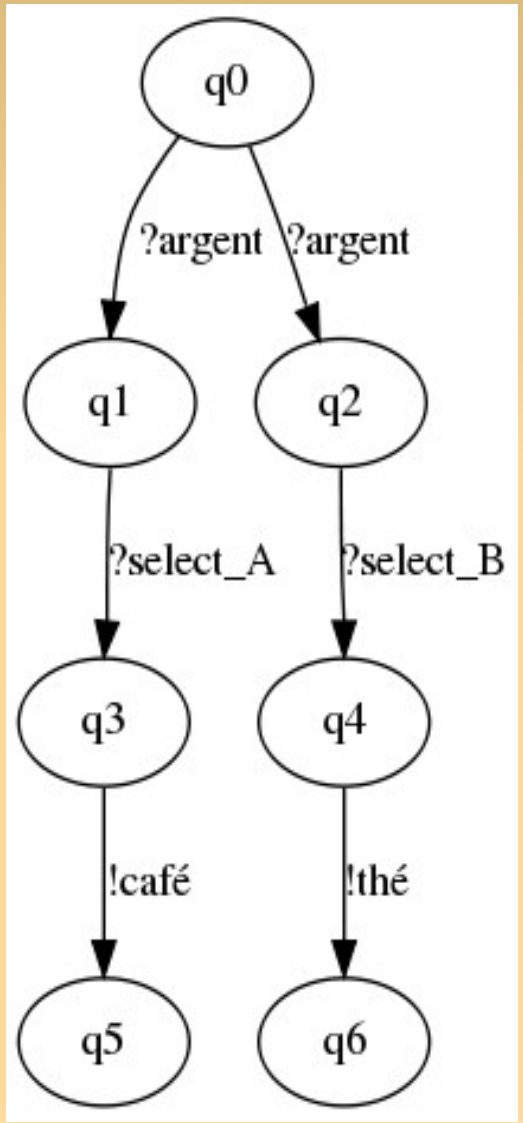
Partie 2 – « Langages » de processus
+ outillage pour modélisation et vérification

- Partie 1 – modèle comportemental de système
- Partie 2 – ”langages” de processus
- Partie 3 – propriétés

Systèmes simples

- format Aldébaran, fichier .aut
  format CADP utilisé par plusieurs outils
- des (0,7,7) ← état initial / nb. trans. / nb. états
  (0,"argent",1)
  (0,"argent",2)
  (1,"select_B",3) ← transitions
  (2,"select_A",4)
  (3,"_the",5)
  (4,"_cafe",5)
  [5] ← état puits

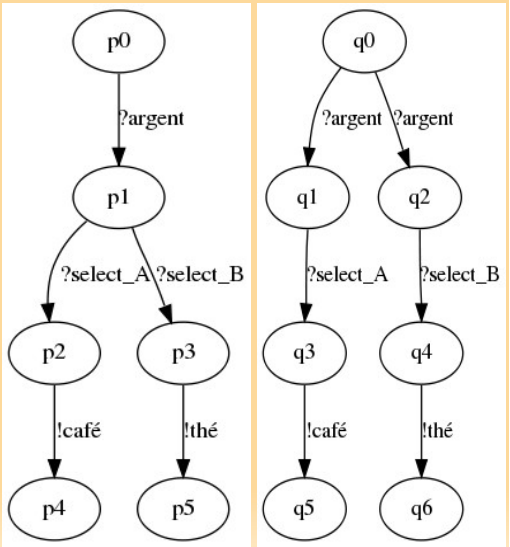


ATTENTION
?a → a
!a → _a

ATTENTION
État initial : 0
N° états en séquence

Processus

- un système (simple) peut aussi être défini par un processus (simple) en mcrl2
- P ::= a *(action, incl. tau)*
  | P1.P2 *(séquence)*
  | P1+P2 *(choix)*
  | delta *(inaction / deadlock)*
- proc M1 = argent.select_A._cafe
  + argent.select_B._the;
- proc M2 = argent.(select_A._cafe
  + select_B._the);



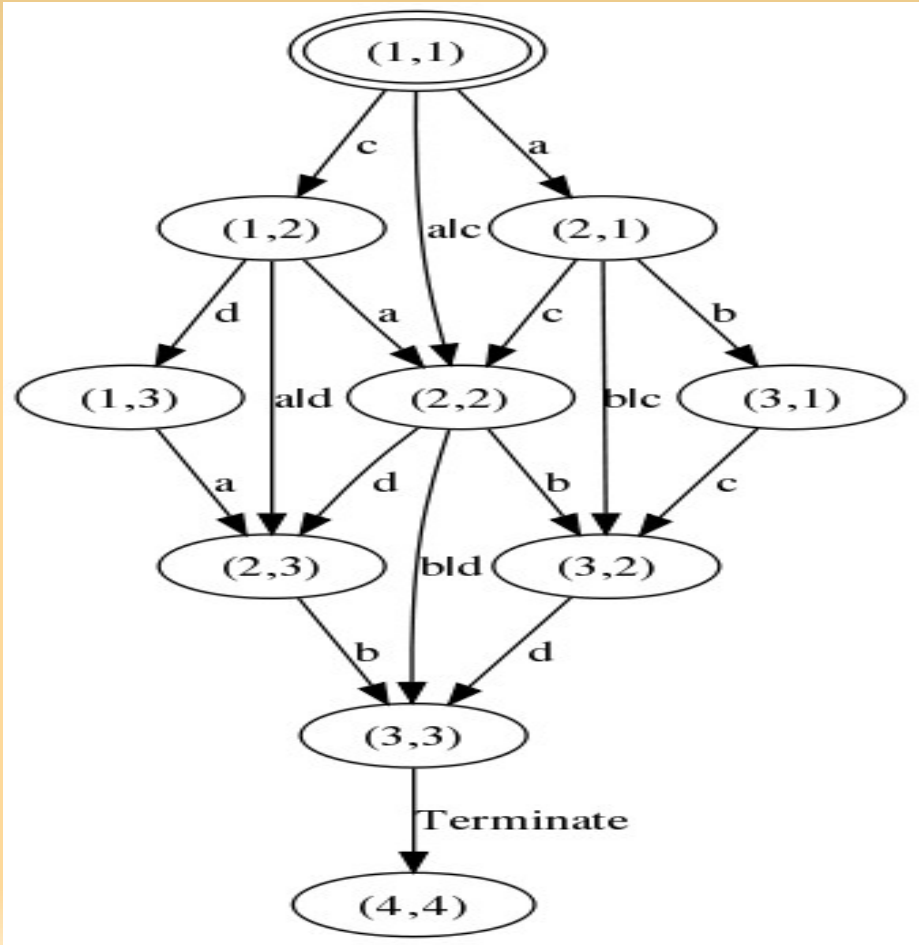
Processus

- utilisation de processus annexes
  proc p = def
  P ::= a | P1.P2 | P1+P2 | delta | **p**
- proc ServiceCafe = select_A._cafe;
  proc ServiceThe = select_B._the;

  proc M1 = argent.ServiceCafe
  + argent.ServiceThe;
  proc M2 = argent.(ServiceCafe + ServiceThe)

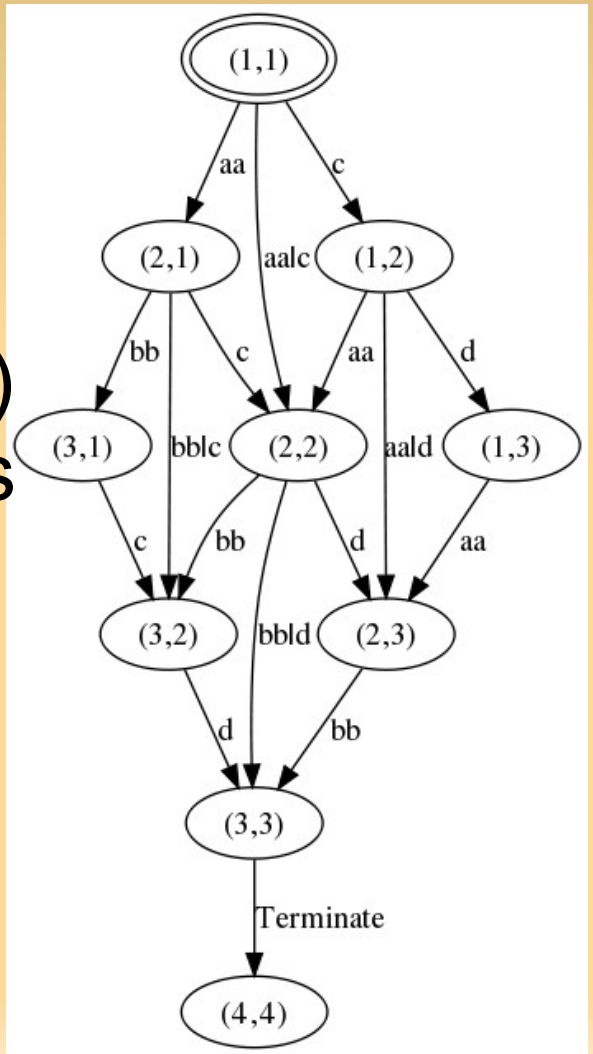
Processus

- composition parallèle
  P ::= a | P1.P2 | P1+P2 | delta | p
  | **P1||P2**
- proche du produit cartesien avec epsilon
- multi-actions
- proc P = a.b;
  proc Q = c.d;
  proc M = (P||Q);



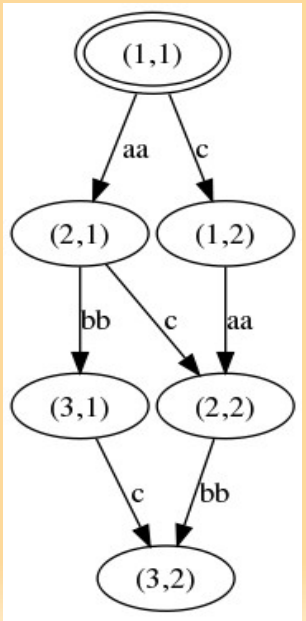
Processus

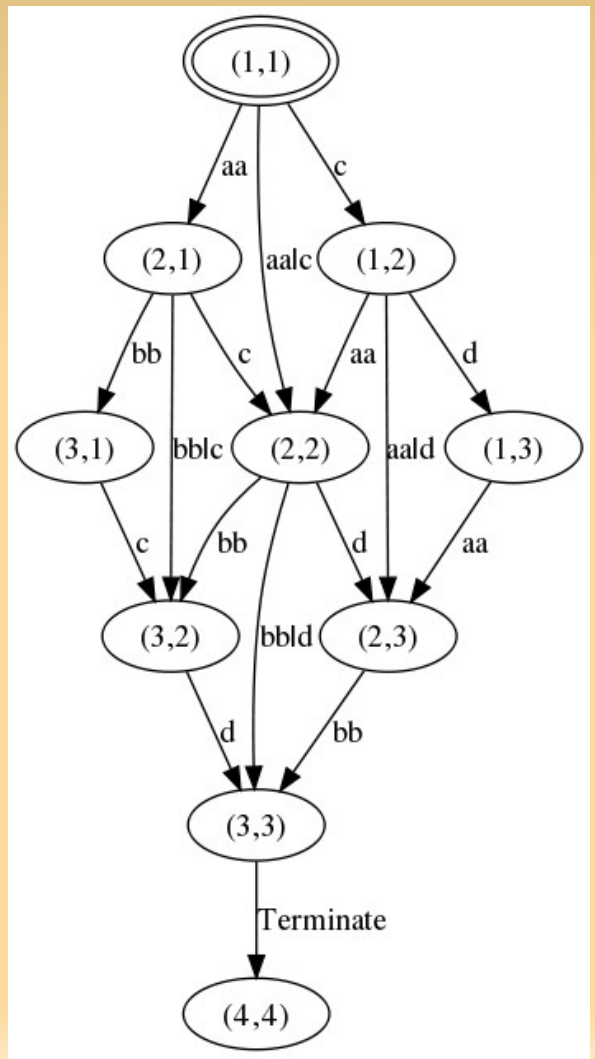
- renommage
  P ::= a | P1.P2 | P1+P2 | delta | p
  | P1||P2
  | **rename(f,P)**
- rename(f,P) correspond à P[f] (LTS) mais avec support des multi-actions
- proc P = a.b;
  proc Q = c.d;
  proc M = P||Q;
  proc Mr =
  rename({a->aa,b->bb},M);



Processus

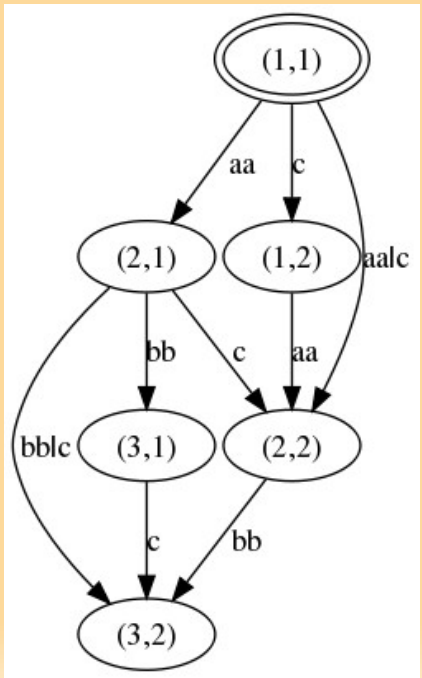
- autorisation
  P ::= a | P1.P2 | P1+P2 | delta | p
  | P1 || P2
  | rename(f,P) | **allow(B,P)**
- allow(B,P) correspond à P#B (LTS)
- proc MrA =
  allow({aa,bb,c},Mr);

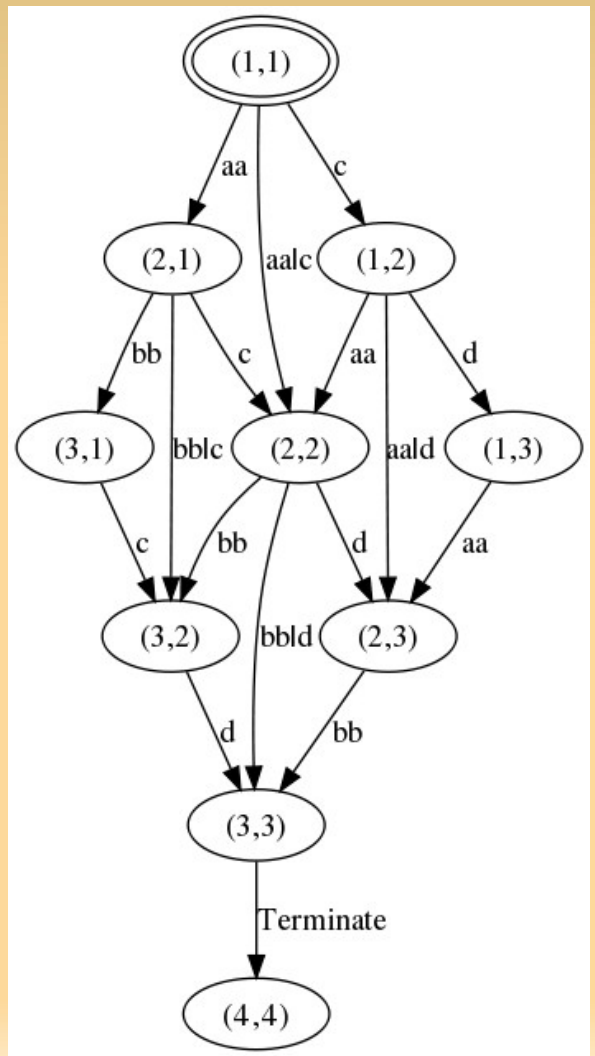




Processus

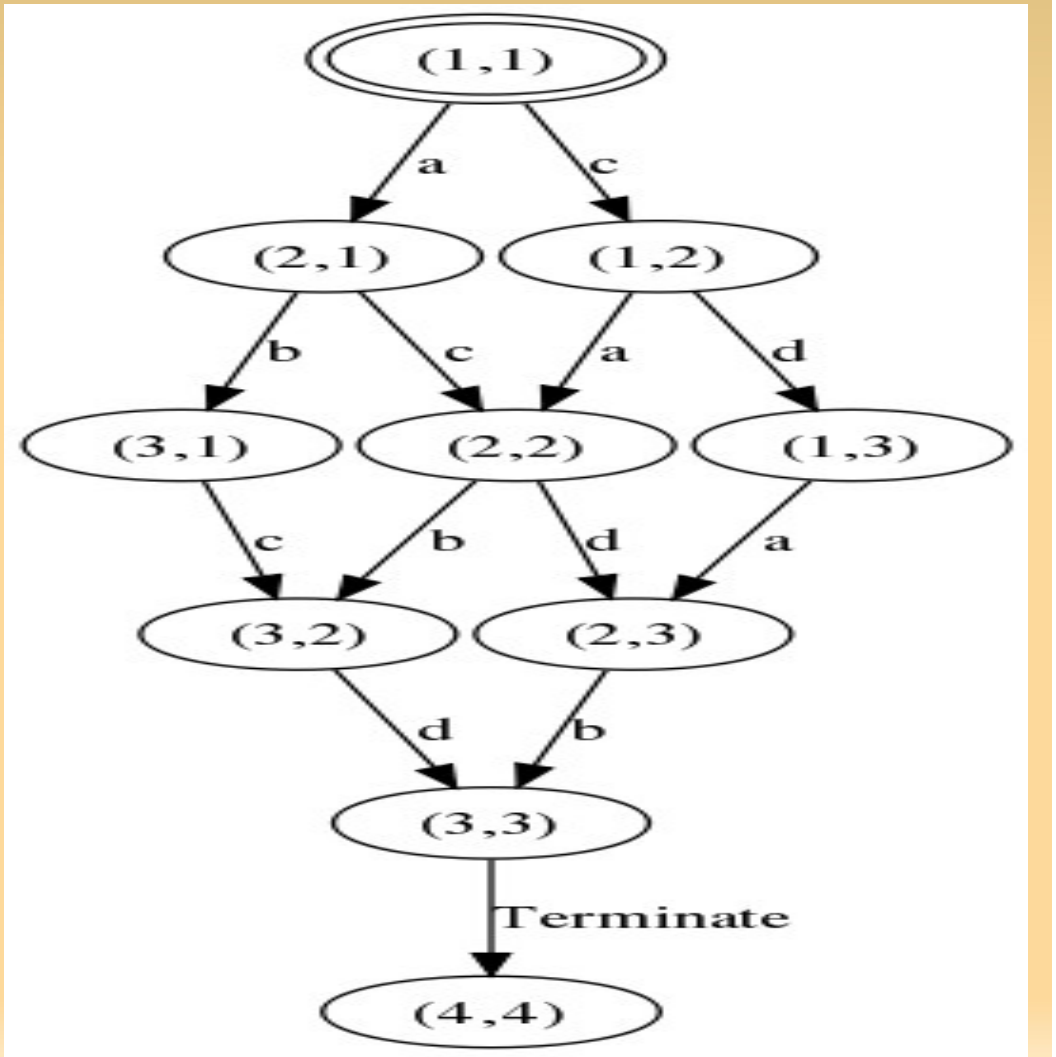
- autorisation
  P ::= a | P1.P2 | P1+P2 | delta | p
  | P1 || P2
  | rename(f,P) | allow(B,P)
  | **block(B,P)**
- block(B,P) correspond à P\B (LTS)
  ”dual” de allow
  mais support pour multi-actions
- proc MrB = block({d},Mr);





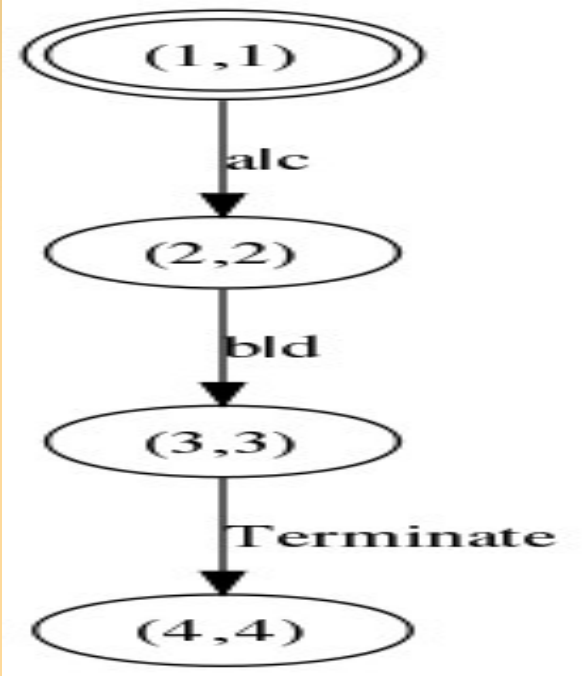
Composition parallèle

- produit libre ?
- utiliser allow(B,P);
- proc P = a.b;
  proc Q = c.d;
  proc M = allow({a,b,c,d},
  (P||Q));



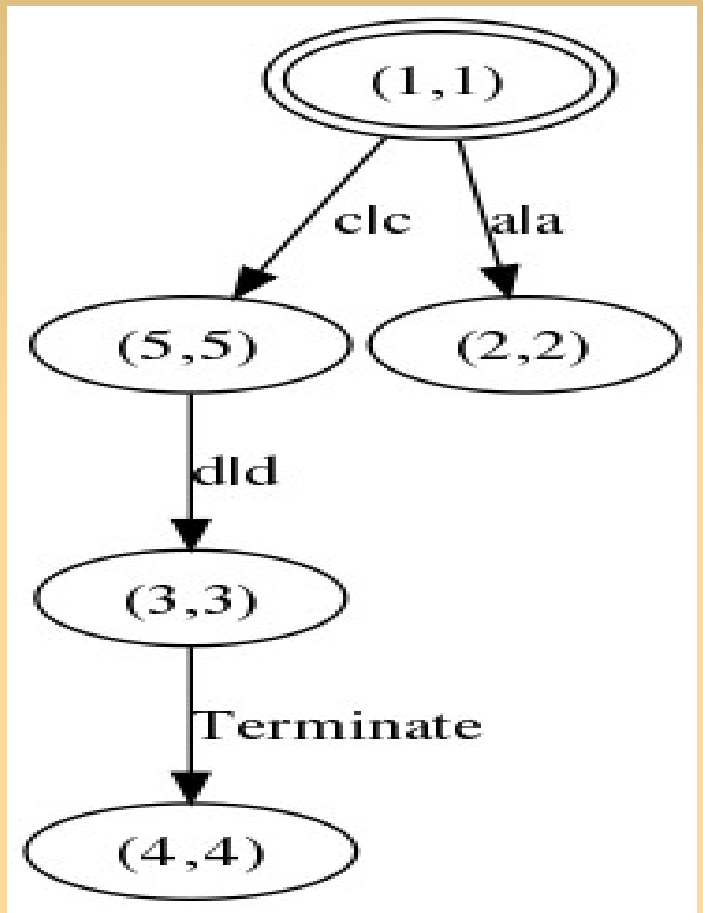
Composition parallèle

- produit cartésien ?
- utiliser allow(B,P);
- proc P = a.b;
  proc Q = c.d;
  proc M = allow({a|c,a|d,b|c,b|d},
  (P||Q));



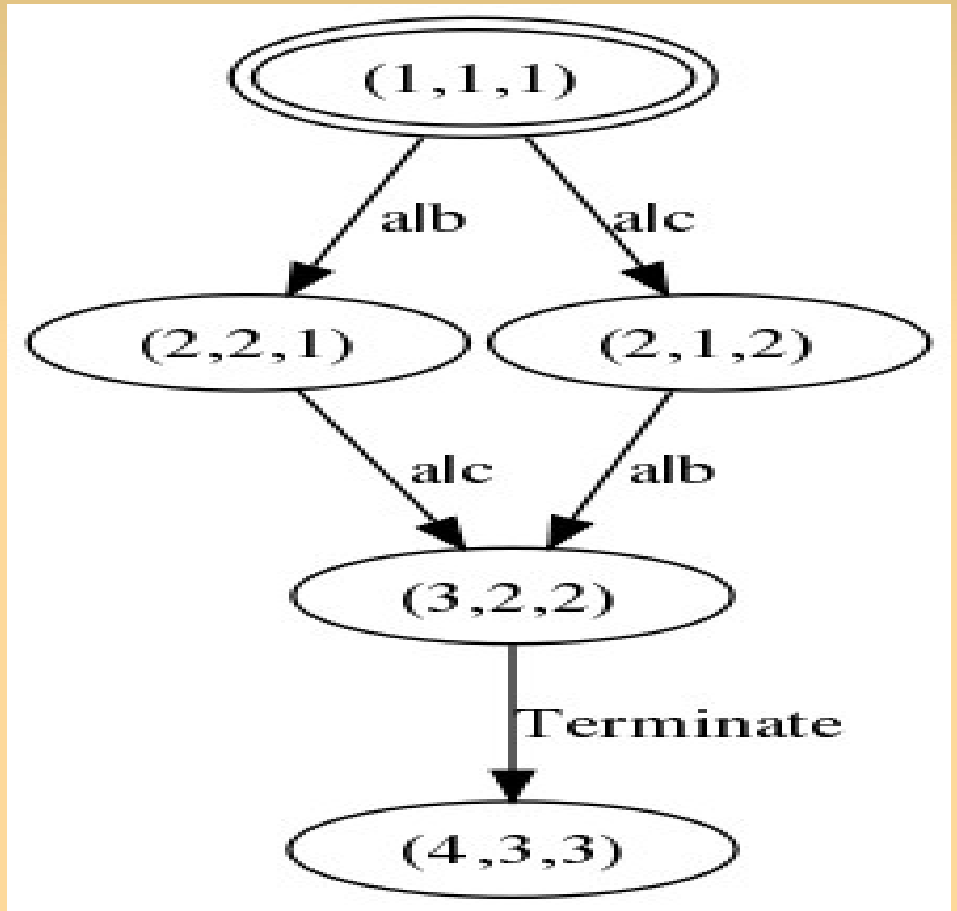
Composition parallèle

- produit synchronisé ?
- utiliser allow(B,P);
- proc P = a.b + c.d;
  proc Q = a.c + c.d;
  proc M = allow({a|a,b|b,c|c,d|d},
  (P||Q));



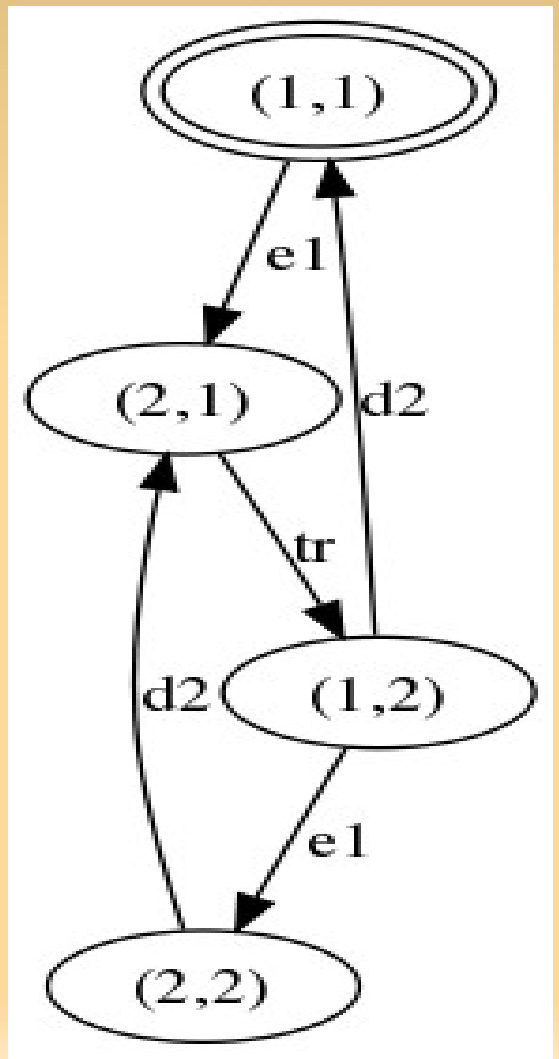
Composition parallèle

- produit vectoriel ?
- vous l'aurez compris : utiliser allow(A,P)
- proc P = a.a;
  proc Q = allow({b,c},
  b||c);
  % ou Q = b.c + c.b
  proc M = allow({a|b,a|c},
  P||Q);



Processus

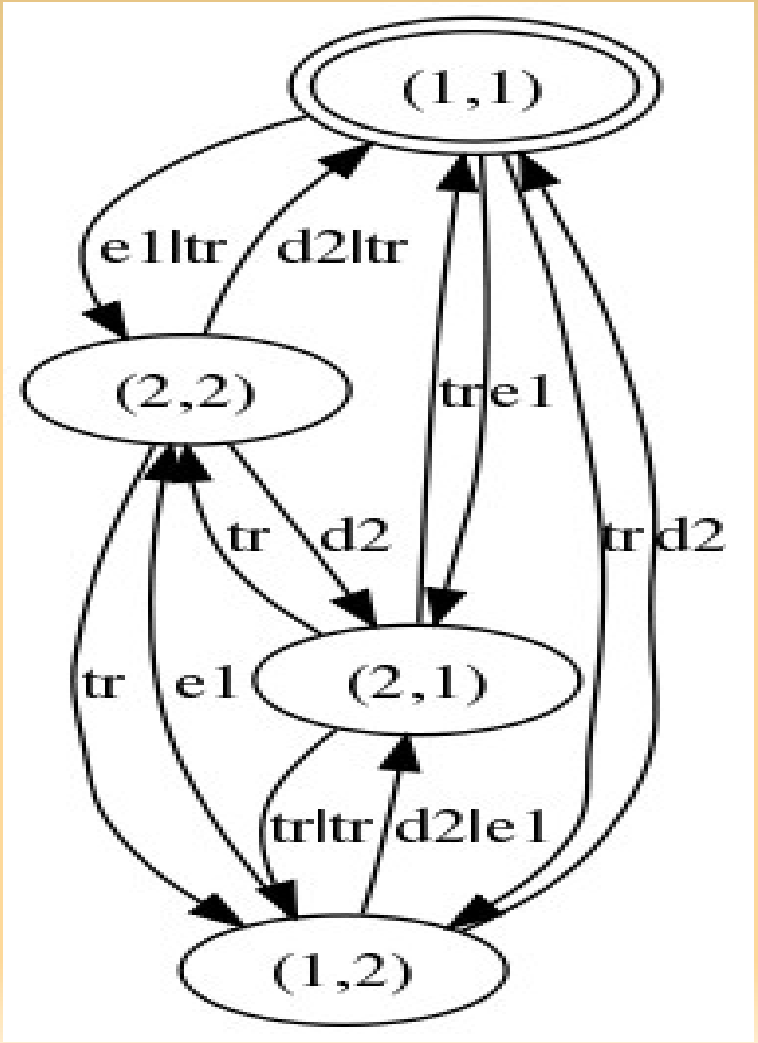
- pour la ”communication”, vous pouvez utiliser **comm({ci},P)** où:
  - chaque ci est une communication
    ex: a|b->comm
  - P est un processus
- proc P1 = e1.d1.P1;
  proc P2 = e2.d2.P2;
  proc BP = allow({e1,d2,tr},
  comm({d1|e2->tr},P1||P2));
- attention à l'oubli du allow !



Processus

- pour la ”communication”, vous pouvez utiliser comm({ci},P) où:
  - chaque ci est une communication ex: a|b->comm
  - P est un processus
- pas de communication implicite (différent de CCS,CSP,LOTOS)
- proc P1 = e1.tr.P1;
  proc P2 = tr.d2.P2;
  proc BP = P1 || P2;

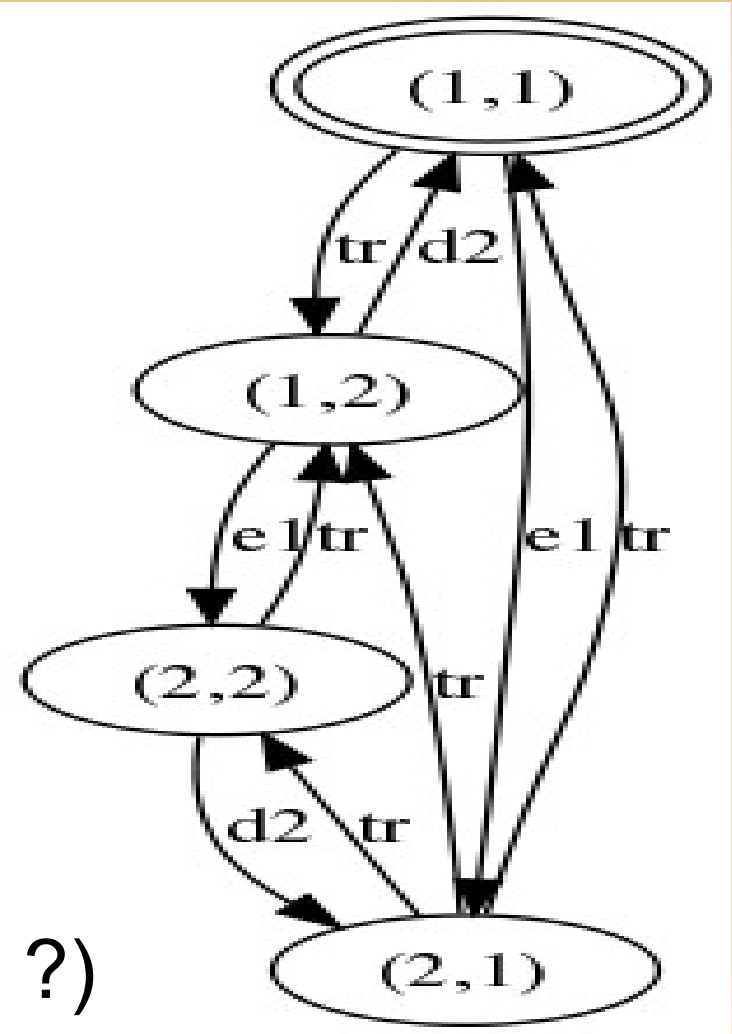
  → pas bon !



Processus

- pour la ”communication”, vous pouvez utiliser comm({ci},P) où:
  - chaque ci est une communication ex: a|b->comm
  - P est un processus
- pas de communication implicite (différent de CCS,CSP,LOTOS)
- proc P1 = e1.tr.P1;
  proc P2 = tr.d2.P2;
  proc BP = allow({e1,d2,tr},
  comm({tr|tr->tr},P1 || P2));

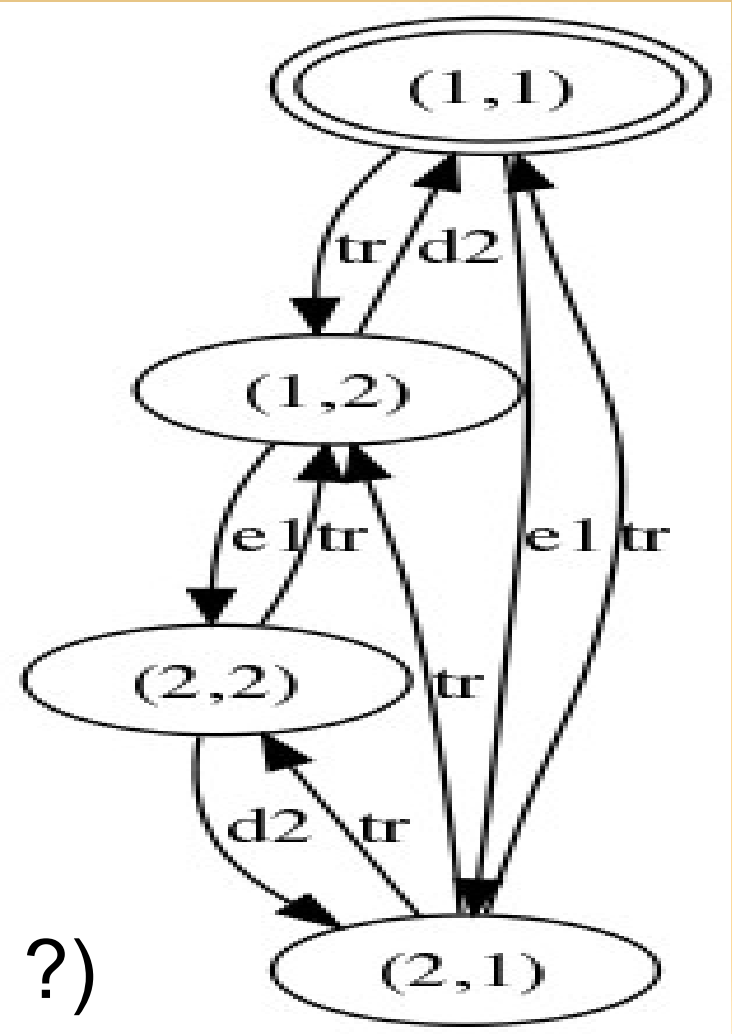
  → pas bon ! (pourquoi ? solution ?)



Processus

- pour la ”communication”, vous pouvez utiliser comm({ci},P) où:
  - chaque ci est une communication ex: a|b->comm
  - P est un processus
- pas de communication implicite (différent de CCS,CSP,LOTOS)
- proc P1 = e1.tr.P1;
  proc P2 = tr.d2.P2;
  proc BP = allow({e1,d2,tr},
  comm({tr|tr->tr},P1 || P2));

  → pas bon ! (pourquoi ? solution ?)

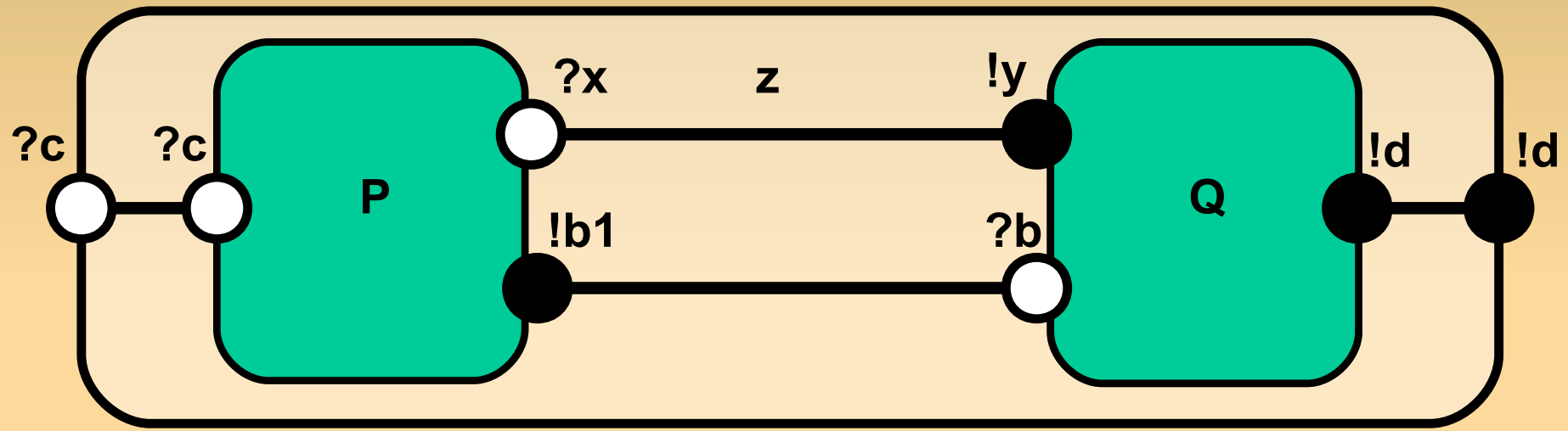


Processus

- pour la ”communication”, vous pouvez utiliser comm({ci},P) où:
  - chaque ci est une communication
    ex: a|b->comm
  - P est un processus
- attention comm({a|b->x,a|c->y},P) interdit !!
  (a dans deux paires de communication)
- si besoin dupliquer les actions correspondantes

Processus

- comment encoder ceci ?

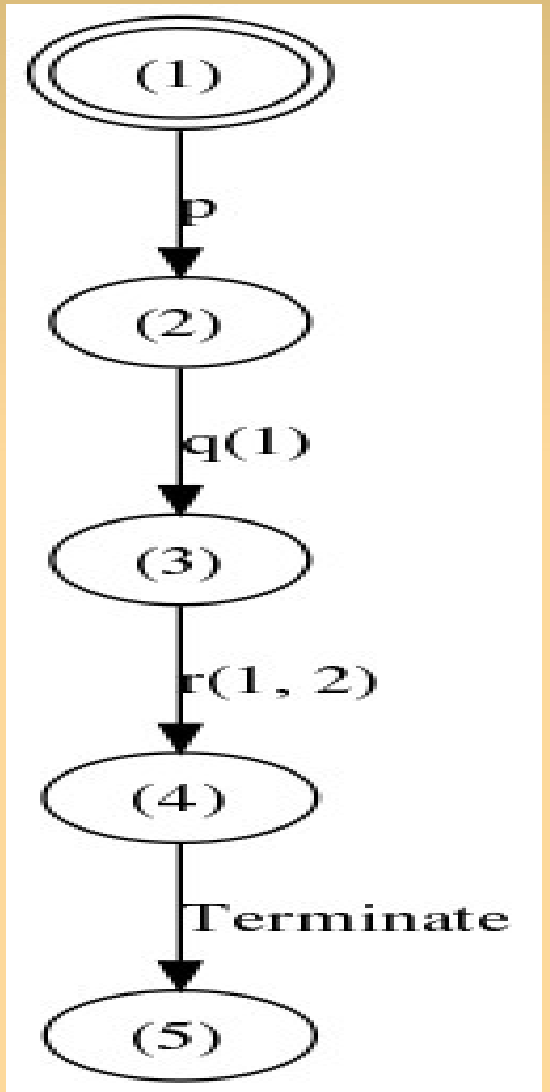


Données

- types de données de base :
  Bool | Pos | Nat | Int | Real
- types collections :
  List[T] | Set[T] | Bag[T]
- types structurés :
  struct c1 (..., p1j:T1j, …) ? r1
  ...
  ci (..., pij:Tij, …) ? ri
  …
  cn (..., pnj:Tnj, …) ? rn
- types définis par l'utilisateur
  → plus sur tout cela en TD
  http://www.mcrl2.org/release/user_manual

Processus et données

- déclaration des actions
  act p;
  q:Nat;
  r:Nat#Nat, …
- définition des processus
  proc P(x:Nat) = p . q(x) . r(x,x+1);
- initialisation
  init P(1);

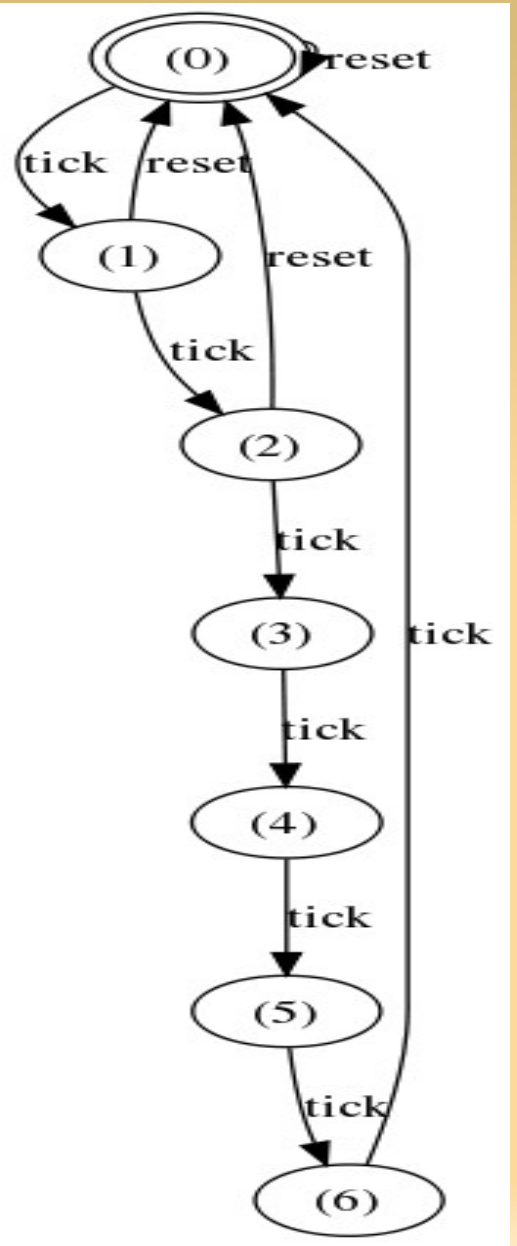


Processus et données

- conditions :
  (condition) → P1 <> P2
- act tick, reset;

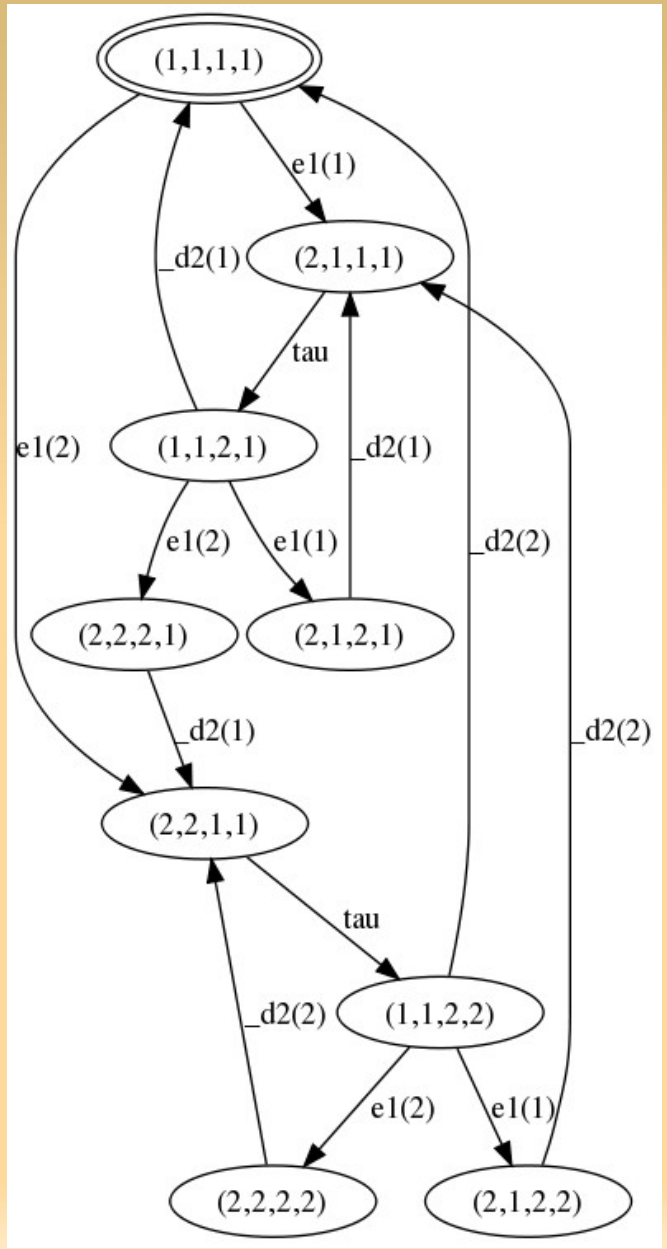
```
proc Clock(n: Nat) =
   (n < 6) -> tick . Clock(n + 1)
            <> tick.Clock(0)
+ (n < 3) -> reset . Clock(0);

init Clock(0);
```



Processus et données

- somme sur données
  d:T . P
- act e,e1,e2,_d,_d1,_d2,tr: Pos;
  proc P =
  sum v:Pos .
  (v<3) -> e(v) . _d(v) . P;
  P1 = rename({e->e1,_d->_d1},P);
  P2 = rename({e->e2,_d->_d2},P);
  M = hide({tr},
  allow({e1,tr,_d2},
  comm({_d1|e2->tr},
  P1||P2)));
  init M;



Utilisation de MCRL2 (bases)

- interface graphique ou outils en ligne de cde.
- compiler code : **mcrl22lps** in.mcrl2 out.lps

  obtention lts : **lps2lts** in.lps out.xxx
  avec xxx dans {aut,dot,fsm,lts,svc}
- visualisation : **ltsgraph** in.lts ou utiliser **dot**
- recherche de blocages: **ltsview** in.lts
- test de preordres: **ltscompare** -p preordre S1 S2
- test d'équivalences: **ltscompare** -e *equiv* S1 S2
- nous verrons pour des formules dans la suite
- mais aussi : animation, vérification de propriétés, conversions de formats, etc … (voir manuel et TD)