Master 2 : Intelligence Artificielle

Logique et Algèbres de Processus

Pr. Mustapha Bourahla

Département d'Informatique Université de M'Sila

Le contenu

- Les algèbres de processus
- LOTOS : Language Of Temporal Ordering Specifications
  - Exemple : système téléphonique
  - Equivalence
  - Récursion
  - Autres opérateurs
- Full LOTOS ou LOTOS complet

Pour le TP, nous utilisons le CADP ("Construction and Analysis of Distributed Processes", connu par "CAESAR/ALDEBARAN Development Package").
c'est un outil qui permet de:

Pour compiler une spécification lotos: caesar spec.lotos
Cette commande génère un fichier spec.bcg qui contient un graph en format BCG (Binary-Coded Graphs).

Pour générer le programme C: caesar -exec spec.lotos
Nous pouvons ajouter des options pour le compilateur C à l'aide de l'option -c

Le graphe spec.bcg peut être afficher à l'aide des commandes: bcg_draw -ps spec.bcg (pour générer le postscript spec.ps), puis gs spec.ps (pour l'afficher)
Si l'option -ps n'est pas donnée le bcg_draw affiche le graphe directement

Pour la simulation, nous utilisons la commande bcg_open spec.bcg ocis
ocis - Open/Caesar Interactive Simulator

La commande bcg_open spec1.bcg bisimulator spec2.bcg pour tester l'équivalence (forte, strong par défaut) entre spec1 et spec2

Pour le model-checking, nous utilisons la commande;
bcg_open spec.bcg evaluator4 prop.mcl
Où le fichier prop.mcl contient la propriété selon le langage MCL (Model Checking Language)

Les algèbres

- Les algèbres traitent d'expressions formées de constantes, de variables et d'opérateurs
- Elles sont fournies de règles pour transformer les expressions: simplification, expansion…
- Dans les algèbres de processus, les constantes et les variables représentent des processus

Les algèbres de processus

- Dans les algèbres de processus les systèmes de processus communicants sont représentés par des expressions de caractère algébrique, appelées :
  - **Expressions de comportement, behaviour expressions**
  - A[]B pour dire que A et B sont en alternative, la prochaine action doit être prise ou de l'expression A, ou de l'expression B (l'autre étant ensuite écartée)
    - Parfois aussi écrite A+B
  - A||B pour dire que les processus A et B sont en exécution parallèle, la prochaine action sera prise de A et B conjointement (composition synchrone)
  - Etc.

Propriétés algébriques des expressions de comportement

- Commutativité du choix
  - A[]B = B[]A
- Commutativité de la composition parallèle
  - A||B = B||A
- Zéro absorption
  - A[]stop = stop[]A = A
  - A||stop = stop||A = stop
- Associativité
  - A[](B[]C) = (A[]B)[]C
  - A||(B||C) = (A||B)||C

Expressivité des algèbres de processus

- Les algèbres de processus fournissent des formalismes dans lesquels il est possible de ***prouver*** que la composition de deux processus est égale à un troisième processus

Différentes algèbre de processus

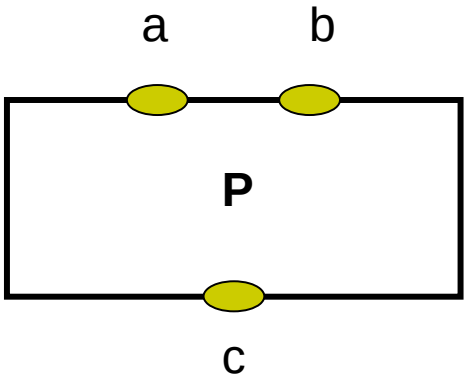
- Malheureusement, il n'y a pas encore accord concernant les algèbres de processus
- Plusieurs algèbres ont été étudiés, et chaque groupe de travail continue à développer la sienne…
- Milner a développé le CCS Calculus of Communicating Processes, dans les années 1970-1980
  - Il a développé ultérieurement ce concept dans le π-calculus
- Hoare a développé le CSP Communicating Sequential Processes, plus ou moins dans les mêmes années
- LOTOS a été développé dans les années 1980
  - C'est le langage choisi pour ce cours

LOTOS

- ***Language Of Temporal Ordering Specifications***
  - Mais sans relation avec les logiques temporelles que nous allons discuter plus tard
- Langage algébrique pour la spec des protocoles
- Inspiré surtout au CCS de Milner, prend quelques éléments du CSP de Hoare
- Norme internationale de l'ISO
- Une nouvelle norme, appelée Extended LOTOS (E-LOTOS) a été développée aussi, mais elle ne fut jamais implémentée (complexe)
- Vaste théorie
- Utilisé en pratique dans un grand nombre d'applications
- Outils et documentation peuvent être obtenu facilement
  - V. sites Web CADP et WELL

Processus LOTOS

- Un processus LOTOS est une boîte ‘noire’ avec des points de contact avec l’environnement
  - Les *portes* ou *gates*

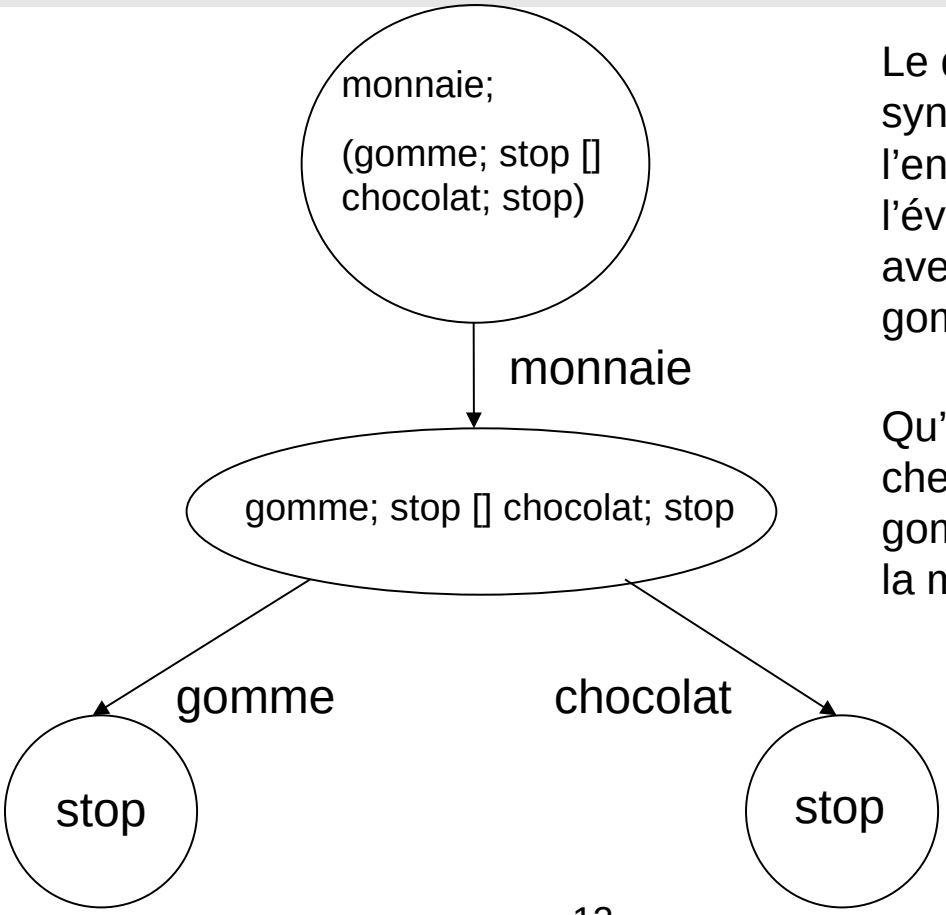



```
process P[a,b,c] :=
        expression de comportement
endproc
```

•L’*expression de comportement, behavior expression*, définit le comportement du système par rapport aux *portes*, l’environnement
•Différents processus ou expressions de comportement peuvent communiquer à travers leur *portes* par composition synchrone (opérateur ||).

Les expressions de comportement décrivent des états

```
process Distributeur [monnaie, gomme, chocolat] :=
        monnaie; (gomme; stop [] chocolat; stop)
endproc
```



Le distributeur est prêt à synchroniser avec l'environnement avec l'événement monnaie, puis avec l'un ou l'autre de gomme ou chocolat

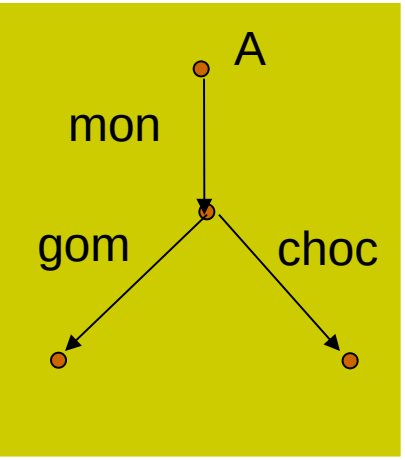
Qu'arrive si l'environnement cherche à toucher à la gomme sans avoir introduit la monnaie?

Action interne i (action tau τ dans CCS)

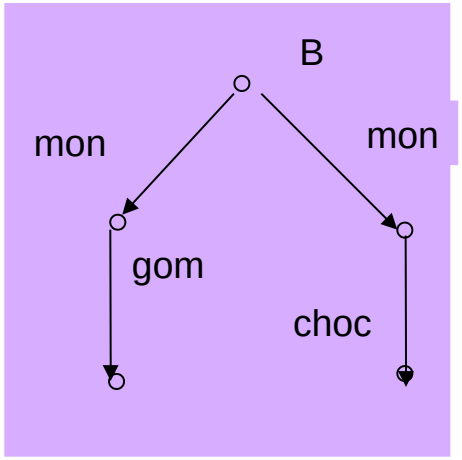
- Les comportements peuvent avoir des actions internes
- Ces actions dénotent un comportement interne de la machine sans vouloir entrer dans les détails
- Détails laissés à des raffinements successifs dans la conception ou dans l'implantation
- L'action interne peut être spécifiée directement
  - mon; (**i**;gom;stop [] choc;stop)
- Ou indirectement (ces deux expressions sont équivalentes)
  - **hide** choc_fini **in** (mon; (choc_fini; gom; stop [] choc;stop))
- L'action interne ne *synchronise* pas avec l'environnement (elle est invisible à l'extérieur)

Comportement non-déterministe

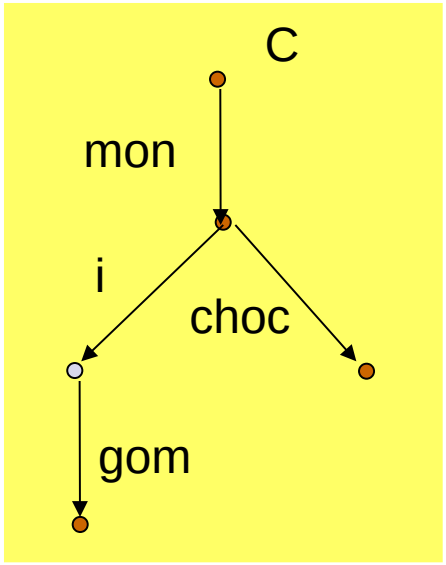
- Voyez-vous la différence? B, C, et D sont des exemples de non-déterminisme



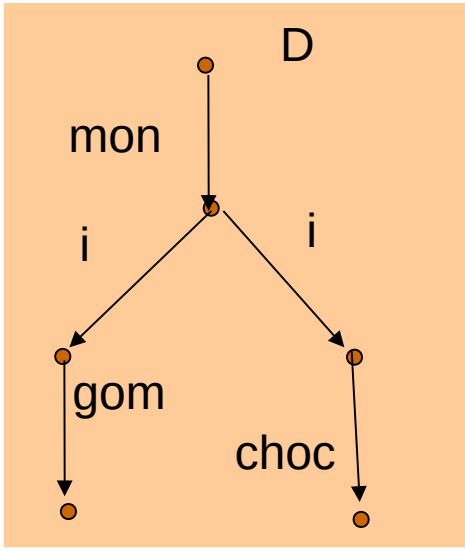
Après mon, A sera capable de synch avec gom ou choc



Après mon, B pourra synch avec l'un de gom ou choc, mais pas l'autre



Après mon, C est capable de synch avec gom, mais pourrait ne pas synch avec choc (après mon, dès qu'il amorce l'action interne, il sera plus capable de faire choc, par contre si gom est proposé, i sera exécuté et puis gom)



Après mon, D décide de synchroniser avec l'un des deux, et pas l'autre (l'environnement ne peut pas distinguer D de B)

En syntaxe LOTOS

- Les expressions LOTOS des quatre arbres de comportement précédents s'écrivent comme suit:
  - A:= mon; (gom;stop [] choc; stop)
  - B := mon; gom ;stop [] mon; choc; stop
  - C := mon; (i; gom; stop [] mon; choc; stop)
  - D := mon; (i; gom; stop [] i; mon; choc; stop)

    - **Les séquences d'actions sont essentiellement les mêmes, cependant il y a des différences concernant les points où on prend les décisions sur ce qui suit**

Action *stop*

- L’action ***stop*** est l’action vide
- Elle ne fait rien, elle n’offre rien à l’environnement
  - Parfois aussi appelée action ***nil***

Opérateurs principaux en LOTOS

- a;B
  - se comporte comme a (une action) puis comme B (une expression de comportement)
- B1 [] B2
  - se comporte ou bien comme B1, ou bien comme B2
- B1 || B2
  - composition synchrone de B1 et B2 (doivent synch sur toutes leurs actions)
- B1 ||| B2
  - entrelacement de B1 et B2
- B1 |[a,b,c…]| B2
  - B1 et B2 doivent synchroniser sur les actions a,b,c et s'entrelacent par rapport aux autres actions
- **hide** a,b,c… **in** B
  - B est exécuté, mais chaque fois qu'une action a, b, c… est exécutée, elle est remplacée pas l'action interne **i**
  - Cette dernière ne peut pas synchroniser avec autres actions

Règles d’inférence

- La sémantique des opérateurs LOTOS est définie de manière précise par des *règles et axiomes d’inférence*

- Disant:
  - étant donnée une expression de comportement
  - une *action*
  - transforme l’expression de comportement dans une autre expression de comportement

Règles d'inférence: préfixe a; B

- Axiome d'inférence pour le préfixe:

$$a; B \xrightarrow{a} B$$

Si nous avons a;B *et a synchronise avec l'environnement*, a;B devient B.

$$monnaie; café; stop \xrightarrow{monnaie} café;stop$$

$$café; stop \xrightarrow{café} stop$$

Règles d'inférence: choix B1 [] B2

$$\frac{B1 \xrightarrow{a1} B1'}{B1 \,[]\, B2 \xrightarrow{a1} B1'}$$

$$\frac{B2 \xrightarrow{a2} B2'}{B1 \,[]\, B2 \xrightarrow{a2} B2'}$$

La règle de gauche dit:

Si B1 peut synchroniser avec l'environnement sur l'action a1 donnant B1' comme résultat

Alors B1[]B2 peut synchroniser avec l'environnement sur l'action a1 donnant B1' comme résultat

L'alternative qui a été choisie est continuée

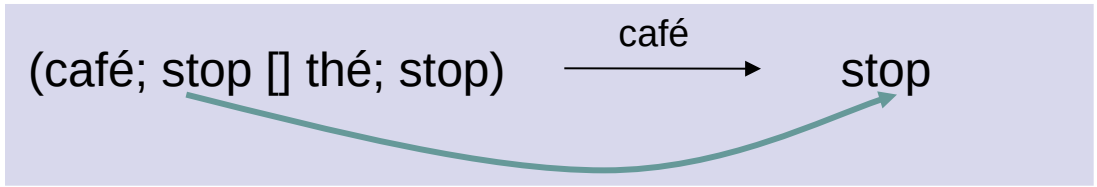
L'autre est écartée

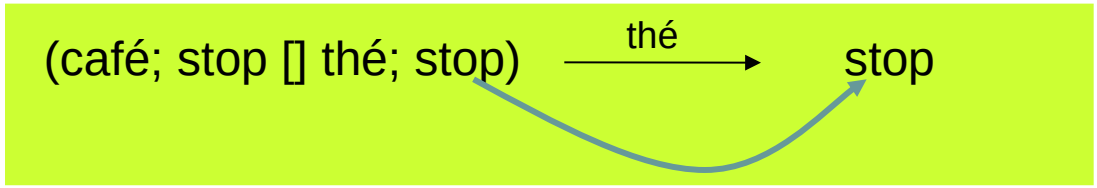
Quand plusieurs règles sont données pour un seul comportement, une seule est exécutée chaque fois

Exemple de []

$$\frac{B1 \xrightarrow{a1} B1'}{B1 \,[]\, B2 \xrightarrow{a1} B1'}$$

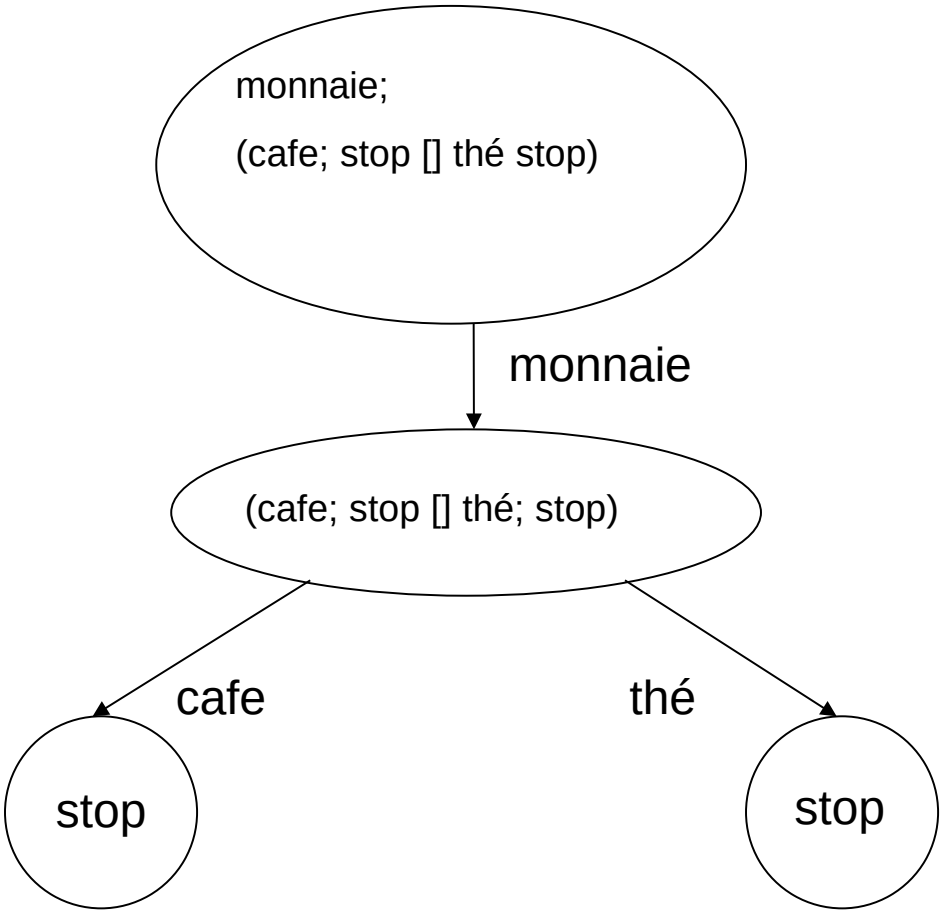
$$\frac{B2 \xrightarrow{a2} B2'}{B1 \,[]\, B2 \xrightarrow{a2} B2'}$$





Ces deux possibilités sont exclusives: si l'environnement décide d'effectuer café, l'autre possibilité n'existe plus, et vice-versa

Arbre de comportement:



Exercice n° 1: Lois algébriques pour []

- Les lois algébriques de LOTOS sont une conséquence des règles d'inférence
- Persuadez-vous que ceci est vrai pour les trois lois que nous avons mentionné pour l'opérateur []
  - A[]B = B[]A
  - A[]stop = stop[]A = A
  - A[](B[]C) = (A[]B)[]C

Attention: non-distributivité de ; par rapport à []

- Conséquence de ce que nous disions avant au sujet du nondéterminisme:

a; (b; stop [] c;stop) ≠ a; b; stop [] a; c; stop

Aussi dans le cas où a = i

À noter que le choix de l'action a dans le comportement de gauche transforme ce dernier en (b; stop [] c; stop)
Tandis que le choix de a dans le comportement de droite le transforme ou bien en b; stop ou bien en c; stop, selon l'a qui est choisi

Règles d'inférence: composition synchrone

B1 || B2

$$\frac{B1 \xrightarrow{a} B1' \qquad B2 \xrightarrow{a} B2'}{B1 \,||\, B2 \xrightarrow{a} B1' \,||\, B2'}$$

Si deux expressions de comportement sont prêtes à synchroniser sur une action **a** donc elles peuvent produire une action **a** commune et puis exécuter ce qui reste

Attention: il n'y a pas de synchronisation sur l'action interne **i**

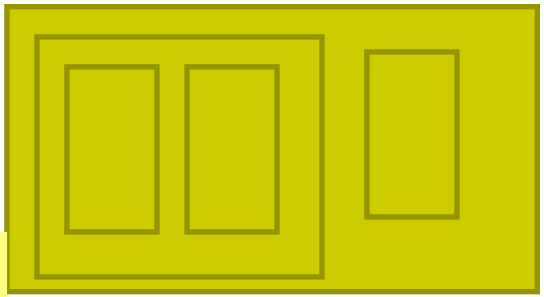
Ni sur le stop évidemment

Exemple

- Marie et Abdel mangent toujours ensemble
  - Ils ont trois actions: Petit dejeuner, dejeuner, souper
  - Marie:= pd; d; s; stop
  - Abdel:= pd; d; s; stop
  - Donc Marie || Abdel = pd; d; s; stop

- Cependant si Abdel n'a pas l'habitude de déjeuner:
  - Marie:= pd; d; s; stop
  - Abdel:= pd; s; stop
  - Donc Marie || Abdel = pd; stop
    - **Après le pd il ne réussissent pas à se mettre d'accord donc impasse!**

Trois identités intéressantes



((a; stop || a; stop) || a; stop) = a; stop

Les deux premiers a synchronisent et offrent un a qui synchronise avec le troisième a.

Donc ce comportement offre l'action a à l'environnement.

Le même résultat pour (a… || (a… || a…)) (donc associativité de ||)

((**hide** a **in** (a; stop || a; stop)) || a; stop) = **i; stop**

Les deux premiers a synchronisent mais l'action a résultante est cachée et devient **i**. Cette action interne est incapable de synchroniser avec le troisième a, qui donc ne peut pas être exécuté.

(**hide** a **in** ((a; stop || a; stop) || a; stop)) = **i; stop**

Les trois a synchronisent mais l'action a résultante est cachée et devient **i**.

Règles d'inférence: entrelacement (interleave)

B1 ||| B2

$$\frac{B1 \xrightarrow{a1} B1'}{B1 \,|||\, B2 \xrightarrow{a1} B1' \,|||\, B2}$$

$$\frac{B2 \xrightarrow{a2} B2'}{B1 \,|||\, B2 \xrightarrow{a2} B1 \,|||\, B2'}$$

Une action est sélectionnée d'un des deux comportements, et exécutée

L'autre partie du comportement peut encore être sélectionnée plus tard

Exemple

- Marie et Abdel n'ont rien à faire l'un avec l'autre
  - Ils ont deux actions: Petit dejeuner, dejeuner
  - Marie:= pd; d; stop
  - Abdel:= pd; d; stop
  - Donc Marie ||| Abdel =

pd ; (d; pd; d; stop [] pd; (d; d; stop [] d; d; stop))
[]
pd ; (d; pd; d; stop [] pd; (d; d; stop [] d; d; stop))

= pd; (d; pd; d; stop [] pd; d; d; stop)

La simplification a été faite utilisant deux fois la loi A[]A=A.
On ne pourrait pas simplifier si on faisait distinction entre les actions de Marie et Abdel: pd_M, pd_A etc. Cependant ce résultat est logique du point de vue d'un cuisinier qui doit préparer des pd ou d identiques pour les deux.

Exercice n° 2: Lois algébriques pour ||

- Les lois algébriques de LOTOS sont une conséquence des règles d'inférence
- Persuadez-vous que ceci est vrai pour les trois lois que nous avons mentionné pour l'opérateur ||
  - A||B = B||A
  - A||stop = stop||A = stop
  - A||(B||C) = (A||B)||C

Enfin, si vous faites le travail vraiment bien, vous verrez que la deuxième loi n'est pas vraie dans un cas particulier…

Parallélisme général B1|[A]|B2

- Synchronisation sur quelques actions (ensemble A), entrelacement sur d'autres:
  - combine les règles des compositions synchrone et asynchrone

$$\frac{B1 \xrightarrow{a} B1' \qquad B2 \xrightarrow{a} B2'}{B1 \,|[A]|\, B2 \xrightarrow{a} B1' \,|[A]|\, B2'} \text{ si } a \in A$$

Synchronisation sur les portes de l'ensemble A (A est une liste de portes)

$$\frac{B1 \xrightarrow{a1} B1'}{B1 \,|[A]|\, B2 \xrightarrow{a1} B1' \,|[A]|\, B2} \text{ si } a1 \notin A$$

$$\frac{B2 \xrightarrow{a2} B2'}{B1 \,|[A]|\, B2 \xrightarrow{a2} B1 \,|[A]|\, B2'} \text{ si } a2 \notin A$$

Entralacement sur les portes qui ne sont pas dans A

Exemple

- Marie et Abdel font séparément le petit déjeuner et le souper, cependant ils déjeunent toujours ensemble
  - Marie:= pd; d; s; stop
  - Abdel:= pd; d; s; stop
  - Donc Marie |[d]| Abdel =

pd; pd; d; (s; s; stop [] s; s; stop)
[] pd; pd; d; (s; s; stop [] s; s; stop)

= pd; pd; d; s; s; stop

Si Abdel n’a pas l'habitude de dejeuner, le résultat est:
pd; d; s; stop |[d]| pd; s; stop = pd; pd; stop

Exemple

(a; b; stop [] c; d; stop) |[a,b]| (a; b; stop [] d; f; stop)

=

a; b; stop [] (c; d; stop ||| d; f;
stop)

=

a; b; stop []
(c; (d; d; f; stop [] d; (d; f; stop [] f; d; stop)) []
d; (c; (d; f; stop [] f; d; stop) [] f; c; d; stop))

Exercice: Dessiner les arbres pour comprendre mieux!

Exemple

Actions identiques qui ne sont pas dans l'ensemble de synch entrelacent

a; b; c; stop |[b]| a; b; d; stop

=

a; a; b; (c; d; stop [] d; c; stop)
[]
a; a; b: (c; d; stop [] d; c; stop)

=

(à cause de la loi A[]A =A)

a; a; b; (c; d; stop [] d; c; stop)

Exécution des règles d'inférence

- L'exécution d'une spec LOTOS transforme la spec en utilisant les règles d'inférence
- La spec courante (représentant l'état global courant) peut être transformée en utilisant n'importe quelle règle applicable
- L'arbre qui représente toutes les transformations possibles est le système de transition étiqueté du système
- Il est aussi l'arbre d'accessibilité montrant toutes les transitions possibles d'état du système spécifié
- Cet arbre peut aussi être représenté comme expression LOTOS
- L'impasse-deadlock est le cas où aucune règle d'inférence ne peut être appliquée
- Impasse et **stop** sont exactement la même chose en LOTOS:
  - Il n'y a pas de règles d'inférence pour **stop**

Quelques exemples additionnels

pour étude individuelle – dessiner les arbres

(a; b; stop) |[b]| (c; b; stop) = (a; c; b; stop) [] (c; a; b; stop)

(i; b; stop) |[b]| (c; b; stop) = (i; c; b; stop) [] (c; i; b; stop)

(i; b; stop) |[b]| (i; b; stop) = (i; i; b; stop) [] (i; i; b; stop) = (i;i;b;stop)

(a; b; stop) |[b]| (b; c; stop) = a; b; c; stop

(a; b; stop) |[a,b]| (b; a; stop) = stop = (a; b; stop) || (b; a; stop)

Impossibilité d'accord sur la 1ère action

(a; b; stop [] d; f; stop) |[a,b]| (a; b; c; stop [] i; stop) =

(a; b; c; stop [] d; (f; i; stop [] i; f; stop) [] i; d; f; stop)

Labelled Transition System

A **Labelled Transition System** Sys is a 4-tuple $<S, A, T, s_0>$ where

(i) S is a non-empty set of **states**,

(ii) A is a set of **actions**,

(iii) T is a set of **transition relations** $T_a \subseteq S \times S$, one for each $a \in A$;

T_a is a set of **transitions** of the form: cur $\xrightarrow{a}$ next, where cur, next $\in S$

(iv) $s_0 \in S$ is the **initial state** of Sys.

A **state** is unambiguously identified by a **behaviour expression**

An **action** is of the form $\mathbf{gv_1 \ldots v_n}$ where g is a gate name and the v_i are values of some sort

We define: name $(gv_1 \ldots v_n) = g$

There is a distinguished (internal) action: i, which has no associated value.

There is a distinguished (terminating) gate name: δ

But we will consider first Basic LOTOS (without data types)

Operational semantic rules for Basic LOTOS

$$a;P \xrightarrow{a} P \qquad \text{exit} \xrightarrow{\delta} \text{stop} \qquad \frac{P \xrightarrow{a} P'}{P \,[]\, Q \xrightarrow{a} P'}$$

$$\frac{P \xrightarrow{a} P'}{P \,|[\Gamma]|\, Q \xrightarrow{a} P' \,|[\Gamma]|\, Q} (a \notin \Gamma \cup \{\delta\}) \qquad \frac{P \xrightarrow{a} P',\ Q \xrightarrow{a} Q'}{P \,|[\Gamma]|\, Q \xrightarrow{a} P' \,|[\Gamma]|\, Q'} (a \in \Gamma \cup \{\delta\})$$

$$\frac{P \xrightarrow{a} P'}{\text{hide } \Gamma \text{ in } P \xrightarrow{a} \text{hide } \Gamma \text{ in } P'} (a \notin \Gamma) \qquad \frac{P \xrightarrow{a} P'}{\text{hide } \Gamma \text{ in } P \xrightarrow{i} \text{hide } \Gamma \text{ in } P'} (a \in \Gamma)$$

$$\frac{P \xrightarrow{a} P'}{P >> Q \xrightarrow{a} P' >> Q} (a \neq \delta) \qquad \frac{P \xrightarrow{\delta} P'}{P >> Q \xrightarrow{i} Q}$$

$$\frac{P \xrightarrow{a} P'}{P \,[>\, Q \xrightarrow{a} P' \,[>\, Q} (a \neq \delta) \qquad \frac{P \xrightarrow{\delta} P'}{P \,[>\, Q \xrightarrow{\delta} P'} \qquad \frac{Q \xrightarrow{a} Q'}{P \,[>\, Q \xrightarrow{a} Q'}$$

$$\frac{P[g_1/h_1, \ldots g_n/h_n] \xrightarrow{a} P',\ Q[h_1, \ldots h_n] := P}{Q\,[g_1, \ldots g_n] \xrightarrow{a} P'}$$

Exercice n° 3:

En utilisant les règles d’inference dessiner les arbres (STE) de

process one [a,b,c]

a; (b; stop [] c; stop)

endproc

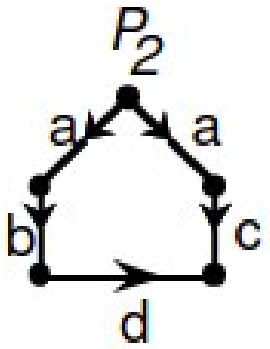
process two [a,b,c]

a; b; stop [] a; c; stop

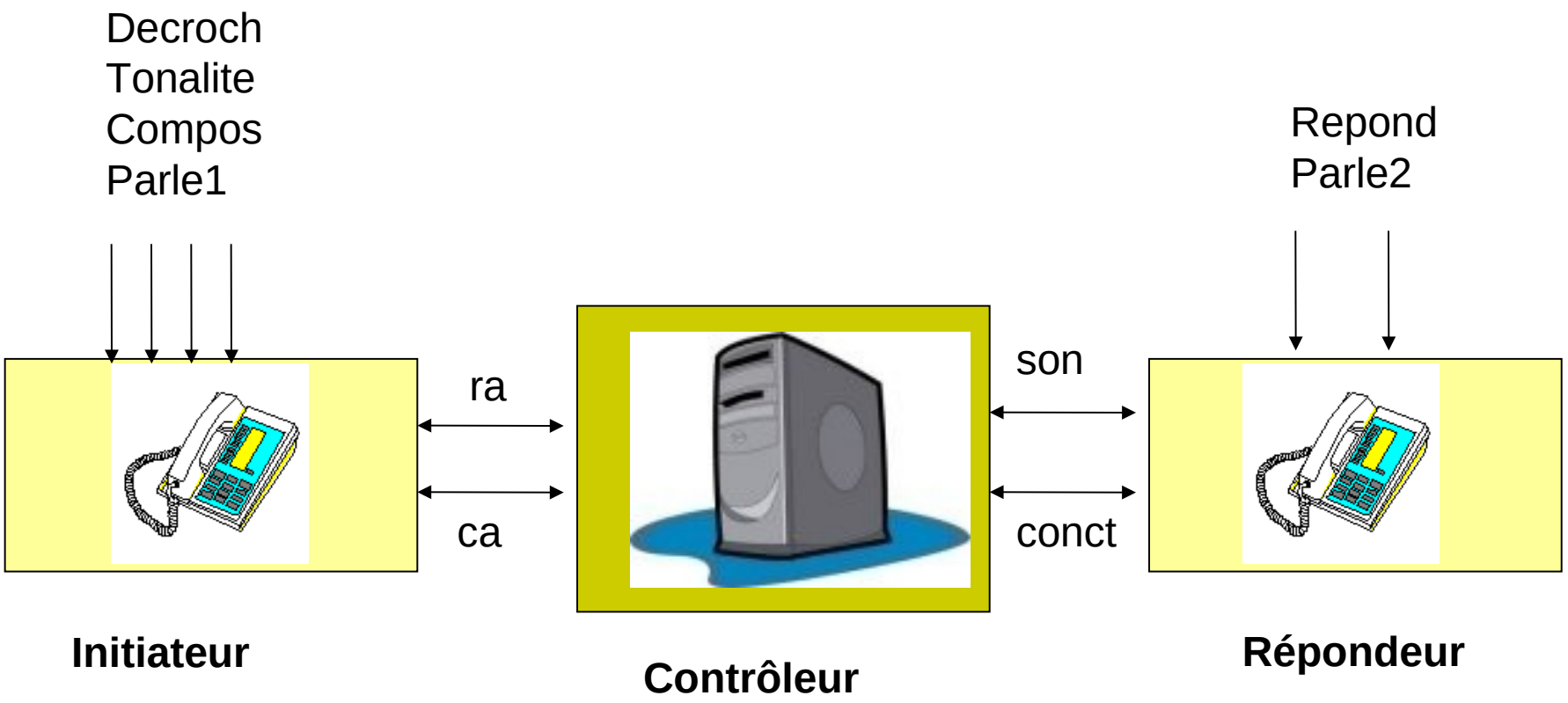
Endproc

P1 := a; (b; d; stop [] c; stop)

P2 := a; b; d; stop [] a; c; stop



Exemple: système téléphonique



Exemple: système téléphonique

Syntaxe simplifiée…

process phone:= initiateur |[ra,ca]| controleur |[son,conct]| repondeur

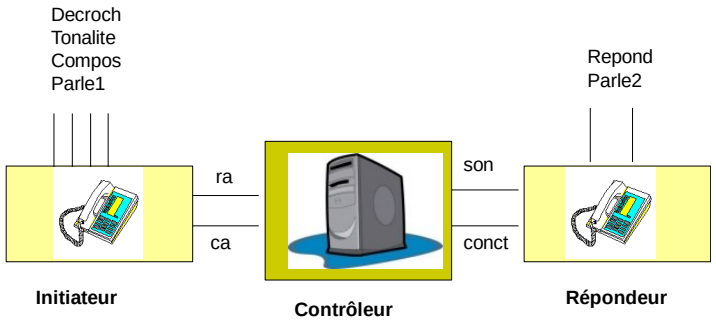
process initiateur:= decroch; tonalite; compos; ra; ca; parle1; stop

process controleur:= ra; son; conct; ca; stop

process repondeur:= son; repond; conct; parle2; stop

Une séquence d'exec. possible:

decroch; tonalité; compos; ra; son; repond; conct; ca; parle1; parle2; stop



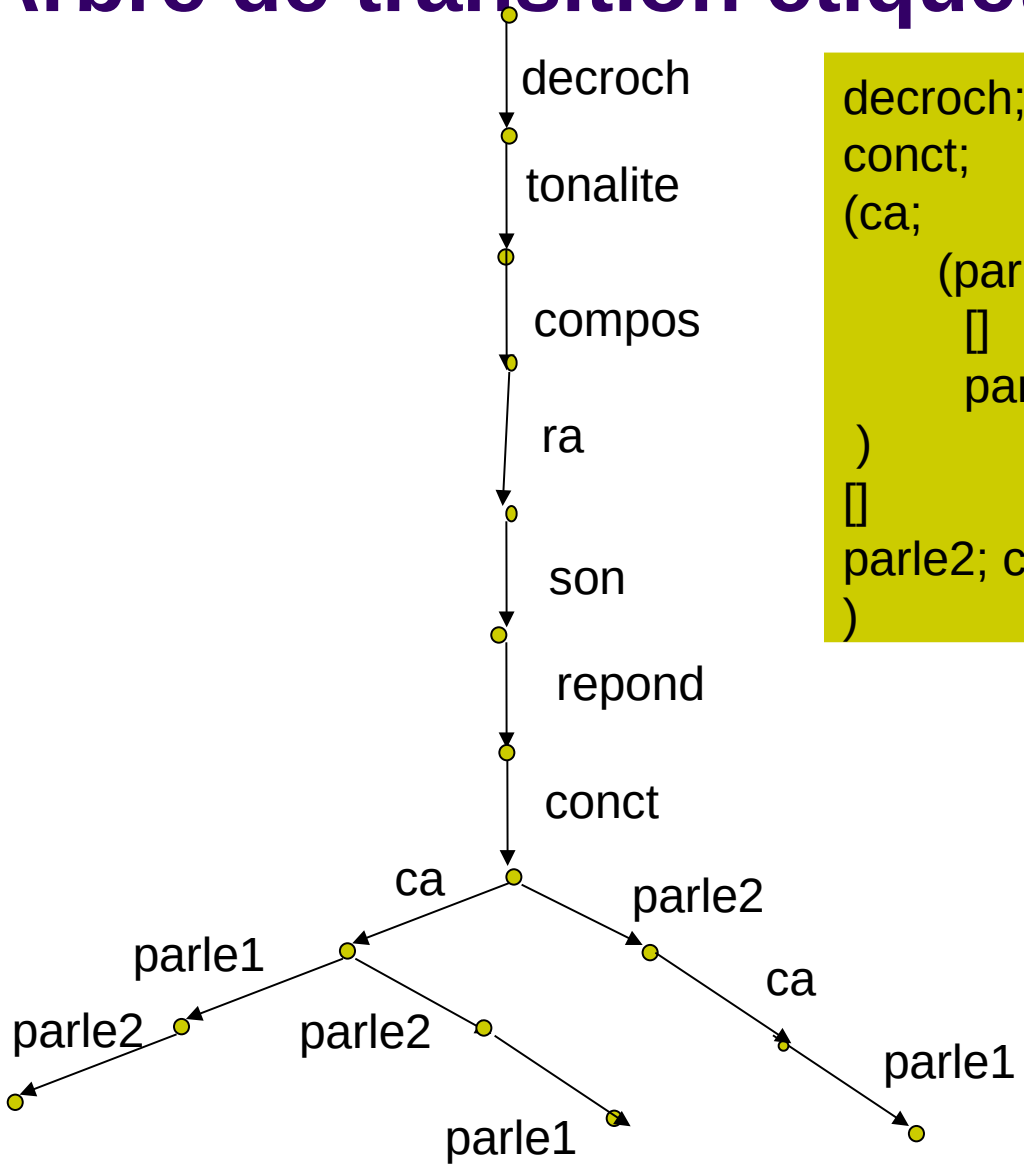
Expansion de la spec téléphonique

- Montre en forme de spec LOTOS *toutes* les séquences possibles

```
decroch; tonalite; compos; ra; son; repond; conct;
(ca;
       (parle1; parle2; stop
         []
         parle2; parle1; stop)
 )
[]
parle2; ca; parle1; stop
)
```

Cette *expansion* montre une situation imprévue dans le protocole!

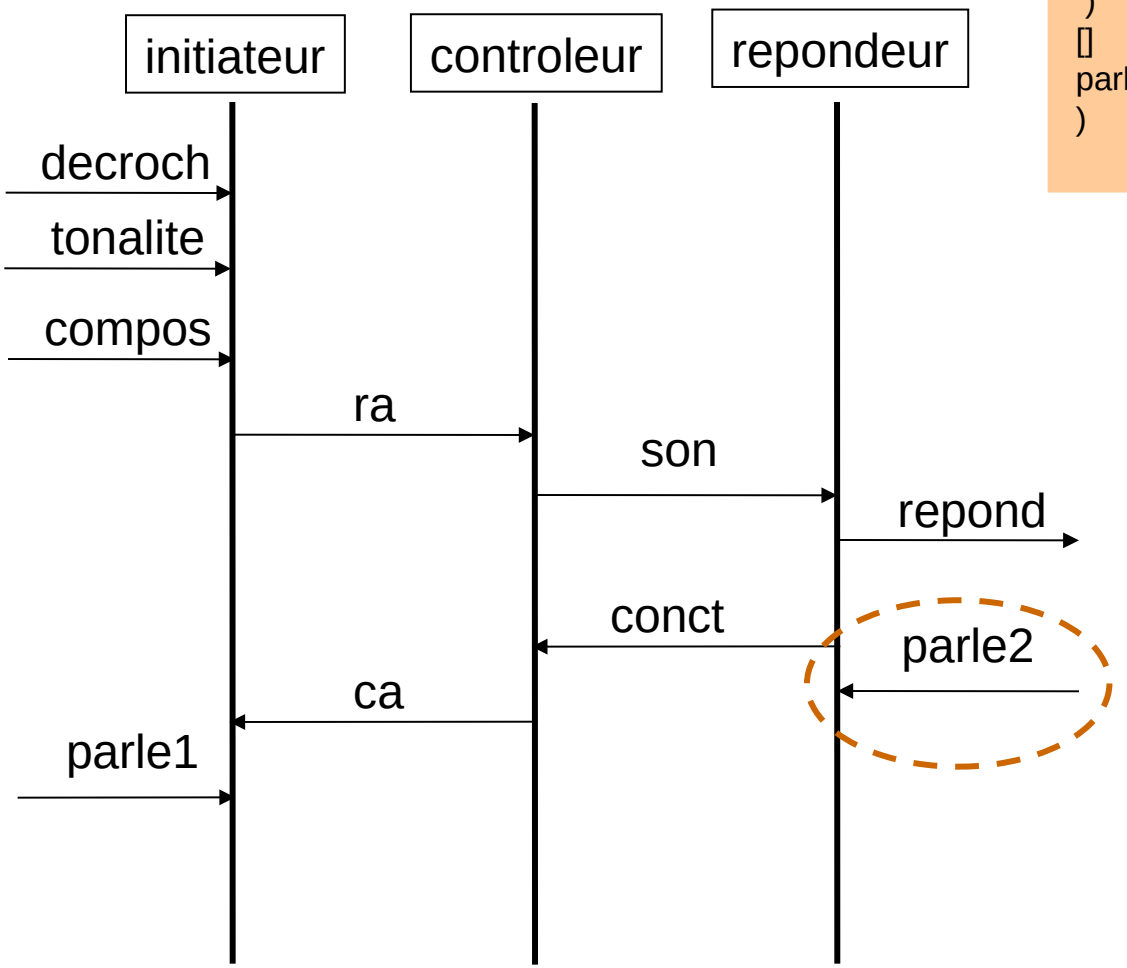
Arbre de transition étiqueté




```
decroch; tonalite; compos; ra; son; repond;
conct;
(ca;
      (parle1; parle2; stop
        []
        parle2; parle1; stop)
 )
[]
parle2; ca; parle1; stop
)
```

Message Sequence Chart

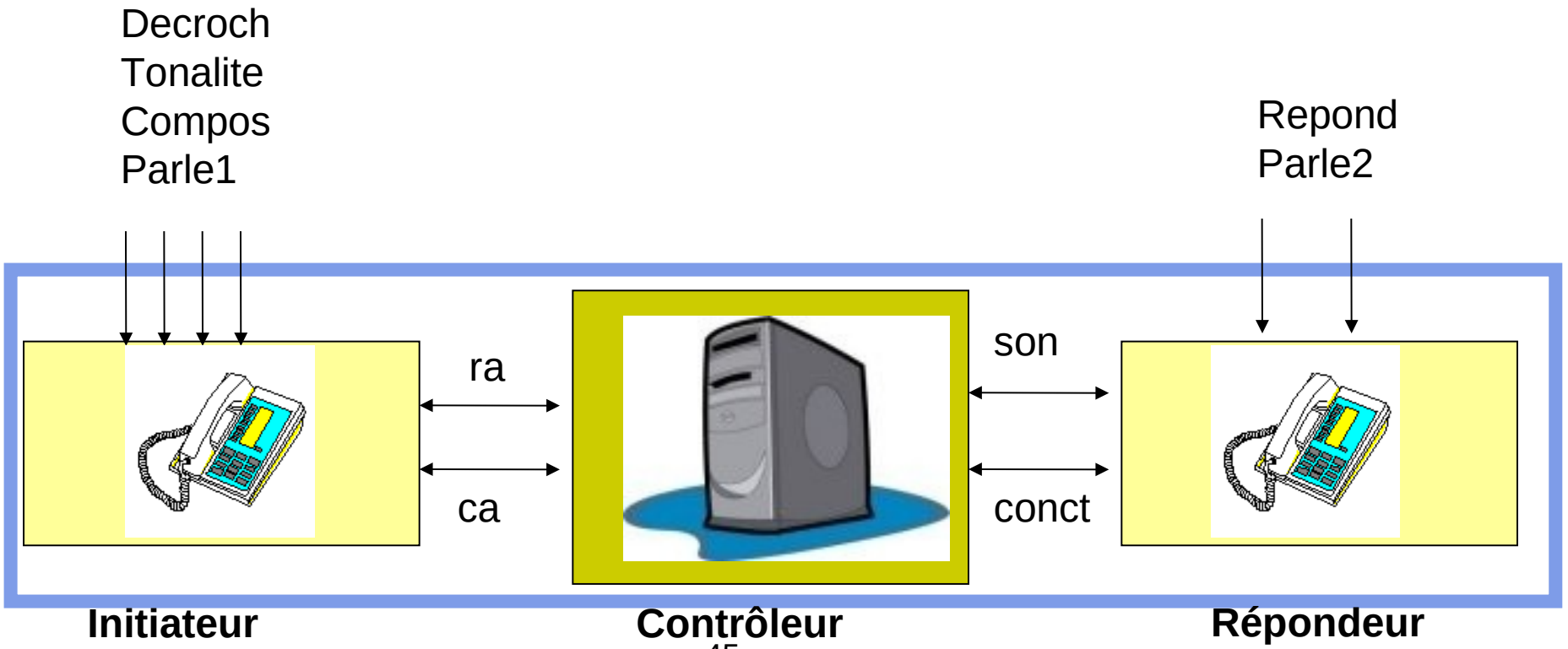
```
decroch; tonalite; compos; ra; son; repond; conct;
(ca;
      (parle1; parle2; stop
       []
       parle2; parle1; stop)
 )
[]
parle2; ca; parle1; stop
)
```



Possibilité de perte de *parle2* car la connexion pourrait ne pas être encore complètement établie.
Malheureusement ceci est inévitable car les trois entités ne peuvent pas communiquer directement et simultanément.
Le problème n'est pas grave si on pense que la séquence *conct ca* sera exécutée très rapidement, sans donner au répondeur le temps de parler

Masquage d'événements

- La manière qu'on a fait, l'environnement (l'usager) doit participer à tous les événements!
  - Nous cacherons maintenant les événements entre les téléphones et le contrôleur



Masquage d'événements dans le système
(utilisant la spec initiale)

process phone:= **hide** ra, ca, conct **in**

(initiateur |[ra,ca]| controleur |[son,conct]| repondeur)

=

decroch; tonalité; compos; **i**; son; repond; **i**;
(**i**;
(parle1; parle2; stop
[]
parle2; parle1; stop)
)
[]
parle2; **i**; parle1; stop
)

Montre seulement le comportement visible du téléphone

decroch; tonalite; compos; ra; son; repond; conct;
(ca;
(parle1; parle2; stop
[]
parle2; parle1; stop)
)
[]
parle2; ca; parle1; stop
)

Élimination d'actions internes

- Les actions internes peuvent être éliminées, sauf les premières après un choix!

```
decroch; tonalité; compos; son; repond;
(i;
      (parle1; parle2; stop
        []
        parle2; parle1; stop)
 )
[]
parle2; parle1; stop
)
```

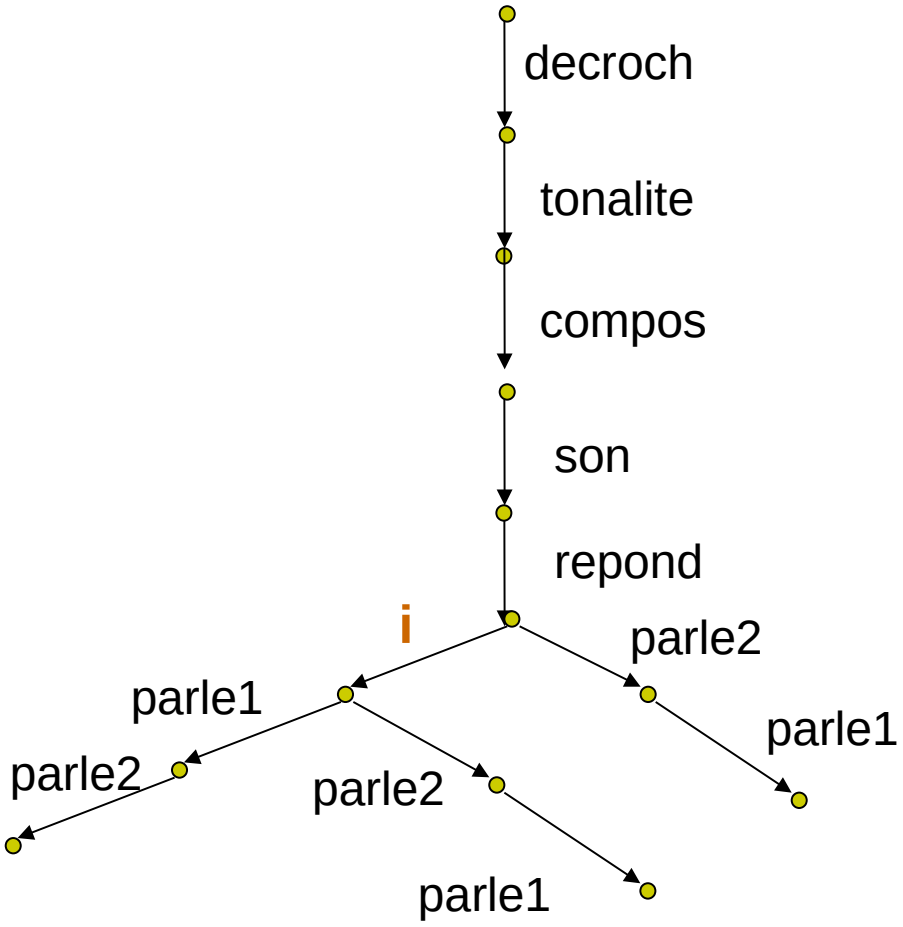
```
decroch; tonalité; compos; i; son; repond; i;
(i;
      (parle1; parle2; stop
        []
        parle2; parle1; stop)
 )
[]
parle2; i; parle1; stop
)
```

observez le signal ca est devenu interne: i

il peut enlever à 2 la possibilité de parler sans attendre le signal interne ca

donc l'action interne **i** qui reste est significative et ne peut pas être enlevée

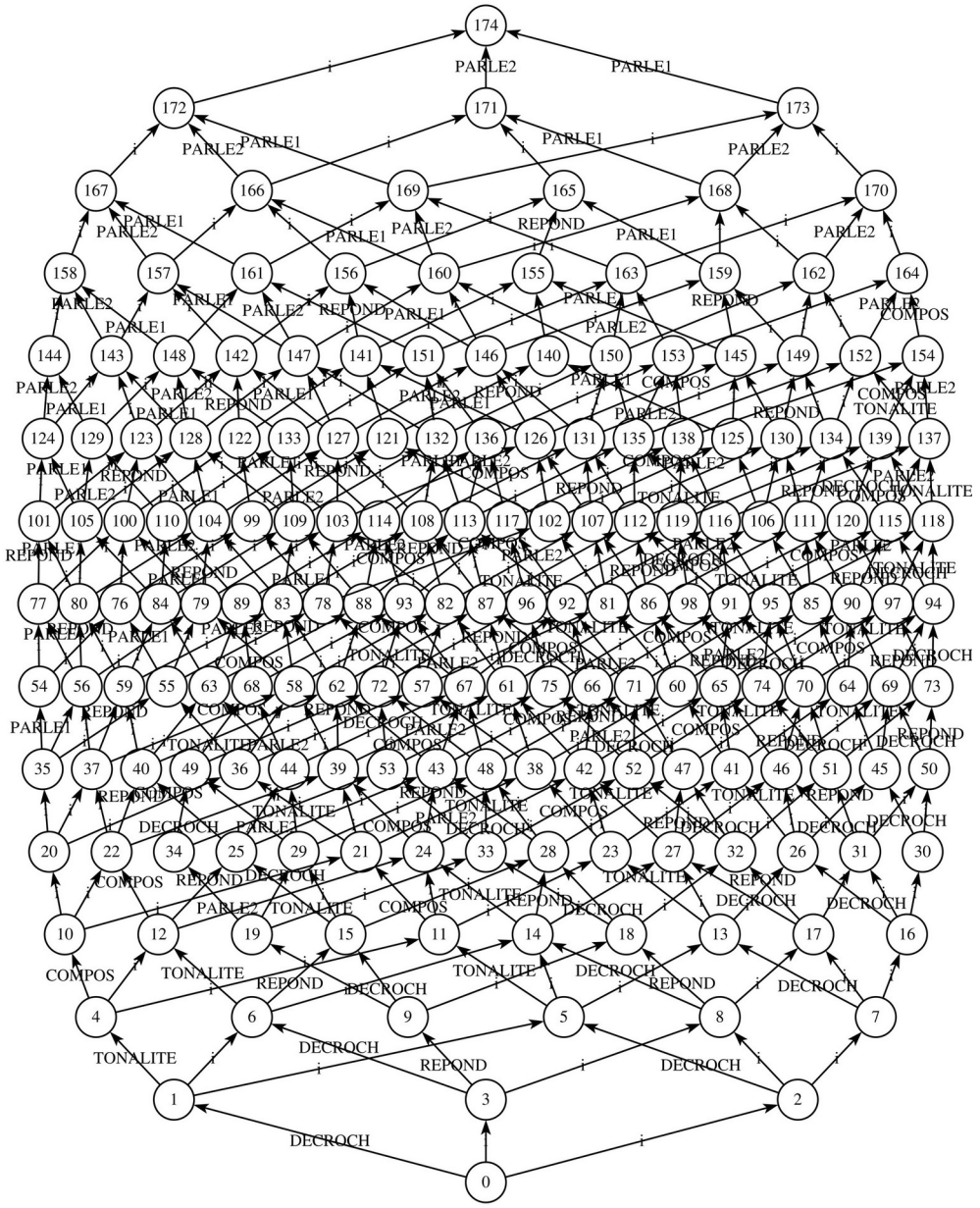
Arbre de transition étiqueté



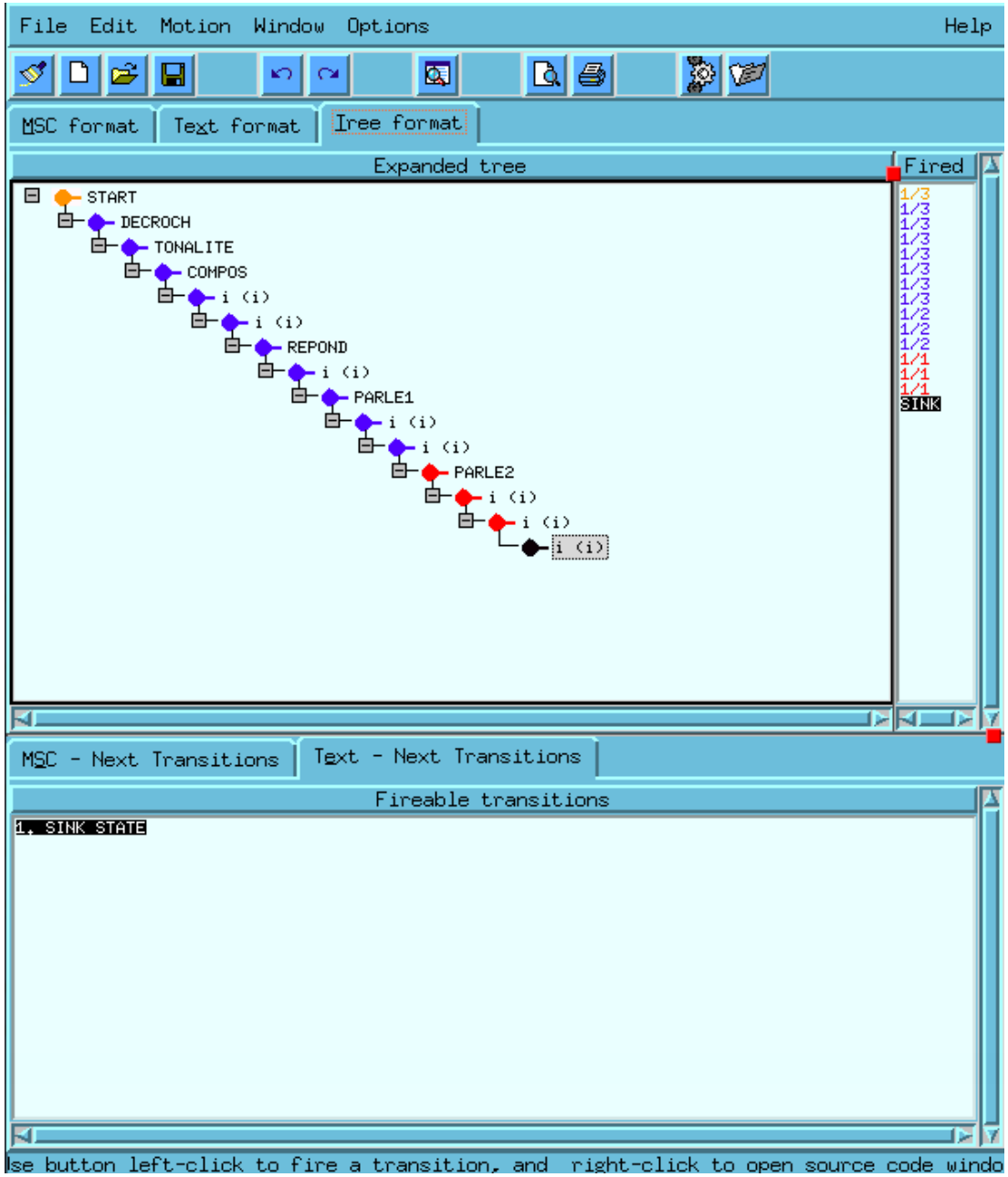

```
decroch; tonalité; compos; son; repond;
(i;
      (parle1; parle2; stop
        []
        parle2; parle1; stop)
 )
[]
parle2; parle1; stop
)
```

```
specification phone [decroch, tonalite, compos, parle1, repond, parle2] : noexit behaviour
        hide ra, ca, son, conct in
                (initiateur[decroch, tonalite, compos, parle1]
                        |[ra, ca]|
                controleur
                        |[son, conct]|
                repondeur[repond, parle2])
  where
    process initiateur [decroch, tonalite, compos, parle1] : noexit :=
        hide ra, ca in
                (decroch, tonalite, compos, ra, ca; parle1; stop)
    endproc
    process controleur : noexit :=
        hide ra, ca, son, conct in (ra; son; conct; ca; stop)
    endproc
    process repondeur [repond, parle2] : noexit :=
        hide son, conct in (son; repond; conct; parle2; stop)
    endproc
endspec
```

../cadp/com/bcg_draw phone.bcg



../cadp/com/bcg_open phone.bcg ocis



< parle1 * > true and < parle2 * > false

../cadp/com/bcg_open phone.bcg evaluator4 phone_prop.mcl
bcg_open: using ``/home/bourahla/cadp/bin.x64/evaluator4.a''
bcg_open: running ``evaluator4 phone_prop.mcl'' for ``./phone.bcg''

FALSE

Théorème de l'expansion

- Toute expression LOTOS, impliquant n'importe quels opérateurs, peut être réécrite comme expression LOTOS
  - impliquant seulement **[]** et **;**
  - cependant cette *expansion* peut être infinie
  - cette *expansion* représente l'arbre d'accessibilité de l'expression originale

Information architecturale

- Pourquoi avons-nous écrit la spec originale en utilisant des processus parallèles
  - Au lieu de l'écrire dans sa forme 'expanded'?
  - Parce que la première spec contenait des informations architecturales qui sont perdues dans l'expansion
    - **Les trois composantes ont disparu dans l'expansion**
    - **Mais l'expansion est utile pour la vérification**

Équivalence

- La théorie de LOTOS propose un certain nombre de notions d'équivalence
- La plus utilisée est appelée
  - **Equivalence observationnelle** ou **bisimulation faible**
  - Elle permet de simplifier les expressions LOTOS en enlevant des alternatives répétées et des actions internes
  - Attention: une action interne peut être due au cachement (hide) d'une action visible à un niveau plus détaillé
  - En général, une action interne peut être éliminée si elle n'est pas dans un contexte de choix

Bisimulation forte et faible

- Deux comportements sont **fortement bisimilaires** si pour chaque action que l'un peut offrir, l'autre peut offrir la même action et les deux se transforment dans deux comportements qui sont aussi fortement bisimilaires
  - Une loi de bisimulation forte que nous avons utilisée: **A [] A ~ A**
  - Je n'aurai pas dû écrire A [] A = A
- Deux comportement sont **faiblement bisimilaires** si pour chaque *séquence* d'actions *visibles (non-interne)* que l'un peut offrir, l'autre peut offrir la même séquence d'actions et les deux se transforment dans deux comportements qui sont aussi faiblement bisimilaires
- Ces deux concepts sont importants *bien au delà* des algèbres de processus et LOTOS
  - Faire recherche Web sur le mot 'bisimulation' (ou 'bissimulation')

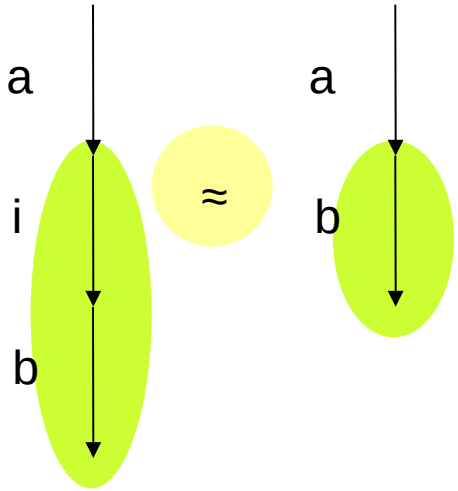
Bisimulation faible

- Deux processus P et Q sont en relation de bisimulation *faible* si:
  - Chaque fois que **P** peut proposer une ***séquence d'actions externes* s** se transformant dans un processus **P'**
  - Q peut aussi proposer **s** se transformant dans un processus **Q'** qui est en relation de bisimulation faible par rapport a **P'**
  - Et viceversa, échangeant **P** et **Q**
  - Les actions internes sont ignorées

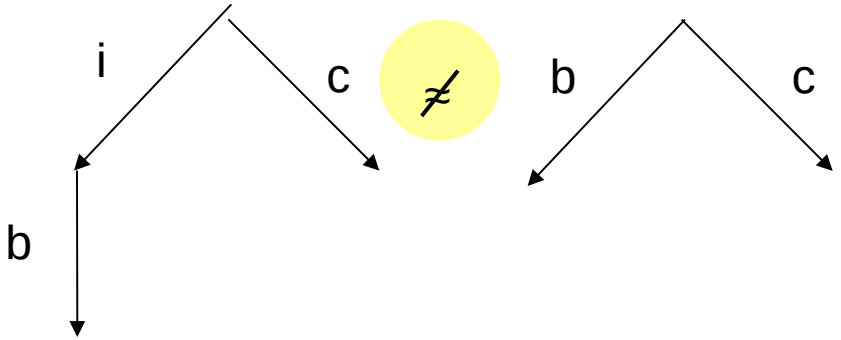
- La chose essentielle à savoir est que pour réduire une expression ou un arbre d'accessibilité contenant actions internes, il est possible d'éliminer toutes les actions internes qui *ne sont pas* des premières actions dans un choix
  - V. exemple précédent

Bisimulation faible ≈, exemples

- En principe:



a; i; b; … ≈ a; b; …



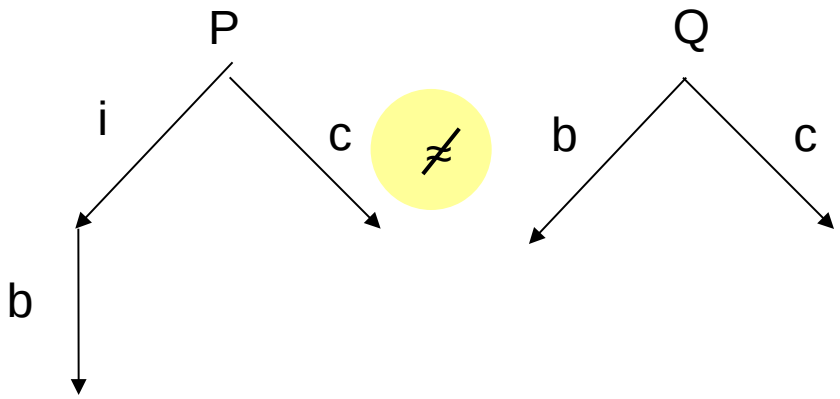
(i;b … [] c …) ≠ (b… [] c…)

(b … [] i; c …) ≠ (b… [] c…)

a; (i;b … [] c …) ≠ (a;(b… [] c…)

a; (i;b … [] i; c …) ≠ (a;(b… [] c…)

Pourquoi



Prenons la séquence d'actions externes *vide*.

Pour cette séquence, P (executant i) peut se transformer dans le comportement b;stop qui n'offre que b.

Pour cette même séquence, Q ne peut rien faire et se transforme en lui-même.

b;stop et (b;stop []c;stop) ne sont pas faiblement bisimilaires car le premier ne peut pas proposer c

Donc P et Q ne sont pas faiblement bisimilaires.

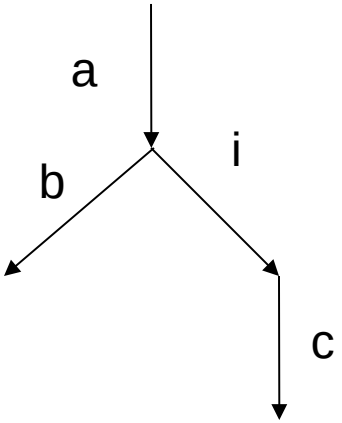
Équivalence de test

- Considère seulement les résultats d'interactions d'un système avec son environnement, sans considérer les états internes

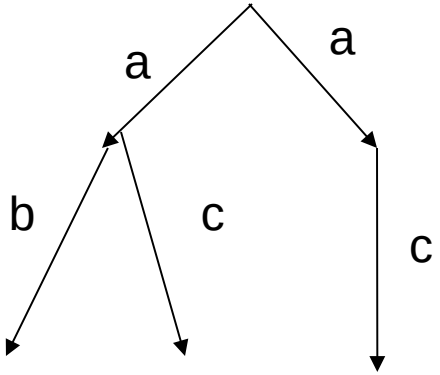
- Une caractérisation précise de l'équivalence de test existe, mais elle est plus difficile à expliquer (et à calculer) que la bisimulation

Relation entre bisimulation et équivalence de test

- On peut prouver que si deux expressions de comportement sont bisimilaires, alors on ne peut pas les distinguer par des séquences de test
- Cependant il y a des cas où deux processus ne sont pas bisimilaires, et encore on ne peut pas les distinguer par test (voir l'exemple ci-dessous)



a; (b;stop [] i; c; stop)



a; (b;stop [] c;stop) [] a; c; stop

Ces comportements ne sont pas bisimilaires, MAIS

Si l'environnement fait a;c

Ceci est toujours accepté par les deux

Si l'environnement fait a;b

Ceci pourrait être accepté ou non par les deux

Donc,

Si deux comportement sont bisimilaires, ils sont équivalents pour le test

Mais des comportements peuvent être équivalents pour test, mais pas bisimilaires

Récursion

- La récursion est interprétée comme remplacement d'un identificateur de processus par sa définition:

A := a; A
=
a; a; A
=
a; a; a; A

Etc.

Un exemple de comportement non-déterministe

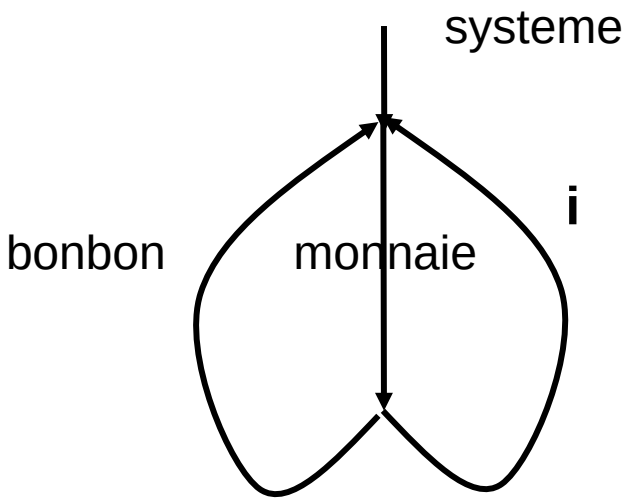
```
process systeme [monnaie, bonbon] :=
      hide candi in
                  (distributeur  [monnaie, bonbon, candi]
                  |[candi]|
                   diable [candi])
endproc

where

process distributeur [monnaie, bonbon, candi] :=
      monnaie;
      (bonbon; distributeur [monnaie, bonbon, candi]
                  []
      candi; distributeur [monnaie, bonbon, candi] )
endproc

process diable [candi] :=
      candi; diable [candi]
endproc
```

Après avoir pris la monnaie, le distributeur peut aléatoirement [] décider de

1) Donner le bonbon au client
2) Ou synchroniser avec le diable qui mange le candi! Ceci génère une action interne (hide)
3) Ce cycle continue à jamais (récursivité)

Équivalent par *bisimulation faible* à:

```
process systeme[monnaie,bonbon] :=
        monnaie;
                (bonbon; systeme [monnaie, bonbon]
                []
                i; systeme [monnaie, bonbon])
```



Ce deuxième processus est une *expansion* du précédent

L'usager pourra entrer la monnaie et ne pas voir le bonbon qui sera parfois consommé par le diable

```
specification systeme [monnaie, bonbon] : noexit :=
hide candi in
(distributeur  [monnaie, bonbon, candi]
       |[candi]|
        diable [candi])


where

process distributeur [monnaie, bonbon, candi] : noexit :=
monnaie;
(bonbon; distributeur [monnaie, bonbon, candi]
       []
candi; distributeur [monnaie, bonbon, candi] )
endproc

process diable [candi] : noexit :=
candi; diable [candi]
endproc

endspec
```

```
specification systeme2 [monnaie,bonbon] : noexit behaviour
(systeme [monnaie,bonbon])
where
process systeme[monnaie,bonbon] : noexit :=
monnaie; (bonbon; systeme [monnaie, bonbon] [] i; systeme [monnaie, bonbon])
endproc
endspec
```

Opérateurs additionnels

- LOTOS a aussi les opérateurs suivants:
  - **>>** enable : A >> B veut dire que après une **exit** de A on fait B
    - **Comme stop, exit termine un processus mais s'il y a un enable il permet de passer au processus suivant**
  - **[>** disable: A [> B veut dire que n'importe quand pendant l'exécution de A, B peut interrompre A en initiant son exécution
    - **A n'est plus repris.**

Full LOTOS (LOTOS Complet)

- Ce que nous avons vu est le LOTOS de base (basic LOTOS) sans la possibilité d'exprimer données et valeurs
- En Full LOTOS il est possible de définir des données et d'introduire des données dans les actions
  - a!x
    - **Veut dire que le processus offre la valeur de la variable x à la porte a**
  - a?x: nat
    - **Veut dire que le processus attend un nombre naturel x à la porte a**
  - a ?x !y
    - **En même temps, le processus accepte une valeur et en offre une autre**
- Nous avons la même règle de synchronisation que pour LOTOS de base:
  - Deux actions synchronisent si elles sont identiques
  - P. ex. a!3 et a?x:nat synchronisent car:
    - **Elles proposent la même porte a**
    - **L'une propose un entier précis tandis que l'autre propose un entier quelconque**
      - **a?x:nat est équivalent à**
        - **a!0 [] a!1[] a!2 [] a!3…**

Expressions gardées

- Voir ‘garded commands’ dans quelques langages de programmation

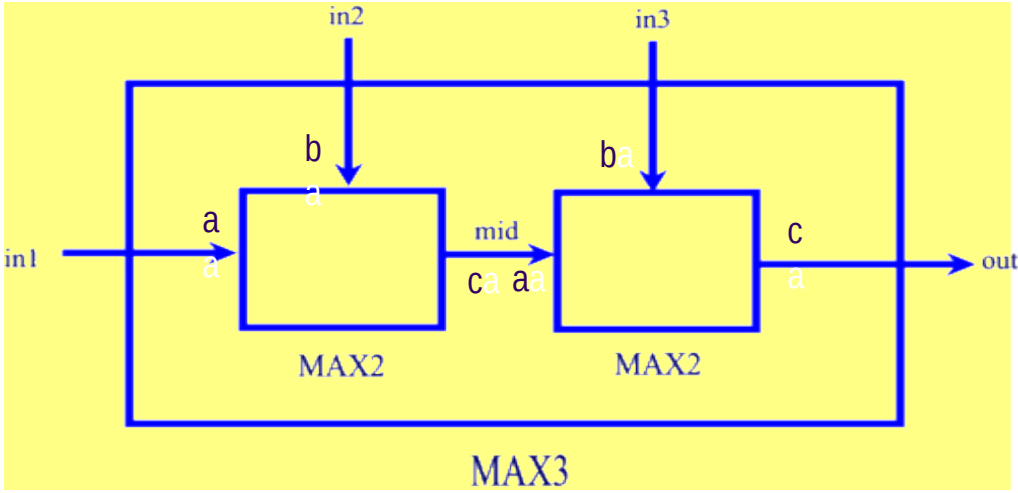
```
    [x>0] -> process1
[] [x=5] -> process2
[] [x<9] -> process2
```

Observes la possibilité d’exprimer le **nondéterminisme** (trois possibilités dans le cas de x=5)

Une spécification simple en ‘full LOTOS’

```
Specification Max3 [in1, in2, in3, out]:noexit
type natural is
    sorts nat
    opns zero: →nat
         succ: nat → nat
         largest: nat, nat →nat
     eqns ofsort nat
          forall x:nat
          largest(zero, x) = x
          largest(x, y) = largest(y, x)
          largest(succ(x), succ(y)) = succ(largest(x, y))
endtype (* natural *)

behaviour
       hide mid in
            (Max2[in1, in2, mid] |[mid]| Max2[mid, in3, out])    //crée deux instances du processus Max2, synch. par la porte mid
where
    process Max2[a, b, c] : noexit :=
        a ?x:nat; b ?y:nat; c !largest(x,y); stop
        []
        b ?y:nat; a ?x:nat; c !largest(x,y); stop
    endproc (*Max2*)
endspec (*Max3*)
```

Définit un processus à 4 portes qui accepte trois nombres naturels (type nat) dans un ordre quelconque et sort le plus grand



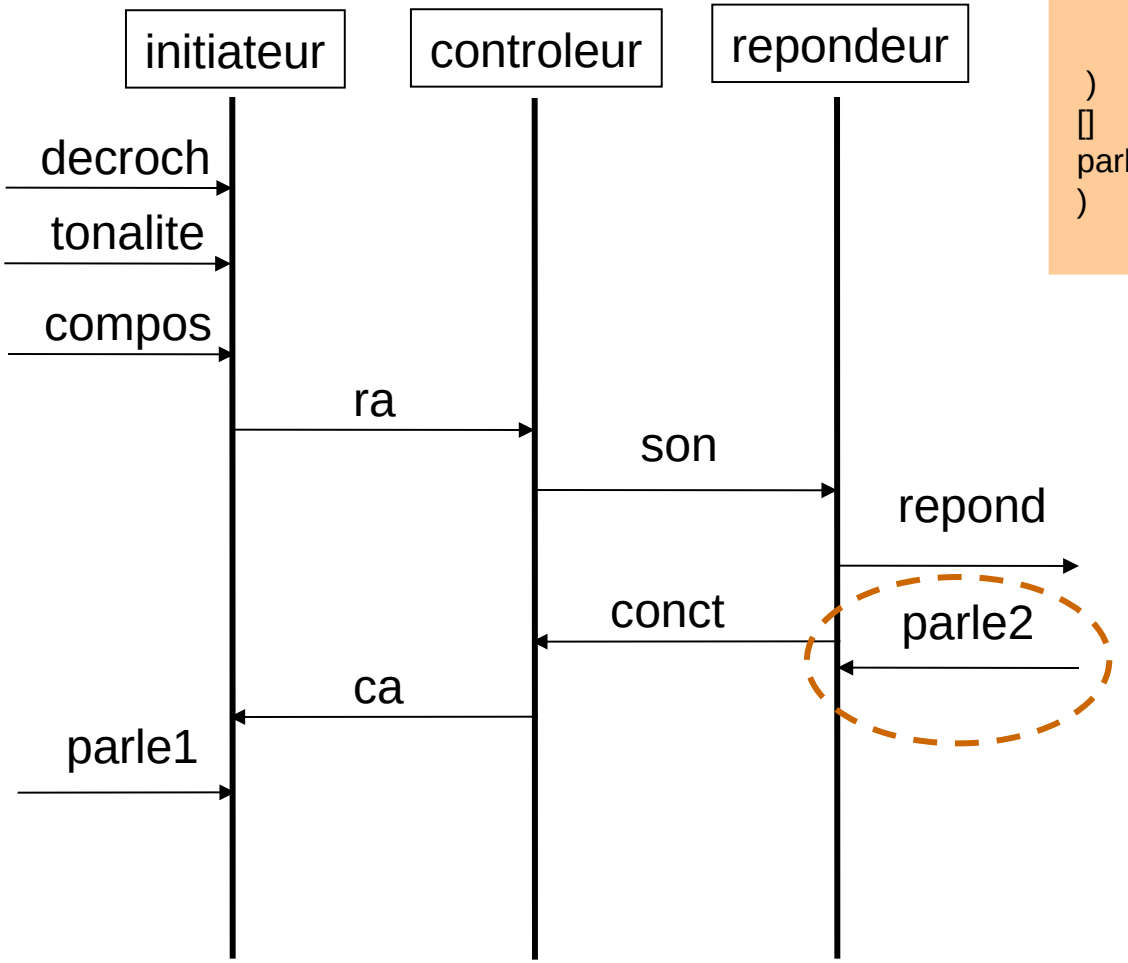
Explication pratique de la notation pour types de données

- Pour calculer largest(1,2)
  - 1 s'écrit succ(0) et 2 s'écrit succ(succ(0)) donc le terme au complet s'écrit comme suit:
  - largest(succ(0),succ(succ(0)))
    - **Pour calculer le résultat on applique les équations données dans la spécification comme suit:**
    - **largest(succ(0),succ(succ(0)))**
    - **= succ(largest(0, succ(0))**
    - **= succ(succ(0))**
      - **Ce résultat final réprésente en effet la valeur 2**
- Cette simple notation algébrique est importante à comprendre car elle est à la base de plusieurs notations informatiques
  - Abstract Data Types ou ADTs, on trouve cette notation en SDL aussi

EXPLICATIONS SUPPLÉMENTAIRES SUR UN PROBLÈME MENTIONNÉ…

Message Sequence Chart

```
decroch; tonalite; compos; ra; son; repond; conct;
(ca;
      (parle1; parle2; stop
       []
       parle2; parle1; stop)
 )
[]
parle2; ca; parle1; stop
)
```



Possibilité de perte de *parle2* car la connexion n'est pas encore complètement établie.
Malheureusement ceci est inévitable car les trois entités ne peuvent pas communiquer directement et simultanément.
Mais il n'y aura pas de perte si *conct* et *ca* sont exécutés très rapidement, ce qui correspond à la pratique et à la solution suivante.

Comment on pourrait penser à réparer ce problème

- Ajouter ***ca*** au répondeur et à l'ensemble de synchronisation pour les derniers deux processus
- Les trois doivent donc sync sur ***ca***

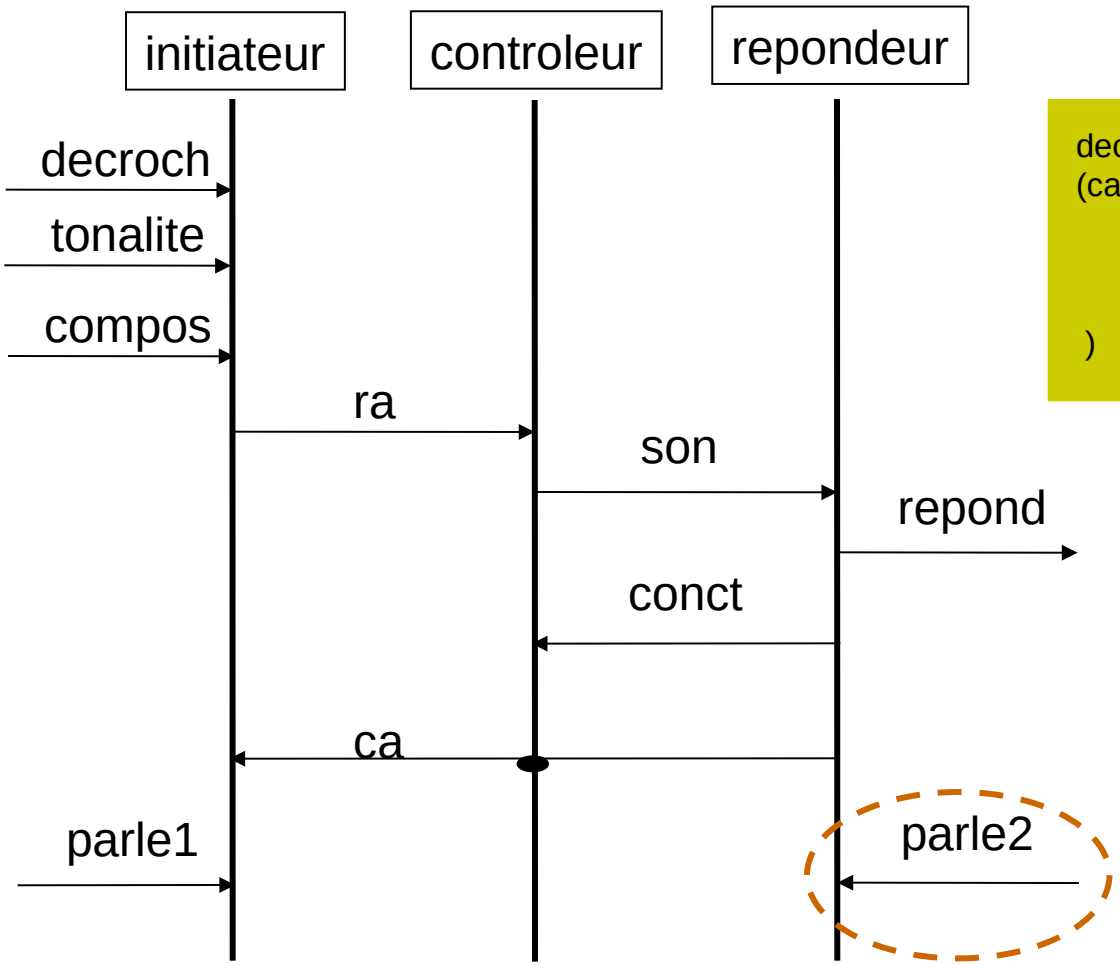
process phone:= initiateur |[ra,ca]| controleur |[son,conct,**ca**]| repondeur

process repondeur:= son; repond; conct; ca; parle2; stop

Expansion:

```
decroch; tonalite; compos; ra; son; repond; conct;
(ca;
      (parle1; parle2; stop
        []
        parle2; parle1; stop)
 )
```

MSC pour comportement corrigé



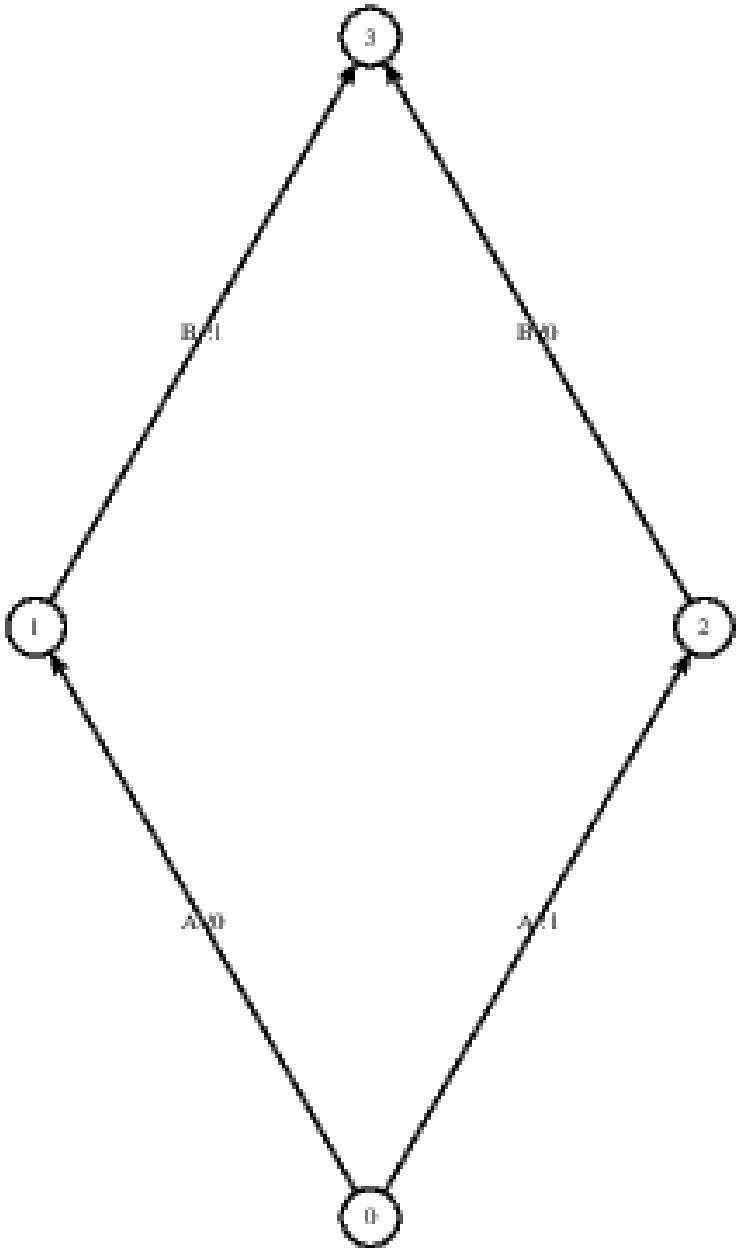

```
decroch; tonalite; compos; ra; son; repond; conct;
(ca;
      (parle1; parle2; stop
        []
        parle2; parle1; stop)
 )
```

Cependant l'implémentation de cette synchronisation à trois reste problématique car une communication directe entre initiateur et répondeur n'est pas possible

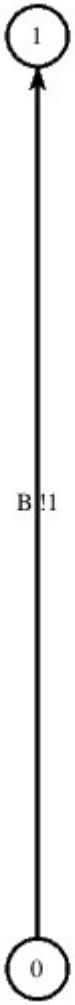
Exercices sur les circuits logiques:

```
specification circuit_logique [a,b,c] : noexit
type BIT is
   sorts BIT
   opns 0 (*! constructor *),
        1 (*! constructor *) : -> BIT
        not : BIT -> BIT
   eqns
      ofsort BIT
         not (0) = 1;
         not (1) = 0;
endtype

behaviour
   gate_not[a, b]
where
   process gate_not[a, b] : noexit :=
     a ?aa:Bit; b !not(aa); stop
   endproc
endspec
```

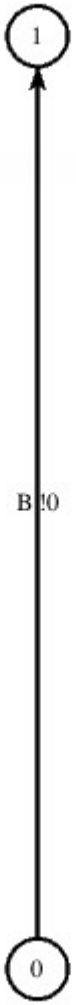
Essayer de faire des exercices dans ce domaine.


3
B/1
B/0
1
2
A/0
A/1
0

```
specification circuit_logique [a,b,c] : noexit
type BIT is
   sorts BIT
   opns 0 (*! constructor *),
        1 (*! constructor *) : -> BIT
        not : BIT -> BIT
   eqns
      ofsort BIT
         not (0) = 1;
         not (1) = 0;
endtype

behaviour
   gate_not[a, b](0)
where
   process gate_not[a, b] (aa : BIT) : noexit :=
     b !not(aa); stop
   endproc
endspec
```


1
B!1
0

```
specification circuit_logique [a,b,c] : noexit
type BIT is
   sorts BIT
   opns 0 (*! constructor *),
        1 (*! constructor *) : -> BIT
        not : BIT -> BIT
   eqns
      ofsort BIT
         not (0) = 1;
         not (1) = 0;
endtype

behaviour
   gate_not[a, b]
where
   process gate_not[a, b] : noexit :=
     let aa:Bit = 1 in (
     b !not(aa); stop)
   endproc
endspec
```


1
B 0
0

Exercice n° 4:

Ecrire des spécifications lotos pour les circuits logiques: and, or et xor.

Dessiner leurs systèmes à transitions étiquettées