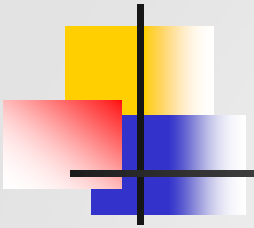


# Développement et modèles formels

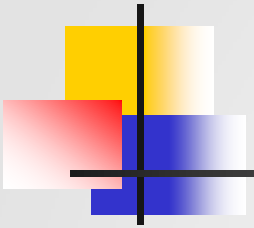
**Jean Fanchon**

LAAS-CNRS

*fanchon@laas.fr*

Journées Envol, Biarritz, 21-25/1/2012

Avec quelques emprunts à D.Peled et A. Bouajjani



# Pourquoi des modèles ?

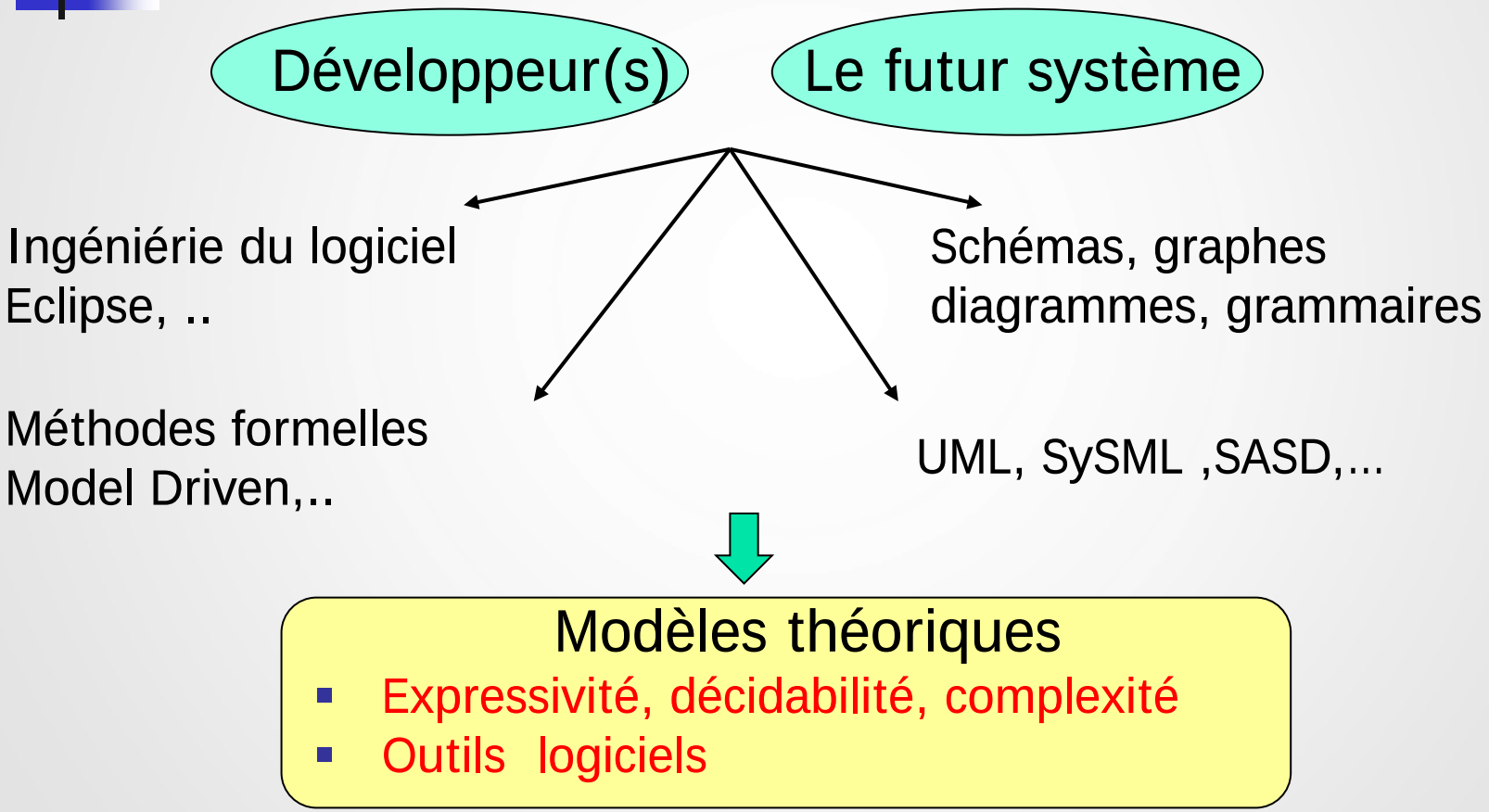
Développeur(s)

Le futur système

- Décrire, schématiser
- Communiquer/ collaborer / maintenir
- Représenter : code $\neq$ exécutions/comportement
- Structurer : composants, interactions
- Abstraire : niveaux de raffinement
- Spécifier : code $\neq$ propriétés

programme commenté, prose explicative, diagrammes, flowcharts, arbres, graphes, automates, grammaires etc...

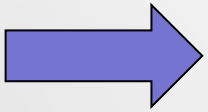
# Les choix du développeur



# Modèles pour éviter erreurs?

On veut faire des programmes qui marchent bien, dont on soit sûr (qu'on a pris toutes les précautions pour qu'il soit le plus sûr possible)

- Simulations et tests
  - prouvent qu'il y a une (des) erreur(s) en les exhibant
  - ne prouvent pas l'absence d'erreur
- Montrer la correction
  - Pas de comportements indésirables
  - Buts attendus réalisés



Les modèles formels permettent des preuves

# Un panorama du formel

On doit représenter les programmes et leurs comportements

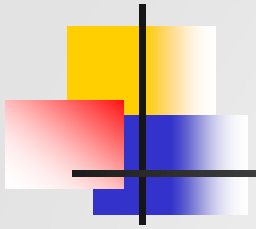
- Machines
  - Syntaxe: définitions finies, i.e. programmes, automates, RdP , etc
  - Sémantique:  définition de leurs effets et/ou comportements

On doit spécifier , vérifier leurs propriétés , i.e. ça fonctionne bien

- Logiques
  - Langages de spécification des propriétés des comportements
  - Méthodes de vérification  d'un modèle v.à.v d'une propriété

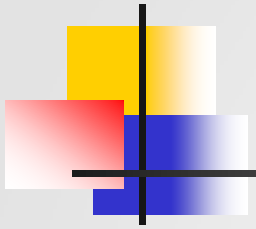
On doit construire des programmes complexes à partir de composants

- Algèbres
  - Syntaxe: opérateurs de composition de machines
  - Sémantique: opérations sur les comportements des composants



# Plan

- Introduction
- Machines
  - Syntaxe ≠ sémantique
  - Automates étendus
  - Produits d'automates
  - MSC et automates communicants
- Logiques
  - Exprimer les propriétés des systèmes
  - Logiques temporelles
  - Model-checking régulier
  - Le cas des MSC
- Conclusions

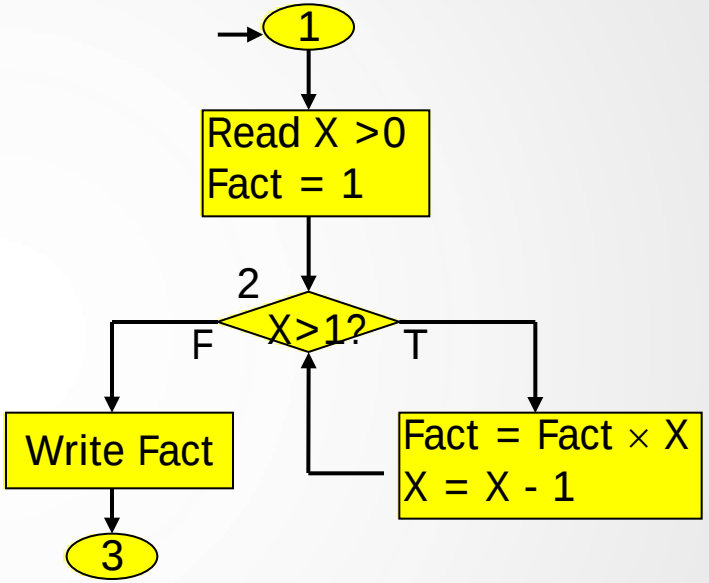


# Types de programmes

- Programmes séquentiels
  - Controle fini, nombre fini de registres
    - Bornés , non bornés
  - Procédures et récursivité , controle fini + pile
- Programmes concurrents/distribués
  - Threads parallèles (séquentiels)
    - nombre fixe
    - dynamique, nombre non borné
- Communication/synchronisation
  - Mémoire partagée
  - Rendez-vous
  - Canaux
    - Bornés, non bornés
    - Ordonnés ( FIFO ), non ordonnés, à pertes

# Syntaxes et sémantique

```
Read X >0
Fact = 1
While (X>1) {
    Fact = Fact × X
    X = X – 1
}
Write Fact
```

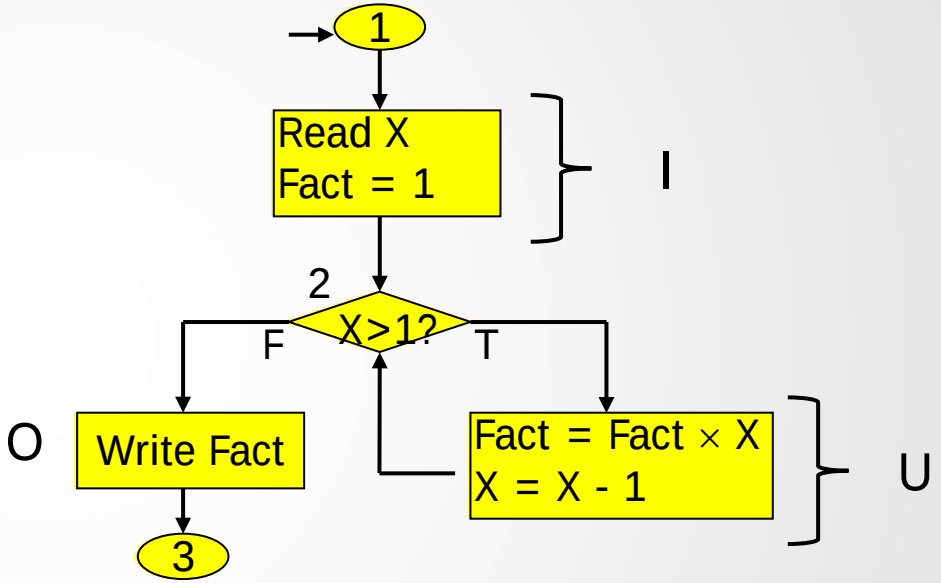



Configuration = (compteur ordinal, valeur de X , valeur de Fact)

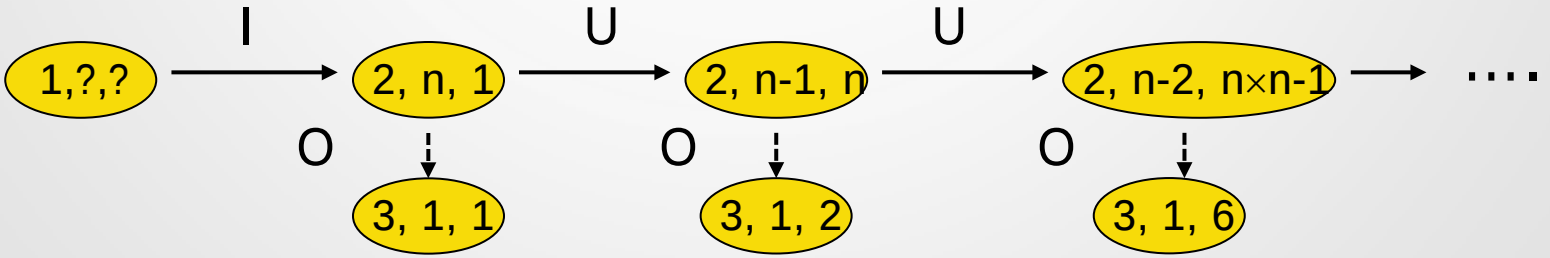

transition entre 2 configurations

# Syntaxes et sémantique

```
Read X
Fact = 1
While (X>1) {
    Fact = Fact × X
    X = X – 1
}
Write Fact
```

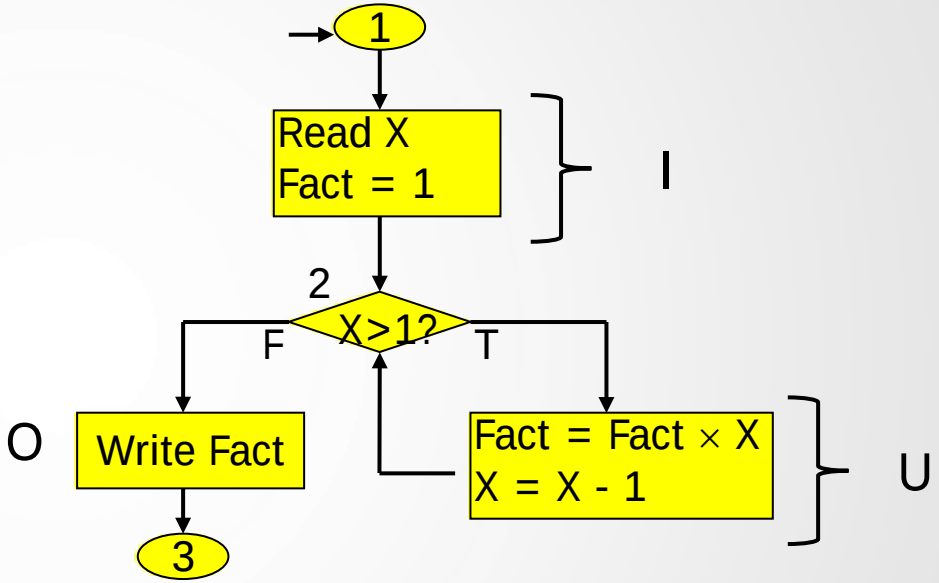



Sémantique opérationnelle : modèle des comportements

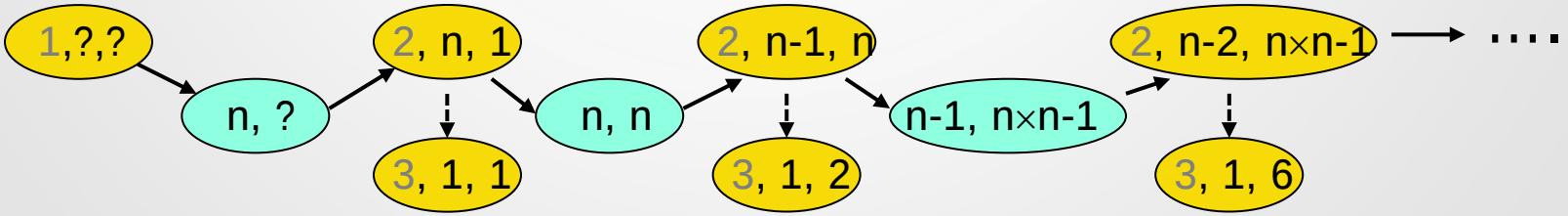


# Syntaxes et sémantique

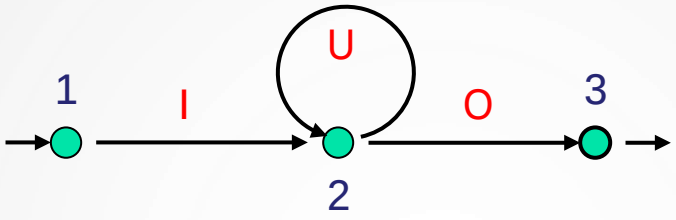
```
Read X
Fact = 1
While (X>1) {
      Fact = Fact × X
      X = X – 1
}
Write Fact
```




Sémantique opérationnelle : modèle des comportements



# Automates




- Alphabet : {I,U,O}
- Etats : {1,2,3}
- Transitions : {(1, I, 2), (2, U, 2), (2, O, 3)} (origine, étiquette, extrémité)
- Etat initial : 1 , Etat final : 3
- Chemin : séquence de transitions dont l'extrémité est l'origine de la suivante

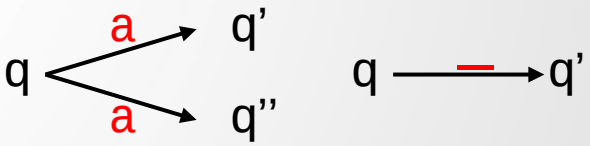
  (1, I, 2) (2, U, 2) (2, U, 2) (2, O, 3)
- Langage : ensemble des mots ( séquences ) de I,U,O qui étiquètent un chemin entre 1 et 3 I.U.U.O

# Automates

Automate sur alphabet fini $\Sigma$ (Symboles, actions, signaux, procédures,..)

$$A=(Q , T , q_{in} , F)$$

- Q ensemble fini d'états
- T ensemble de transitions $T \subseteq Q \times \Sigma \times Q$ , triplets (q, a, q')
  - Soit $T: Q \times \Sigma \rightarrow Q$ déterministe
  - Soit $T: Q \times \Sigma\_ \rightarrow 2^Q$ non-déterministe
- $q_{in}$ état initial , F états finaux

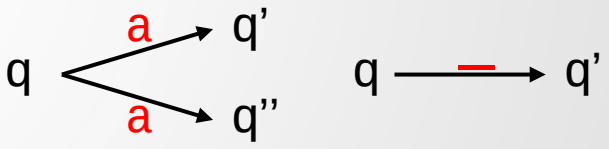


# Automates

Automate sur alphabet $\Sigma$ (Symboles, actions, signaux, procédures,..)

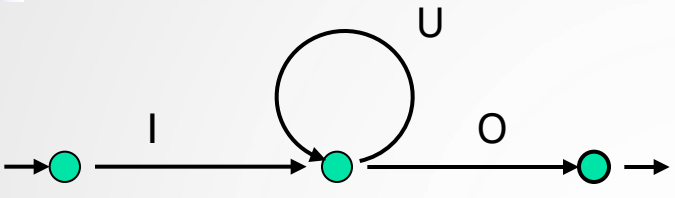
$$A=(Q , T , q_{in} , F)$$

- Q ensemble fini d'états
- T ensemble de transitions $T \subseteq Q \times \Sigma \times Q$ , triplets (q, a, q')
  - Soit $T: Q \times \Sigma \rightarrow Q$ déterministe
  - Soit $T: Q \times \Sigma\_ \rightarrow 2^Q$ non-déterministe



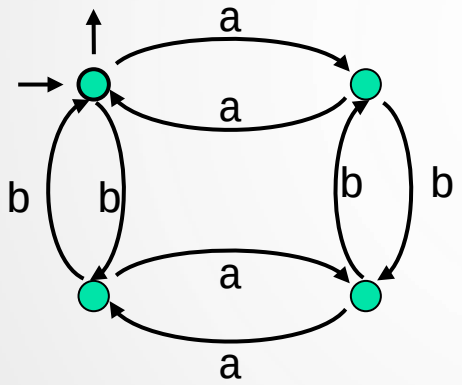

- $q_{in}$ état initial , F états finaux
- Langage fini : ensemble des séquences $a_0a_1...a_n \in \Sigma^*$ qui étiquettent un chemin de $q_{in}$ à un état de F
- Langage infini : ensemble des séquences infinies qui passent par un état de F un nombre infini de fois.

# Automates




Alphabet : {I,U,O}

Langage : I.{U}*.O




Alphabet : {a,b}

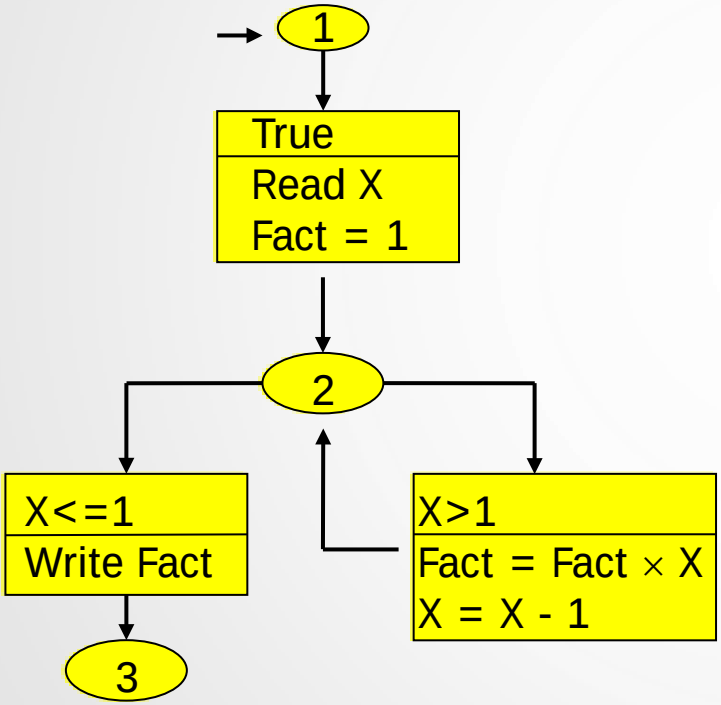
Langage : mots sur {a,b} avec
nombre pair de a
et nombre pair de b

Le langage d'un automate peut se décrire par une expression rationnelle

- . concaténation
- \+ union
- * itération

Init . ( Lock1.Calcul1.Unlk1 + Lock2.Calcul2.Unlk2 )* . Fin

# Automates étendus

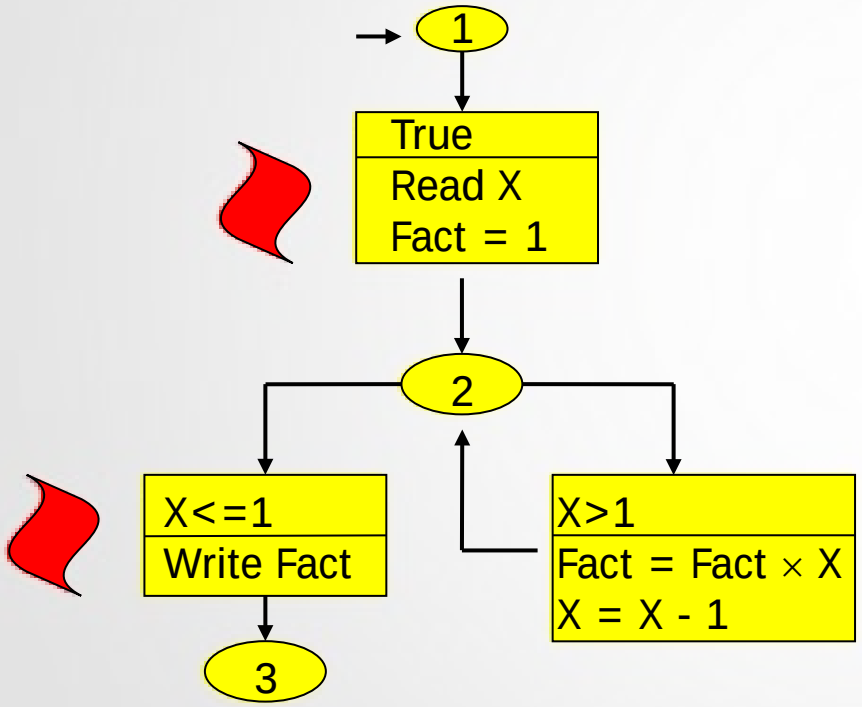



I = (1, <True, Init>, 2)

U = (2, <(X>1), Update>, 2)

O = (2, <(X<=1), Output>, 3)

# Automates étendus




I = (1, Read , <True, Init>, 2)

U = (2, _ , <(X>1), Update>, 2)

O = (2, Write , <(X<=1), Output>, 3)

# Automates étendus

- Automate + ensemble fini de variables $X=(x_1 , x_2 \ldots x_n )$

  de types *booléen, entier*, *pile, objet de classe toto* ...

- Etat des variables $\sigma: X \longrightarrow D$ $\sigma(X)=(v_1 , v_2 \ldots, v_n )$

  D domaines de valeurs , $\Omega$ ensemble des états

- Configuration : (état de l'automate, état des variables) $(q, \sigma)$
- Transitions : tuplets (q, a, <c,o>, q') où <c,o> est composé
  - d'une condition sur les variables $c : \Omega \longrightarrow \{O,1\}$ (True)
  - d'une opération sur les variables $o : \Omega \longrightarrow \Omega$ (Nop)

# Sémantique opérationnelle

- Configuration = $(q, \sigma)$
- On passe de $(q_1, \sigma_1)$ à $(q_2, \sigma_2)$ s'il existe une transition

  $t=( q_1, a, <c,o>, q_2)$ telle que $c(\sigma_1) = 1$ et $\sigma_2 = o( \sigma_1)$

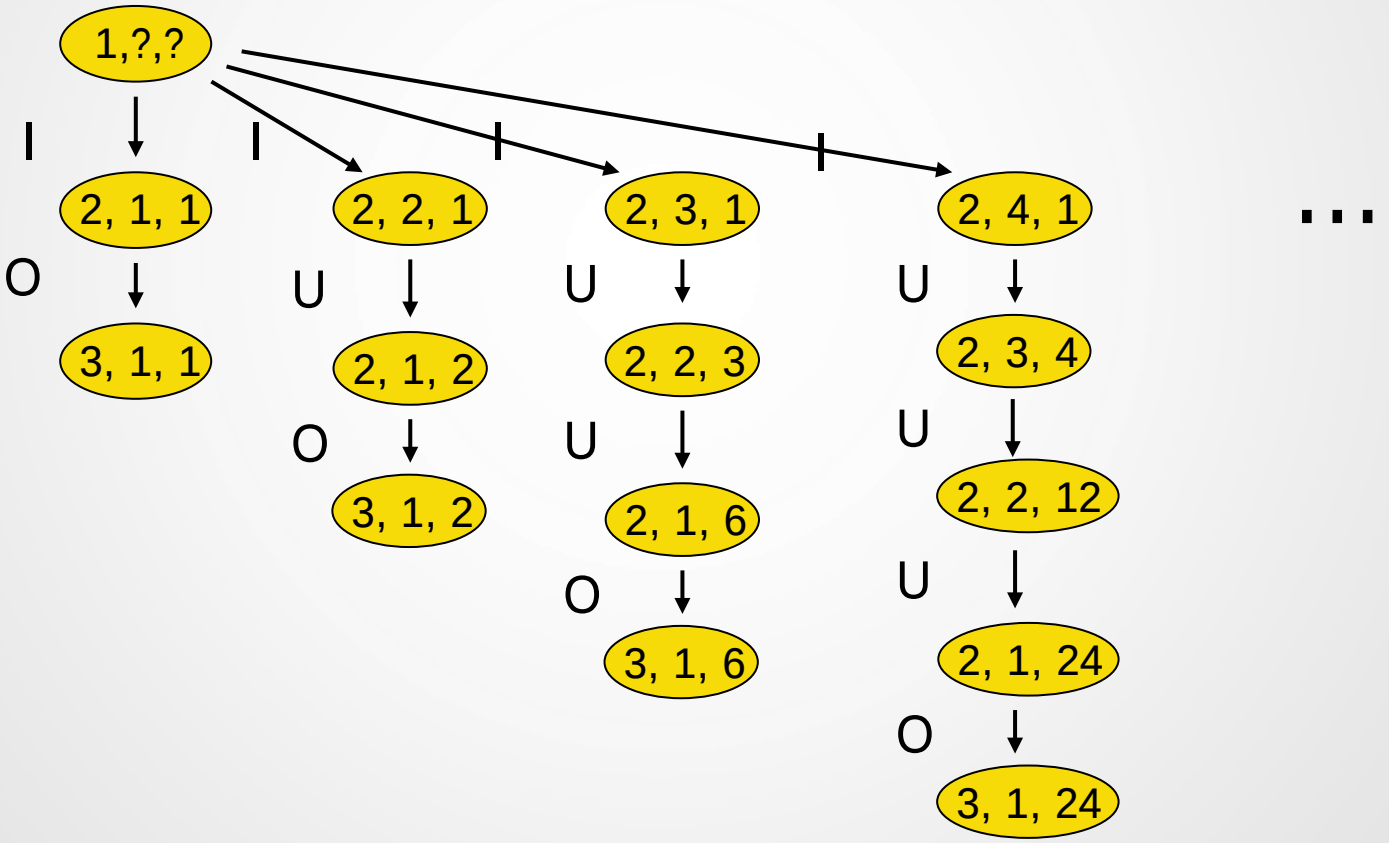
  On note $(q_1, \sigma_1) \xrightarrow{t} (q_2, \sigma_2)$

- Exécution séquentielle

  $(q_1, \sigma_1) \xrightarrow{t_1} (q_2, \sigma_2) \xrightarrow{t_2} \dots (q_n, \sigma_n)$

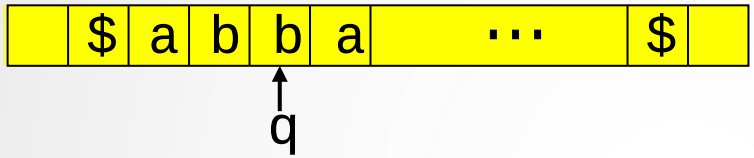
- Graphe des configurations (graphe d'états )
  - sommets $\rightarrow$ configurations accessibles
  - arcs $\rightarrow$ transitions étiquetées entre configurations

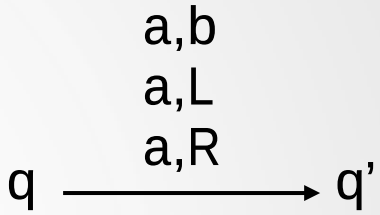
# Graphe d'états



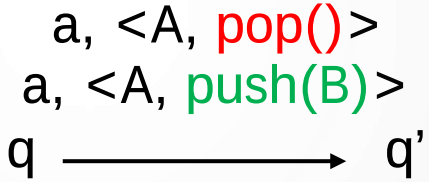
# Automates étendus

- Automate avec bande infinie



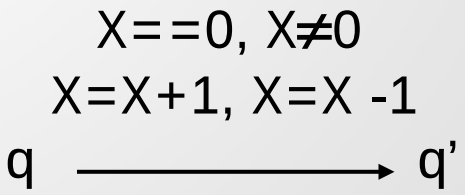

- Automate à pile(s)

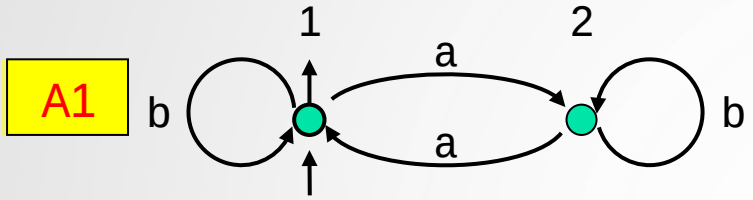

Call(q'),< ?, push(q'')>
Return, <q', pop()>

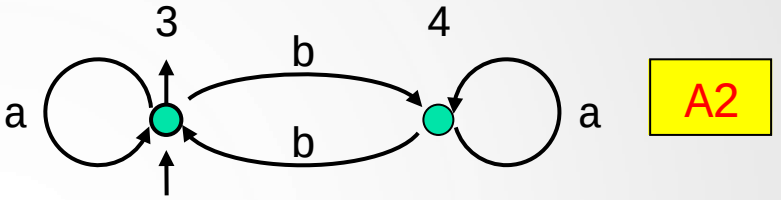
- Automate à file(s)


- Automate à compteur(s)

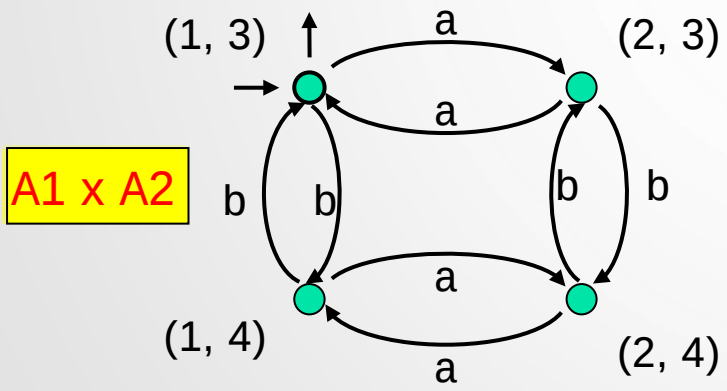
# Produit d'Automates




Langage(A1)= mots sur {a,b} avec nombre pair de a

Langage(A2)= mots sur {a,b} avec nombre pair de b



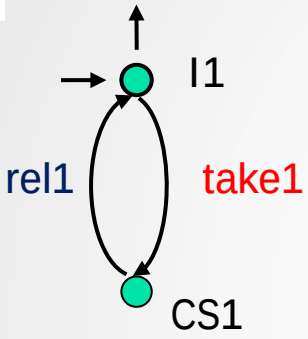

transitions du produit **A1 x A2**

$((q_1 , q_2), a , (q_1', q_2'))$ ssi

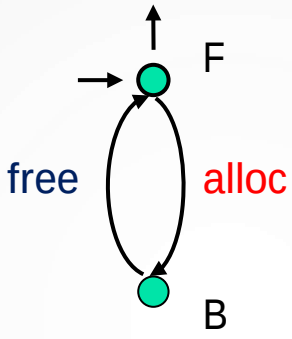
- $(q_1, a , q_1')$ dans A1
- $(q_2, a , q_2')$ dans A2

Langage(A1 x A2) = Langage(A1) ∩ Langage(A2)

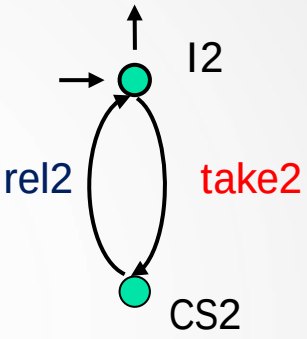
# Produits synchronisés (I)




Utilisateur 1




Ressource exclusive

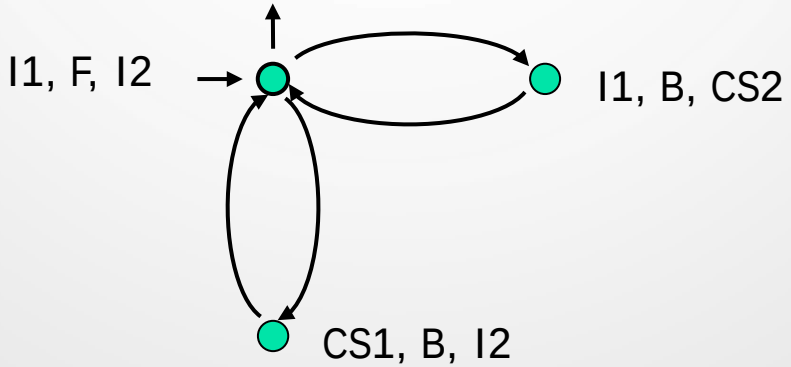



Utilisateur 2

Alphabet de Synchronisation:

* = inactivité

{(take1, alloc, *), (*, alloc, take2), (rel1, free, *), (*, free, rel2)}



# Produits synchronisés

Automates sur alphabets $\Sigma_i$ : $A_i=(Q_i , T_i , I_i , F_i)$ i=1...n

Un alphabet de synchronisation constitué de vecteurs d'actions:

$\Sigma_s \subseteq (\Sigma_1 \cup \{*\}) \times (\Sigma_2 \cup \{*\}) \times \ldots\ldots (\Sigma_n \cup \{*\})$ * = inactivité

# Produits synchronisés

Automates sur alphabets $\Sigma_i$ : $A_i=(Q_i , T_i , I_i , F_i)$ $i=1...n$

Un alphabet de synchronisation constitué de vecteurs d'actions:

$\Sigma_s \subseteq (\Sigma_1 \cup \{*\}) \times (\Sigma_2 \cup \{*\}) \times ..... (\Sigma_n \cup \{*\})$ * = inactivité

Le produit des $\{A_i\}$ synchronisé par $\Sigma_s$ :

- Etats $Q = Q_1 \times Q_2 \times ... \times Q_n$
- Transitions $T \subseteq Q \times \Sigma_s \times Q$

$((q_1 , q_2 ... q_n ), (a_1 , a_2 ... a_n ), (q_1', q_2' ... q_n') \in T$ :

# Produits synchronisés

Automates sur alphabets $\Sigma_i$ : $A_i=(Q_i , T_i , I_i , F_i)$ $i=1...n$

Un alphabet de synchronisation constitué de vecteurs d'actions:

$\Sigma_s \subseteq (\Sigma_1 \cup \{*\}) \times (\Sigma_2 \cup \{*\}) \times ..... (\Sigma_n \cup \{*\})$ * = inactivité

Le produit des $\{A_i\}$ synchronisé par $\Sigma_s$ :

- Etats $Q = Q_1 \times Q_2 \times ... \times Q_n$
- Transitions $T \subseteq Q \times \Sigma_s \times Q$

  $((q_1 , q_2 ... q_n ), (a_1 , a_2 ... a_n ), (q_1', q_2' ... q_n') \in T$ :

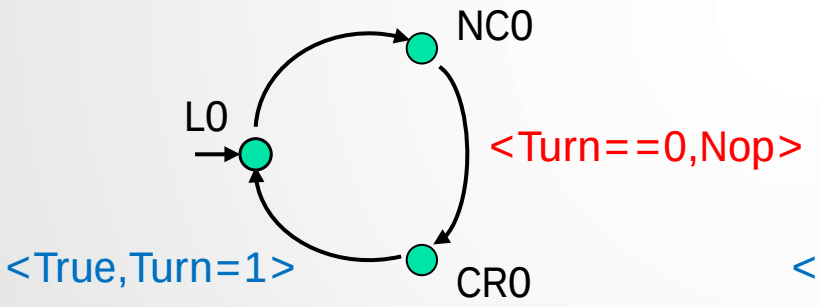
  - Si $a_i = *$ , alors $q_i = q_i'$
  - Sinon $(q_i, a_i , q_1') \in T_i$
- Etat initial $I = (I_1 , I_2 , ... , I_n )$
- Etats finaux $F = F_1 \times F_2 ... \times F_n$

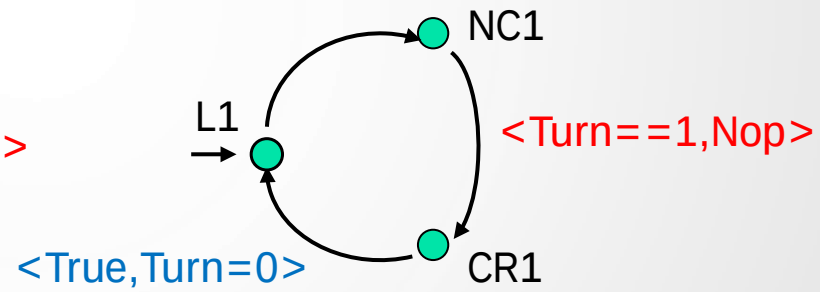
# Produits synchronisés (II)

L0:While True do
  NC0:wait( Turn==0);
  CR0:Turn=1
endwhile

||

L1:While True do
  NC1:wait( Turn==1);
  CR1:Turn=0
endwhile



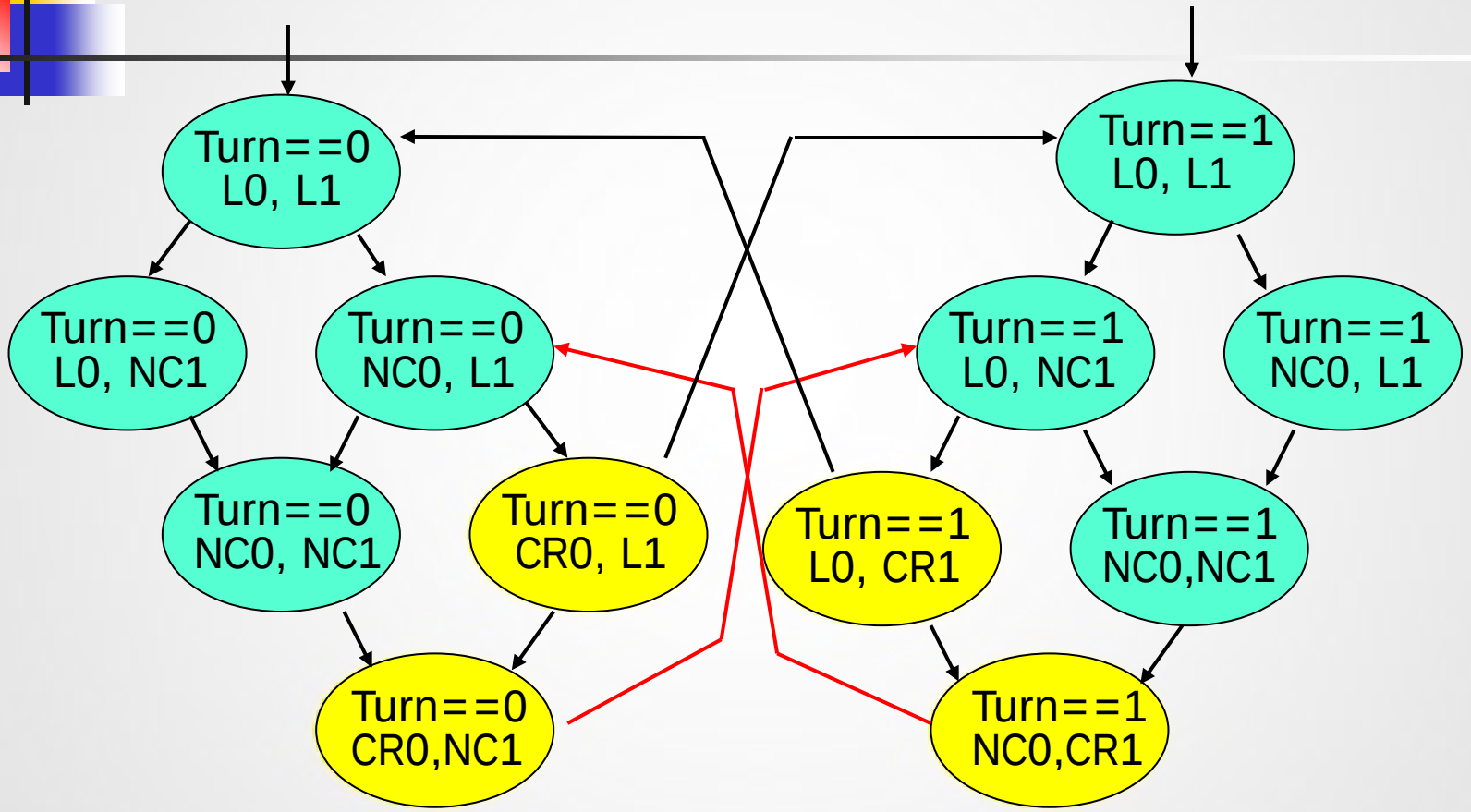




$$\Sigma_s \subseteq (\Sigma_0 \times \{*\}) \cup (\{*\} \times \Sigma_1)$$

Configuration : (PC0, PC1, Turn)

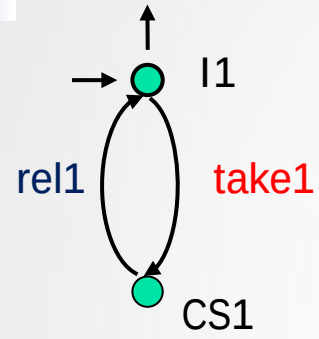
Initialement: PC0=L0 et PC1=L1

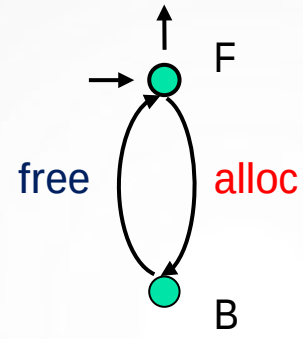
# Le graphe d'états

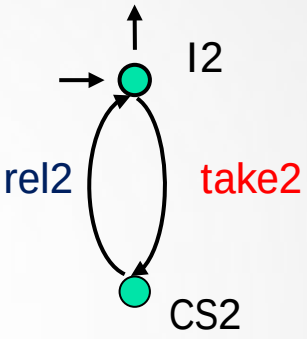


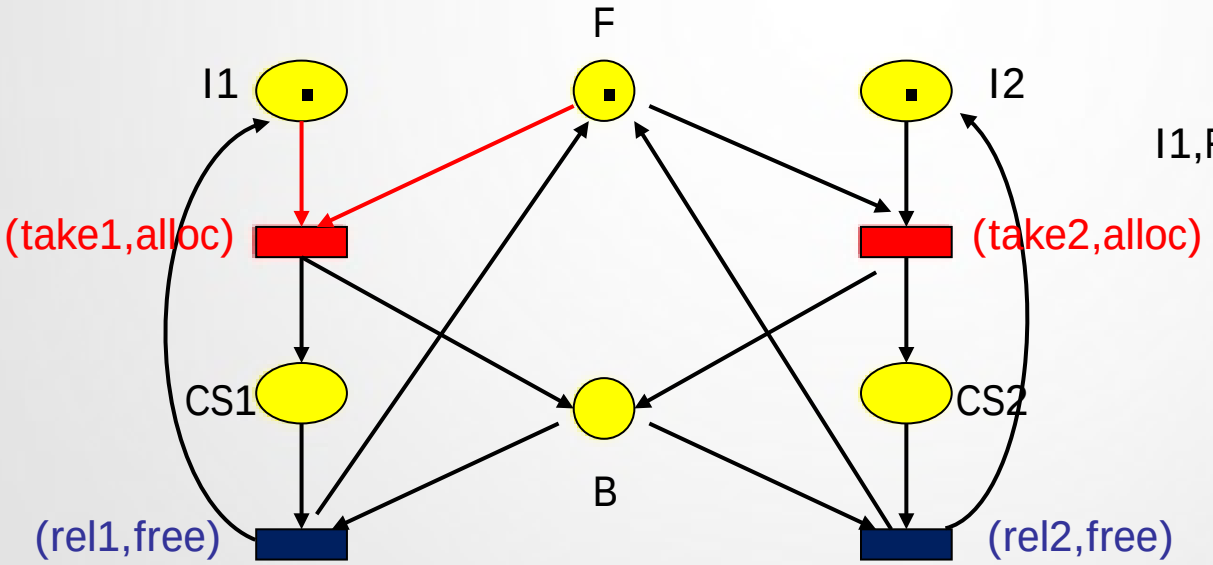

Invariant: $\neg(PC0==CR0 \wedge PC1==CR1)$

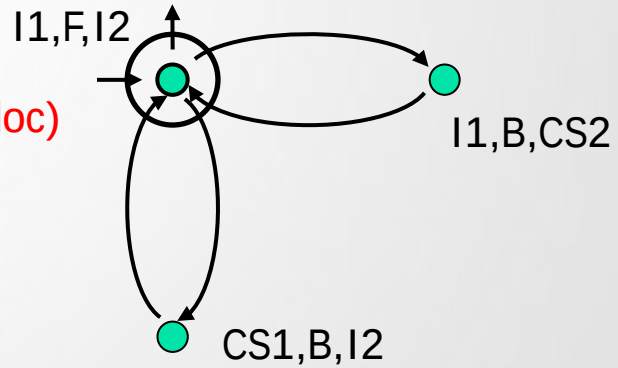
# Produits et Réseaux de Petri



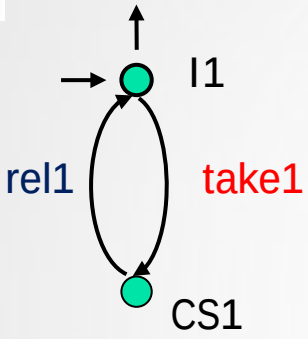



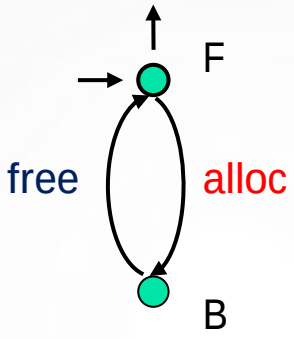



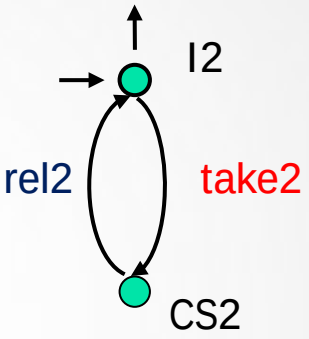



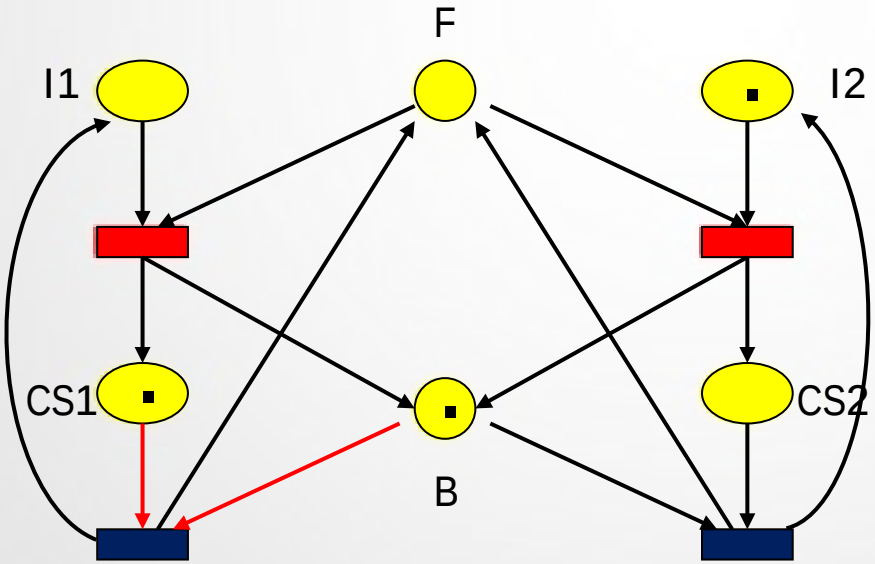



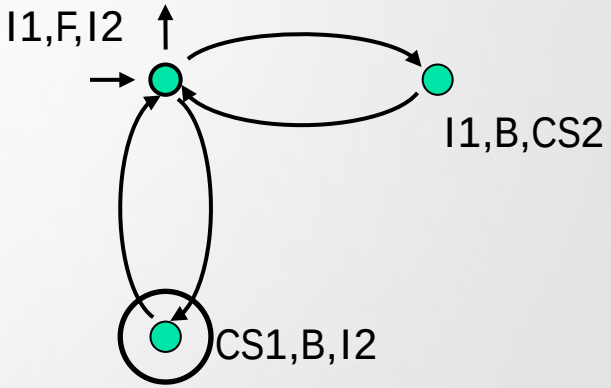
# Produits et Réseaux de Petri

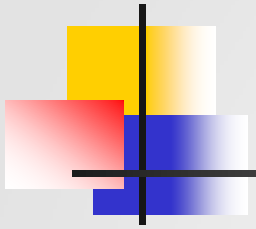


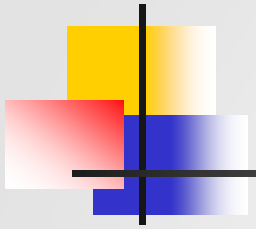





# Abstractions utiles

Les automates sont utiles en tant qu'abstractions des programmes.

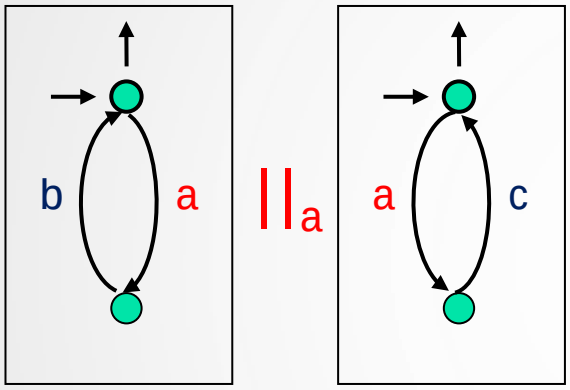
- Projeter sur des composantes des états
  - Les valeurs d'une ou plusieurs variables finies
  - Le *status* d'un processus Unix, des threads
  - L'état des connexions
- Projeter sur des ensembles d'actions cohérents
  - Les synchronisations avec l'environnement
  - Les appels/retours de procédures (récursivité)
  - Les lock et unlock des mutex (blocages)
  - Les émissions et réceptions de messages
- Renommages, quotients, ...



# Plan

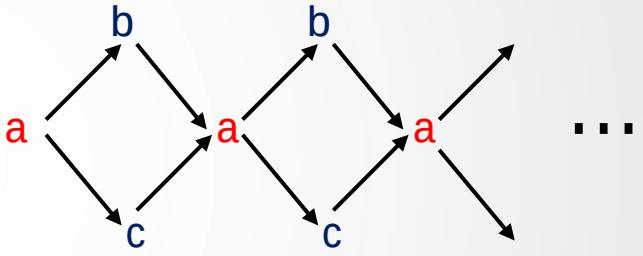
- Introduction
- Machines
  - Syntaxe et sémantique
  - Automates étendus
  - Produits d'automates
  - MSC et automates communicants
- Logiques
  - Exprimer les propriétés des systèmes
  - Logiques temporelles
  - Model-checking régulier
  - Le cas des MSC
- Conclusions

# Sémantiques concurrentes




abcacba...

Séquence d'actions




Relation de causalité

Produits synchronisés, réseaux de Petri , automates communicants etc...:
causalité/indépendance entre évènements

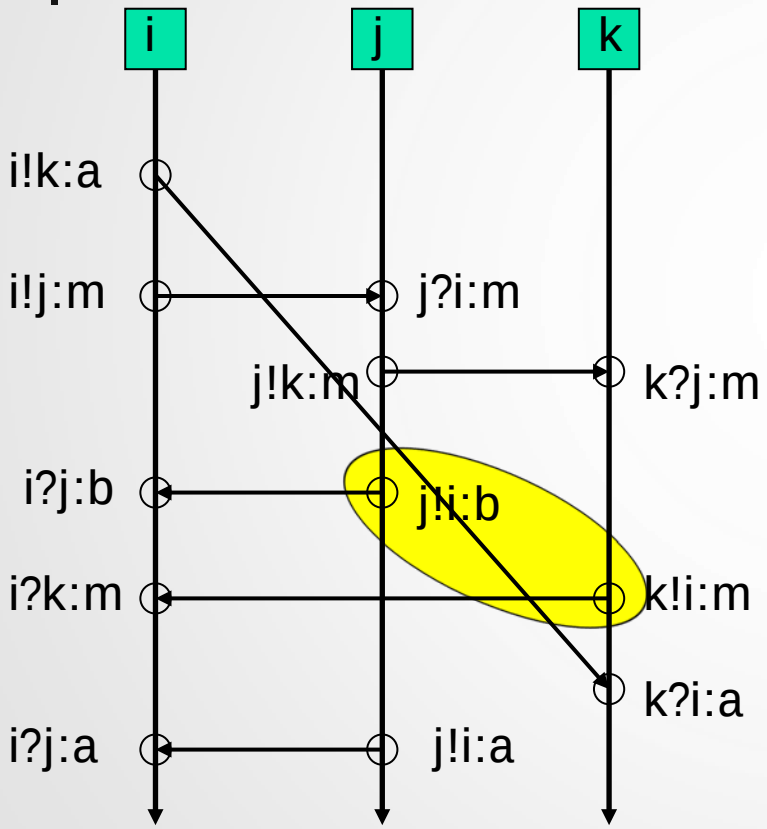
Modèles de comportement représentant la causalité:

- Ordres partiels /graphes acycliques étiquetés
- Ex: Message Sequence Charts (MSC)

Consistance avec la sémantique séquentielle :

les entrelacements coincident avec les séquences d'exécution

# Message sequence charts



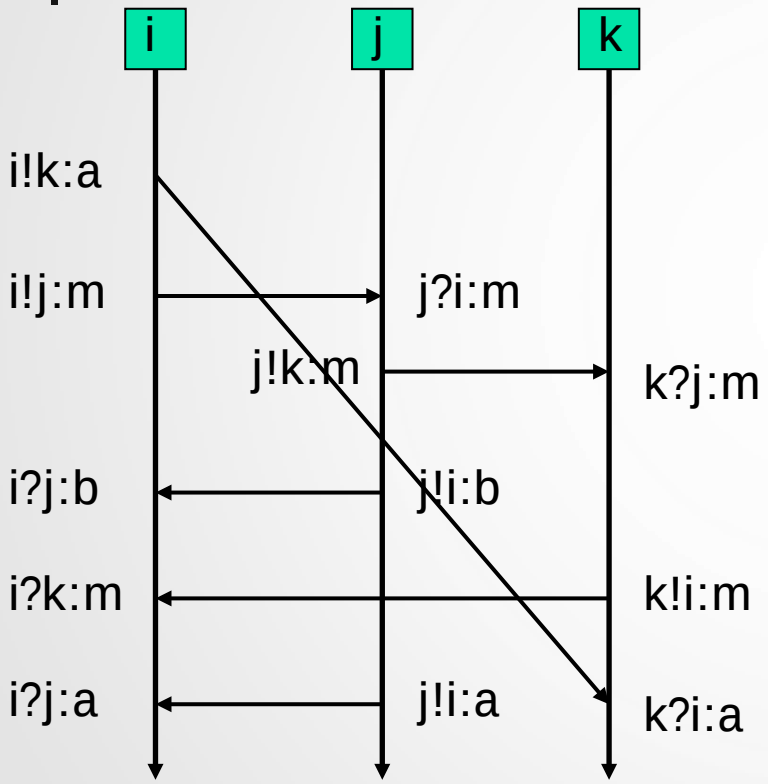

Décrit une exécution d'un système distribué:

- ensemble d'évènements ○
- étiquetés par des actions i!k:a
- relation de précédence entre évènements

Graphe étiqueté acyclique

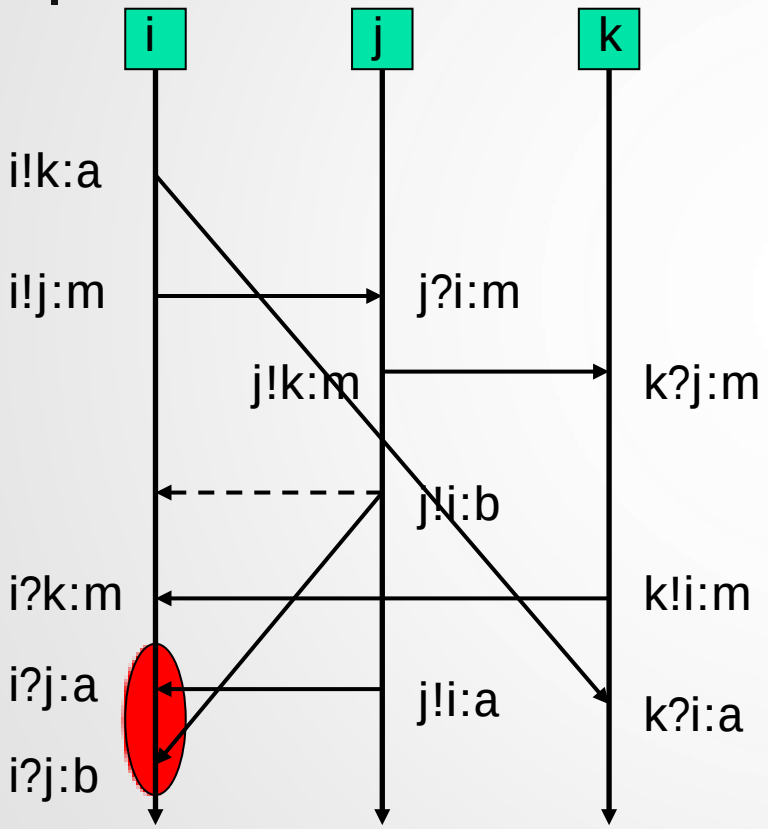
- Ordre partiel sur evènements
  - Evènements dépendants
  - Evènements concurrents

# Message sequence charts



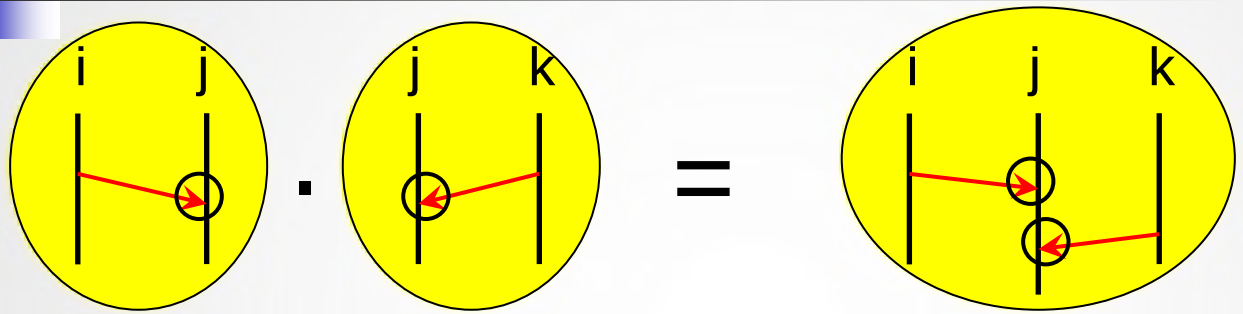

- Ensemble fini de processus i, j, k
- Ensemble de messages m, a, b
- Pour chaque instance (i): séquence de sommets étiquetés par
  - émissions de message vers {j,k}
    i!k:a
  - réceptions de message de {j,k}
    i?j:b
- Arcs entre émissions et réceptions

# Message sequence charts




- Ordre FIFO de réception des messages entre deux instances

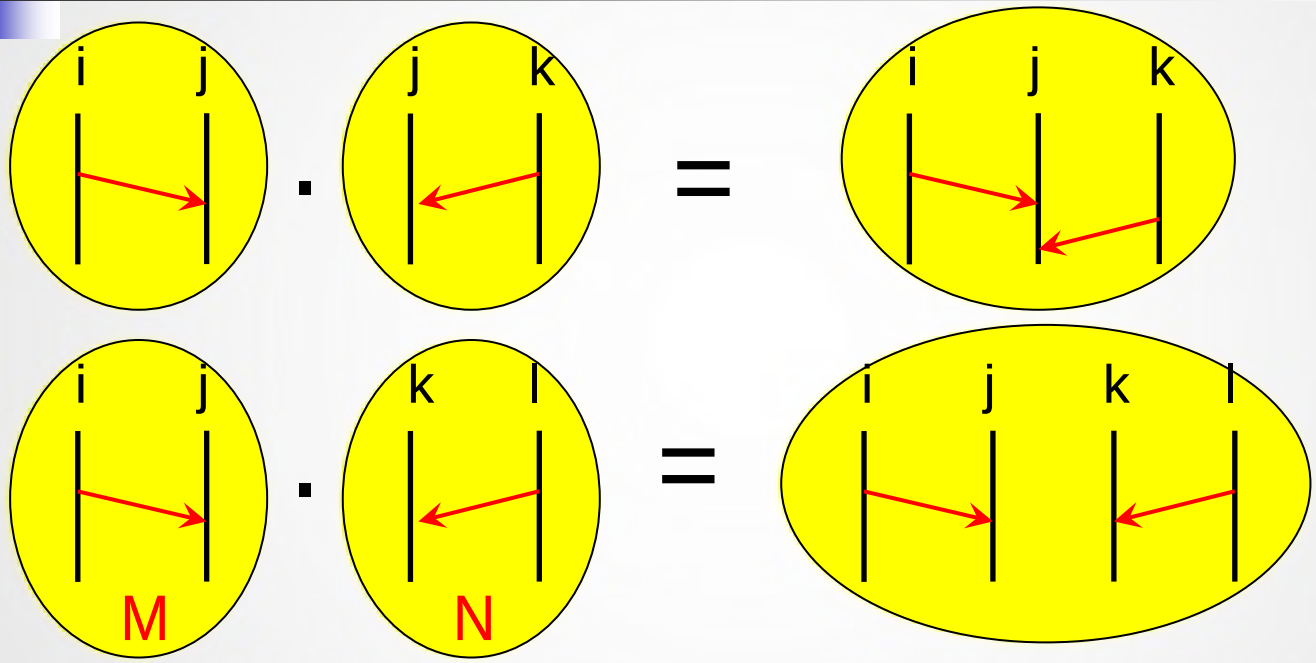
# Composition de MSC




on conserve la structure des composants

on n’ajoute d’arcs qu’entre évènements d’une même instance

# Composition de MSC

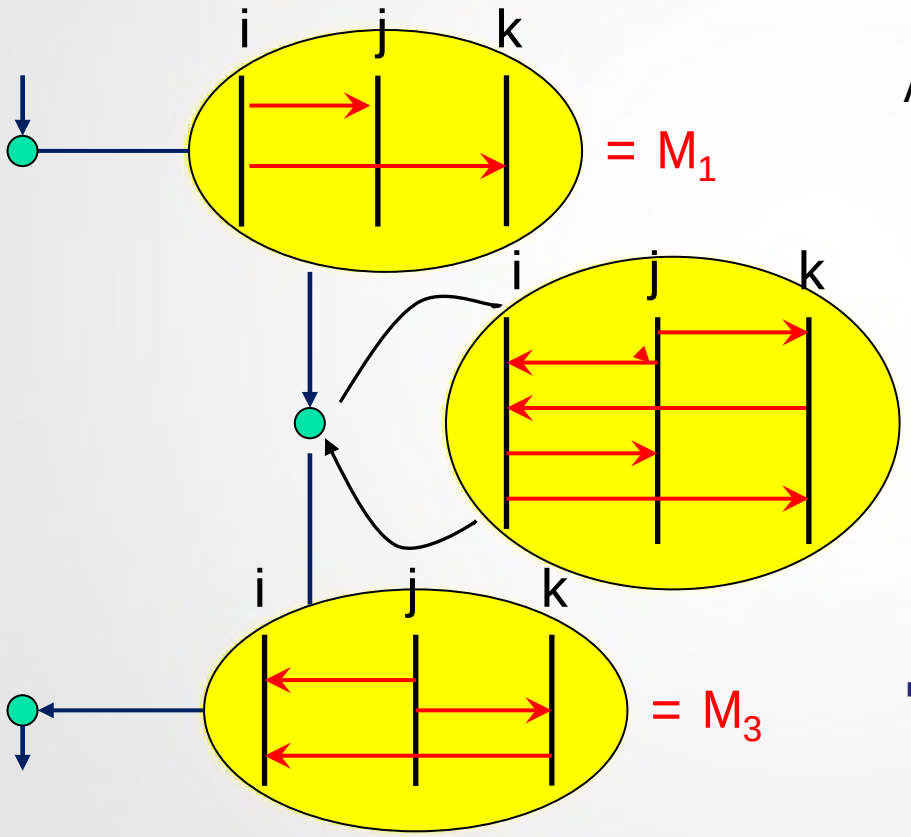



Si A,B et C sont des MSC , A.B.C est un MSC

Les MSC indépendants ( instances disjointes ) commutent

M.N=N.M

# Graphes de MSC (HMSC)



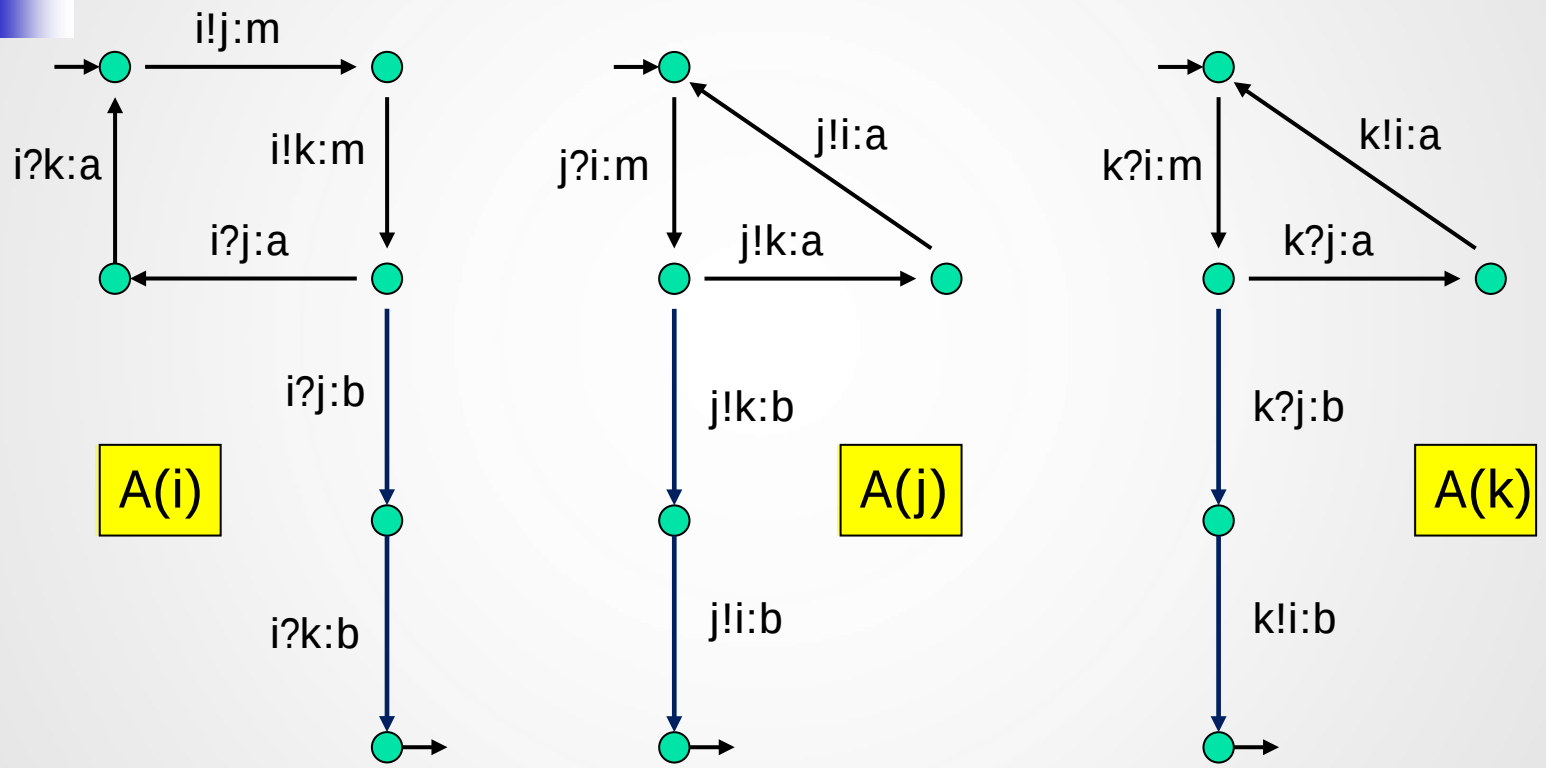

Automate sur un alphabet de MSC

Langage : $M_1 . M_2 *. M_3$

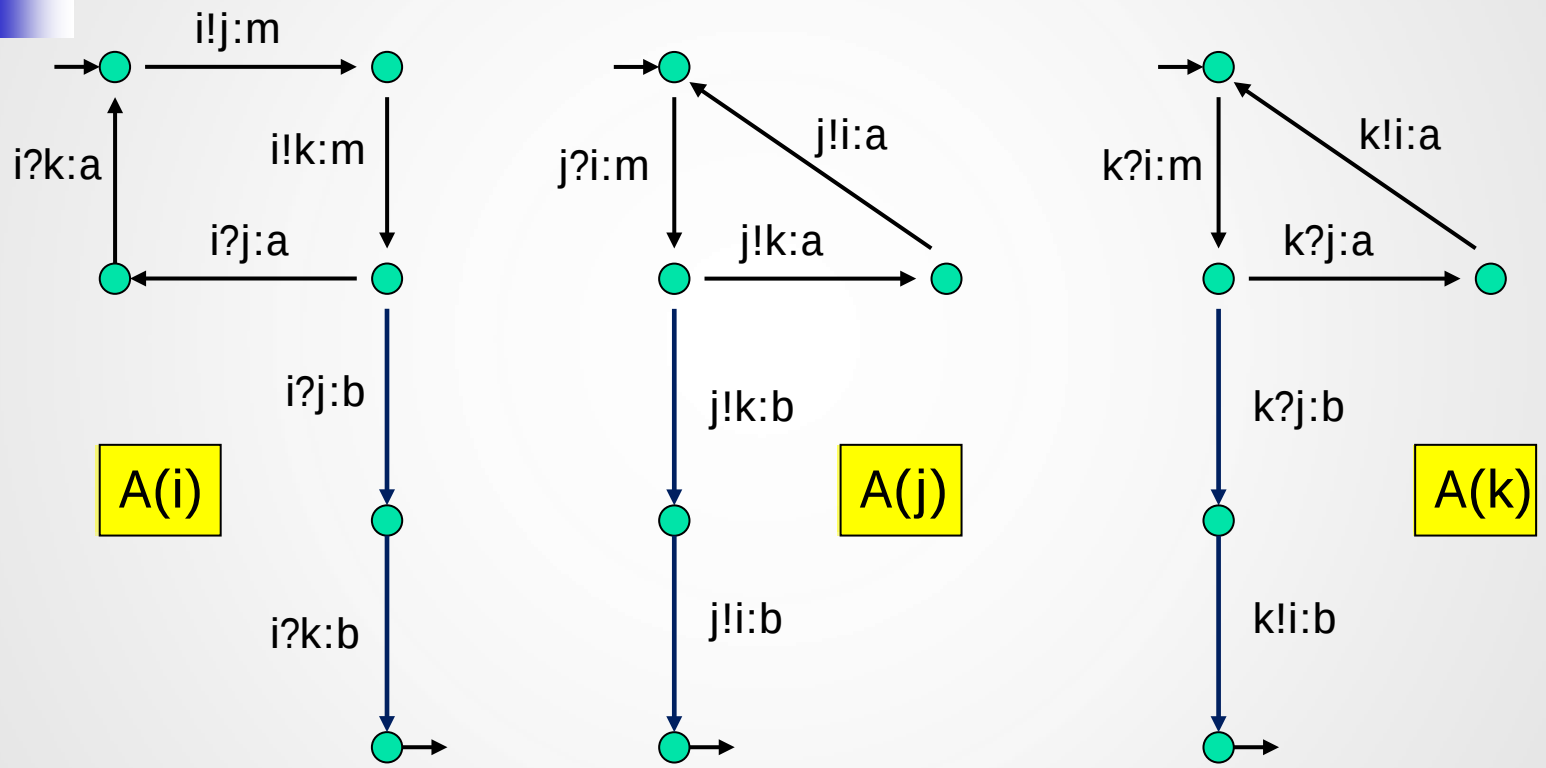
- Canaux bornés : Loop-st-connected
  - les MSC des boucles sont fortement connexes

Langages de MSC finiment engendrés

# Automates communicants



# Automates communicants

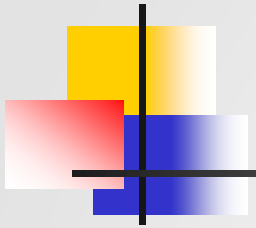



L'ensemble {A(i),A(j),A(k)} a le même comportement distribué,

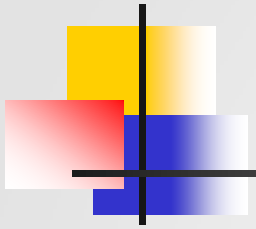
i.e. le même langage de MSC, que le HMSC précédent

Implémentation distribuée d'une spécification centralisée

# Automates communicants CFSM



- Instances $I=\{1....n\}$
- Automates $A_1, A_2 \ldots A_n$ , canaux $C \subseteq I \times I$ , messages $M$
- Alphabets des $A_i$
  - Emissions de message m dans canal c : c ! m ( cf i ! j : m)
  - Réceptions de message m par canal c : c ? M ( cf i ? j : m)
- Configurations
  - Couple (états des automates, états des canaux)
  - Etat de canal = suite de messages émis non reçus ( mot de $M^*$)
  - $((q_1, q_2.. q_n), (s_1, s_2.. s_{|C|}))$



# Automates communicants

- Instances $I=\{1....n\}$
- Automates $A_1, A_2 \ldots A_n$ , canaux $C \subseteq I \times I$ , messages $M$
- Alphabets des $A_i$
  - Emissions de message m dans canal c : c ! m ( cf i ! j : m)
  - Réceptions de message m par canal c : c ? M ( cf i ? j : m)
- Configurations
  - Couple (états des automates, états des canaux)
  - Etat de canal = suite de messages émis non reçus ( mot de $M^*$)
  - $((q_1, q_2.. q_n), (s_1, s_2.. s_{|C|}))$
- Transitions globales : un seul automate avance
  - c ! m : dépose un message m en queue de canal c
  - c ? m : prélève un message m en tête de canal c s'il est présent
  - Change d'état local
- Un MSC unique déduit de chaque séquence d'exécution (Fifo)

# Message sequence charts

Instances $I$

Pour tout ensemble de MSC $L \subseteq MSC(I)$

à canaux bornés

| ∃ aut. communicant $A$ | ⟷ | ∃ Graphe de CMSC $G$ (*) |
|---|---|---|
| $L = L(A)$ | | $L = L(G)$ |

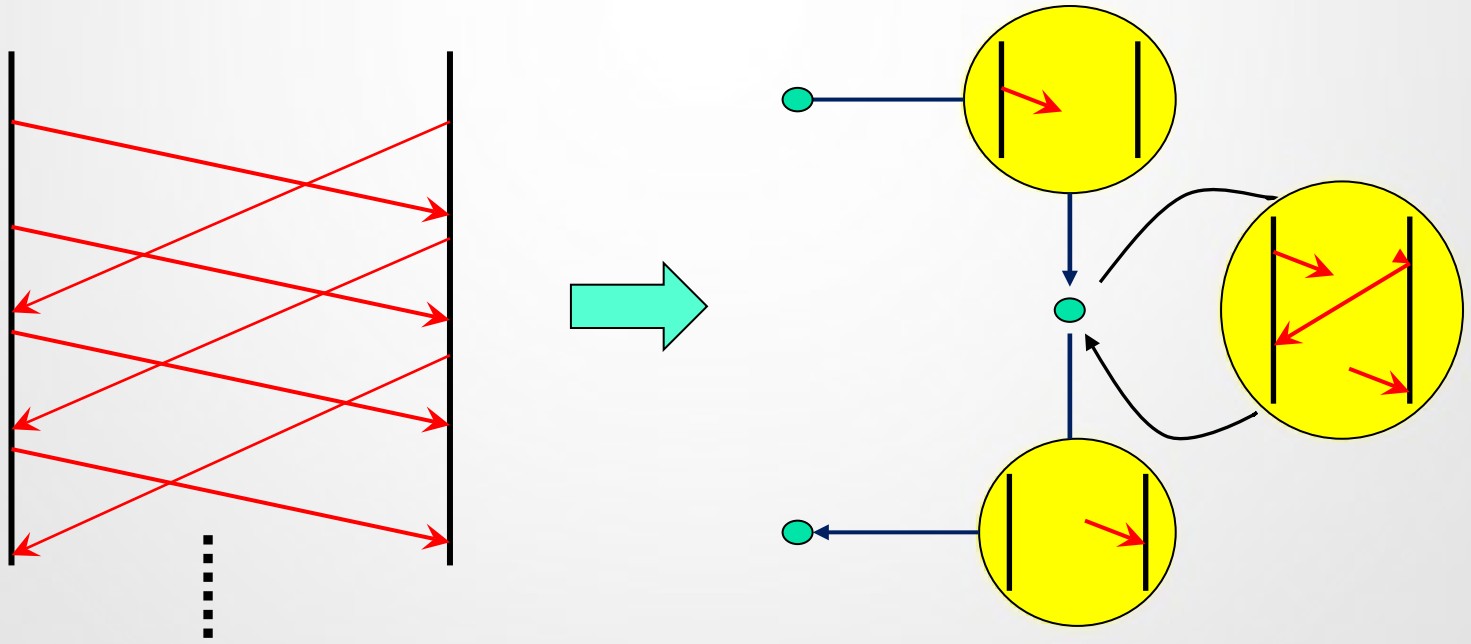
$$L = L(A) = L(G)$$

(*) Hypothèse sur les boucles : loop st-connected

Pour les AC, même avec un seul canal, la finitude des canaux est indécidable (≠ ils sont bornés par un entier donné k)

# Graphes de MSC composables

Les AC , même à canaux bornés, peuvent avoir un langage de MSC non décomposables en nombre fini de MSC : on introduit des émissions et réceptions isolées.



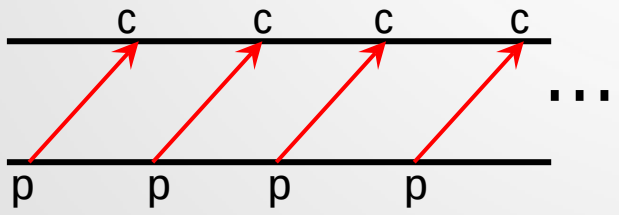
# Message sequence charts

Pour tout ensemble de MSC *L*
à canaux existentiellement bornés

| ∃ aut. communicant *A* | ⟺ | ∃ Graphe de CMSC *G* (**) |
|---|---|---|
| *L* = L*(A)* | | *L* = L*(G)* |

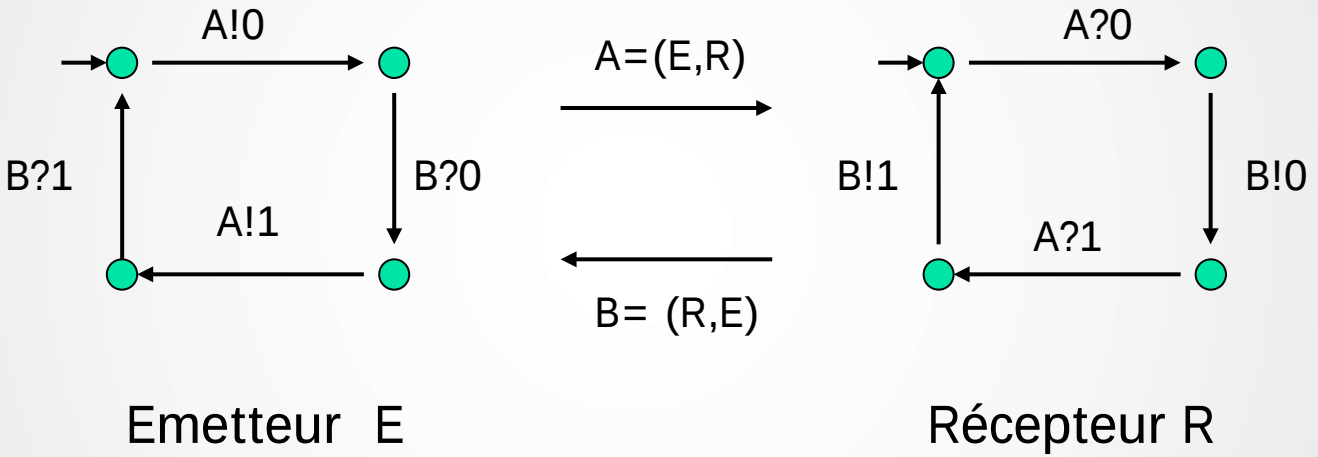
M a des canaux existentiellement bornés s'ils sont bornés pour un certain ordre séquentiel des évènements de M

Ex : le producteur consommateur est Exist$^{t}$ borné par 1



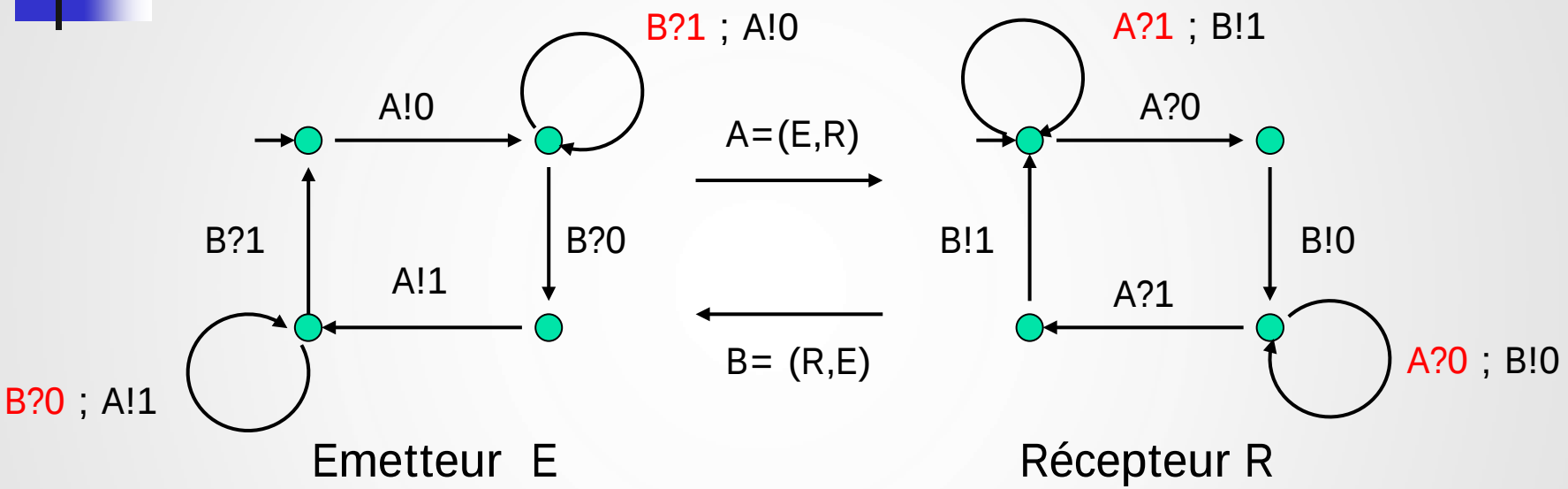

pcpcpcpc...

# Protocole du Bit alterné

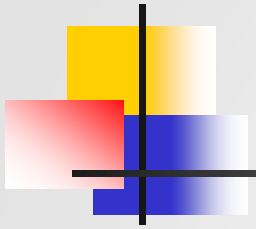



- Etats des canaux : {0, 1}
- Et si les canaux perdent des messages ?

# Protocole du Bit alterné

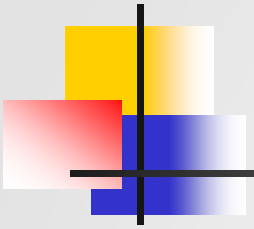



- Etats des canaux : séquences de {0, 1}
- Les canaux peuvent perdre des messages
- Emetteur : émissions de 0 (resp. 1) jusqu'à réception de 0 (resp. 1)
- Récepteur : émissions de 0 (resp. 1) jusqu'à réception de 1 (resp. 0)



# Plan

- Introduction
- Machines
  - Syntaxe et sémantique
  - Automates étendus
  - Produits d'automates
  - MSC et automates communicants
- Logiques
  - Exprimer les propriétés des systèmes
  - Logiques temporelles
  - Model-checking régulier
  - Le cas des MSC
- Conclusion



# Logiques et modèles

- Exprimer les propriétés des graphes d'états
  - Le système finit par s'arrêter : terminaison ou blocage
  - On peut atteindre un état particulier : accessibilité
  - Tous les états vérifient une propriété donnée: invariant
  - Toute requête finit par être satisfaite
  - On atteint un état qui vérifie la condition p, peut exécuter l'action a etc
  - Relations entre états : logiques temporelles
- Décidabilité
  - Espace d'états finis :

    Accessibilité, invariant, logiques temporelles, etc
  - Espaces d'états infinis

    Dépend de l'expressivité du modèle

# Expressivité et décidabilité

- Automate fini + 1 pile ; Automate fini + 1 compteur

  Accessibilité décidable

- Réseaux de Petri , automates à compteurs restreints

  Accessibilité décidable

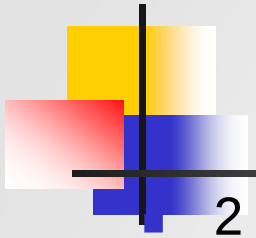
- Machines de Turing : Automate fini + 1 bande infinie

  Accessibilité indécidable

- Automate fini + 2 piles ; + 2 compteurs ; + 1 file

  Accessibilité indécidable ( $\Leftrightarrow$ Machines de Turing)

Recherche et résultats pour de nombreuses classes composées modélisant les différents types de programmes



# Logiques Temporelles

- 2 niveaux de propriétés des états d'un système
  - Propriétés « atomiques » des états
    - x + 3y < 2 , pile vide — state based logic
    - on peut exécuter " a " — action based logic
  - Propriétés de leur futur ( leurs états successeurs dans le modèle) exprimées par des opérateurs appelés modalités

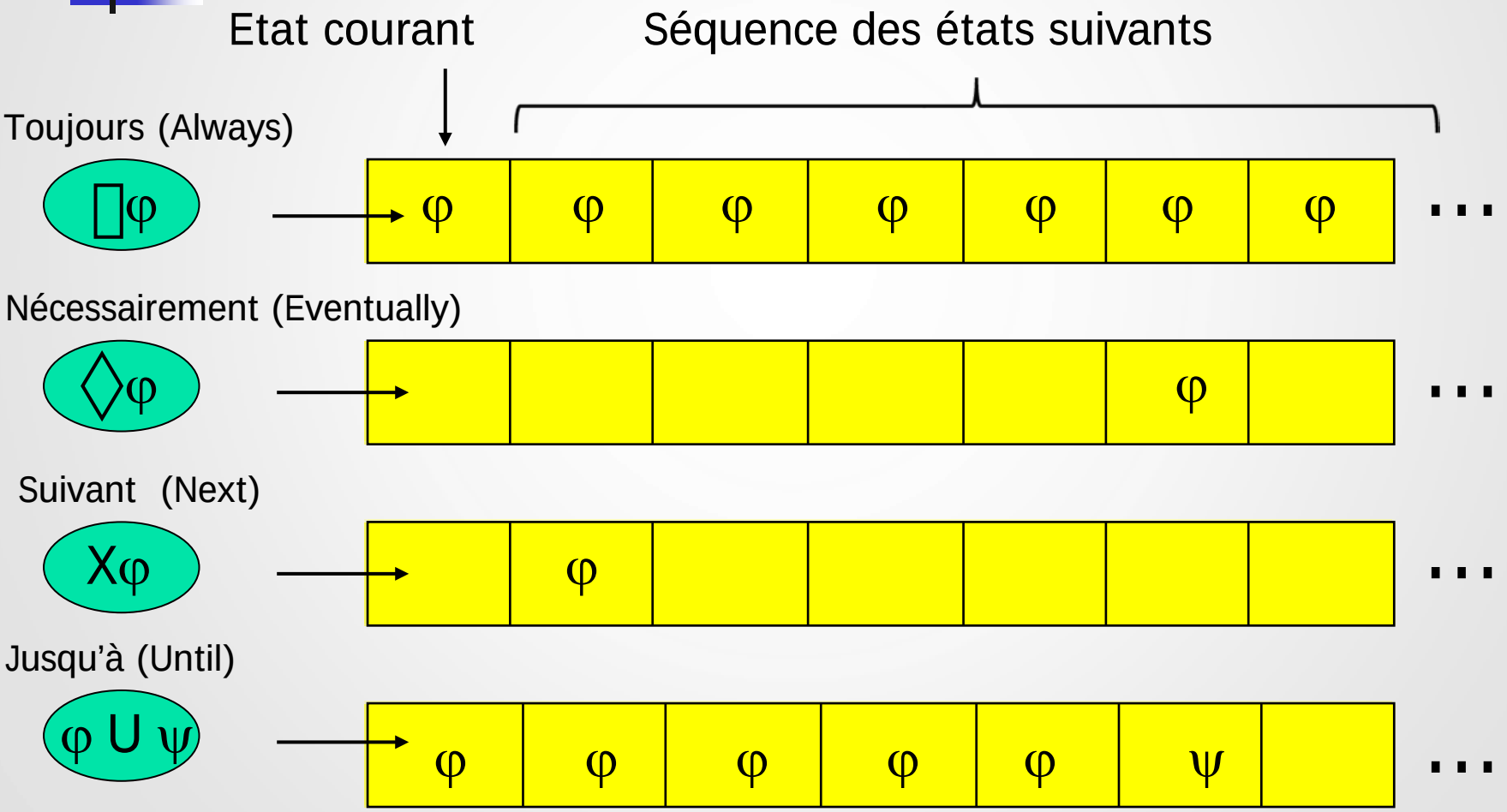
# Logiques Temporelles

- 2 niveaux de propriétés des états d'un système
  - Propriétés « atomiques » des états
    - x + 3y < 2 , pile vide — state based logic
    - on peut exécuter " a " — action based logic
  - Propriétés de leur futur ( leurs états successeurs dans le modèle) exprimées par des opérateurs appelés modalités
- 2 types de logiques (modalités) selon qu'on exprime les propriétés
  - Des exécutions séquentielles : temps linéaire (Linear Temporal Logic,...). Chaque état a au plus un successeur, et un seul futur
    - " de cet état on atteint un état qui vérifie p " , " on exécute a et l'état suivant vérifie p "

# Logiques Temporelles

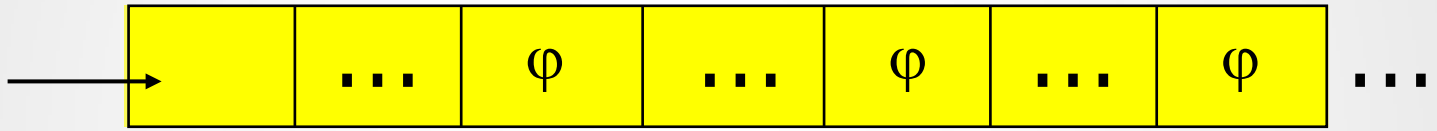
- 2 niveaux de propriétés des états d'un système
  - Propriétés « atomiques » des états
    - x + 3y < 2 , pile vide — state based logic
    - on peut exécuter " a " — action based logic
  - Propriétés de leur futur ( leurs états successeurs dans le modèle) exprimées par des opérateurs appelés modalités
- 2 types de logiques (modalités) selon qu'on exprime les propriétés
  - Des exécutions séquentielles : temps linéaire (Linear Temporal Logic,...). Chaque état a au plus un successeur, et un seul futur
    - " de cet état on atteint un état qui vérifie p " , " on exécute a et l'état suivant vérifie p "
  - Des graphes : temps avec branchements (CTL, CTL*...). Un état peut avoir plusieurs successeurs directs et plusieurs futurs
    - " dans un des futurs de cet état, dans tous les futurs, on atteint un état qui vérifie p "

# Modalités de LTL

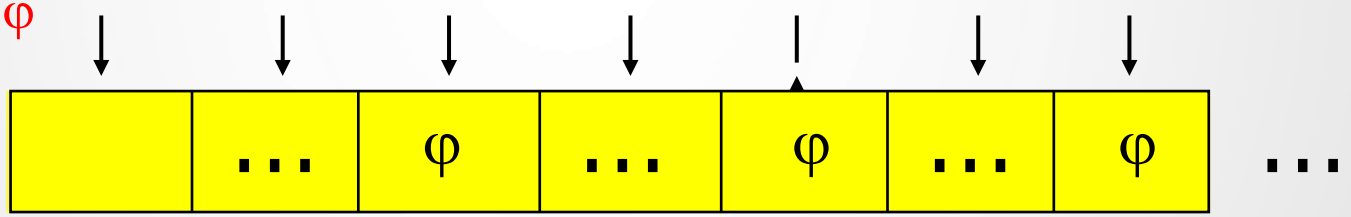


# Combinaisons

- □◇ φ : φ est vrai un nombre infini de fois dans le futur

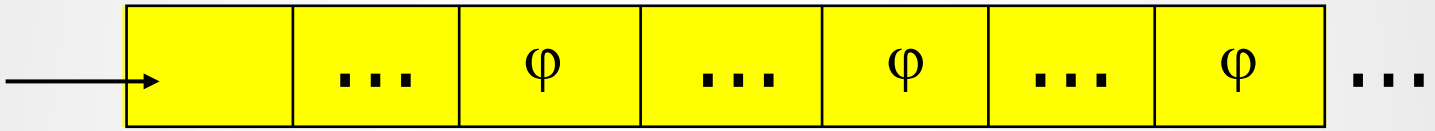



- ◇ φ

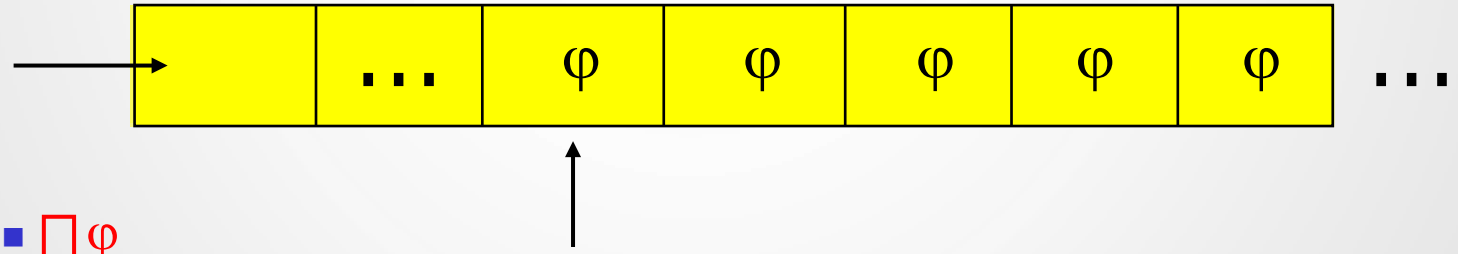


# Combinaisons

- $\Box\Diamond\varphi$ : $\varphi$ est vrai un nombre infini de fois dans le futur



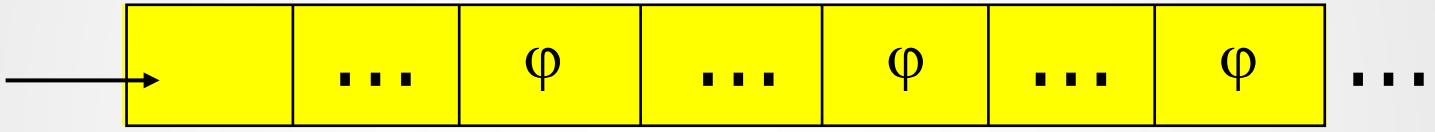

- $\Diamond\Box\varphi$ : à partir d'un état futur, $\varphi$ est toujours vrai



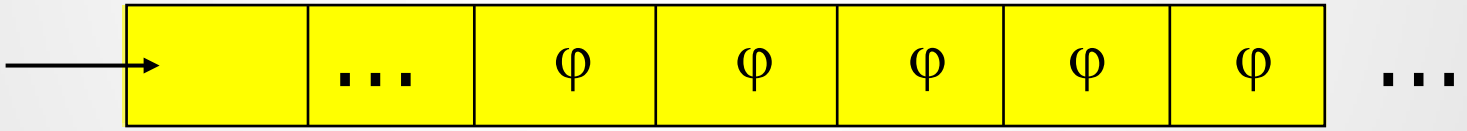

- $\Box\varphi$

# Combinaisons

- $\Box\Diamond\varphi$ : $\varphi$ est vrai un nombre infini de fois dans le futur




- $\Diamond\Box\varphi$ : à partir d'un état futur $\varphi$ est toujours vrai



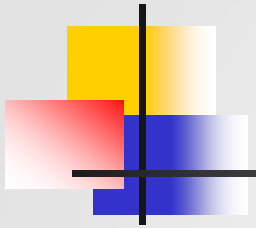

- $\Box(\text{Turn}=0 \Rightarrow \Diamond\, \text{Turn}=1)$ : toujours si un état vérifie Turn = 0, alors dans son futur un état vérifie Turn = 1

- $\bigwedge_i \Box(\ \text{take}_i \Rightarrow X\ (\neg\ \text{take}_i\ U\ \text{release}_i))$ ??

# Problèmes liés à la logique

Une logique $\mathcal{L}$, une classe de modèles $\mathcal{C}$ (séquences, graphes , etc )

- Satisfiabilité
  - Une formule $\varphi$ de $\mathcal{L}$
  - Existe-t-il un modèle $M \in \mathcal{C}$ qui satisfait $\varphi$ ?
- Model checking
  - Une formule $\varphi$
  - Un modèle $M \in \mathcal{C}$
  - Est-ce que $M$ satisfait $\varphi$ , noté $M \models \varphi$



# Model-checking régulier

Alphabet $\Sigma$

Pour tout ensemble de mots $L \subseteq \Sigma^*$

$L$ est le langage d'un automate fini

$\Leftrightarrow$

$L$ est l'ensemble des mots satisfaisant une formule MSO (*)

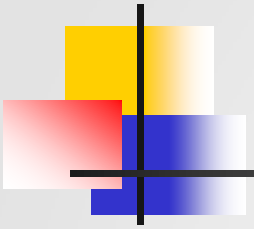
En particulier pour toute formule LTL $\varphi$ il existe un automate fini $A_\varphi$ tel que

$$L(A_\varphi) = L(\varphi)$$

(*) Logique Monadique du Second Ordre, plus expressive que les logiques temporelles

# Model-checking régulier

- Le modèle: un graphe d'états fini $S$
- La propriété: une formule LTL $\varphi$
- On montre que $S \models \varphi$ en montrant que $L(S) \subseteq L(\varphi)$ , c'est-à-dire $L(S) \cap L(\neg\varphi) = \varnothing$ (mots de $S$ qui ne satisfont pas $\varphi$)
  - On construit l'automate $A_{\neg\varphi}$
  - On fait le produit $S \times A_{\neg\varphi}$
  - On vérifie que $L(S) \cap L(A_{\neg\varphi}) = L(S \times A_{\neg\varphi}) = \varnothing$
- Problèmes
  - Construire l'automate $A_{\neg\varphi}$
  - Exp$^{\text{tiel}}$ en taille de $\varphi$ , poly en taille de $S$



# Limites du Model-checking régulier ?

- De nombreux protocoles sont à états finis
- De nombreux programmes permettent des abstractions à états finis: on valide un sur-ensemble des exécutions
- On peut décomposer un programme et valider les composants séparément
- Nombreuses techniques pour réduire la taille de l'espace d'états (BDDs, Partial Order Reduction)
- Performance et mémoire des ordinateurs

# Variantes du MC

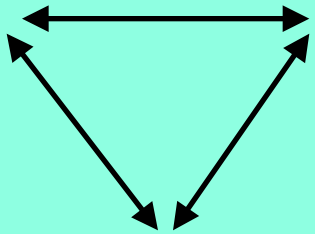
- Model-checking à la volée (on the fly)
  - On vérifie le graphe d'états au fur et à mesure de sa construction
- Weighted LTL


- Bounded model-checking
  - On vérifie sur des séquences d'au plus k états (avec ou sans boucles)
- Counter example Guided Abstraction Refinement (CEGAR)
  - Si le model-checking d'une abstraction donne un contre–exemple, on vérifie qu'il existe dans le modèle complet. Si oui on raffine l'abstraction en utilisant le contre-exemple.

# Model-checking et MSC

Instances $I$
Pour tout ensemble de MSC $L \subseteq MSC(I)$
à canaux bornés

$\exists$ aut. communicant $A$
$L = L(A)$

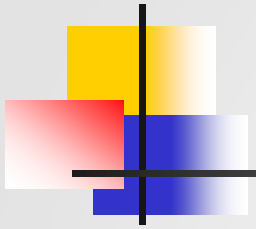


$\exists$ Formule MSOL $\varphi$
$L = L(\varphi)$

$\exists$ Graphe de CMSC $G$ (*)
$L = L(G)$

Model-checking d'automates communicants

# Modèles temporisés



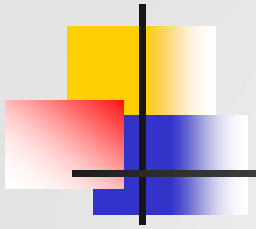
- Automates temporisés
- Réseaux de Petri  temporisés
- Automates communicants temporisés
- Algèbres de processus temporisés

- Automates probabilistes

# Outils et liens

- en.wikipedia.org/wiki/List_of_model_checking_tools
- Spin (Promela, SDL) spinroot.com/spin/whatispin.html
- SMV (Symbolic Mchecker) www.cs.cmu.edu/~modelcheck/tour.htm
- Tina (Time Petri Nets + LTL Mchecking ) www.laas.fr/tina/
- Uppaal (Timed models) www.uppaal.org/
- Petri Nets tools www.informatik.uni-hamburg.de/TGI/PetriNets/tools/
- MSC tools
  www-i2.informatik.rwth-aachen.de/Research/AG/MCS/MSC/Tools/msc.html
- SMA Smyle Modeling Approach
- Altarica https://altarica.labri.fr/forge/
- CADP cadp.inria.fr/
- PRISM (Automates probabilistes,..) www.prismmodelchecker.org/
- YASM, Java Pathfinder

# Conclusion



- Comme tout formalisme:apprentissage
- Diagrammes avec sémantique
- Guide  à la compréhension/structuration
- Vérification
- Communiquer/collaborer/maintenir
- Outils logiciels

# Problème de la terminaison des programmes

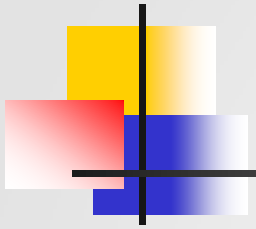
- Chaque programme p codé par un entier #p unique
- Halt(x,y) : prédicat « le programme p tq #p=x termine sur input y »
  .
- Supposons Halt(x,y) calculable , alors il existe un programme P(x)

  [A] IF Halt(x,x) GOTO A
- P(x) termine ssi Halt(x,x) est faux
- Soit #P = t on a pour tout x:

  HALT(t,x) $\Leftrightarrow$ ~HALT(x,x), et donc pour x = t

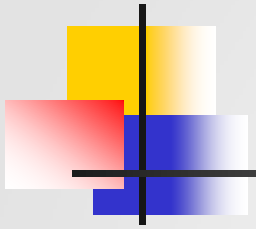
  HALT(t,t) $\Leftrightarrow$ ~HALT(t,t) Contradiction

Ref : Davies, Weyuker. *Computability, Complexity and Languages.*



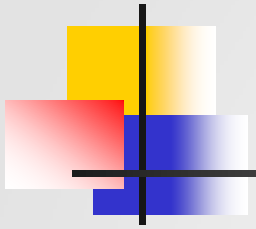
# Types de programmes

- Programmes séquentiels
  - Controle fini, nombre fini de registres
    - Bornés , non bornés
  - Procédures et récursivité , controle fini + pile
- Programmes concurrents/distribués
  - Threads parallèles (séquentiels)
    - nombre fixe
    - dynamique, nombre non borné
- Communication/synchronisation
  - Mémoire partagée (atomicité)
  - Rendez-vous
  - Canaux
    - Bornés, non bornés
    - Ordonnés ( FIFO ), non ordonnés, à pertes



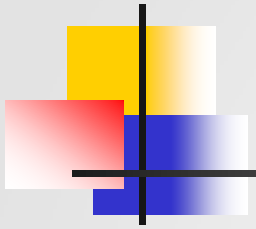
# Programmes itératifs

- #Thread fixe + variables bornées
  - Etats expo en #threads et “ variables
  - Décidable , Access Pspace-complet
- #Thread fixe + canaux  bornées
  - Etats expo en # threads et # canaux
  - Décidable , Access Pspace-complet
- # Thread non borné
  - Indécidable avec threads ids , décidable sinon
- Creation dyn de thread non récursifs et anonymes
  - Décidable , coverability in PNets



# Programmes récursifs

- Concurrent recursive programs
  - 2 threads = MT
- Concurrent asynchronous programs
  - Appels stockés, exécutions asynchrones
- Multiset PDS
  - Task sequential recursive + dynamic task creation
  - Decidable  EXPspace complet
- Message passing + procédures
  - Indecidable even with bounded channels
  - Decidable classes



# Récursivité / dynamicité

- Dynamic nets of Finite state processes
- Automates communicants + creation dynamique d'AC
  - Dynamic MSC languages
  - Dynamic communicating automata
- Programmes concurrent récursifs
  - Concurrent pushdown systems
- Programmes récursifs communicants
- Dynamic nets of PDS