état interne> and <état perçu> alors <action>

R1

Si j'ai soif et
je vois du café sur la table et
je suis loin de la table
alors je m'approche
de la table

R2

Si j'ai soif et
je vois du café sur la table et
je suis près de la table,
alors je prend le café

Différence actions situées-FMS

- ◆ **FSM: l'important c'est l'état (l'activité en cours). Et les perceptions contribuent à modifier l'activité**
 - **Avantages:** permet de caractériser un comportement procédural (faire ça, puis ça) qui est déclenché en fonction de conditions extérieures
 - **Inconvénient:** difficultés à prendre en compte ce qui n'est pas décrit pas l'automate
- ◆ **Actions situées: l'important ce sont les perceptions**
 - **Avantages:** permet de prendre en compte l'état immédiat tel qu'il est, là maintenant
 - **Inconvénient:** difficulté à caractériser des suites d'action

Codage des actions situées

◆ Perceptions

- Perceptions externes (vision, ouïe, radars, etc..)
 - ☞ Eventuellement issues d'une phase de reconnaissance.
Ex: si je perçois un ennemi...
- Capteurs sur des données corporelles du robot..
 - ☞ Ex: si j'ai faim, si mes points de vie < valeur, etc..

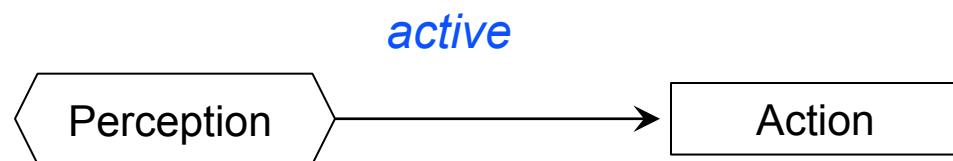
◆ Actions (ou tâches)

- Actions élémentaires
 - ☞ Avancer, reculer, fuir, aller vers la perception, etc..
- Actions composites
 - ☞ Rapporter nourriture

Notation graphique des actions situées

Perception

Action

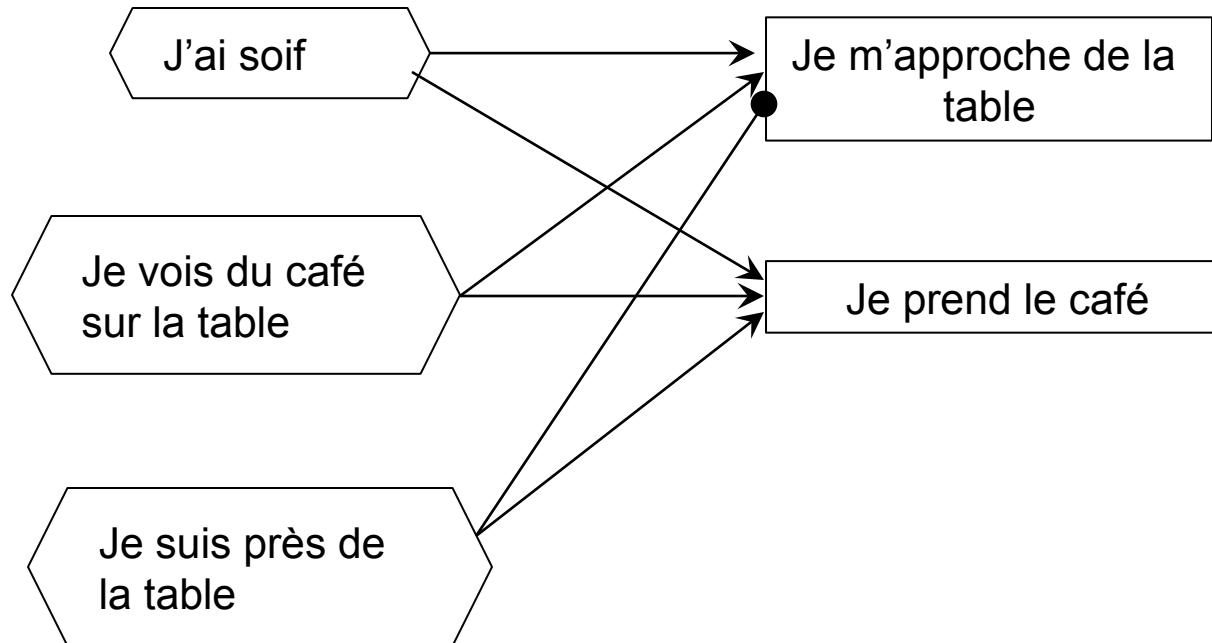


Si perception alors action

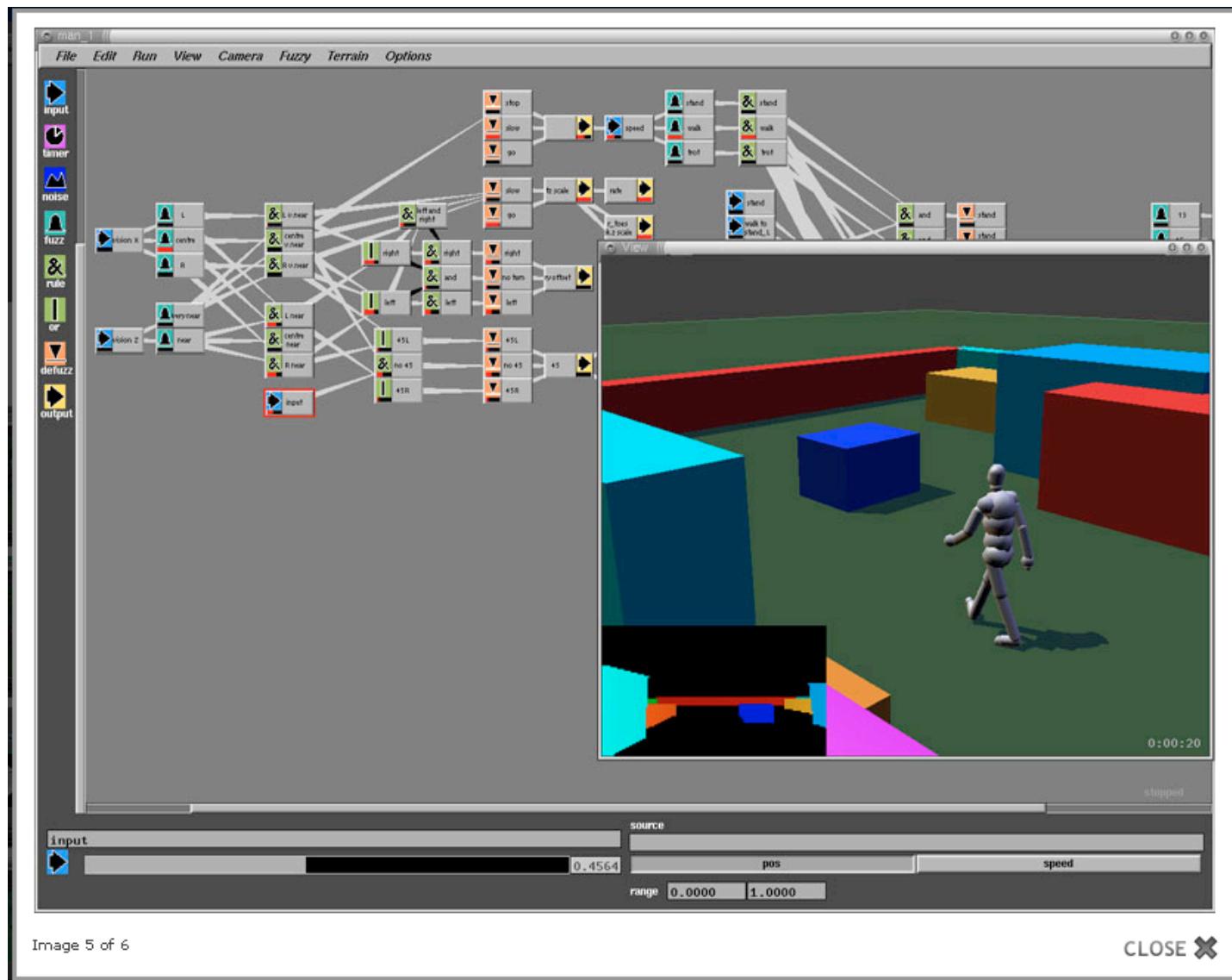


Si perception alors \neg action

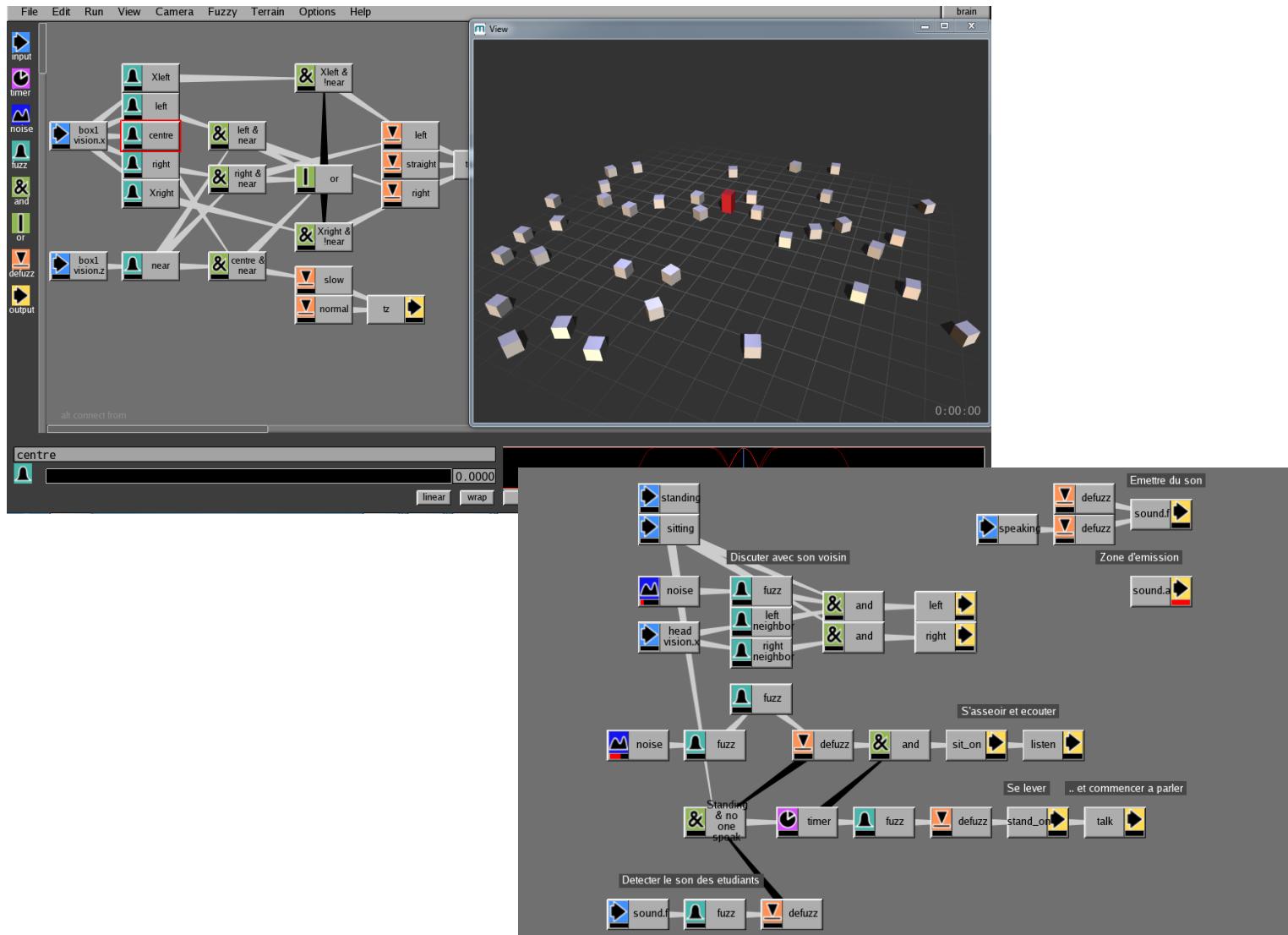
Exemple du café



Ex: architecture de Massive

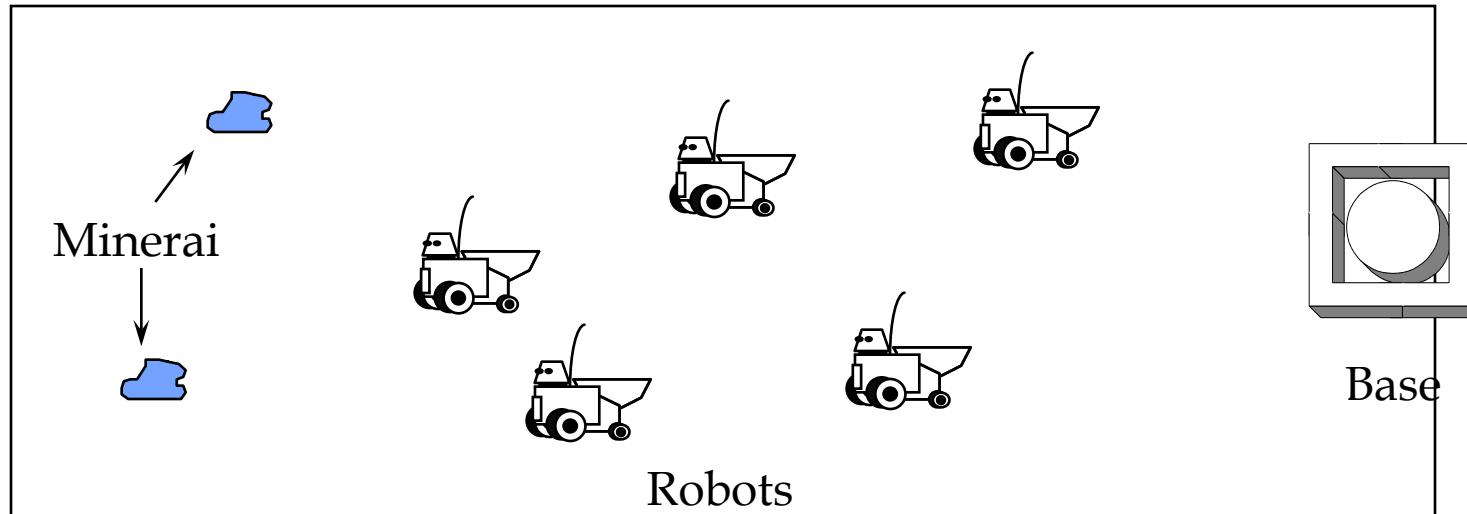


Massive - Brain

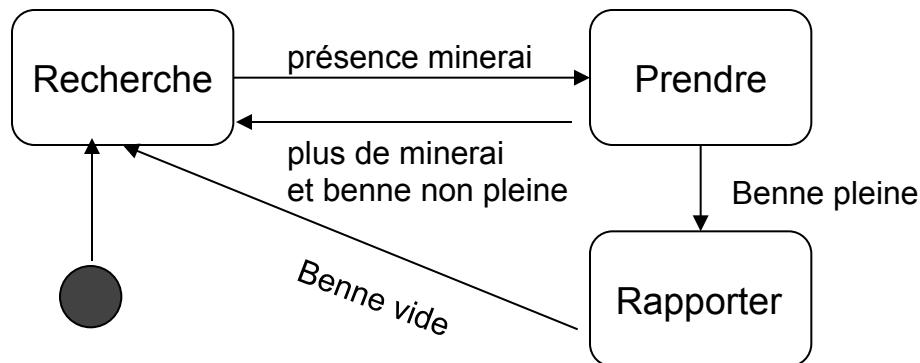


Robots (fourmis) explorateurs

- ◆ Un ensemble de robots doivent collecter et ramener des échantillons de minerai à la bases (identique au pb du fourrage des fourmis)
- ◆ Problème: comment décrire leur comportement et leur technique de coordination afin qu'ils remplissent leur mission.
- ◆ Hypothèses:
 - ◆ Ils ne peuvent pas s'envoyer des messages par radio.
 - ◆ Les robots sont tous similaires (totipotence)



Solution à base de FSM



Règles d'action situées: sans coordination

Premières règles

Règle d'exploration

si je ne porte rien
et je ne perçois pas de minerai
alors marche aléatoire

Règle rapporter

Si je porte du minerai
et je ne suis pas à la base
alors retourner à la base

Règle trouver

Si je ne porte rien
et je perçois du minerai
then I take a sample of it

Règle déposer

Si je porte du minerai
et je suis à la base
alors déposer du minerai

**Aucune coordination
entre les agents**

Règles d'action située: avec coordination

Idée: utiliser des marques à la manière des fourmis (et du petit poucet) pour retrouver l'emplacement du mineraï

Règle explorer (changée)

Si je ne porte rien et je ne perçois aucun mineraï et je ne vois pas de marques alors marche aléatoire

Règle Rapporter (changée)

Si je porte du mineraï et je ne suis pas à la base alors revenir à la base et déposer 1 marque

Règle SuivreMarques

Si je ne porte rien et je ne perçois aucun mineraï et je vois des marques alors aller vers les marques

Une situation en cours

Cf. Démo sous MadKit

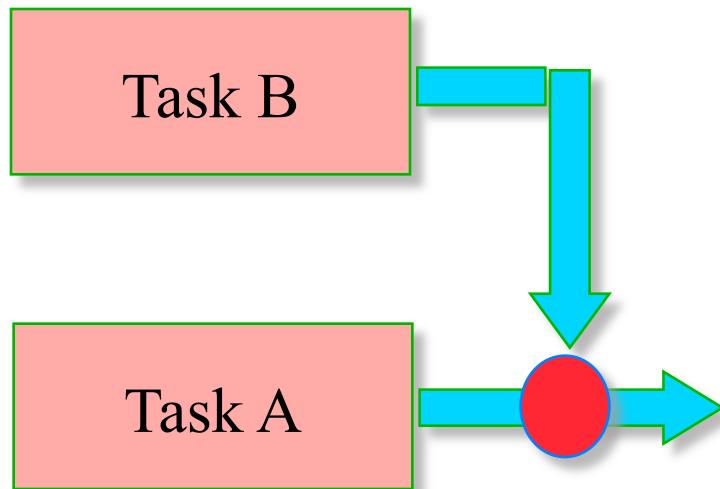
Comment organiser un ordre dans toutes les règles?

◆ Réponse: utiliser une architecture qui hiérarchise les règles

- Règles réflexes ou de survie
- Règles liées à la fonction de l'agent
- Règles par défaut

Architectures d'agents réactifs #2

◆ Subsumption architecture (R. Brooks)



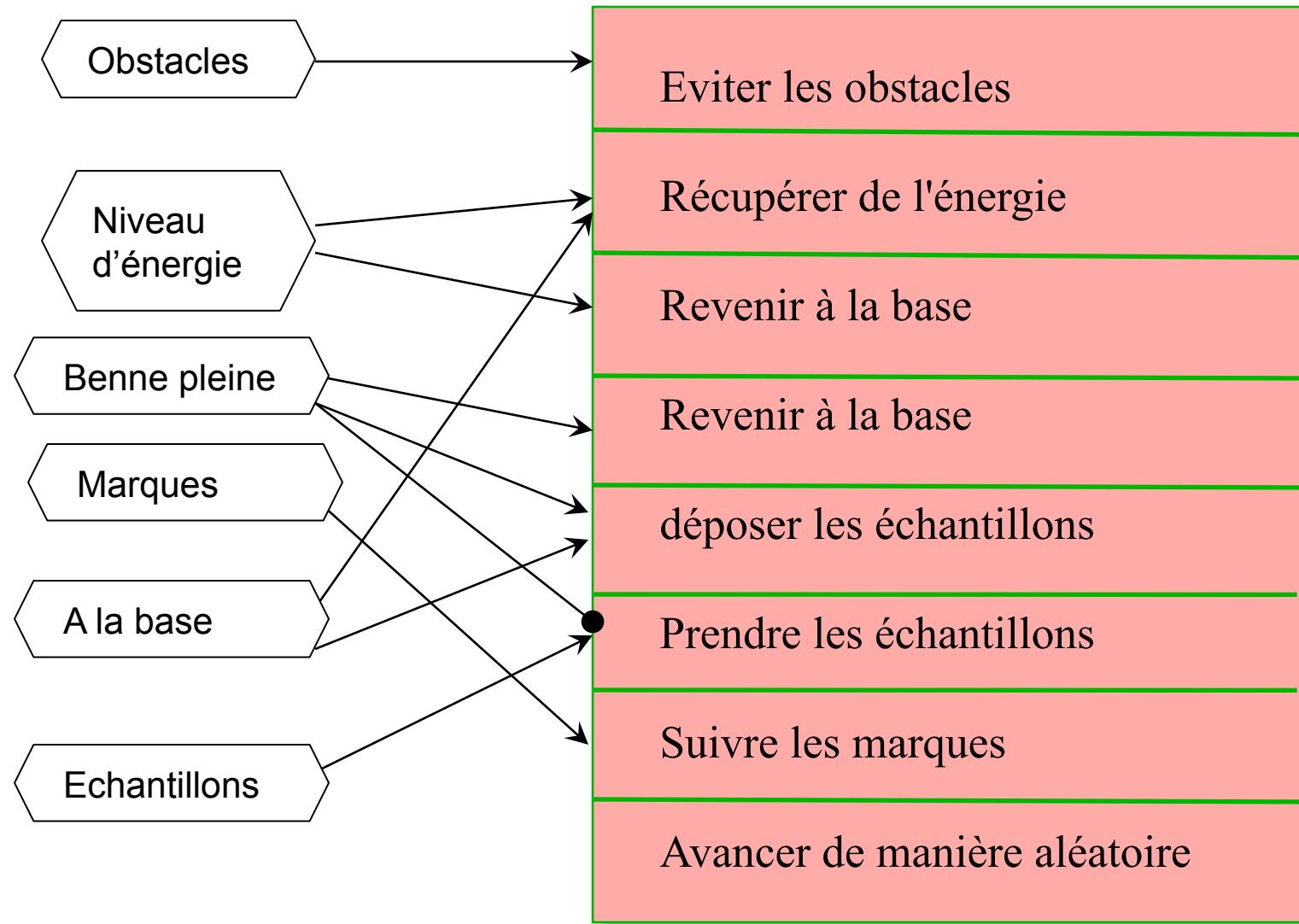
Eviter les dangers

Maintenir un niveau
d'énergie

Accomplir une tâche
productive

Avancer de manière
aléatoire (comportement
par défaut)

Exemple: les robots fourrageurs



Implémenter architecture de subsomption

Récupérer toutes les infos (percepts)

```
si obstacle() alors éviterObstacle(); break
si niveauEnergie < seuil alors revenirBase(); break
si niveau Energie < seuil et alabase()
    alors récupérerEnergie; break
si bennePleine et alaBase() alors viderBenne(); break
si bennePleine alors revenirBase(); break
si presenceEchantillon() alors remplirBenne(); break
si presenceMarques() alors suivreMarques(); break
avancerAleatoire()
```

On récupère les informations, et on hiérarchise les décisions

Problème

◆ Que faire quand de nombreuses choses doivent être faites sans hiérarchie a priori

◆ Exemple pour un animal :

- Trouver de la nourriture et de l'eau
- Echapper aux prédateurs et aux événements
- Maintenir un niveau de chaleur
- Dormir
- Se reproduire

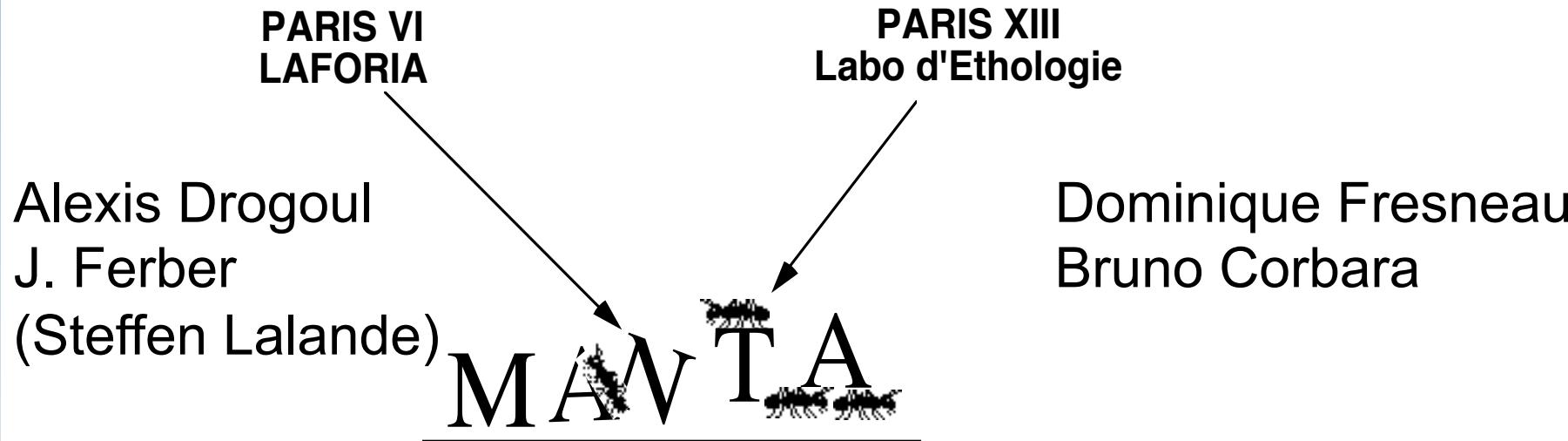
◆ Exemple pour un personnage dans un jeu vidéo:

- Aller chercher des ressources
- Attaquer la base ennemie
- Défendre
- Construire
- Explorer l'espace

Tâches en compétition

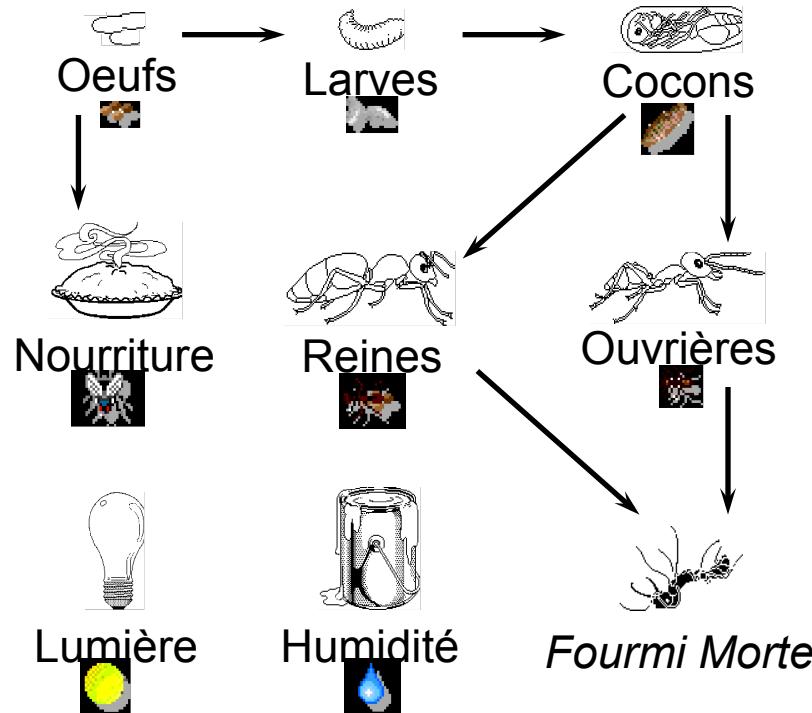
- ◆ Permet de prendre en compte plusieurs activités possibles
 - Création d'une notion de rôle: plus on fait quelque chose plus on a tendance à faire la même chose...

Un exemple: le projet Manta



**Modélisation
comportementale d'une
société de fourmis
Ectatomma ruidum et
l'étude de l'émergence de
structures sociales au sein
d'une colonie**

Les agents Manta

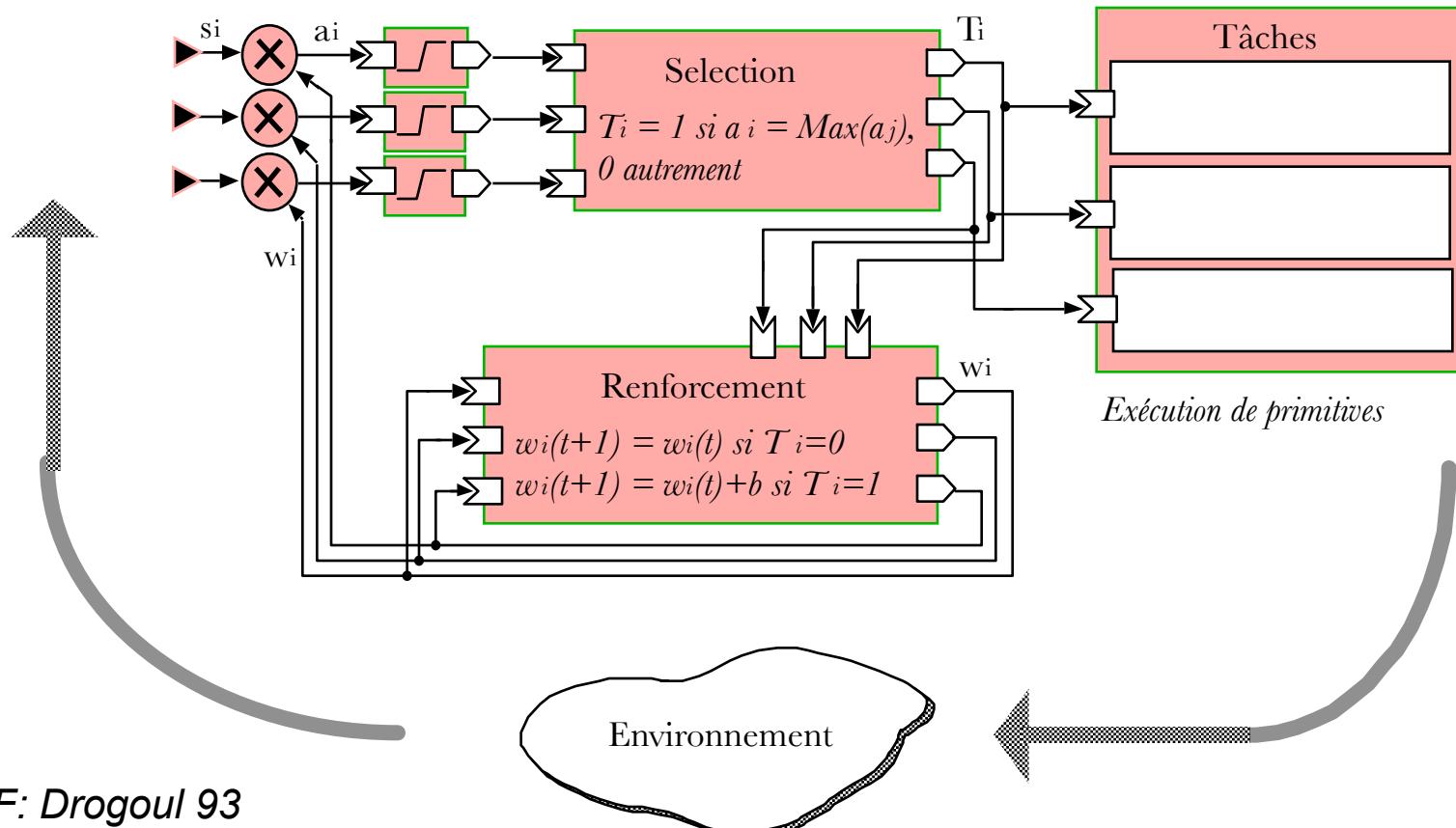


- ◆ Les agents Manta possèdent une architecture de type «tâches en compétition»

Architectures d'agents réactifs #4

◆ Architectures à base de tâches en compétition (cf. projet Manta)

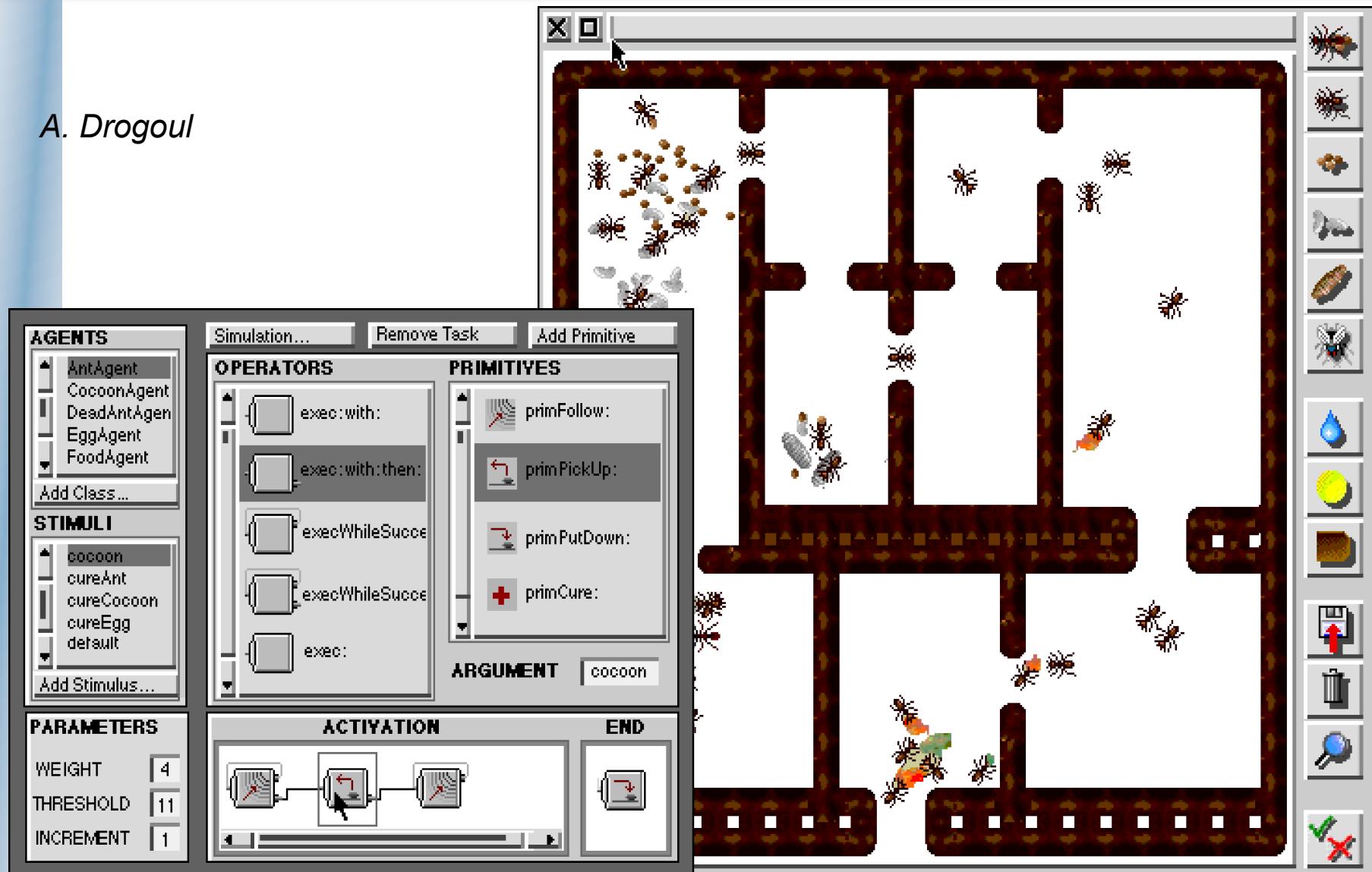
Stimuli Seuils



EMF: Drogoul 93

Manta: le laboratoire virtuel

A. Drogoul



Manta: expériences



Dynamique Démographique



- ◆ 300 sociétés artificielles de fourmis (depuis leur fondation jusqu'à l'âge adulte)



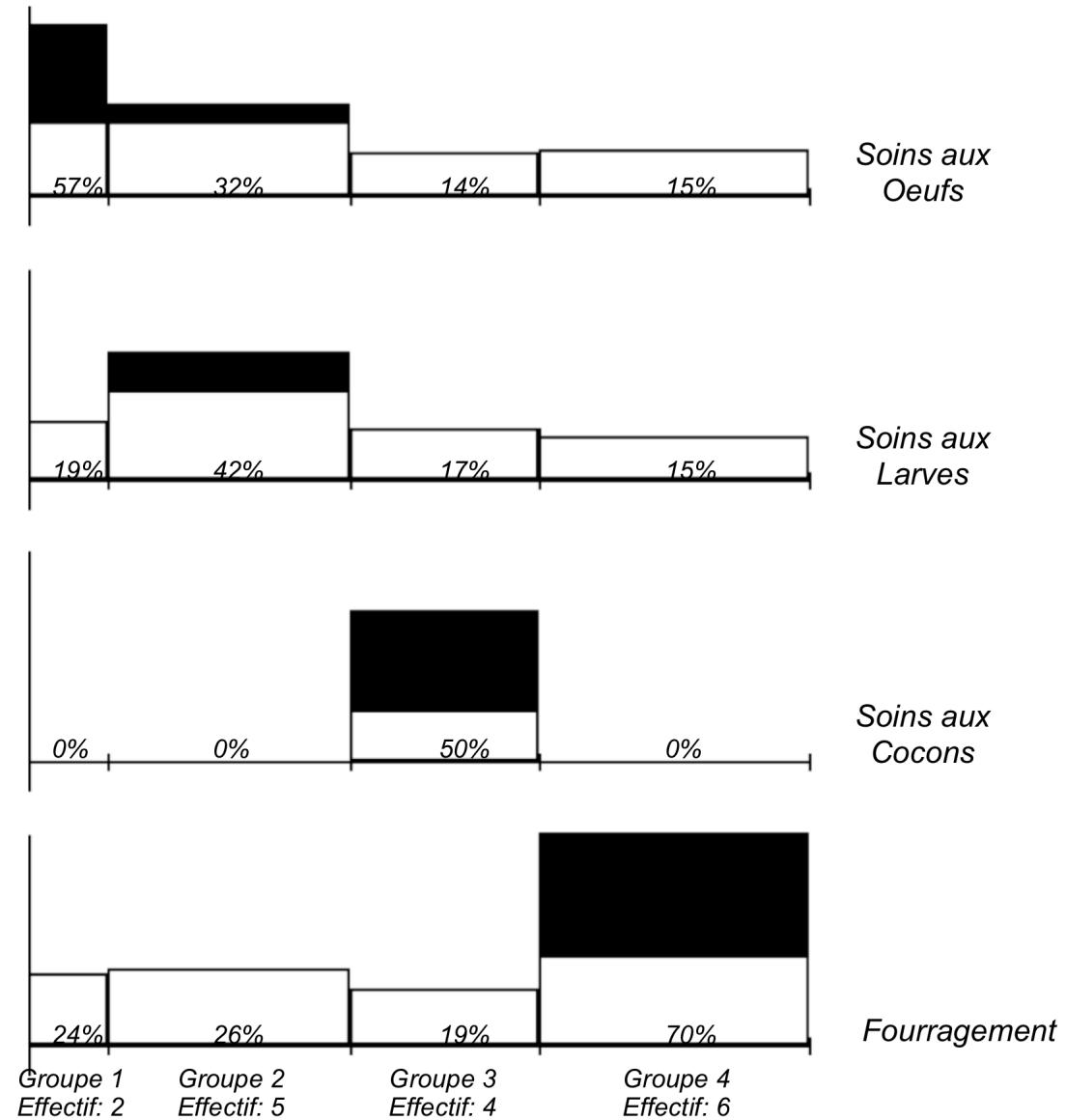
Organisation Sociale

Evolution de l'Organisation

Evolution avec Restriction de la Nourriture

Manta: division du travail

Répartition en groupes fonctionnels
(exp. P_5, 17 individus)



Emergence et adaptation collective

◆ L'adaptation sociale des fourmis: la sociotomie

- Que se passe-t-il si l'on divise une société en deux..
Est ce que chaque moitié peut redonner une société?

◆ Survie et apparitions de formes collectives

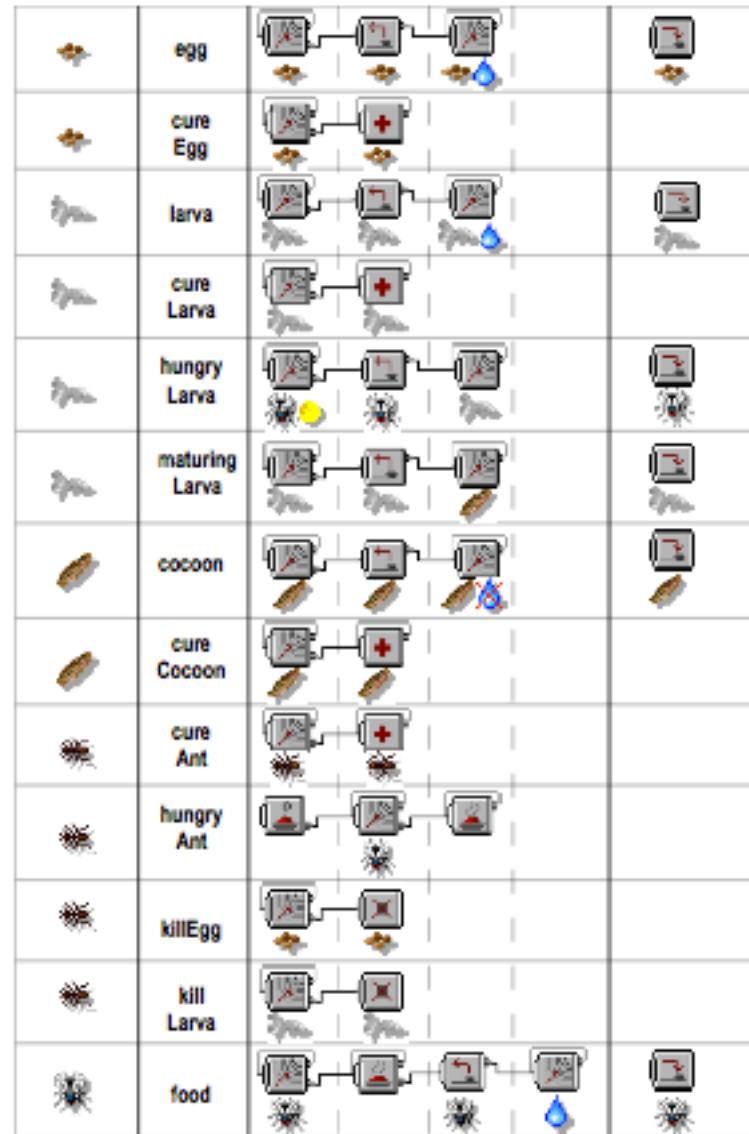
Actions

◆ Primitives

- Aller vers <stimulus>, Fuir <stimulus>
- Prendre <objet>, déposer
- Procurer des soins <agent>
- Manger

◆ Tâches=combinaison de primitives

- Ex: soigner cocons =
 - suivre ‘demande-soin-cocon’
 - soigner cocon
- Ex: donner à manger aux larves =
 - suivre ‘nourriture’
 - prendre #nourriture
 - suivre ‘faim-larve’
 - si #larve déposer



L'importance du système de renforcement

- ◆ **Tout fonctionne à partir de renforcements**

$\text{niveau}[i] = \text{poids}[i] * \text{stimulus}[i]$

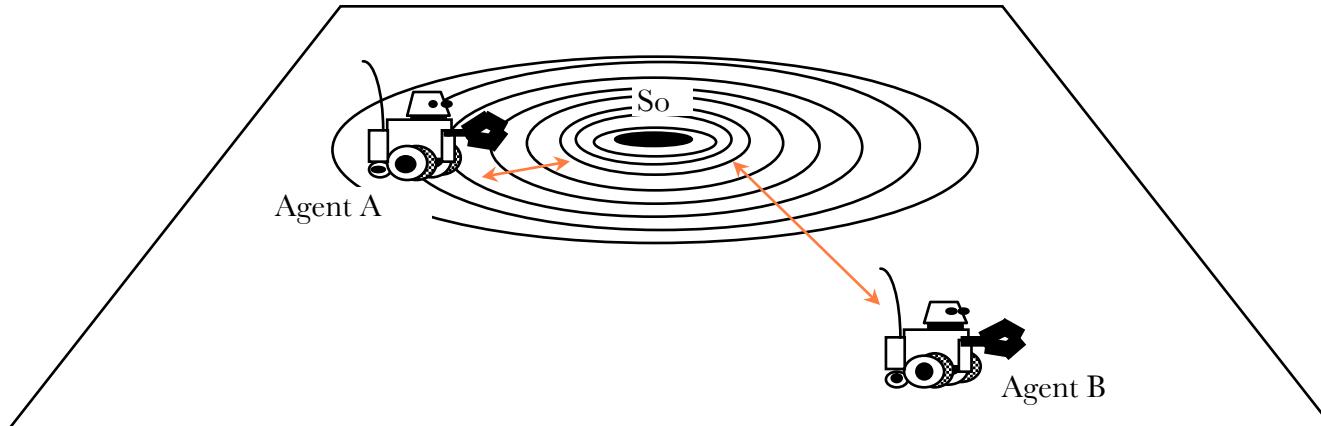
si $\text{niveau}[i] < \text{seuil-min}$ then -1 (ou 0 = valeur qui indique une non prise en compte de la tâche)

- ◆ **Sélectionne la tâche i dont le niveau est le plus élevé**

Tache-courante = tache[i] tq $\text{niveau}[i] = \max(\text{niveau}[j], \text{pour tout } j)$

- ◆ **On peut y ajouter une fonction de « lassitude » pour éviter les « addictions »**

Distance et signaux



L'émetteur envoie un signal qui se propage dans l'environnement. L'intensité du signal décroît en fonction de la distance (et du temps):

$$V(x) = V(x_0)/\text{dist}(x, x_0)$$

Implémenter EMF en NetLogo

```
<breed>-own [ctask triggers weights tasks]

to go-wolf
    set triggers compute-trigger 1 []
    decide ;; and reinforce
end

to-report get-pheromone [i]
    if i > 1 [report 0]
    ifelse i = 0
        [report <smell1> ;; fonction ad-hoc malheureusement...
        [report <smell2>]
end

;; compute the list of thresholdss
;; trigger = weight * stimulus
;; if trigger below a specific threshold then we put -1 in the list..
to-report compute-trigger [i lst]
    ifelse (i < 0)
        [report lst]
        [let trigger get-pheromone i * item i weights ;; compute trigger
        ifelse trigger < min-threshold
            [report compute-trigger (i - 1) fput -1 lst]
            [report compute-trigger (i - 1) fput trigger lst]]
end
```

EMF #2

```
;; returns the index where the value is maximal in the list lst..
;; Uses terminal recursion
to-report select-max [lst i val index]
  if (empty? lst) [report index]
  ifelse (first lst > val)
    [report select-max (bf lst) (i + 1) (first lst) i]
    [report select-max bf lst (i + 1) val index]
end

;; decision function
to decide
  let index select-max triggers 0 min-threshold -1
  ifelse (index >= 0)
    [reinforce index
      run item index tasks]
    [set color red wiggle ]
end

;; reinforcement function of the ith weight
to reinforce [i]
  ;; a definir...
end
```

Architectures réactives

◆ FSM:

- permet de bien décrire des comportements lié à des activités bien réglées..
- Pb: trop d'engagement.. Nécessite des techniques particulières pour gérer les réflexes et besoins de survie.

◆ Actions situées et Subsomption

- Permet de décrire des comportements fluides liés aux perceptions
- Pb: pas assez d'engagement... Les agents perdent ce qu'ils sont en train de faire...

◆ Compétition entre tâches

- Intégrer les FSM (les tâches) et les perceptions (déclenchement sur les actions).