

# Planification

ASP  
*Calcul des situations*  
*Planification non linéaire*

## IAMSI

Intelligence Artificielle et  
Manipulation Symbolique de l'information

Cours 8



# Programmation planification en ASP

- **Représentation**
  - *objets*
  - *états*
- **Buts**
- **Mouvements**
  - *effets*
  - *inertie*
  - *contraintes*
- **Génération**



# Programmation planification en ASP

- **Représentation**
  - *objets*
  - *états*
- **Buts**
- **Mouvements**
  - *effets*
  - *inertie*
  - *contraintes*
- **Génération**



# Programmation en ASP

## Représentation des états

- Les blocs et les instants sont représentés avec des nombres:

block(1..6).

time(0..5).

- Représenter les états avec des prédictats:

– Etat initial: 0

on(1, 2, 0).

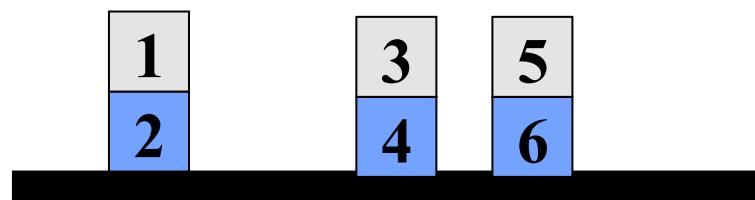
on(2, table, 0).

on(3, 4, 0).

on(4, table, 0).

on(5, 6, 0).

on(6, table, 0).



TABLE



# Programmation en ASP

## Représentation des objets (*block, poignets*)

- Les lieux sont représentés par des nombres. Il y en a au moins autant que de blocs... + la table

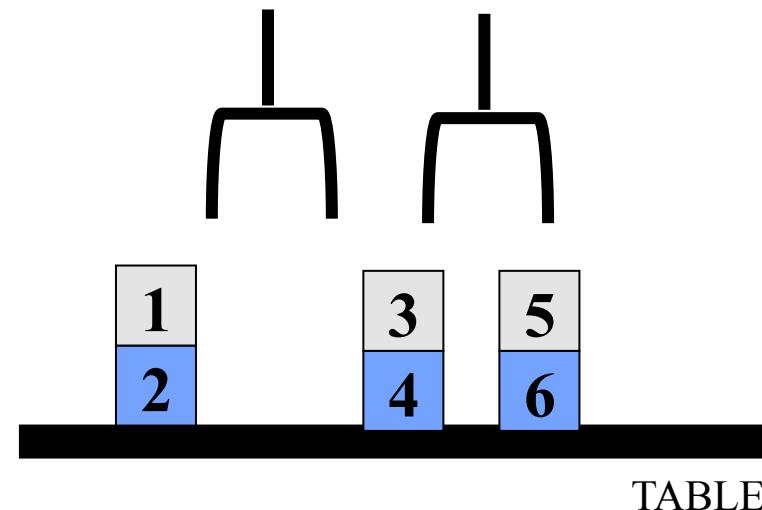
```
block(1..6).
```

```
time(0..5).
```

```
location(B) ←  
block(B).
```

```
location(table)
```

- Nombre de poignets: variable...



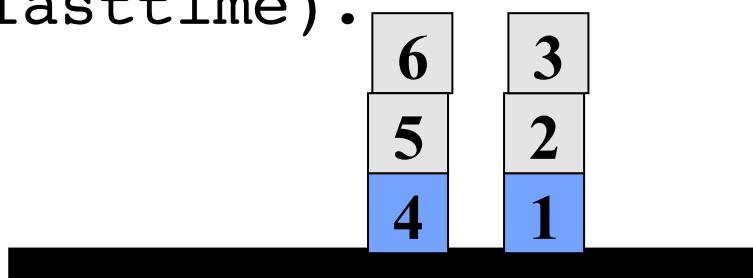
# Programmation planification en ASP

- **Représentation**
  - *objets*
  - *états*
- **Buts**
- **Mouvements**
  - *effets*
  - *inertie*
  - *contraintes*
- **Génération**



# But

```
:‐ not on(3, 2, lasttime).  
:‐ not on(2, 1, lasttime).  
:‐ not on(1, table, lasttime).  
:‐ not on(6, 5, lasttime).  
:‐ not on(5, 4, lasttime).  
:‐ not on(4, table, lasttime).
```



TABLE



# Programmation planification en ASP

- **Représentation**
  - *objets*
  - *états*
- **Buts**
- **Mouvements**
  - *effets*
  - *inertie*
  - *contraintes*
- **Génération**



# Mouvements

- **Effet du mouvement d'un bloc**

```
on(B, L, T+1) :- move(B, L, T), T < lasttime.
```

- **Inertie**

```
on(B, L, T+1) :-  
    on(B, L, T),  
    not -on(B, L, T+1),  
    T < lasttime.
```

- **Position unique de chaque bloc**

```
-on(B, L1, T+1) :- on(B, L, T+1),  
                  neq(L, L1).
```



# Contraintes

- **Deux blocs ne peuvent pas se trouver au-dessus d'un même bloc**  
:- 2 {on(BB, B, T) : block(BB)}.
- **Un bloc ne peut être mu si quelque chose se trouve au-dessus**  
:- move(B, L, T), on(B1, B, T).
- **Un bloc ne peut pas être mu sur un bloc qui est en mouvement**  
:- move(B, B1, T), move(B1, L, T).



# Programmation planification en ASP

- **Représentation**
  - *objets*
  - *états*
- **Buts**
- **Mouvements**
  - *effets*
  - *inertie*
  - *contraintes*
- **Génération**



# Génération avec 1 poignet

```
{move(BB, LL, T):block(BB):location(LL)}1 :-  
    T < lasttime.
```



# Programmation en ASP

- Les blocs et les instants sont représentés avec des nombres:

```
block(1..6).
```

```
time(0..5).
```

- Représenter les états avec des prédictats:

- Etat initial: 0

```
on(1, 2, 0).
```

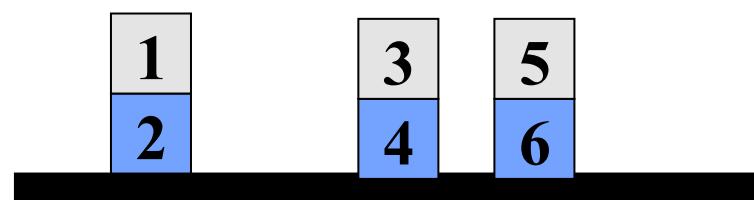
```
on(2, table, 0).
```

```
on(3, 4, 0).
```

```
on(4, table, 0).
```

```
on(5, 6, 0).
```

```
on(6, table, 0).
```

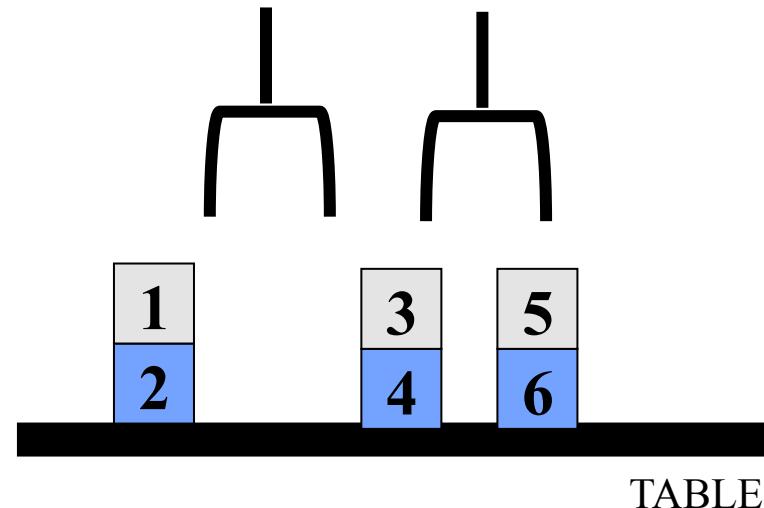


TABLE



# Programmation en ASP (*suite*)

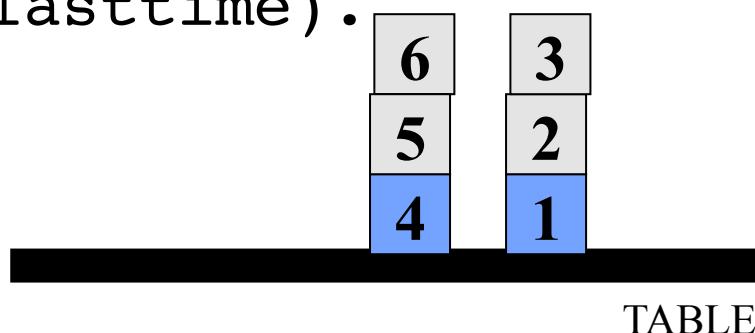
- Les lieux sont représentés par des nombres. Il y en a au moins autant que de blocs... + la table  
`block(1..6).`  
`time(0..5).`  
`location(B) ← block(B).`  
`location(table)`
- Nombre de poignets: variable...



# But

```
:‐ not on(3, 2, lasttime).  
:‐ not on(2, 1, lasttime).  
:‐ not on(1, table, lasttime).  
:‐ not on(6, 5, lasttime).  
:‐ not on(5, 4, lasttime).  
:‐ not on(4, table, lasttime).
```

Fin cours 6



TABLE



# Génération avec n poignets

## AnsProlog\* - 2 poignets

```
{move(BB, LL, T) : block(BB) :- location(LL)}2 :-  
T < lasttime.
```

## Clingo - 2 poignets

```
{move(BB, LL, T) : block(BB), location(LL)}=2 :-  
T < lasttime.
```



# Mouvements – AnsProlog\*

- **Effet du mouvement d'un bloc**

```
on(B, L, T+1) :- move(B, L, T), T < lasttime.
```

- **Inertie**

```
on(B, L, T+1) :-  
    on(B, L, T),  
    not -on(B, L, T+1),  
    T < lasttime.
```

- **Position unique de chaque bloc**

```
-on(B, L1, T+1) :- on(B, L, T+1),  
                  neq(L, L1).
```



# Mouvements – Clingo

- **Effet du mouvement d'un bloc**

```
on(B, L, T+1) :- move(B, L, T),  
block(B), time(T), location(L), T < lasttime.
```

- **Inertie**

```
on(B, L, T+1) :- block(B), time(T),  
location(L),  
on(B, L, T),  
not -on(B, L, T+1),  
T < lasttime.
```

- **Position unique de chaque bloc**

```
-on(B, L1, T+1) :- block(B), time(T),  
location(L), location(L1),  
on(B, L, T+1), L != L1.
```



# Contraintes – AnsProlog\*

- **Deux blocs ne peuvent pas se trouver au-dessus d'un même bloc**

```
:– 2 {on(BB, B, T) : block(BB)}.
```

- **Un bloc ne peut être mu si quelque chose se trouve au-dessus**

```
:– move(B, L, T), on(B1, B, T).
```

- **Un bloc ne peut pas être mu sur un bloc qui est en mouvement**

```
:– move(B, B1, T), move(B1, L, T).
```



# Contraintes - Clingo

- Deux blocs ne peuvent pas se trouver au-dessus d'un même bloc

```
:‐ 2 {on(BB, B, T) : block(BB)},  
      block(B), time(T).
```

- Un bloc ne peut être mu si quelque chose se trouve au-dessus

```
:‐ move(B, L, T), on(B1, B, T),  
      block(B), block(B1), location(L), time(T).
```

- Un bloc ne peut pas être mu sur un bloc qui est en mouvement

```
:‐ move(B, B1, T), move(B1, L, T),  
      block(B), block(B1), location(L), time(T).
```



# Anomalies de Susmann avec ASP

- Etat initial

```
on(3, 1, 0).
```

```
on(1, table, 0).
```

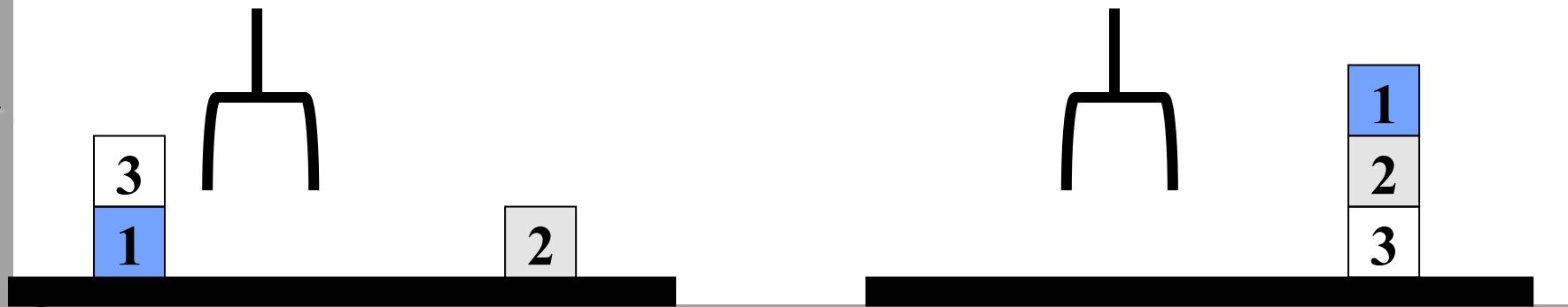
```
on(2, table, 0).
```

- But

```
: - not on(2, 3, lasttime).
```

```
: - not on(1, 2, lasttime).
```

```
: - not on(3, table,  
lasttime).
```

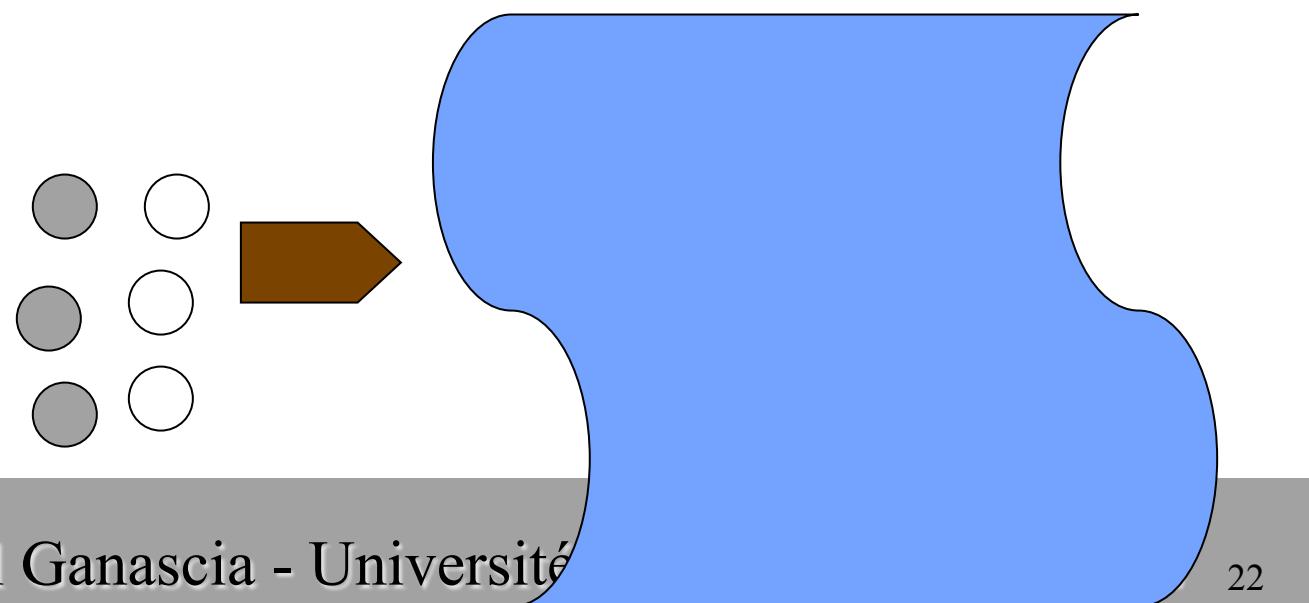


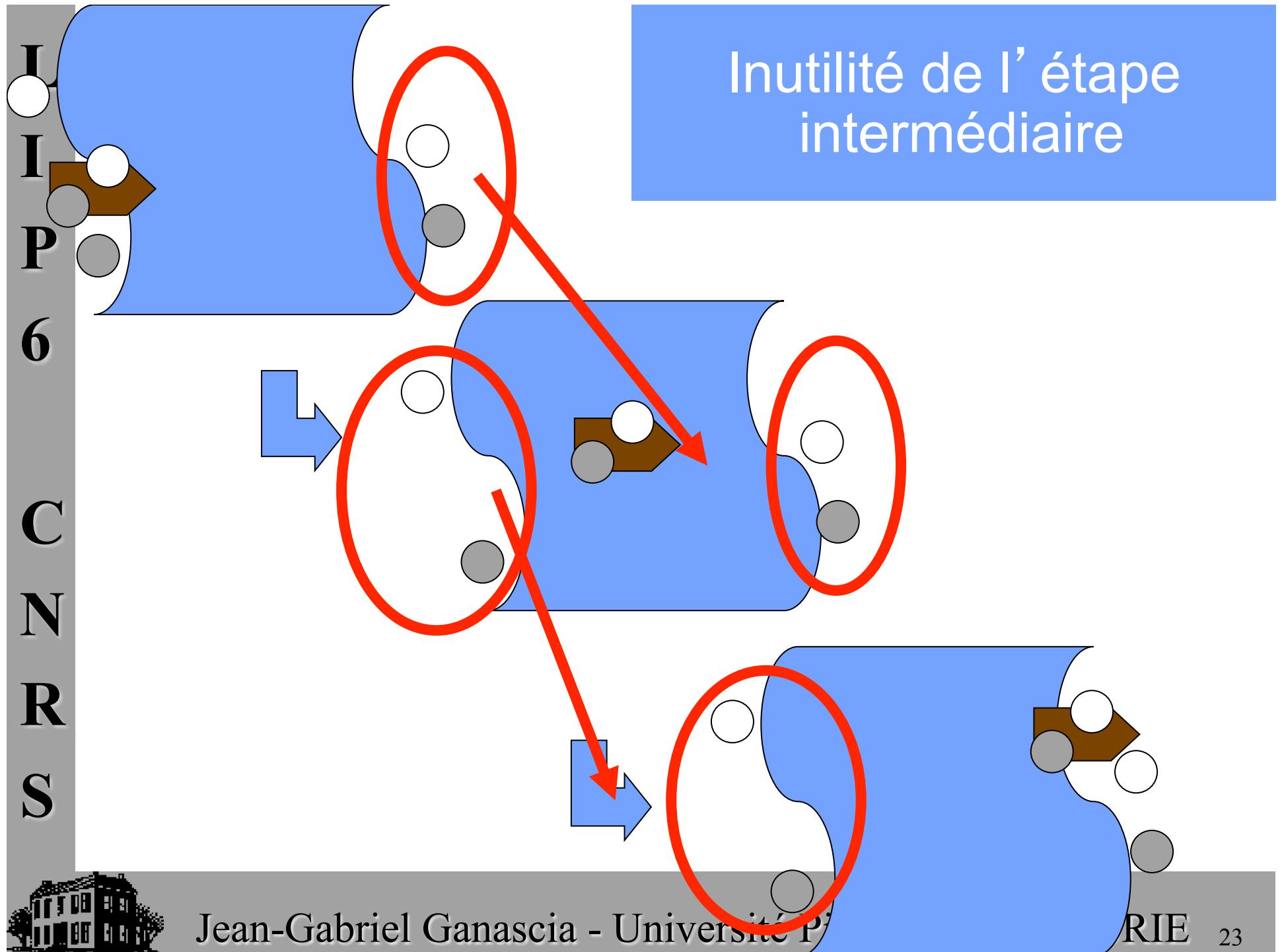
# Exemple de problème

« Les missionnaires et les cannibales »

*trois missionnaires et trois cannibales se trouvent sur les rives d'un fleuve en pleine Amazonie. La barque ne supporte pas plus de deux passagers, et le nombre de missionnaires doit partout et toujours être supérieur ou au moins égal à celui des cannibales pour éviter les drames... Comment faire pour que tous traversent, sans perte d'aucune sorte ?*

N  
R  
S





# Symétrie du problème

- **Représentation des états:** Il suffit de considérer le nombre de missionnaires et de cannibales sur la rive gauche, puis de préciser la position de la barque  
*(missionnaires Rive-gauche,  
cannibales Rive-Gauche,  
position barque)*
- *missionnaires Rive-gauche +  
missionnaires Rive droite = 3*
- **Etat initial:** *(3, 3, Gauche)*
- **Etat final:** *(0, 0, Droite)*
- **Nombre d' états:**  $4 \times 4 \times 2 = 32$



# Programmation planification en ASP

- **Représentation**
  - *objets*
  - *états*
- **Buts**
- **Mouvements**
  - *effets*
  - *inertie*
  - *contraintes*
- **Génération**



# Programmation planification en ASP

- **Représentation**
  - *objets*
  - *états*
- **Buts**
- **Mouvements**
  - *effets*
  - *inertie*
  - *contraintes*
- **Génération**



# Programmation en ASP

## Représentation des états

- Etat courant:

```
etat_courant(NCG, NMG, n°, PBarque) .
```

- Etat initial:

```
etat_courant(3, 3, 0, gauche) .
```

```
ajout(position(cannibale, 3), 0),
```

```
ajout(position(missionnaire, 3), 0),
```

```
ajout(position(barque, gauche), 0)) .
```



# Programmation planification en ASP

- **Représentation**
  - *objets*
  - *états*
- **Buts**
- **Mouvements**
  - *effets*
  - *inertie*
  - *contraintes*
- **Génération**



# Programmation en ASP

## Représentation des buts

- Buts:

```
but(T) :-
```

```
etat_courant(0, 0, T, droite).
```

```
but_atteint :- but(T).
```

```
:- not but_atteint.
```

% **contrainte sur le but**



# Programmation planification en ASP

- **Représentation**
  - *objets*
  - *états*
- **Buts**
- **Mouvements**
  - *effets*
  - *inertie*
  - *contraintes*
- **Génération**



# Missionnaires et cannibales en ASP

- Effet de l'événement aller (A, N) sur l'état X1 suivant l'état T:

```
ajout(position(A, N1-N), X1) :- N1 >= N,  
suivant(T, X1), evenement(aller(A, N), T),  
ajout(position(A, N1), T).
```

```
retrait(position(A, N1), X1) :- N1 >= N,  
suivant(T, X1), evenement(aller(A, N), T),  
ajout(position(A, N1), T).
```

```
ajout(position(barque, droite), X1) :-  
suivant(T, X1), evenement(aller(A, N), T),  
ajout(position(barque, gauche), T).
```

```
retrait(position(barque, gauche), X1) :-  
suivant(T, X1), evenement(aller(A, N), T),  
ajout(position(barque, gauche), T).
```



# I

## Missionnaires et cannibales en ASP

- Effet de l'événement retour (A, N) sur l'état X1 suivant l'état T:

```
ajout(position(A, N1+N), X1) :- N1+N<=NbreMax,  
suivant(T, X1), evenement(retour(A, N), T),  
ajout(position(A, N1), T).
```

```
retrait(position(A, N1), X1) :- N1+N<=NbreMax,  
suivant(T, X1), evenement(retour(A, N), T),  
ajout(position(A, N1), T).
```

```
ajout(position(barque, gauche), X1) :-  
suivant(T, X1), evenement(retour(A, N), T),  
ajout(position(barque, droite), T).
```

```
retrait(position(barque, droite), X1) :-  
suivant(T, X1), evenement(retour(A, N), T),  
ajout(position(barque, droite), T).
```



# Inertie

- **Inertie:** si un prédicat n'est pas affecté, il reste identique...

```
ajout(position(A,N) ,X1) :-  
    suivant(T,X1) ,  
    ajout(position(A,N) ,T) ,  
    not retrait(position(A,N) ,X1) .  
  
ajout(position(barque,Pb) ,X1) :-  
    suivant(T,X1) ,  
    ajout(position(barque,Pb) ,T) ,  
    not  
    retrait(position(barque,Pb) ,X1) .
```



# Cohérence

```
: - ajout(position(barque, gauche), T),  
  ajout(position(barque, droite), T).
```

```
: - neq(N, N1),  
  ajout(position(A, N), T),  
  ajout(position(A, N1), T).
```



# Contraintes 1

- Impossible de faire traverser à plus d'agents qu'il n'y en a sur une rive

```
: - evenement(aller(A,N),T),  
  ajout(position(A,N1),T), N > N1. % on ne peut  
  augmenter le nombre d'individus à gauche à  
  l'aller  
: - evenement(retour(A,N),T),  
  ajout(position(A,N1),T), N+N1 > NbreMax. % on  
  ne peut ramener plus de monde qu'il n'y en a  
  
: - evenement(aller(A, N), T),  
  evenement(aller(A, N1), T), neq(N, N1).  
: - evenement(retour(A, N), T),  
  evenement(retour(A, N1), T), neq(N, N1).
```



# Contraintes 2

- Le nombre de cannibales ne peut pas surpasser le nombre de missionnaires

```
: - ajout(position(cannibale,N),T),  
    ajout(position(missionnaire,N1),T),  
    N > N1,  
    N1 > 0.
```

```
: - ajout(position(cannibale,N),T),  
    ajout(position(missionnaire,N1),T),  
    N1 < NbreMax,  
    N < N1.
```



# Contraintes 3

- Les agents ne peuvent traverser que s'ils sont du même côté que le bateau

```
: - not ajout(position(barque, gauche), T),  
evenement(aller(A,N), T) .
```

```
: - not ajout(position(barque, droite), T),  
evenement(retour(A,N), T) .
```



# Contraintes 4

- Il ne doit pas y avoir plus de deux personnes dans la barque (et moins de 0)

```
: - evenement(aller(cannibale,N),T),  
  evenement(aller(missionnaire,N1),T), N+N1 > 2.  
  
: - evenement(retour(cannibale,N),T),  
  evenement(retour(missionnaire,N1),T), N+N1 > 2.  
  
: - evenement(aller(cannibale,N),T),  
  evenement(aller(missionnaire,N1),T), N+N1 == 0.  
  
: - evenement(retour(cannibale,N),T),  
  evenement(retour(missionnaire,N1),T), N+N1 == 0.  
  
  
: - evenement(aller(cannibale,N),T), N > 2.  
: - evenement(aller(missionnaire,N),T), N > 2.  
: - evenement(retour(cannibale,N),T), N > 2.  
: - evenement(retour(missionnaire,N),T), N > 2.
```



# Programmation planification en ASP

- **Représentation**
  - *objets*
  - *états*
- **Buts**
- **Mouvements**
  - *effets*
  - *inertie*
  - *contraintes*
- **Génération**



# Génération

```
{evenement(aller(A, X), T) : passagers_barque(X) }1  
:- not but(T).
```

```
{evenement(retour(A, X), T) : passagers_barque(X) }1  
:- not but(T).
```



# Langages d'actions

## *généralisation STRIPS*

- Système de transition:  $\langle V, F, A \rangle$   
 $F$ : fluents – prédictats variables dans le temps  
 $V$ : valeurs que les fluents prennent dans les états  
 $A$ : actions
  - i. un ensemble  $S$  (états)
  - ii. une fonction  $V$  de  $F \times S$  dans  $V$
  - iii. un sous-ensemble  $R$  de  $S \times A \times S$   
 $\langle s, A, s' \rangle \in R$  ssi  
     $s'$  est le résultat de l'action  $A$  sur  $s$
- Système de transition propositionnel:  $\langle V, F, A \rangle$   
ssi  $V = \{t, f\}$



# Langage $\mathcal{A}$

- On considère un système de transition propositionnel  $\langle \{t, f\}, F, A \rangle$
- Définition: dans le langage  $\mathcal{A}$ 
  - i. Une *proposition* est une expression de la forme:  **$A$  causes  $L$  if  $F$**  où  $L$  est un littéral et  $F$  une conjonction de littéraux éventuellement vide
  - i. Une *description d'action* est un ensemble de propositions



# Langage $\mathcal{A}$

- **Définition:** soit  $D$  la **description d'une action** dans  $\mathcal{A}$ .  
Le **système de transition**  $\langle S, V, R \rangle$  est défini comme suit:
  - $S$  est l'ensemble des interprétations de  $F$
  - $V(P, s) = s(P)$
  - $R$  est l'ensemble des triplets  $\langle s, A, s' \rangle$  tels que  $E(A, s) \subseteq s' \subseteq E(A, s) \cup s$   
où  $E(A, s)$  liste les effets de l'action  $A$  sur l'état  $s$



# Langage $\mathcal{L}$

- On considère un système de transition propositionnel  $\langle \{t, f\}, F, \{t, f\}^E \rangle$  où  $E$  est un ensemble de noms d'actions
- **Définition: dans le langage  $\mathcal{L}$** 
  - i. Une *formule d'état* est une composition de noms de fluents
  - ii. Une *formule* est une composition de noms de fluents et de noms élémentaires d'actions
  - iii. une *loi statique* est une expression de la forme:  
**caused**  $F$  if  $G$  où  $F$  et  $G$  sont des formules d'états
  - iv. une *loi dynamique* est une expression de la forme:  
**caused**  $F$  if  $G$  after  $U$  où  $F$  et  $G$  sont des formules d'états et  $U$  une formule



# Langage $\mathcal{C}$ (suite)

- Définition: dans le langage  $\mathcal{C}$ 
  - i. Une expression de la forme  **$U$  causes  $F$  if  $G$**  correspond à **caused  $F$  if true after  $GU$**
  - ii. Un expression de la forme **inertial  $F$**  où  $F$  est une formule d'états correspond à **caused  $F$  if  $F$  after  $F$**
  - iii. Un expression de la forme **always  $F$**  où  $F$  est une formule d'états correspond à **caused false if  $\neg F$**



# Utilité des langages d'actions

## Les règles de mouvements des blocs:

- **Génération**

```
{move(BB, LL, T):block(BB):location(LL)}1 :-  
    T < lasttime.
```

- **Effet du mouvement d'un bloc**

```
on(B, L, T+1) :- move(B, L, T), T < lasttime.
```

- **Inertie**

```
on(B, L, T+1) :- on(B, L, T),  
    not -on(B, L, T+1), T < lasttime.
```

- **Position unique de chaque bloc**

```
-on(B, L1, T+1) :- on(B, L, T+1), neq(L, L1).
```

## Sont remplacées dans le langage C par les deux règles:

- **move(b, l) causes on(b, l)**

- **inertial on(b, l)**



# Planification: Approches principales

- GPS
- STRIPS
- « Situation calculus »
- Planification partiellement ordonnée
- Décomposition hiérarchique
- Planification avec contraintes
- Planification réactive



# Représentation Opérateur/action STRIPS – logique

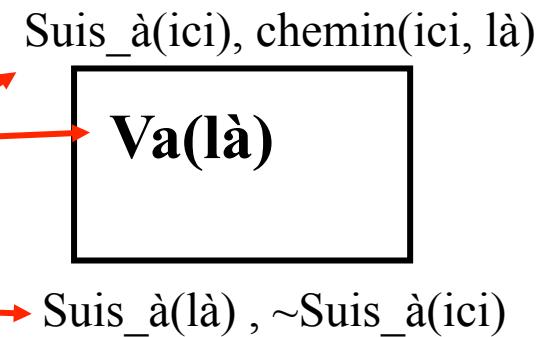
- **Les opérateurs ont trois composants:**
  - Action
  - Précondition – conjonction de littéraux positifs
  - Effet - conjonction de littéraux positifs ou négatifs qui décrivent comment la situation change quand l' opérateur est appliqué

- **Exemple:**

Op[Action: **Va(là)**,

Précondition: **suis\_à(ici) ^ chemin(ici, là)**,

Effet: **Suis\_à(là) ^ ~Suis\_à(ici)]**



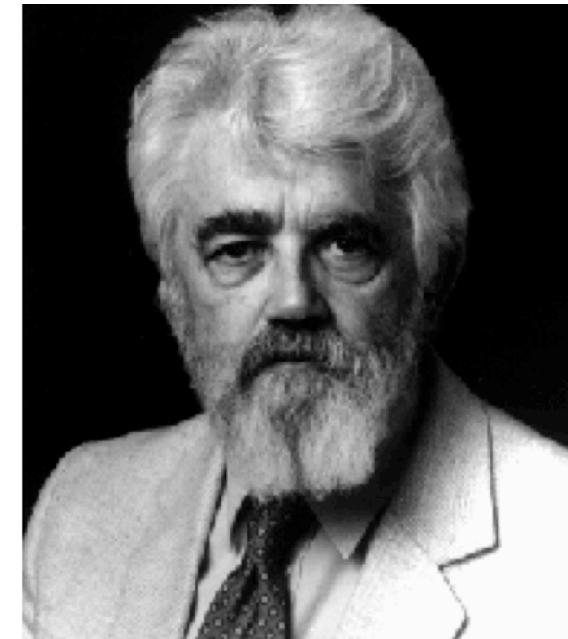
- **Toutes les variables sont universellement quantifiés**
- **Les variables de situation sont implicites**
  - La précondition doivent être vraie dans l' état antérieur à l' action; les effets dans l' état postérieur



# John McCarthy – 1927-2011

## Œuvre

- Mathématique & théorie du calcul
- Intelligence artificielle
- Langages (ALGOL, LISP, Elephant)
- Temps partagé
- Jeux – jeu d'échec
- Calcul des situations
- Logique non-monotones – circonscription
- Créativité
- Philosophie
- Curiosités: fouille de données, logique modale, ...
- Fantaisies



# Le calcul des situations

- **But : représenter les problèmes de planification en utilisant la logique des prédictats de premier ordre**
  - Le calcul des situations que nous permet raisonner au sujet des changements dans le monde
  - Grâce au calcul des situations, on peut utiliser la démonstration automatique de théorèmes pour « prouver » qu’ une séquence particulière d’ action, quand on l’ applique à une situation caractéristique d’ un état du monde, conduira bien à l’ état désiré.



# Calcul des situations

- **Deux notions essentielles:**

- **Les situations:** c'est l'ensemble des états accessibles par le système. En général, les situations ne sont pas données explicitement, mais seulement de façon partielle
- **Les fluents:** ce sont les prédictats ou les termes qui décrivent les évolutions du système
- **Langage**
  - **Holds (pfluent, s)** – pfluent est un prédictat fluent
  - **Value (tfluent, s)** – tfluent est un terme fluent
  - **Result (e, s)** – e est un événement
  - **Occurs (e, s)**
  - **Next (s)**



# L

# Calcul des situations

- **État initial:** une expression logique décrivant (la situation)  $S_0$

$$\text{At(Home, } S_0) \wedge \\ \neg \text{Have(Milk, } S_0) \wedge \neg \text{Have(Bananas, } S_0) \wedge \neg \text{Have(Drill, } S_0)$$

- **État but:**

$$(\exists s) \text{ At(Home, } s) \wedge \text{Have(Milk, } s) \wedge \text{Have(Bananas, } s) \wedge \text{Have(Drill, } s)$$

- **Les opérateurs décrivent la façon dont le monde change sous l' action d'un agent :**

$$\forall (a, s) \text{ Have(Milk, } \text{Result}(a, s)) \Leftrightarrow ((a=\text{Buy(Milk)} \wedge \text{At(Grocery, } s)) \vee \\ (\text{Have(Milk, } s) \wedge a=\text{Drop(Milk)}))$$

- **Result(a, s) nomme la situation résultant de l'exécution de l'action a sur la situation s.**
- **Les séquences d'action sont aussi utiles : Result' (l, s) est le résultat de l'exécution de la liste d'actions (l) en partant de l'état s:**

$$(\forall s) \text{ Result}'([], s) = s$$
$$(\forall a, p, s) \text{ Result}'([a | p], s) = \text{Result}'(p, \text{Result}(a, s))$$

# Calcul des situations -- suite

- **Une solution est un plan qui, quand il est appliqué à l'état initial, conduit à une situation vérifiant les buts :**  
 $\text{At}(\text{Home}, \text{Result}'(p, S_0))$   
   $\wedge \text{Have}(\text{Milk}, \text{Result}'(p, S_0))$   
   $\wedge \text{Have}(\text{Bananas}, \text{Result}'(p, S_0))$   
   $\wedge \text{Have}(\text{Drill}, \text{Result}'(p, S_0))$
- **Sur cet exemple, nous nous attendrions à un plan, c'est-à-dire à une affectation de variables par l'entremise de l'unification, telle que :**  
 $p = [\text{Go}(\text{Grocery}), \text{Buy}(\text{Milk}), \text{Buy}(\text{Bananas}),$   
 $\quad \text{Go}(\text{HardwareStore}), \text{Buy}(\text{Drill}), \text{Go}(\text{Home})]$



# Remarque sur pliage/depliage des programmes

- **Un programme qui contient des termes est dit « plié »**

```
member(X, [X|L]).
```

```
member(X, [_|L]) :- member(X, L).
```

- **On peut le « déplier » (ou l'« aplatisir ») en remplaçant les termes par des atomes**

**–Utilisation d'un prédicat  $co(L, T, Q)$  correspondant au terme  $L = [T|O]$**

```
member(X, L) :- co(L, X, _).
```

```
member(X, L) :- co(L, _, Q),  
               member(X, Q).
```



# Remarque sur pliage/depliage des états

- Un état s correspond à un terme « plié »
- On peut le « déplier » pour obtenir une liste

$s = S_0$  correspond à  $p = []$

$s = \text{Go}(\text{Grocery}, S_0)$  correspond à  $p = [\text{Go}(\text{Grocery})]$

$s = \text{Buy}(\text{Milk}, \text{Go}(\text{Grocery}, S_0))$  correspond à  
 $p = [\text{Go}(\text{Grocery}), \text{Buy}(\text{Milk})]$

$s = \text{Buy}(\text{Bananas}(\text{Buy}(\text{Milk}, \text{Go}(\text{Grocery}, S_0))))$  correspond à  
 $p = [\text{Go}(\text{Grocery}), \text{Buy}(\text{Milk}), \text{Buy}(\text{Bananas})]$

$s = \text{Go}(\text{HardwareStore}, (\text{Buy}(\text{Bananas}(\text{Buy}(\text{Milk}, \text{Go}(\text{Grocery}, S_0)))))$   
correspond à

$p = [[\text{Go}(\text{Grocery}), \text{Buy}(\text{Milk}), \text{Buy}(\text{Bananas}), \text{Go}(\text{HardwareStore})]]$

...

$s = \text{Go}(\text{Home}, \text{Buy}(\text{Drill}, \text{Go}(\text{HardwareStore}, \text{Buy}(\text{Bananas}, \text{Buy}(\text{Milk}, \text{Go}(\text{Grocery}, S_0))))))$  correspond à

$p = [\text{Go}(\text{Grocery}), \text{Buy}(\text{Milk}), \text{Buy}(\text{Bananas}), \text{Go}(\text{HardwareStore}), \text{Buy}(\text{Drill}), \text{Go}(\text{Home})]$



# Programme depliage

- **Programme depliage(S, L) en PROLOG**

*On suppose que les actions ont toutes un argument*

$S_{n+1} = Act(Arg, S_n)$

ex.  $S_1 = go(home, S_0)$ ,  $S_2 = buy(drill, S_1)$ , ...

```
depliage(S, L) :- depliage(S, [], L).
```

```
depliage(S, L, L) :- atomic(S).
```

```
depliage(S, Acc, L) :-  
    S =.. [Act, Arg, Suivant],
```

```
    A =.. [Act, Arg],
```

```
    depliage(Suivant, [A|Acc], L).
```



**Le monde des blocs est un micro monde qui consiste en une table, un ensemble de blocs et une poignée de robot.**

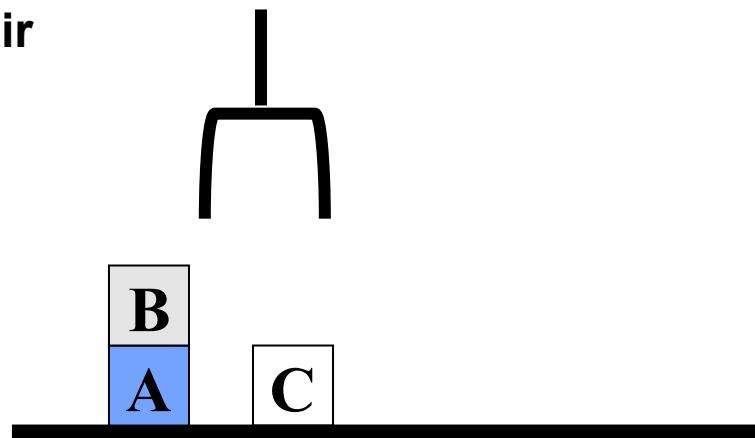
### Contraintes du domaine:

- Il ne peut pas y avoir plus d'un bloc sur un autre
- Il n'y a pas de limite au nombre de blocs présents sur la table
- La poignée du robot ne peut saisir qu'un seul bloc

### Représentation typique:

ontable(a)  
ontable(c)  
on(b,a)  
handempty  
clear(b)  
clear(c)

## Le monde des blocs - rappel



TABLE



# Calcul des situations dans le monde des blocs

- Prédicats:

*on(X, Y, S), ontable(X, S), clear(X, S),  
handempty(S), holding(X, S)*

- Pour caractériser l'action *unstack(X, Y)* on peut écrire:

$$\begin{aligned} [\text{on}(X, Y, S) \wedge \text{handempty}(S) \wedge \text{clear}(X, S)] \Rightarrow \\ [\text{holding}(X, \text{result}(\text{unstack}(X, Y), S)) \wedge \\ \text{clear}(Y, \text{result}(\text{unstack}(X, Y), S))] \end{aligned}$$

- Ceci crée un nouvel état qui correspond à la conquête et l'action de l'état *S* avec l'action *unstack(X, Y)*.



# Calcul des situations dans le monde des blocs

- Voici un autre exemple d'application du calcul des situations au monde des blocs:
  - $\text{Clear}(X, \text{Result}(A, S)) \leftrightarrow$   
[ $\text{Clear}(X, S) \wedge$   
 $(\neg(A = \text{Stack}(Y, X) \vee A = \text{Pickup}(X))$   
 $\vee (A = \text{Stack}(Y, X) \wedge \neg(\text{holding}(Y, S)))$   
 $\vee (A = \text{Pickup}(X) \wedge \neg(\text{handempty}(S) \wedge \text{ontable}(X, S) \wedge \text{clear}(X, S))))]$   
 $\vee [\text{A} = \text{Stack}(X, Y) \wedge \text{holding}(X, S) \wedge \text{clear}(Y, S)]$   
 $\vee [\text{A} = \text{Unstack}(Y, X) \wedge \text{on}(Y, X, S) \wedge \text{clear}(Y, S) \wedge \text{handempty}(S)]$   
 $\vee [\text{A} = \text{Putdown}(X) \wedge \text{holding}(X, S)]$
- Traduction en langage naturel: un bloc est libre si
  - (a) dans l'état précédent il était libre et si nous n'avons ni pris ce bloc ni empilés quoique ce soit dessus,
  - ou (b) nous avons empilé ce bloc sur quelque chose d'autre avec succès,
  - ou (c) il y avait quelque chose dessus que nous avons déplié avec succès,
  - ou (d) ce bloc était en prise et nous l'avons lâché.

# Preuve dans le calcul des situations (1)

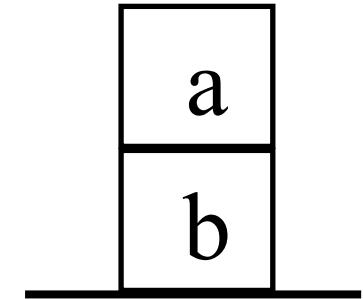
- Nous pouvons maintenant déduire que:

$on(a, b, s0) \wedge ontable(b, s0) \wedge clear(a, s0)$

alors

$holding(a, Result(unstack(a, b), s0)) \wedge$

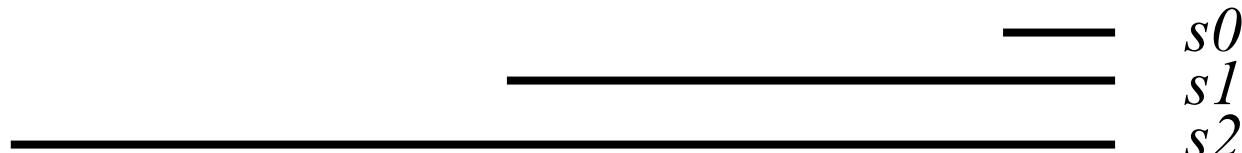
$clear(b, Result(unstack(a, b), s0))$



- Les déductions peuvent continuer. Pour  $putdown(X)$

$holding(X, S) \Rightarrow ontable(X, S) \wedge handempty(S) \wedge clear(X, S))$

$ontable(a, Result(putdown(a), Result(unstack(a, b), s0)))$



# Preuve dans le calcul des situations (2)

- Quand il est valide, l'état est instancié par un terme fonctionnel qui résume la liste des actions qui permettent d'y accéder à partir de l'état initial :

*on(a, b, s5)  $\wedge$  ....*

*s5 = Result(stack(b, c), Result(pickup(b),  
Result(unstack(...))))*

- Problème: on doit expliciter ce qui change et ce qui ne changent pas !

*Si a est rouge dans l'état  $s_n$ , alors il doit rester rouge dans l'état suivant  $s_{n+1}$ !*

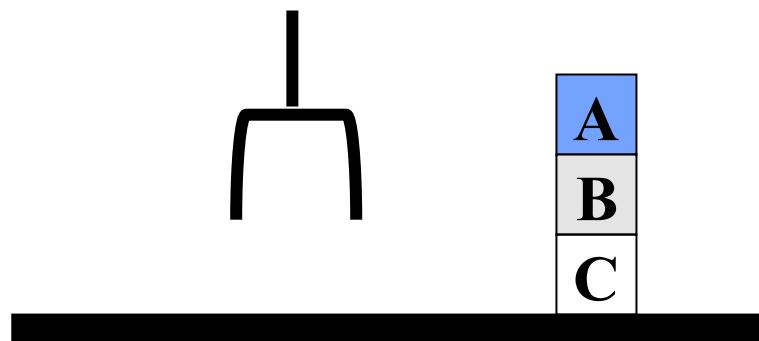
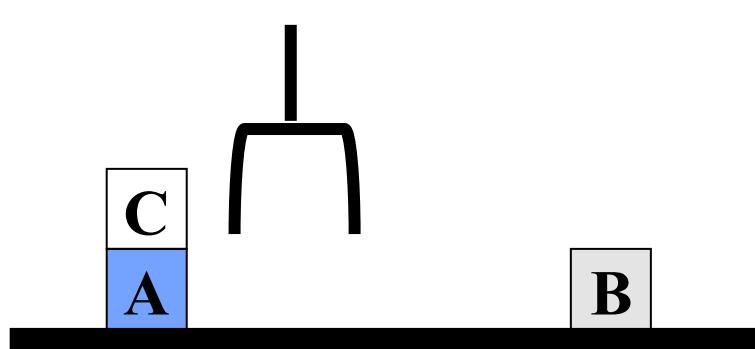


# Une première implémentation

- Etat et buts

```
hold(on(c, a), debut).  
hold(on(a, table), debut).  
hold(on(b, table), debut).  
hold(clear(c), debut).  
hold(clear(b), debut).
```

```
liste_buts([on(a, b), on(b, c)]).
```



Goal state



# Calcul des situations

- **Deux notions essentielles:**

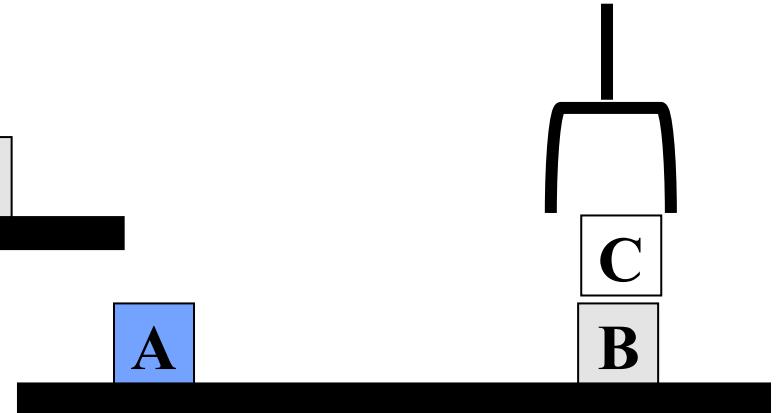
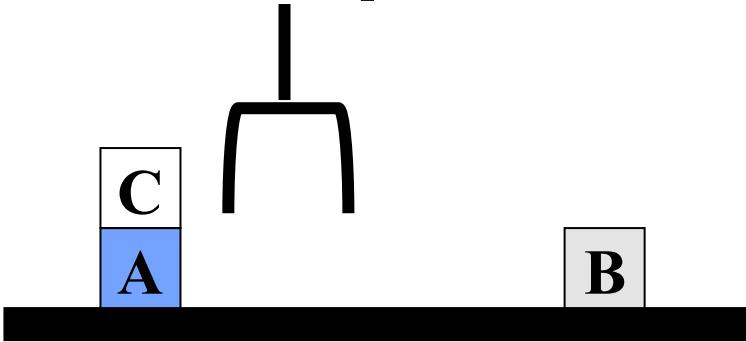
- **Les situations:** c'est l'ensemble des états accessibles par le système. En général, les situations ne sont pas données explicitement, mais seulement de façon partielle
- **Les fluents:** ce sont les prédictats ou les termes qui décrivent les évolutions du systèmes
- **Langage**
  - **Holds (pfluent, s)** – pfluent est un prédicat fluent
  - **Value (tfluent, s)** – tfluent est un terme fluent
  - **Result (e, s)** – e est un événement
  - **Occurs (e, s)**
  - **Next (s)**



# Une première implémentation: actions

- **Actions**

```
ad(on(X, Z), move(X, _, Z)).  
ad(clear(Y), move(_, Y, _)).
```



```
del(on(X, Y), move(X, Y, _)).
```

```
del(clear(Z), move(_, _, Z)).
```

```
cond(move(X, Y, table),  
      [on(X, Y), Y \= table, clear(X)]).
```

```
cond(move(X, Y, Z),  
      [clear(Z), on(X, Y), X \= Z, clear(X)]).
```



# Première implantation: les « fluents »

```
hold(Fait, _) :- toujours(Fait).  
hold(Fait, Etat) :- ad(Fait, Etat).  
hold(Fait, faire(Action, Etat)) :-  
    ad(Fait, Action), cond(Etat).  
hold(Fait, faire(Action, Etat)) :-  
    hold(Fait, Etat), \+ del(Fait, Action).  
  
action_possible(Action, Etat) :-  
    cond(Action, Preconditions),  
    presents(Preconditions, Etat).  
  
presents([], _).  
presents([Fait|SuiteFaits], Etat) :-  
    hold(Fait, Etat), presents(SuiteFaits, Etat).
```



# Première implémentation (fin)

```
but(Etat) :- liste_buts(Buts), presents(Buts, Etat).  
  
etat_possible(debut).  
etat_possible(faire(Action, Etat)) :-  
    etat_possible(Etat), action_possible(Action, Etat).  
  
toujours(X \= Z) :- X \= Z.  
  
jolie_impression(faire(Action, Etat)) :-  
    jolie_impression(Etat), write(Action), write(' -> ').  
jolie_impression(debut) :- write(debut), write(' -> ').
```

**Appel:**

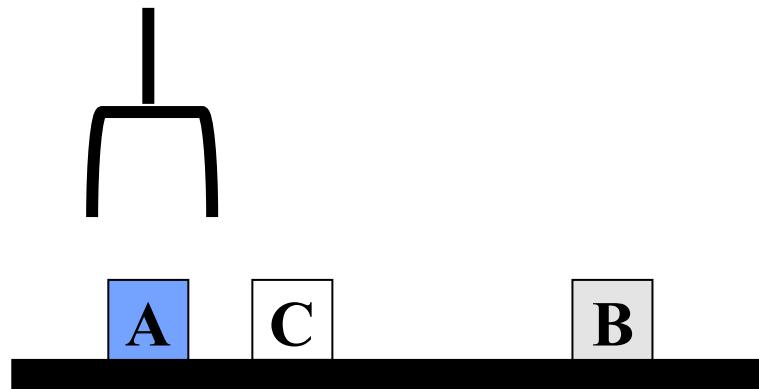
```
?- but(S), etat_possible(S), jolie_impression(S).
```



# Programmation en PROLOG

- **Etat initial s0:**

```
holds_at(ontable(a), s0).  
holds_at(ontable(b), s0).  
holds_at(ontable(c), s0).  
holds_at(clear(a), s0).  
holds_at(clear(b), s0).  
holds_at(clear(c), s0).  
holds_at(handempty, s0).
```



L

I

P

6

C

N

R

S

# Les règles

```
/* empiler(X, Y) */  
cond(empiler(X, Y), S) :-  
    holds_at(hold(X), S),  
    holds_at(clear(Y), S),  
    X \= Y.  
  
ad(empiler(X, Y), S, on(X, Y)).  
ad(empiler(X, Y), S, handempty).  
ad(empiler(X, Y), S, clear(X)).  
del(empiler(X, Y), S, clear(Y)).  
del(empiler(X, Y), S, hold(X)).
```

```
/* saisir(X) */  
cond(saisir(X), S) :-  
    holds_at(ontable(X), S),  
    holds_at(clear(X), S),  
    holds_at(handempty, S).  
  
del(saisir(X), S, ontable(X)).  
del(saisir(X), S, clear(X)).  
del(saisir(X), S, handempty).  
ad(saisir(X), S, hold(X)).
```

```
/* depiler(X, Y) */  
cond(depiler(X, Y), S) :-  
    holds_at(on(X, Y), S),  
    holds_at(clear(X), S),  
    holds_at(handempty, S),  
    X \= Y.  
  
del(depiler(X, Y), S, on(X, Y)).  
del(depiler(X, Y), S, handempty).  
del(depiler(X, Y), S, clear(X)).  
ad(depiler(X, Y), S, clear(Y)).  
ad(depiler(X, Y), S, hold(X)).
```

```
/* déposer(X) */  
cond(deposer(X), S) :-  
    holds_at(hold(X), S).  
  
del(deposer(X), S, hold(X)).  
ad(deposer(X), S, ontable(X)).  
ad(deposer(X), S, clear(X)).  
ad(deposer(X), S, handempty).
```



# Déclenchement d'une règle

```
holds_at(P, r(A, S)) :- setof(Ac, ad(Ac, S, P), L),  
    member(A, L), \+ goal(A), asserta(goal(A)),  
    (declenchee(A, S) ->  
        (asserta(holds_at(P, r(A, S)) : !)),  
        retract(goal(A))) ;  
    (retract(goal(A)), fail)) .
```

```
holds_at(P, r(A, S)) :- \+ var(P), situation(S),  
    holds_at(P, S), declenchee(A, S) ,  
    \+ del(A, S, P) .
```

```
:- dynamic(declenchee/2).  
declenchee(A, S) :- cond(A, S),  
    asserta(declenchee(A, S) : - !) .
```



L

I

P

6

C

N

R

S

# Goal

```
goal([], S, S).  
goal(L, S1, S) :- write('Pile de buts: '), write(L), nl,  
    select(E, L, R), !,  
        ( liste(E) -> (write('but composé: '),  
            write(E), nl,  
            (hold_at(E, S) -> true ;  
            append(E, L, NL),  
                goal(NL, S1, S)))) ;  
    (holds_at(E, S1),  
        write('but: '), write(E),  
        write('; etat: '),  
        write(S1), nl,  
        composition(S1, S2),  
        goal(R, S2, S))).  
  
composition(S, S).  
composition(S1, r(_, S2)) :- composition(S1, S2).  
  
liste(L) :- L =.. [ ' .' | _ ].
```

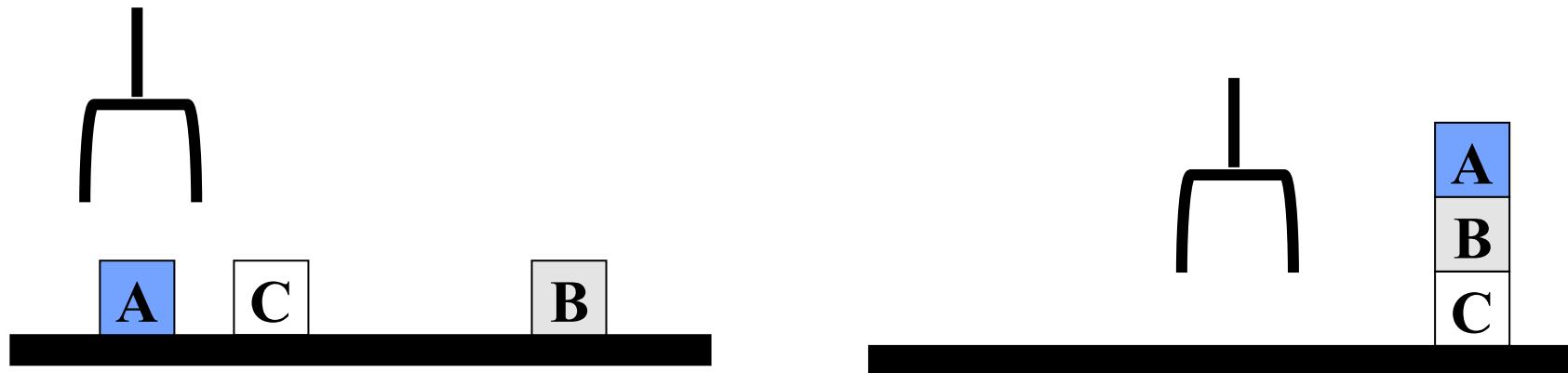


# But

```
?- goal([on(b, c), on(a, b), ontable(c),  
        handempty, clear(a)], _, S).
```

```
?- goal([on(b, c), on(a, b)], _, S).
```

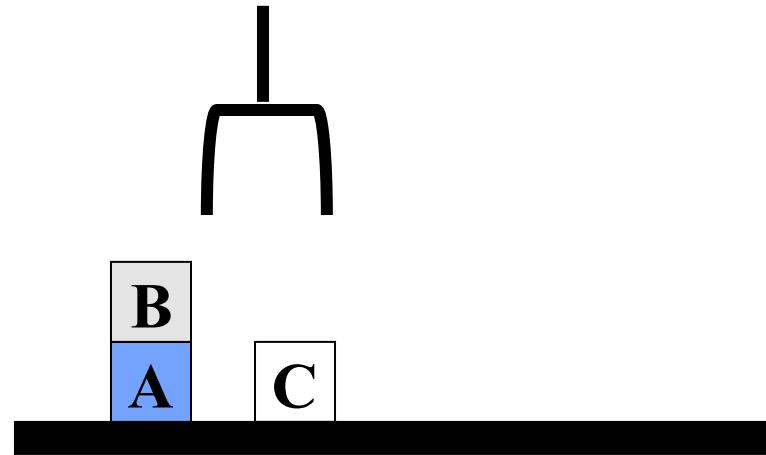
```
S = r(empiler(a, b), r(saisir(a),  
        r(empiler(b, c), r(saisir(b), s0))))
```



# Autre exemple

- **Etat initial s0:**

```
holds_at(ontable(a), s0).  
holds_at(ontable(c), s0).  
holds_at(clear(b), s0).  
holds_at(on(b, a), s0).  
holds_at(clear(c), s0).  
holds_at(handempty, s0).
```



TABLE



# le «frame problem»

- On doit préciser explicitement tout ce qui ne change pas d'un état à un autre, sinon on risque d'obtenir des résultats incomplets

$on(a, b, S) \Rightarrow on(a, b, DO(unstack(c, d), S))$

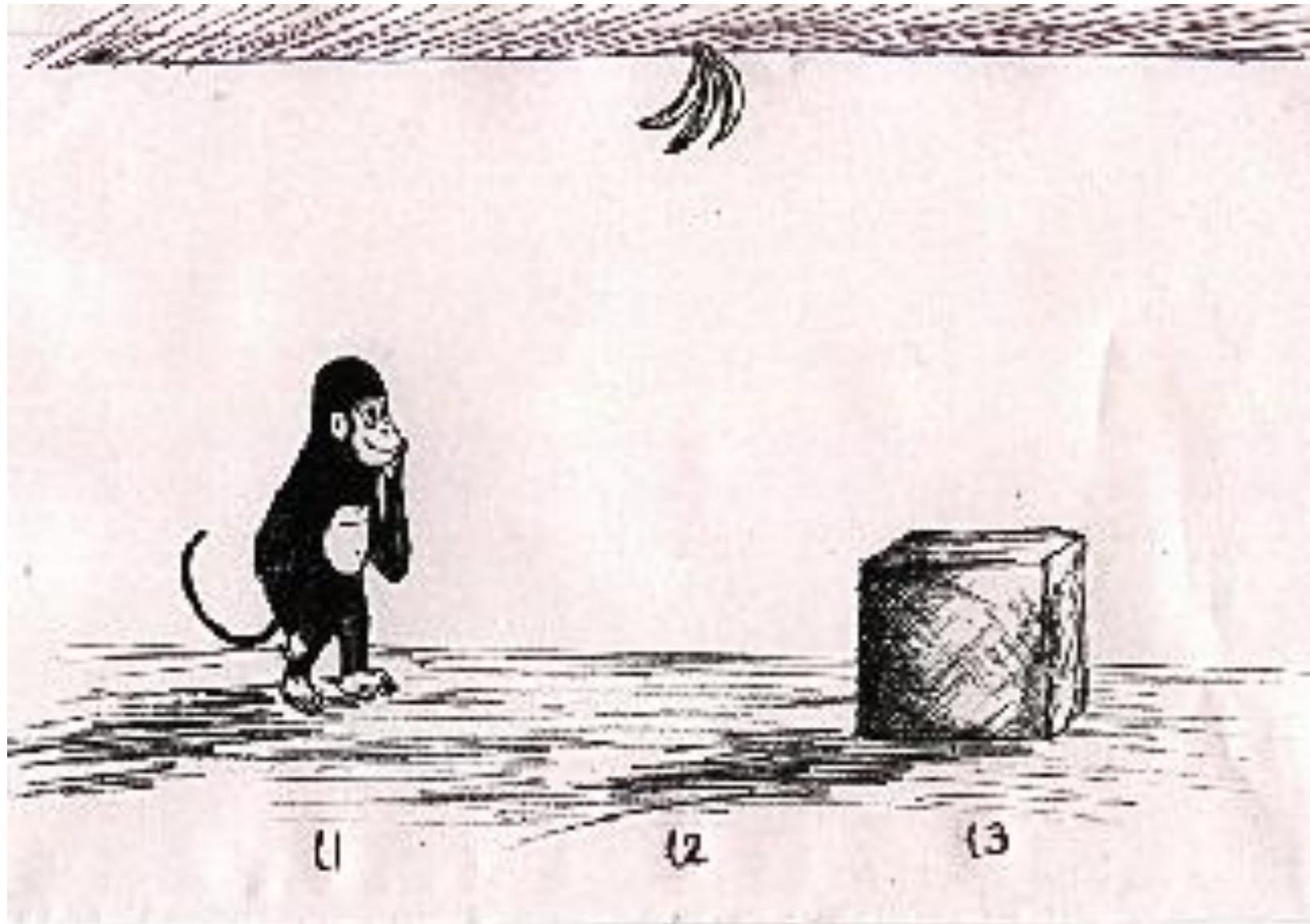
$color(X, red, S) \Rightarrow color(X, red, DO(unstack(Y, Z), S))$

Ces axiomes sont appelés les axiomes du cadre

- C'est une analogie avec la conception des dessins animés
- L'explication de tous les axiomes du cadre est à la fois pénible et inefficace !



# Le « singe et des bananes »



# Le « singe et les bananes »: calcul des situations

- Prédicats de situation

at(ladder,13,s0) .

at(monkey,11,s0) .

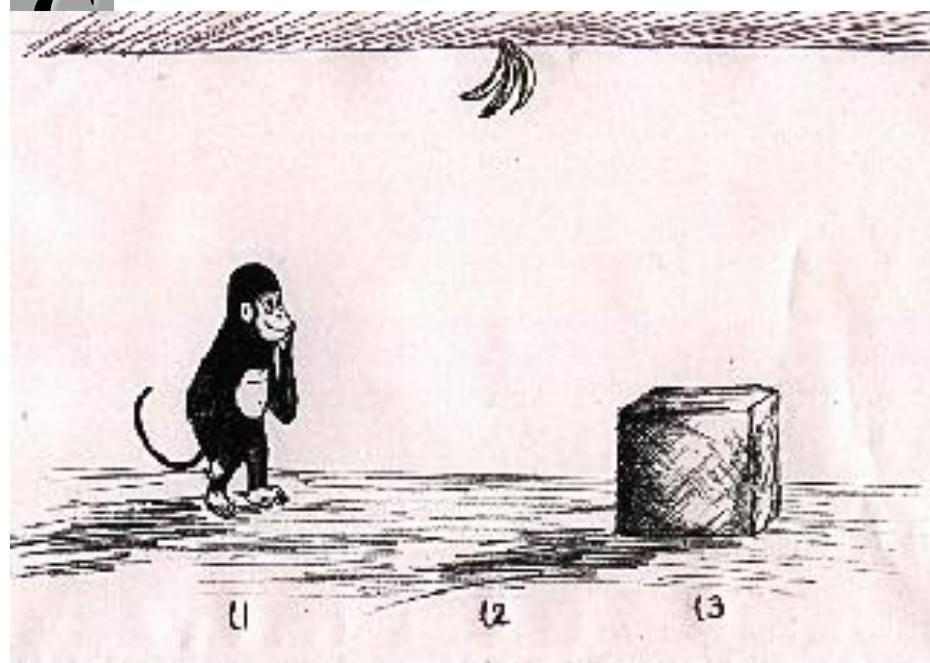
at(bananas,12,s0) .

on(ceiling,bananas,s0) .

on(floor,monkey,s0) .

on(floor,ladder,s0) .

has\_nothing(monkey,s0) .



- Actions: climb, go, drop, grab

at(monkey,L,**go(L,S)**) :-  
on(floor,monkey,S) .

**has\_nothing(monkey,drop(X,S))** :-  
has(monkey,X,S) .

on(ceiling,monkey,  
**climb(ladder,S)**) :-  
can(climb(ladder,S)) .

has(monkey,X,**grab(X,S)**) :-  
can(grab(X,S)) .

at(X,L,S) :- has(monkey,X,S) ,  
at(monkey,L,S) .

on(H,X,S) :- has(monkey,X,S) ,  
on(H,monkey,S) .

# Le « singe et les bananes »: calcul des situations

- Pré-conditions

```
can(grab(X,S)) :-  
    non_monkey(X),  
    at(X,L,S), at(monkey,L,S),  
    has_nothing(monkey,S),  
    on(H,monkey,S), on(H,X,S).
```

```
can(climb(ladder,S)) :-  
    at(ladder,L,S),  
    at(monkey,L,S),  
    on(floor,ladder,S),  
    on(floor,monkey,S),  
    has_nothing(monkey,S).
```

- « Frame axioms »

```
has_nothing(monkey,climb(X,S))  
:- can(climb(X,S)),  
    has_nothing(monkey,S).
```

```
has_nothing(monkey,go(L,S)) :-  
    has_nothing(monkey,S),  
    on(Floor,monkey,S).
```

```
has(monkey,X,go(L,S)) :-  
    has(monkey,X,S),  
    on(floor,monkey,S).
```



L

I

P

6

C

N

R

S



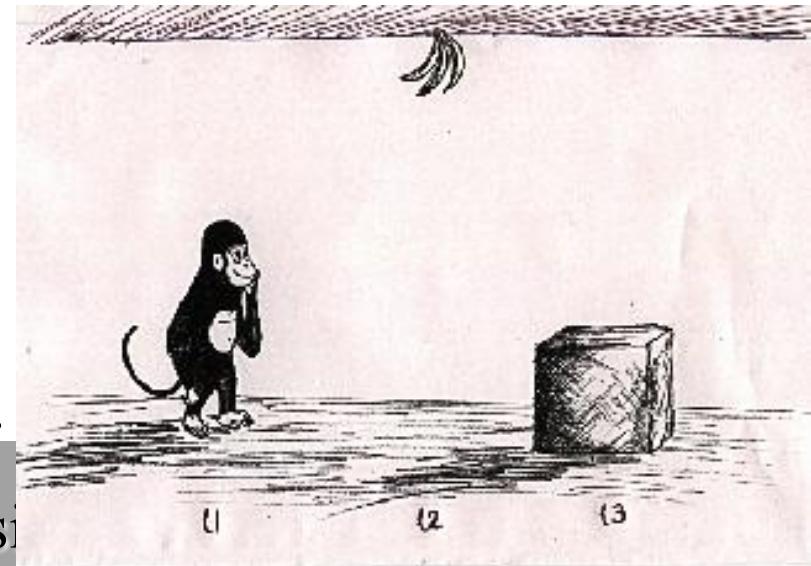
## « singe et bananes » (suite)

```

at(Y,L,grab(X,S)) :- can(grab(X,S)),at(Y,L,S).
at(Y,L,climb(X,S)) :- can(climb(X,S)),at(Y,L,S).
at(X,L,drop(Y,S)) :- has(monkey,Y,S),at(X,L,S).
at(X,L1,go(L2,S)) :- at(X,L1,S), non monkey(X),
    has_nothing(monkey,S), on(floor,monkey,S).
at(X,L1,go(L2,S)) :- at(X,L1,S),non monkey(X),
    has(monkey,Y,S), differ(X,Y), \+on(floor,monkey,S).

on(H,X,go(L,S)) :- on(H,X,S), on(floor,monkey,S).
on(H,X,climb(Y,S)) :- can(climb(Y,S)), on(H,X,S),
    non_monkey(X), has_nothing(monkey,S).
on(floor,Y,drop(Y,S)) :- has(monkey,Y,S).
on(floor,X,drop(Y,S)) :-
    has(monkey,Y,S),
    on(floor,X,S).
on(ceiling,X,drop(Y,S)) :-
    has(monkey,Y,S),
    differ(X,Y),
    on(ceiling,X,S).
on(H,X,grab(Y,S)) :-
    can(grab(Y,S)),on(H,X,S).

```



L

I

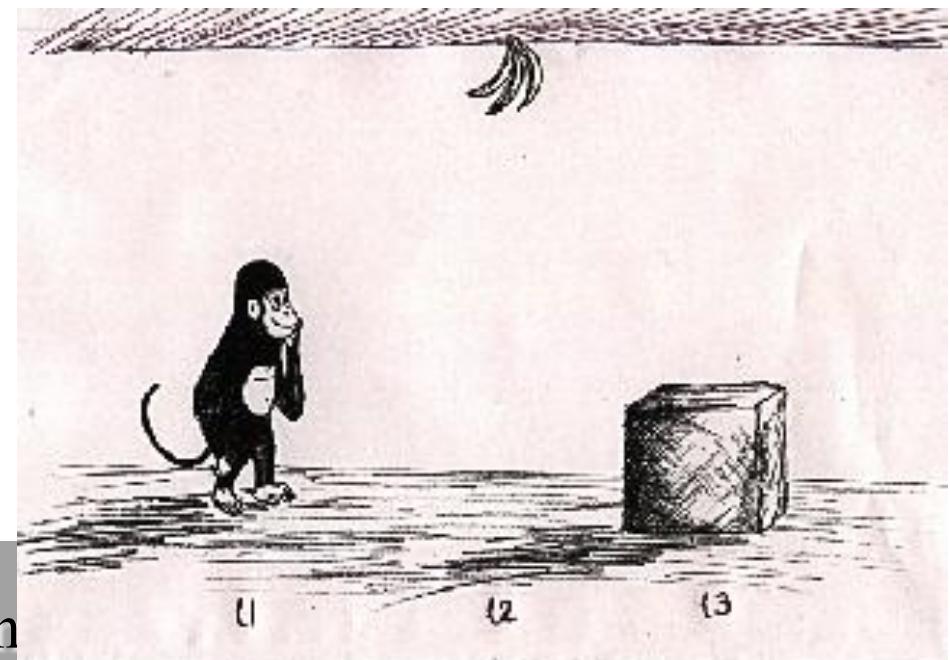
P

6

C  
N  
R  
S

# « singe et bananes » (suite(suite))

```
rewrite(go(Z,go(W,S)),go(Z,S)).  
rewrite(grab(X,drop(X,S)),S).  
rewrite(has(monkey,monkey,S),fail).  
  
non_monkey(ladder).  
non_monkey(bananas).  
differ(ladder,bananas).  
differ(bananas,ladder).
```



1

2

3



# Planification à l'aide du calcul des situations : analyse

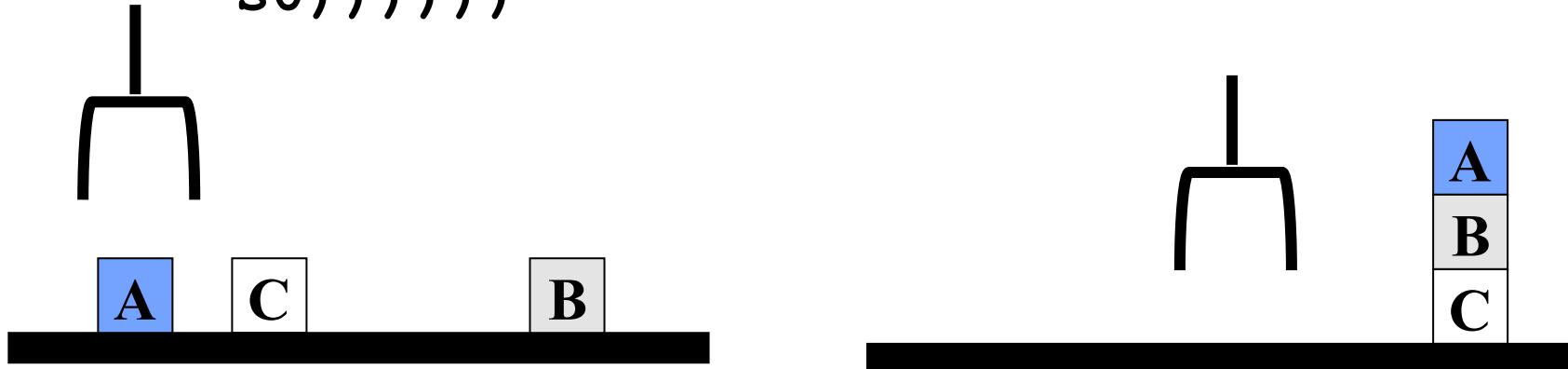
- Complexité rédhibitoire : la démonstration automatique de théorèmes à une complexité algorithmique exponentielle dans le pire des cas
- De plus, la démonstration à l'aide de la méthode de résolution est capable de trouver *une* preuve (c'est-à-dire un plan), mais pas nécessairement un bon plan
- En conséquence, il faut restreindre le langage et faire appel à un algorithme spécialisé (un planificateur) plutôt moins général que le démonstrateur automatique de théorèmes de PROLOG.

Le système STRIPS vu la dernière fois est un exemple de planificateur.



# Exemple de mauvais plan

```
?- goal([on(a, b), on(b, c), ontable(c),  
        handempty, clear(a)], _, S).  
S = r(empiler(b, c), r(saisir(b),  
                      r(deposer(a),  
                      r(depiler(a, b),  
                      r(saisir(a),  
                          s0))))))
```



# Exemple de plan non optimal: nécessité d'appel avec but conjoint

```
?- goal([[on(a, b), on(b, c), ontable(c),  
handempty, clear(a)]], _, S).
```

```

S = r(empiler(a, b), r(saisir(a),
                      r(empiler(b, c),
                        r(saisir(b),
                           r(deposer(a),
                              r(empiler(a, b),
                                r(saisir(a),
                                   s0))))))))).

```



# Planification non-linéaire

- Au lieu de satisfaire les buts séquentiellement nous allons essayer de les satisfaire chacun séparément, puis on verra ce qui est nécessaire pour que le but conjoint soit réalisé
- Si une contradiction intervient, élaguer la branche correspondante
- Besoin de traiter les interactions entre les buts



# Planification partiellement ordonnée

- Un planificateur linéaire construit un plan comme une séquence totalement ordonnée d'étapes de plans
- A planificateur non-linéaire (ou planificateur partiellement ordonné) construit un plan comme un ensemble d'étapes et des contraintes temporelles
  - contraintes de la forme  $S_1 < S_2$  si  $S_1$  doit venir avant  $S_2$ .
- On raffine un plan partiellement ordonné (POP) en:
  - ajoutant une nouvelle étape, ou en
  - ajoutant une nouvelle contrainte aux étapes qui se trouvent déjà dans le plan.
- Un POP peut être linéarisé (converti en un plan totalement ordonné) par un tri topologique



# Moindre engagement

- Les planificateurs binaires incorporent le principe du **moindre engagement**
  - Ne choisir les actions, leur ordonnancement et les liaisons de variables que lorsque c'est absolument nécessaire en laissant toutes les autres décisions à plus tard
  - éviter de s'engager précocement dans des décisions qui n'importent pas vraiment
- Un planificateur linéaire choisit toujours d'adoindre une étape dans un plan **à une place particulière** de la séquence
- Un planificateur non-linéaire choisit d'adoindre une étape et introduit éventuellement **quelques contraintes temporelles**



# Plan non-linéaire

- **Un plan non-linéaire consiste en**
  - (1) **Un ensemble d' étapes** { $S_1, S_2, S_3, S_4\dots$ }
  - Chaque étape à une description d' opérateurs, des préconditions et des postconditions
  - (2) **Un ensemble de liens causaux** { ... ( $S_i, C, S_j$ ) ... }
  - Signifiant par exemple que l' étape  $S_i$  doit être réalisé en précondition  $C$  de l' étape  $S_j$
  - (3) **Un ensemble de contrainte d' ordonnancement** { ...  $S_i < S_j \dots$  }
  - Si l' étape  $S_i$  doit venir avant l' étape  $S_j$
- **Un plan non-linéaire est complet** ssi
  - Toutes les étapes mentionnées en (2) et (3) sont en (1)
  - si  $S_j$  a comme prérequis  $C$ , alors il existe un lien de causal dans (2) de la forme ( $S_i, C, S_j$ ) pour quelque  $S_i$
  - si ( $S_i, C, S_j$ ) est dans (2) et l' étape  $S_k$  est dans (1), et  $S_k$  falsifie ( $S_i, C, S_j$ ) (rend  $C$  faux), alors (3) contient soit  $S_k < S_i$  soit  $S_j < S_k$

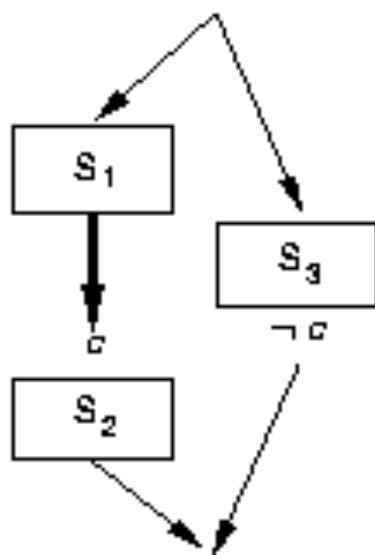


# POP contraintes et heuristiques de recherche

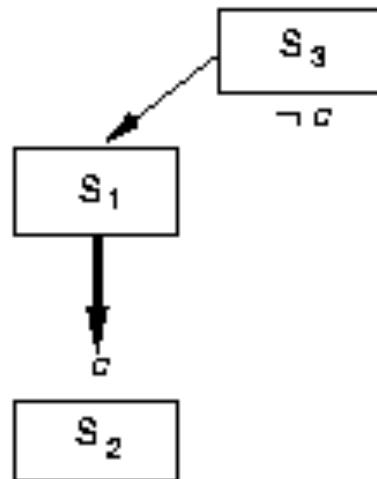
- On ajoute seulement des étapes qui réalisent une précondition qui n'est pas satisfaite
- Utilisation du principe de moindre engagement:
  - Ne pas ordonner des étapes qui n'ont pas besoin de l'être
- Satisfaire les liens causaux  $S_1 \rightarrow S_2$  qui protègent une condition c:
  - Ne jamais ajouter une étape  $S_3$  qui viole c
  - Si une action parallèle menace c (c'est-à-dire si elle a pour effet d'annuler c), résoudre la menace en ajoutant des liens d'ordonnancement:
    - » Ordre  $S_3$  avant  $S_1$  (rétrogradation)
    - » Ordre  $S_3$  après  $S_2$  (promotion)



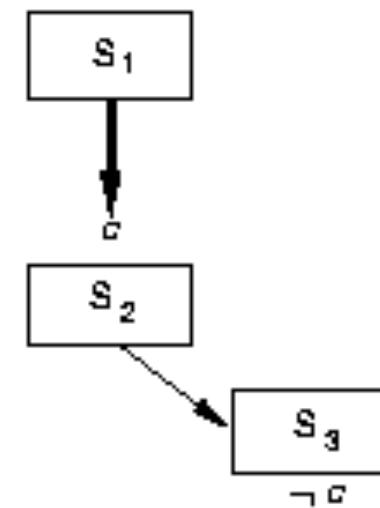
# Résolution de menaces



(a) Menace



(b) Rétrogradation

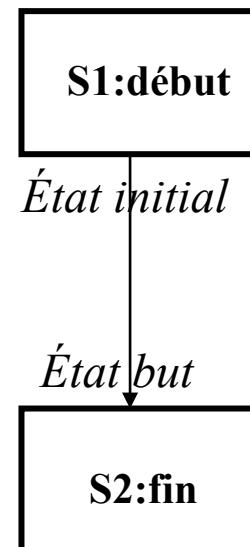


(c) Promotion



# Plan initial

**Tous les plans commencent de la même façon**



# Exemple trivial

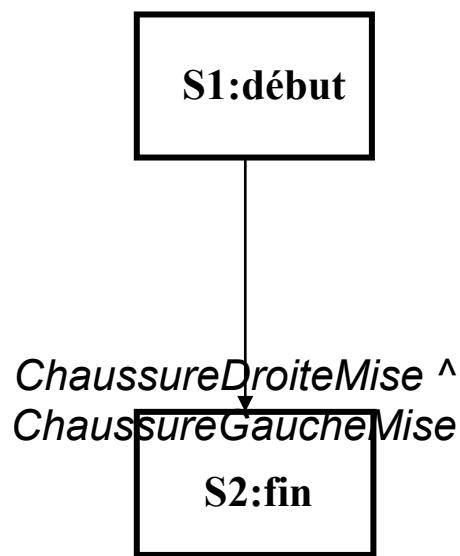
opérateurs:

Op(ACTION: ChaussureDroite, PRECOND: ChaussetteDroiteMise,  
EFFECT: ChaussureDroiteMise)

Op(ACTION: ChaussetteDroite, EFFECT: ChaussetteDroiteMise)

Op(ACTION: ChaussureGauche, PRECOND: ChaussetteGaucheMise,  
EFFECT: ChaussureGaucheMise)

Op(ACTION: ChaussetteGauche, EFFECT: ChaussetteGaucheMise)



Etape: {S1:[Op(Action:début)],

S2:[Op(Action:fin,

Pre: *ChaussureDroiteMise ^*

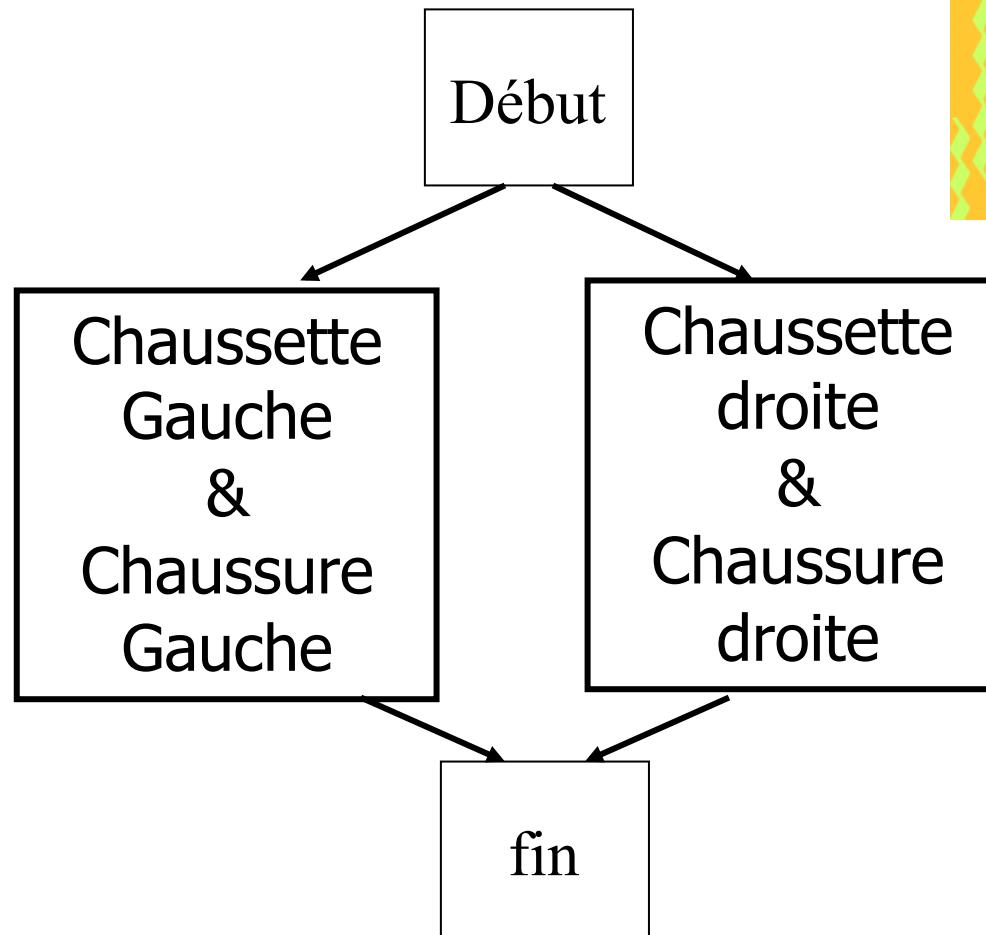
*ChaussureGaucheMise)]}*

Links: {}

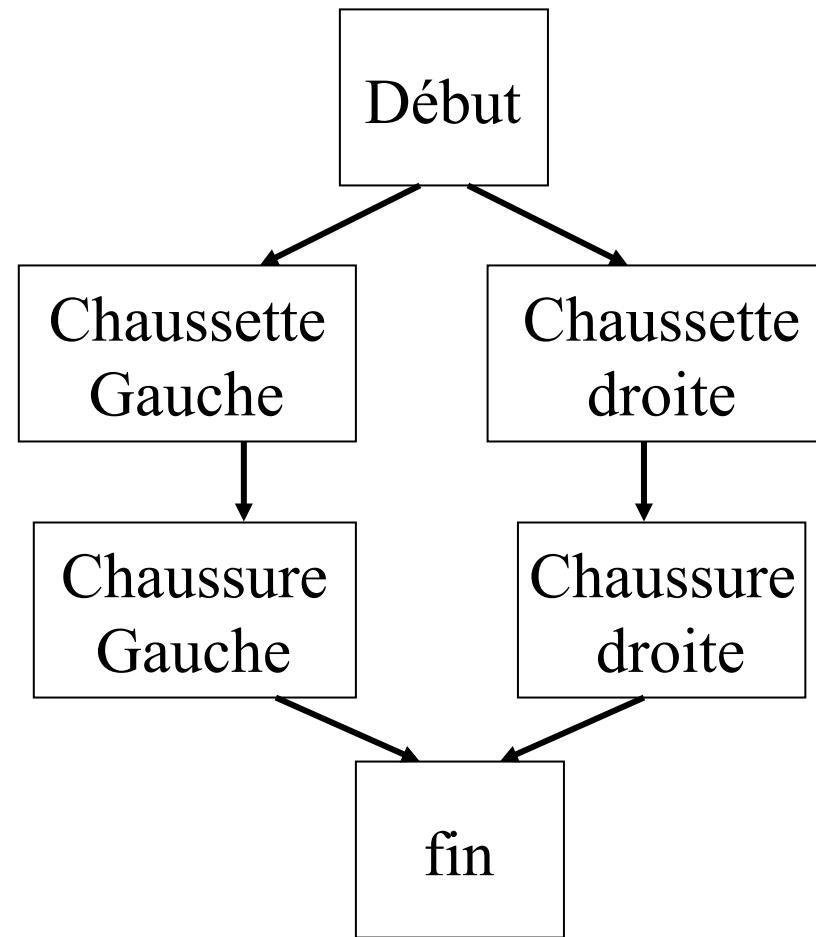
Orderings: {S1 < S2}



# Solution



# Solution



L

I

P

6

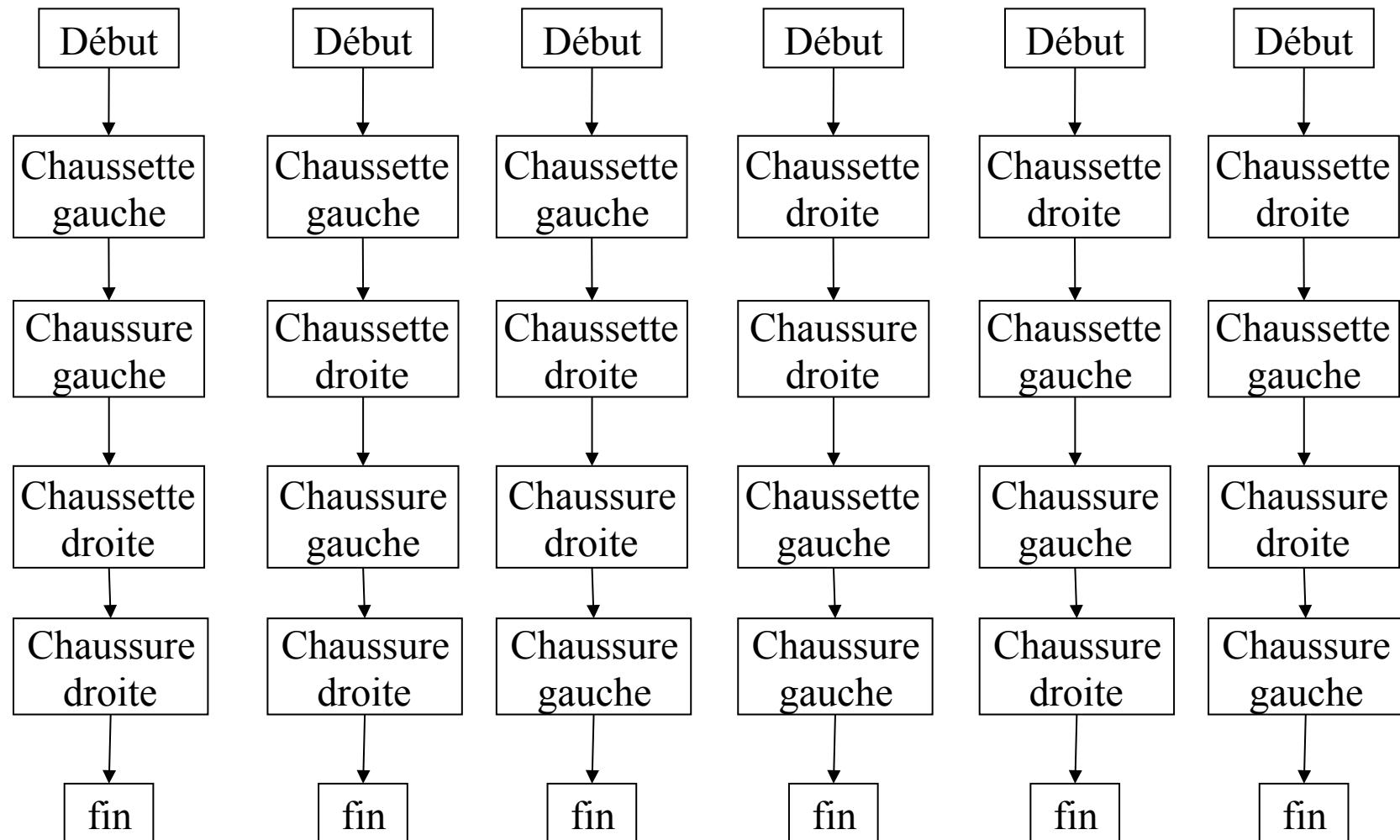
C

N

R

S

# Tous les plans ordonnés...



# Contraintes et heuristiques de recherche

- N'ajouter les étapes que si elles complètent une précondition non réalisée
- Utiliser une approche de « moindre engagement » :
  - Ne pas ordonner les étapes tant qu'elles n'ont pas besoin de l'être
  - Satisfaire les liens causaux  $S_1 \rightarrow S_2$  qui protègent une condition c:
  - Ne jamais ajouter une étape  $S_3$  qui viole c
  - Si une action parallèle ou falsifie c, résoudre cette contrainte en ajoute en un lien d'ordonnancement :
    - » Ordre  $S_3$  avant  $S_1$  (rétrogradation)
    - » Ordre  $S_3$  après  $S_2$  (promotion)



# Système NOAH – Sacerdoti 1975

## « *Nets Of Action Hierarchies* »

- **Représentation des plans partiellement ordonnés**
- **6 différentes types de nœuds:**
  - BUT: but à réaliser – il s'agit d'un état ou d'une action [gris]
  - PHANTOM: condition qui devrait être vraie – si elle l'est (**vert**), sinon (**rouge**)
  - SPLIT: les nœuds connectés à un SPLIT ne sont pas ordonnés [**S**]
  - JOIN: les nœuds connectés sont ordonnés [**J**]



# Système NOAH – Exemple 1

Niveau 1

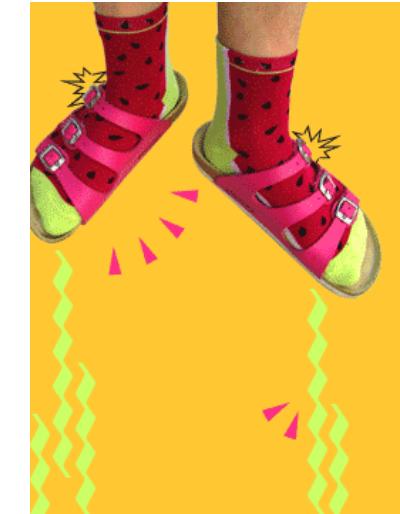
Chausser pieds

Niveau 2

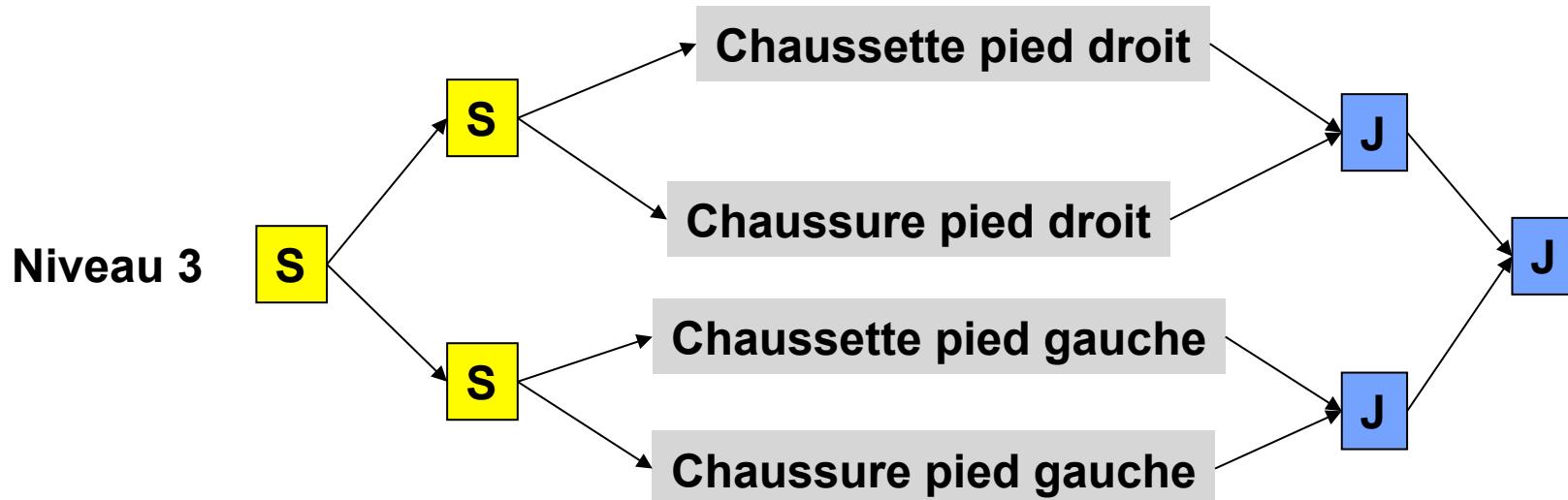
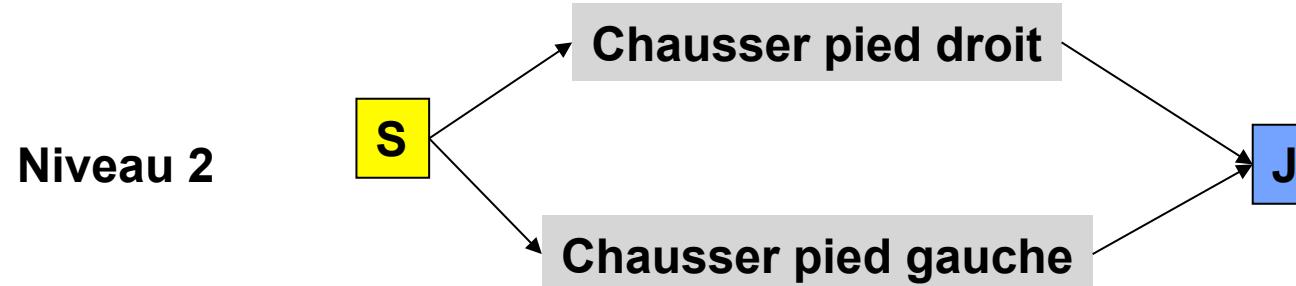
S

Chausser pied droit

Chausser pied gauche



# Système NOAH – Exemple 1



L

I

P

6

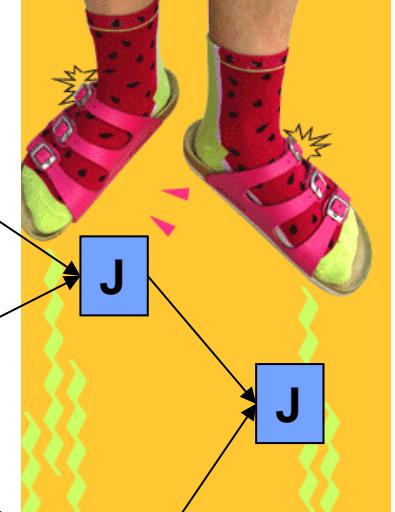
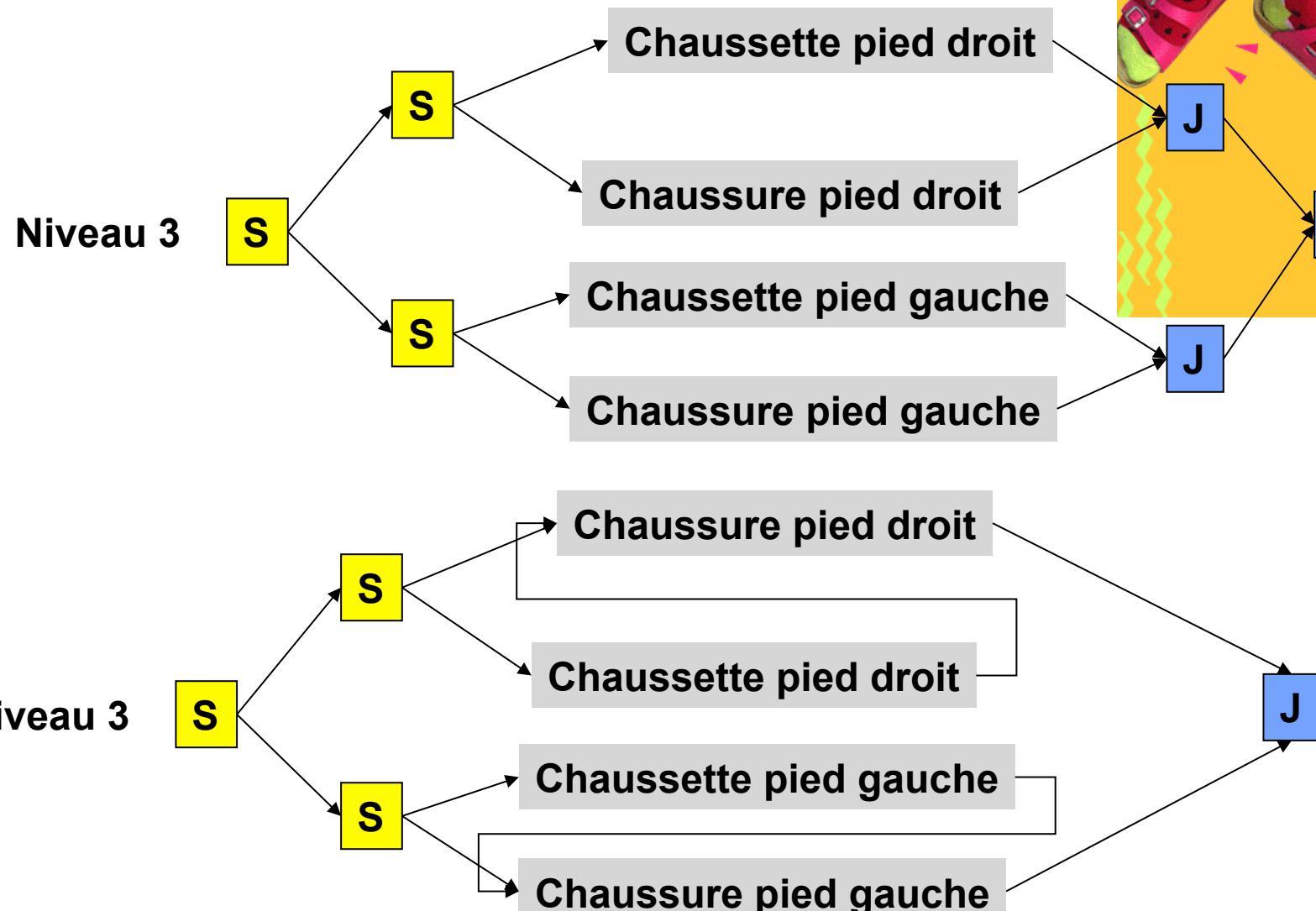
C

N

R

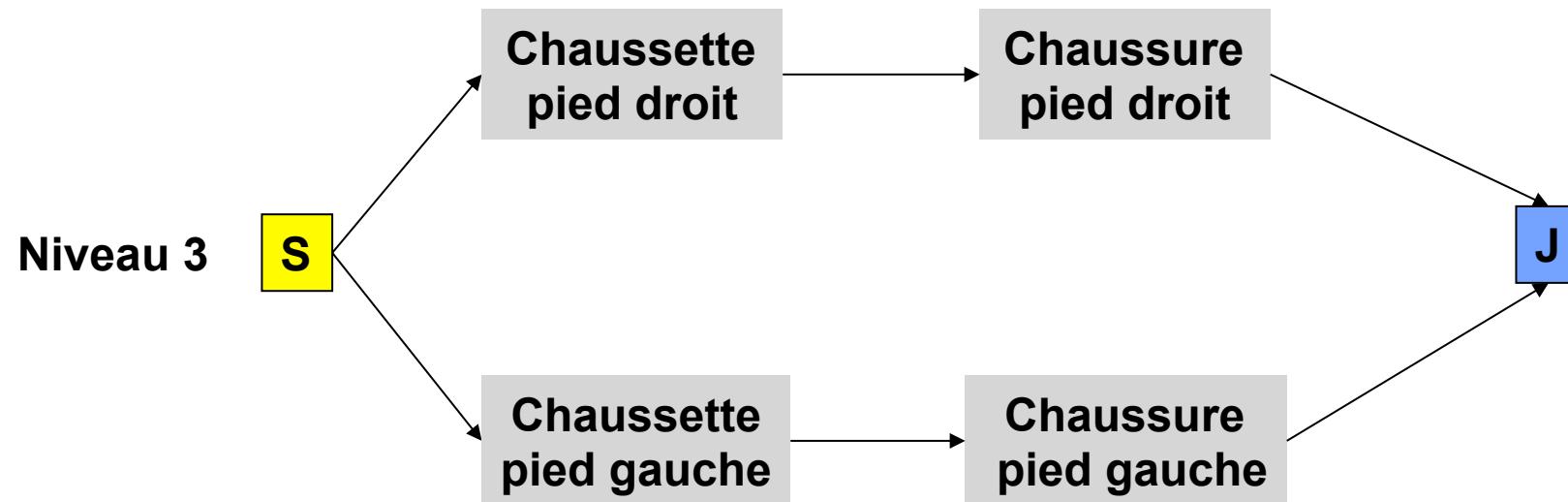
S

# Système NOAH – Exemple 1

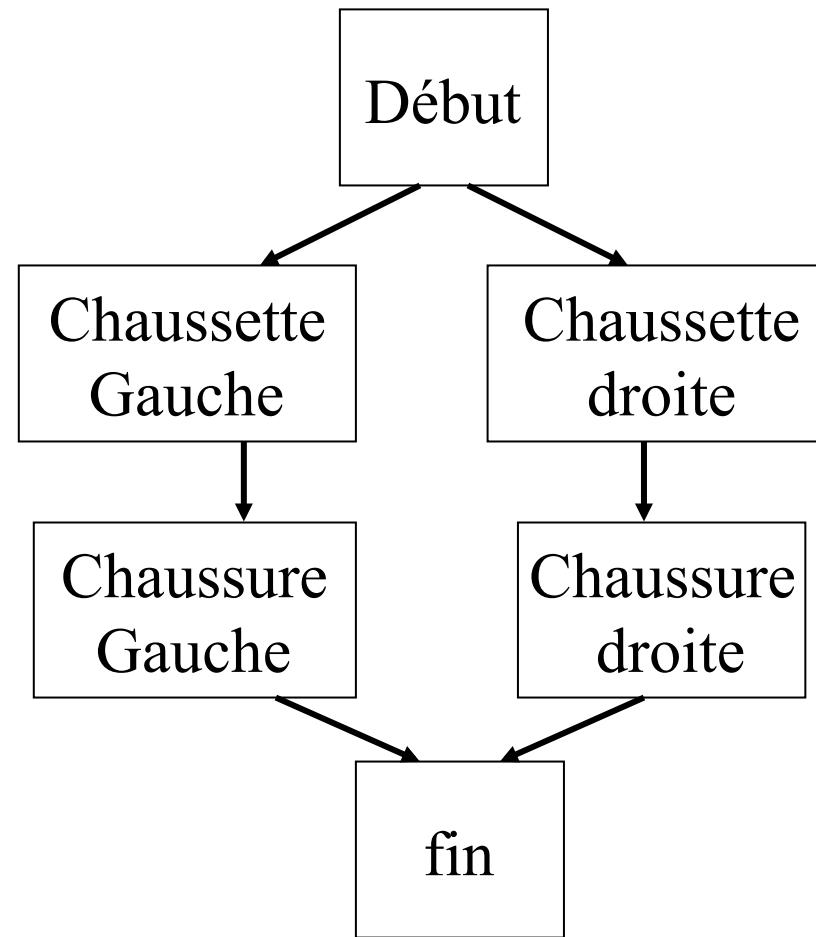


- **Module « Critique »:** donne les contraintes qui permettent d'ordonner des actions - la précondition de chaussure suppose que la chaussette est mise

# Système NOAH – Exemple 1



# Solution



L

I

P

6

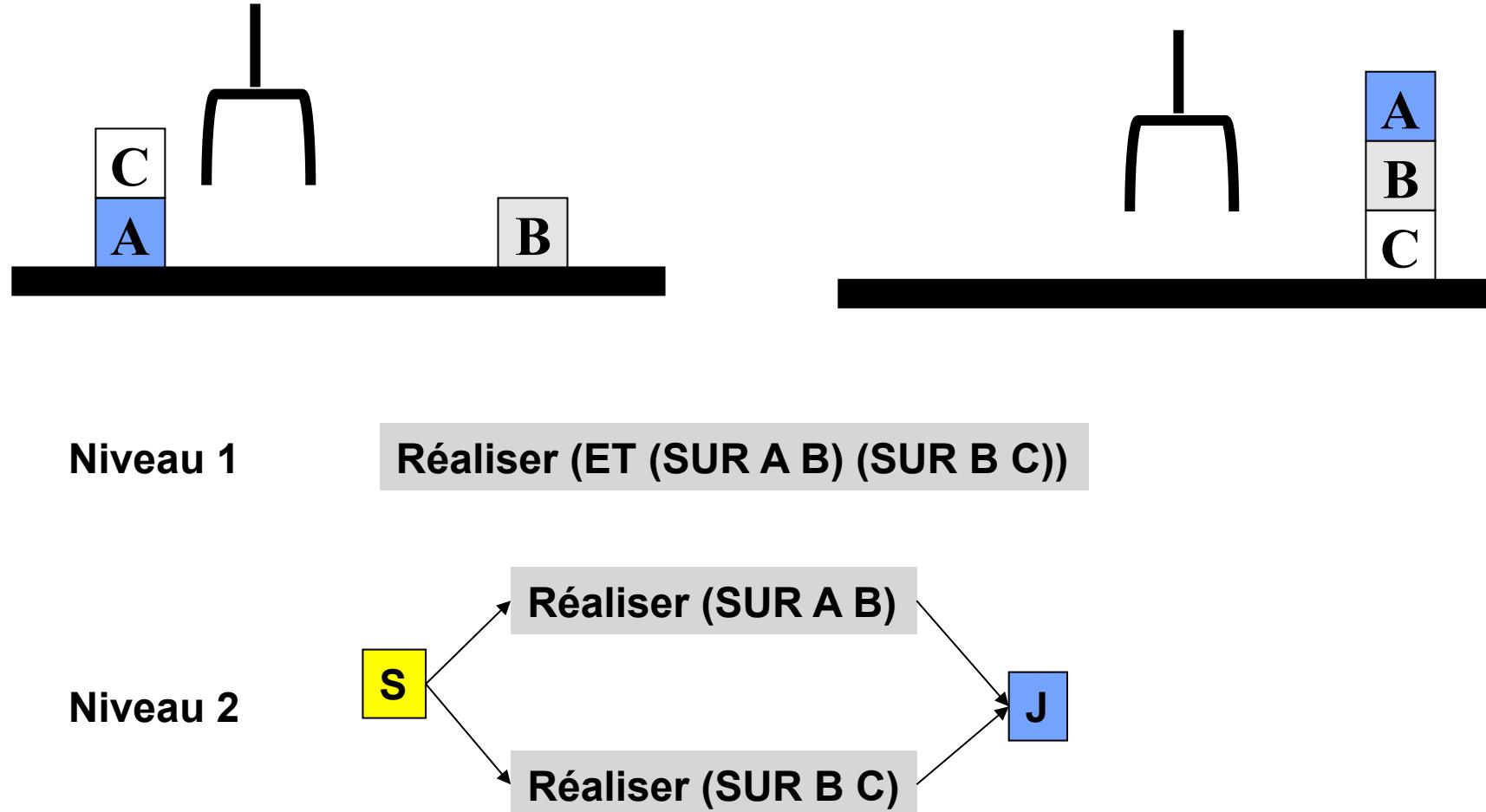
C

N

R

S

# Système NOAH – Exemple



L  
I  
P

6

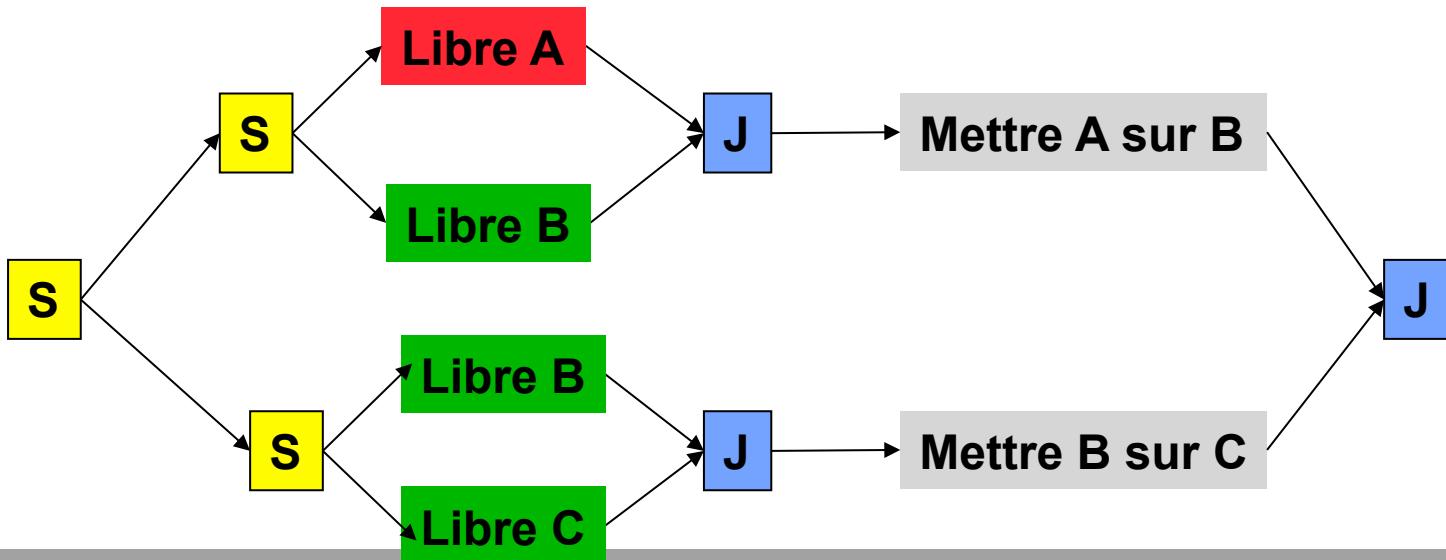
C  
N  
R  
S



Niveau 2



Niveau 3



L

I

P

6

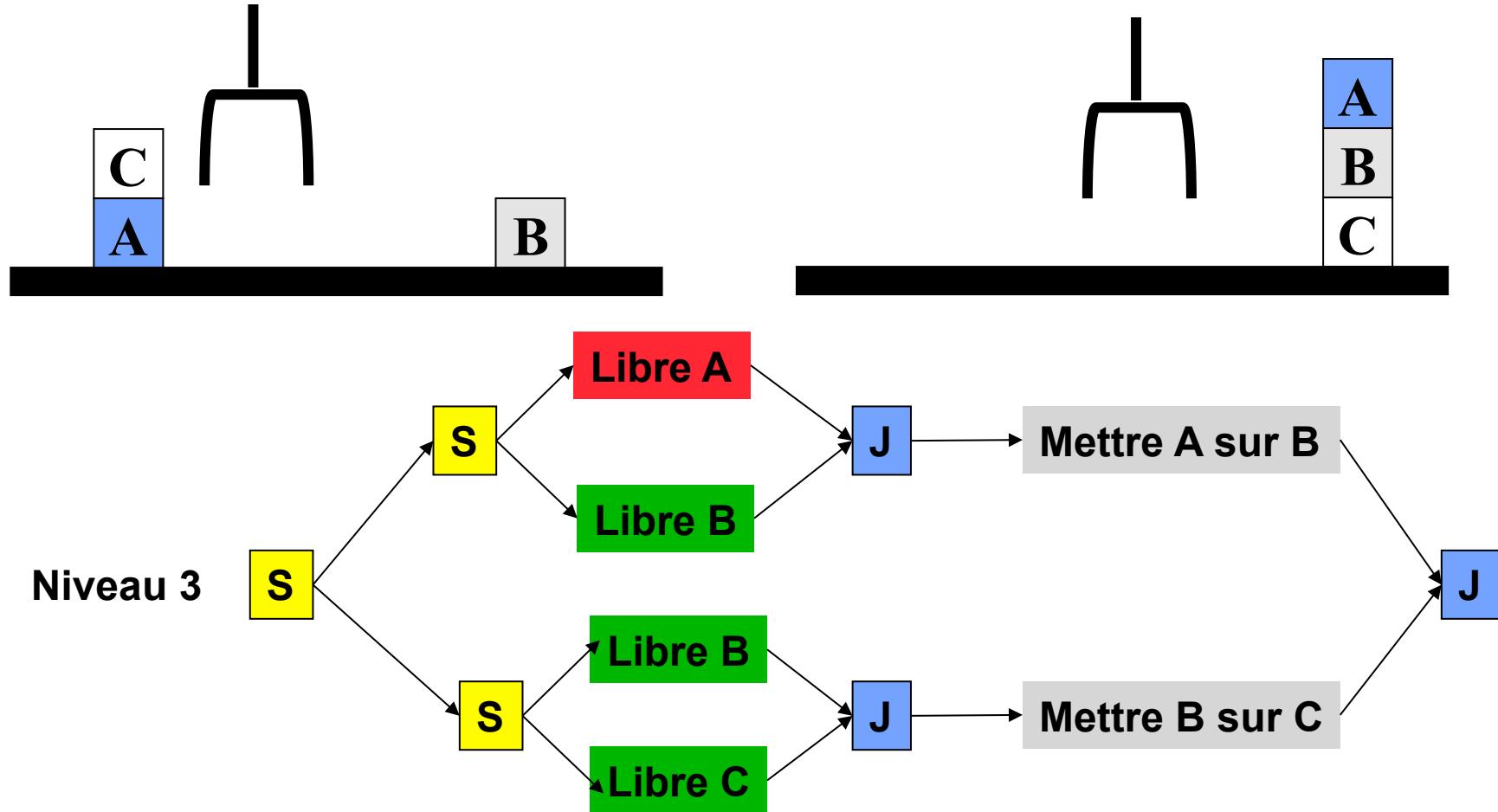
C

N

R

S

# Système NOAH – Niveau 3



- **Module « Critique »: donne les contraintes qui permettent d'ordonner des actions**



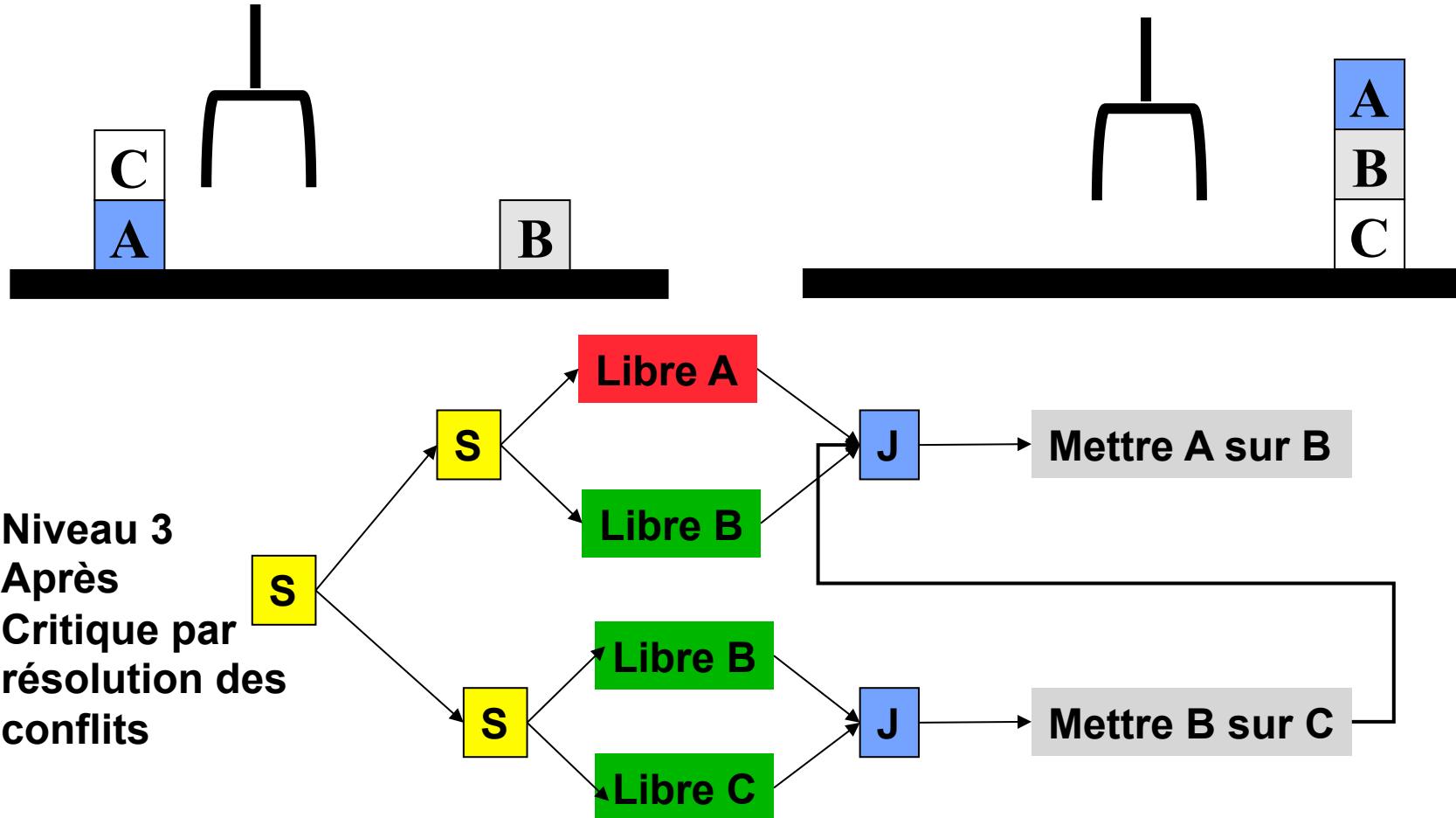
L

I  
P

6

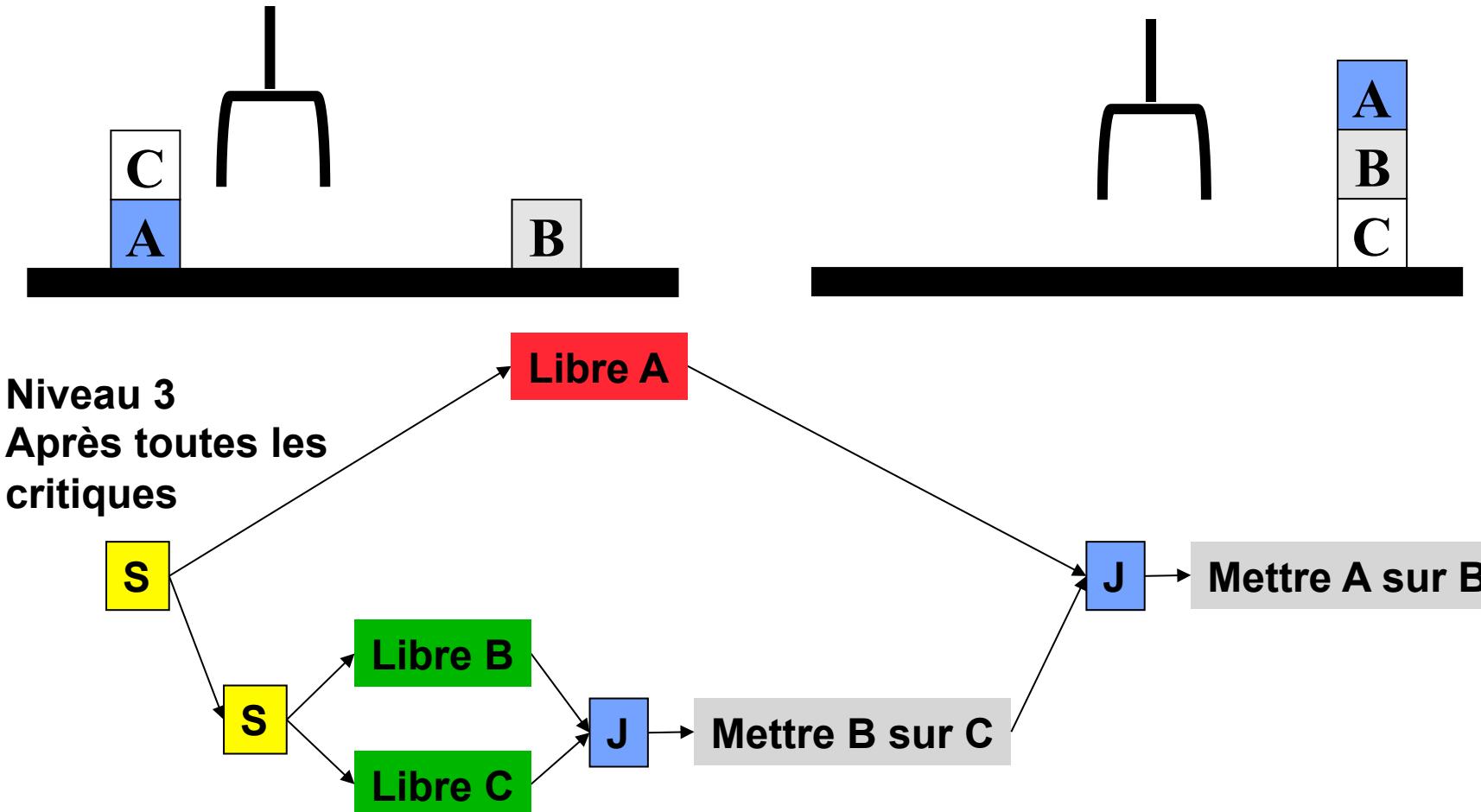
C  
N  
R  
S

# Système NOAH – Niveau 3 (suite)



- **Module « Critique »:** donne les contraintes qui permettent d'ordonner des actions - la précondition de Mettre B sur C (libre B) devient fausse si on met A sur B avant... il faut donc le mettre après

# Système NOAH – Niveau 3 – fin



- **Module « Critique »: donne les contraintes qui permettent d'ordonner des actions - propagation**



L

I

P

6

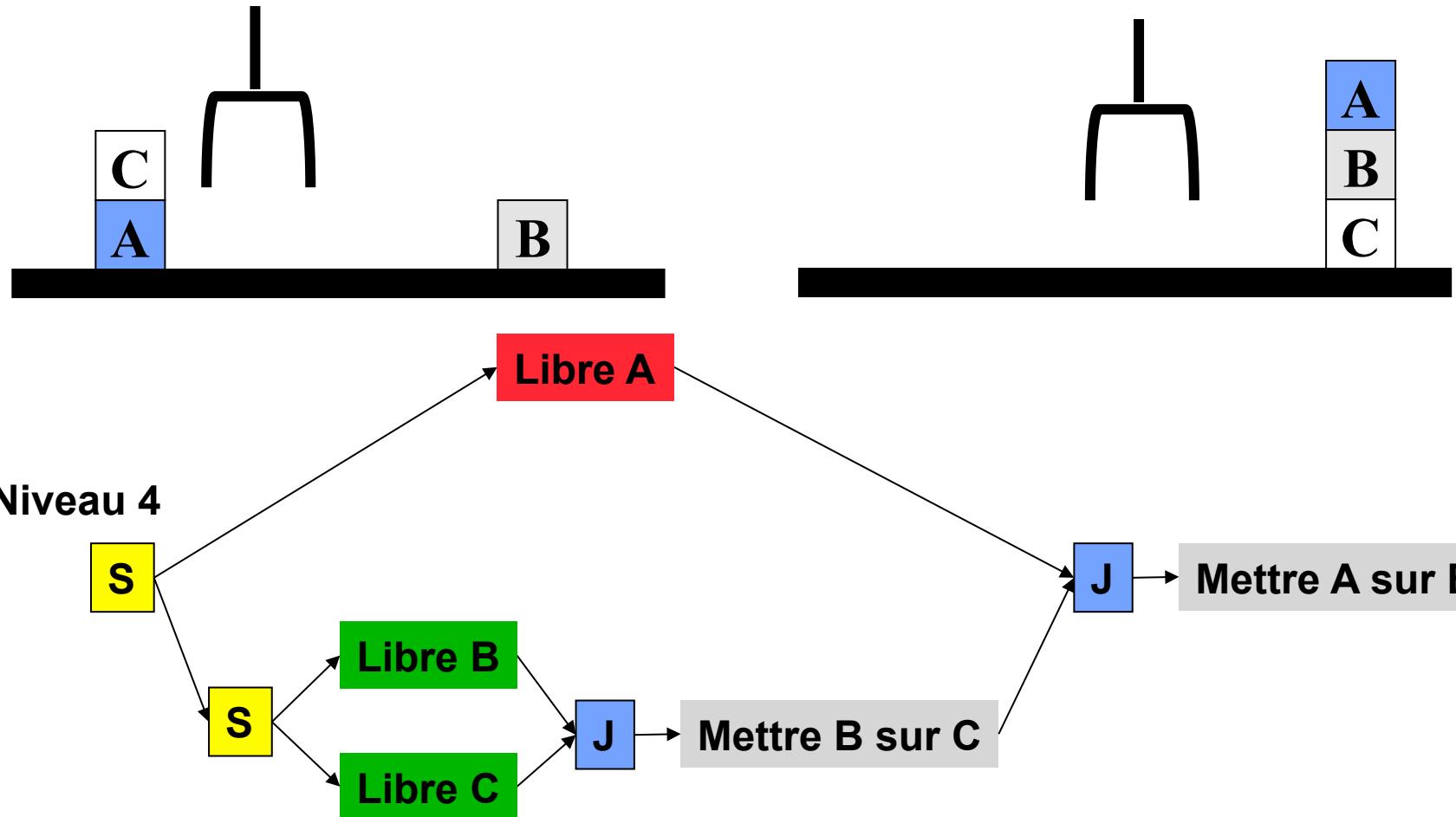
C

N

R

S

# Système NOAH – Niveau 4



L

I

P

6

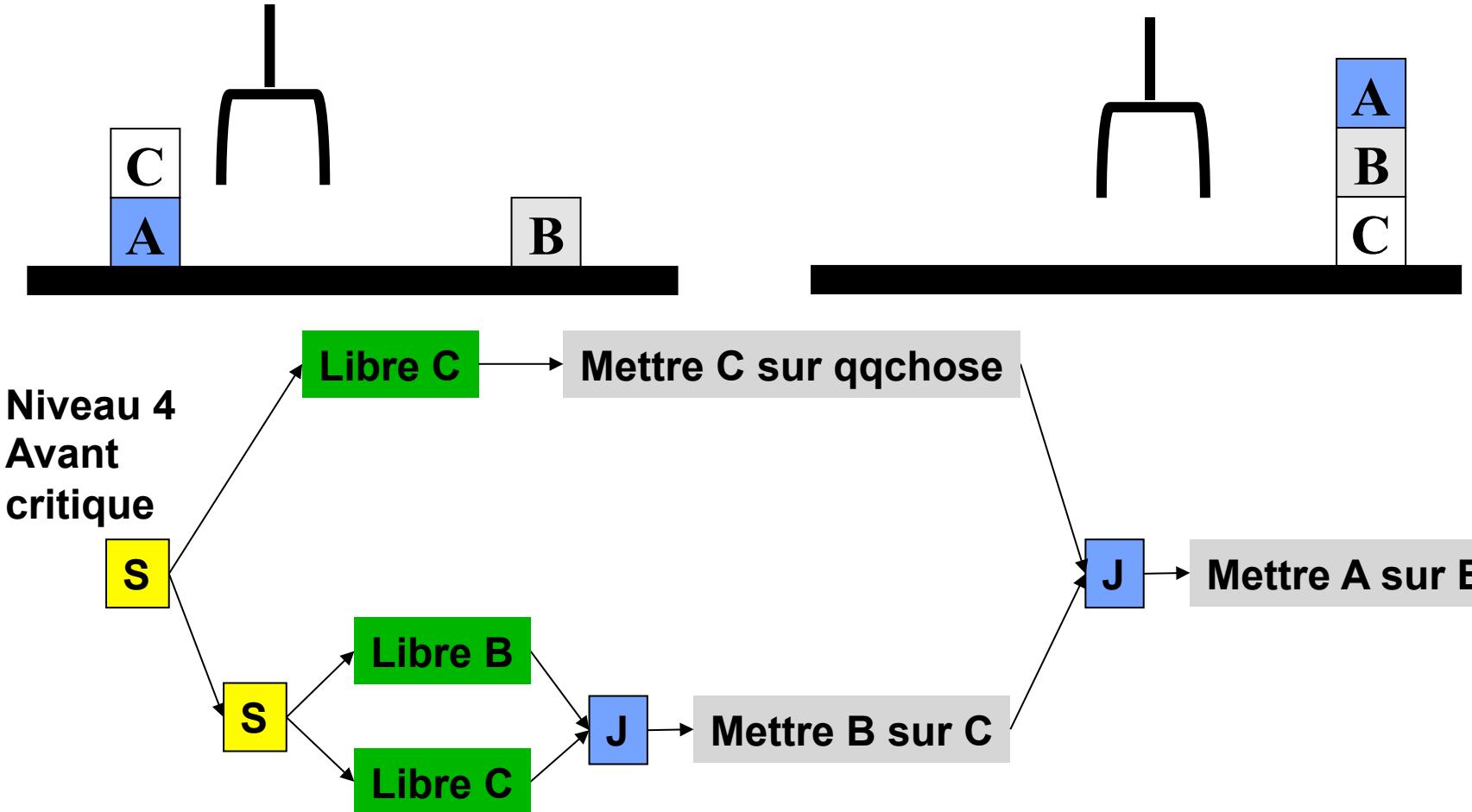
C

N

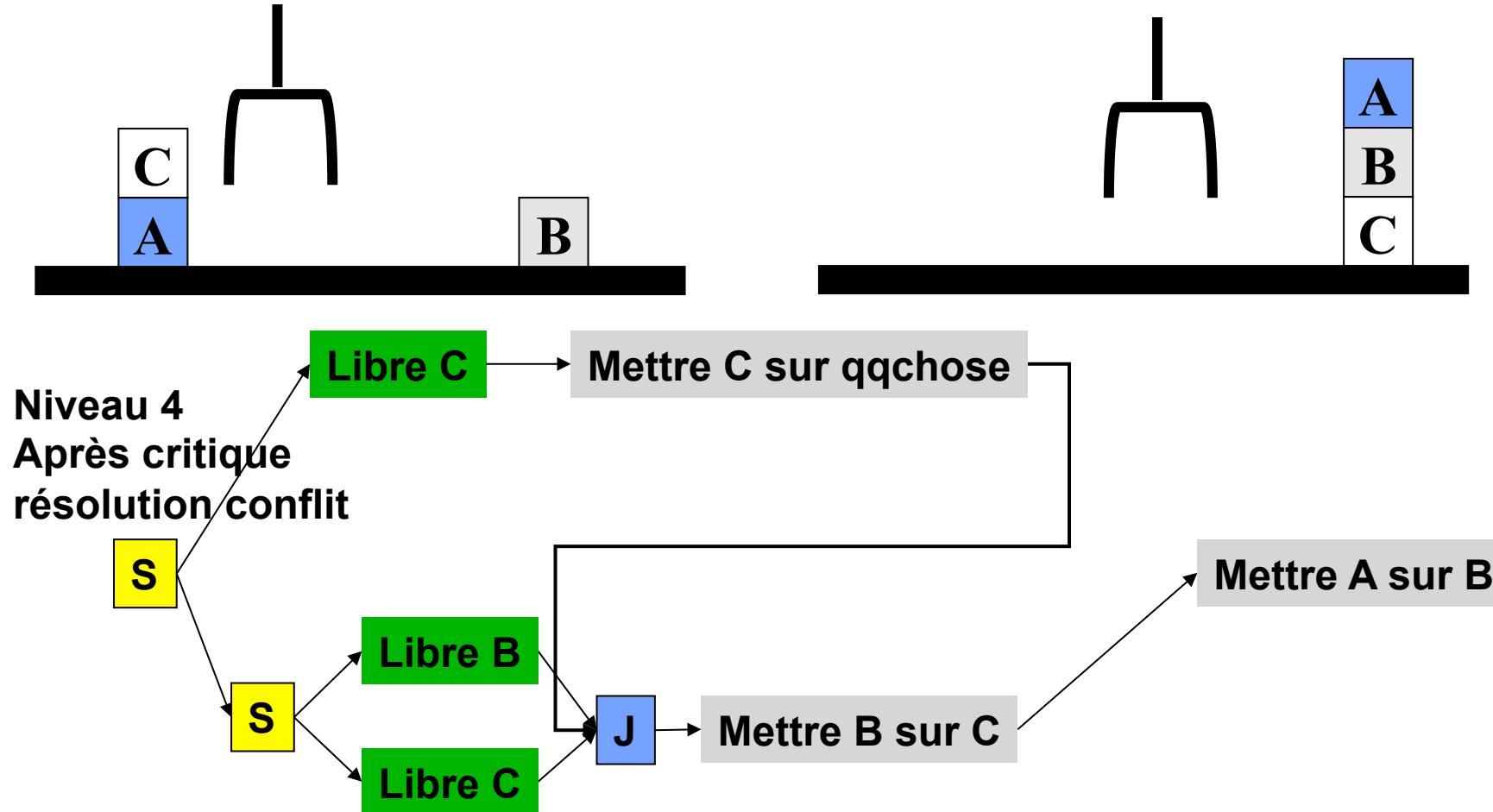
R

S

# Système NOAH – Niveau 4



# Système NOAH – Niveau 4



- **Module « Critique »: pour mettre C sur qqchose, C doit être libre; cette opération doit être effectuée avant mettre B sur C**

L

I

P

6

C

N

R

S

# Système NOAH – Niveau 4

