Introduction au Model Checking

Etienne Renault & Alexandre Duret-Lutz

Avril 2015

https://www.lrde.epita.fr/~renault/teaching/imc/

Organisation du cours (1/2)

- 27 Avril : Introduction Générale & Modélisation

- 4 Mai : Représentations compactes via BDD
  - QCM sur "Efficient Implementation of BDD package" K. Brace, L. Rudell, R. Bryant.

Organisation du cours (2/2)

- 8 Juin : Vérification propriétés LTL & CTL + Emptiness Checks (début)
  - Fiche lecture à choisir parmi :
    - ★ "On the Verification of Temporal Properties" par P. Godefroid et G. Holzmann.
    - ★ "On-the-fly Verification of Linear Temporal Logic" par Jean-Michel Couvreur
    - ★ "Comparison of Algorithms for Checking Emptiness on Büchi Automata" par Andreas Gaiser and Stefan Schwoon
    - ★ "Three SCC-based Emptiness Checks for Generalized Buchi Automata" par E.Renault, A. Duret-Lutz, F. Kordon et D. Poitrenaud

Organisation du cours (3/3)

- 15 Juin : Emptiness Checks (suite) + survol SAT
  - QCM cours précédents + "SAT-solving in practice" K. Claessen, Niklas Een, M. Sheeran, and N. Sörensson.

Fiche de Lecture

Objectif

Vous faire réfléchir sur un papier de recherche (d'un domaine que vous connaissez peu) et être capable de prendre du recul par rapport à ce qui est présenté (sur une page recto/verso maximum !).

Points à aborder (liste non-exhaustive)

- Introduction : problématique, objectif, contexte, domaine d'application, ...
- Analyse de la contribution : nature, plan, argumentation ...
- Références clefs, glossaire, ...
- Remarques personnelles ...

Généralités

Qu'est ce qu'un système ?

Différents types de programmes

- Les programmes *standards*
  1. terminent
  2. produisent des résultats
  3. peuvent être exprimés en terme de pré-conditions et de post-conditions
  4. données complexes mais structure de contrôle simple

  Exemples : Tris, compilateurs, traitement d'images, ...

Différents types de programmes

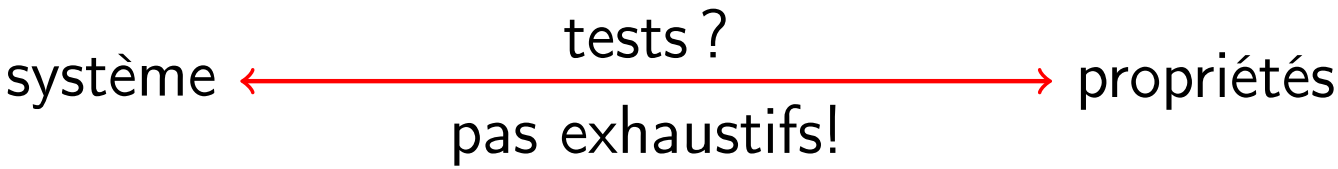
- Les programmes *standards*
  1. terminent
  2. produisent des résultats
  3. peuvent être exprimés en terme de pré-conditions et de post-conditions
  4. données complexes mais structure de contrôle simple

  Exemples : Tris, compilateurs, traitement d'images, ...

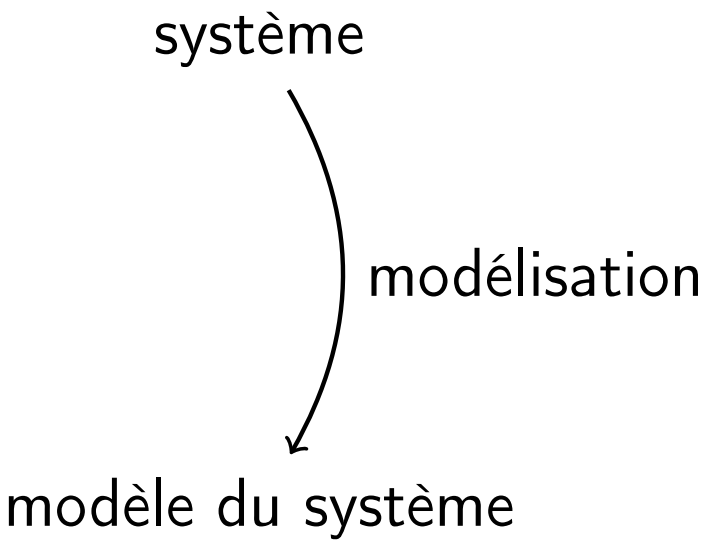
- Les systèmes *réactifs*
  1. ne terminent pas nécessairement
  2. interagissent avec leur environnement
  3. maintiennent une interaction
  4. données simples mais structure de contrôle complexe

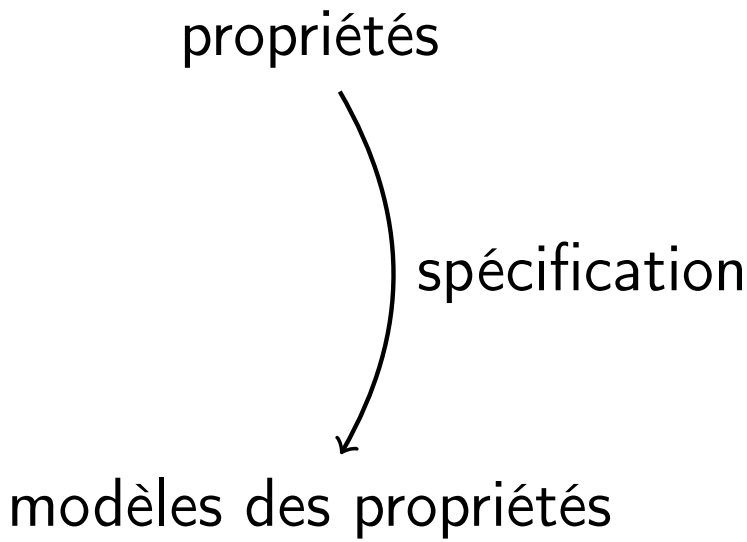
  Exemples : Protocoles, système d'exploitation, ...

Vérification formelle

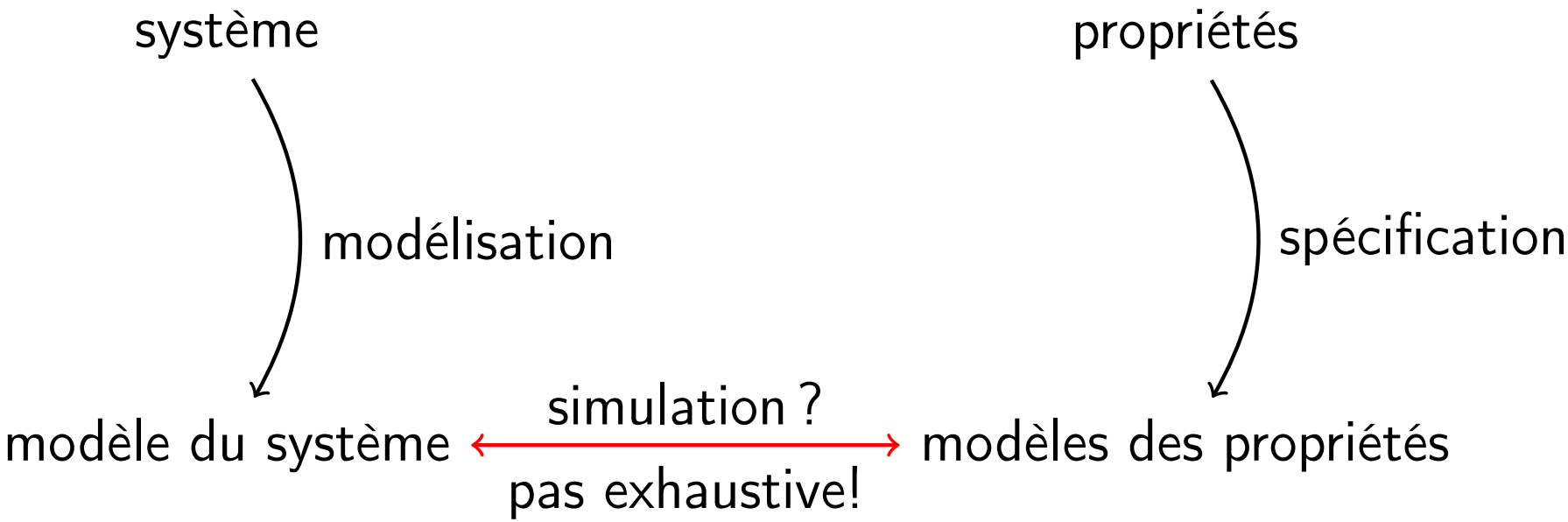


Vérification formelle

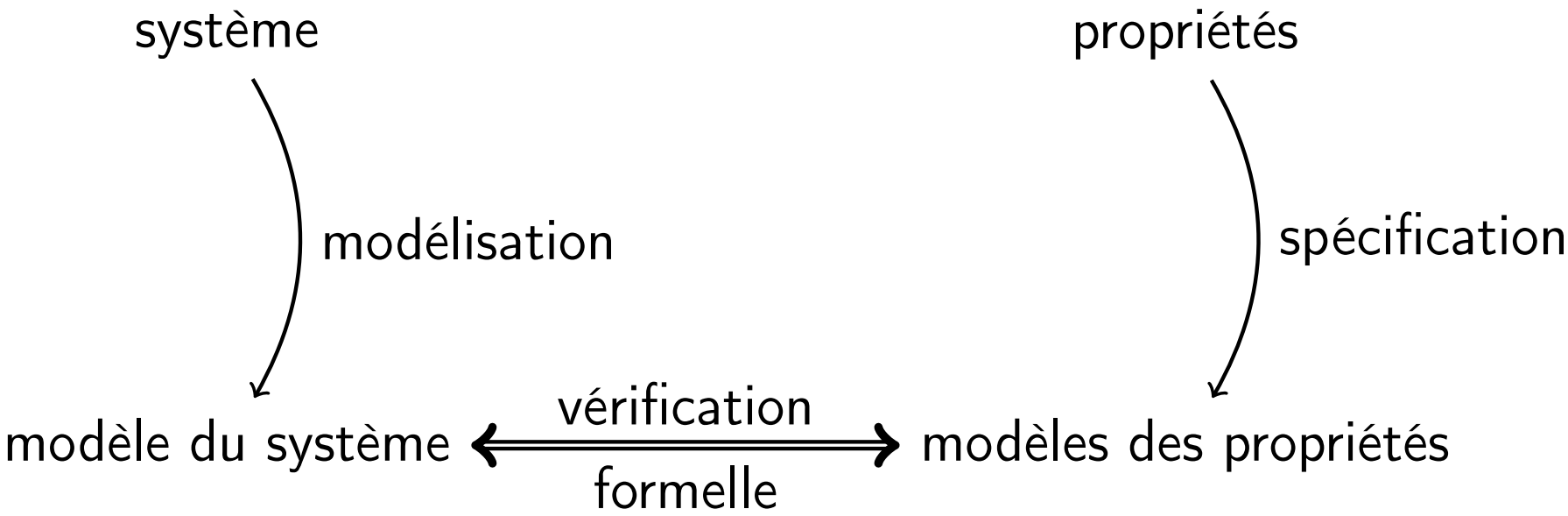




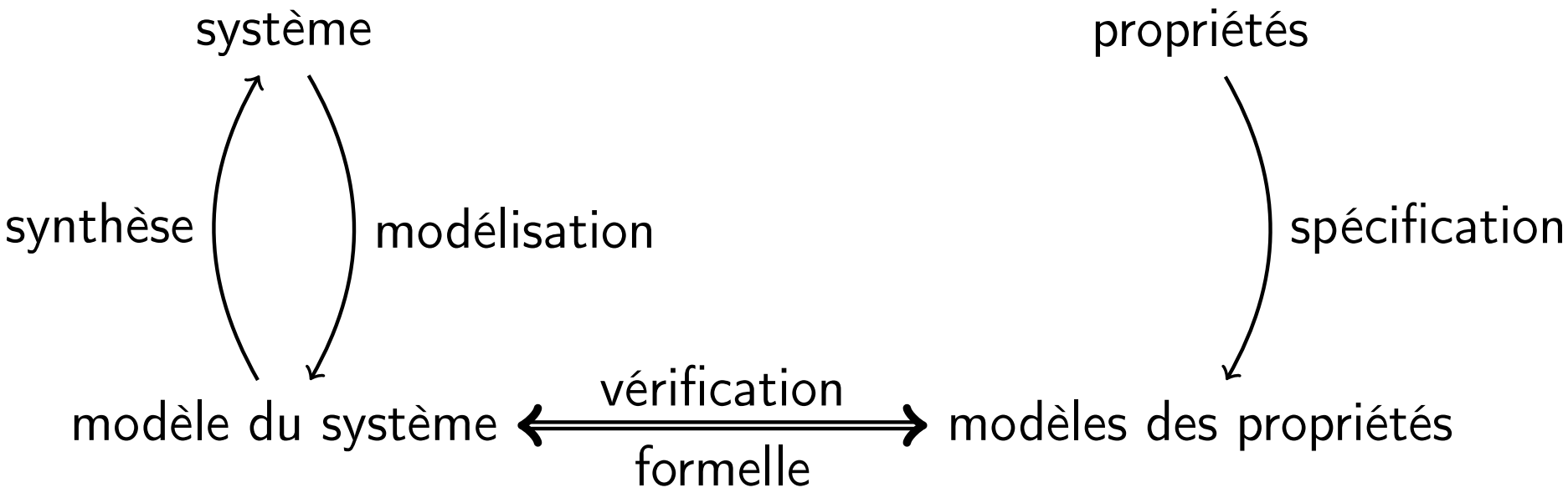
Vérification formelle



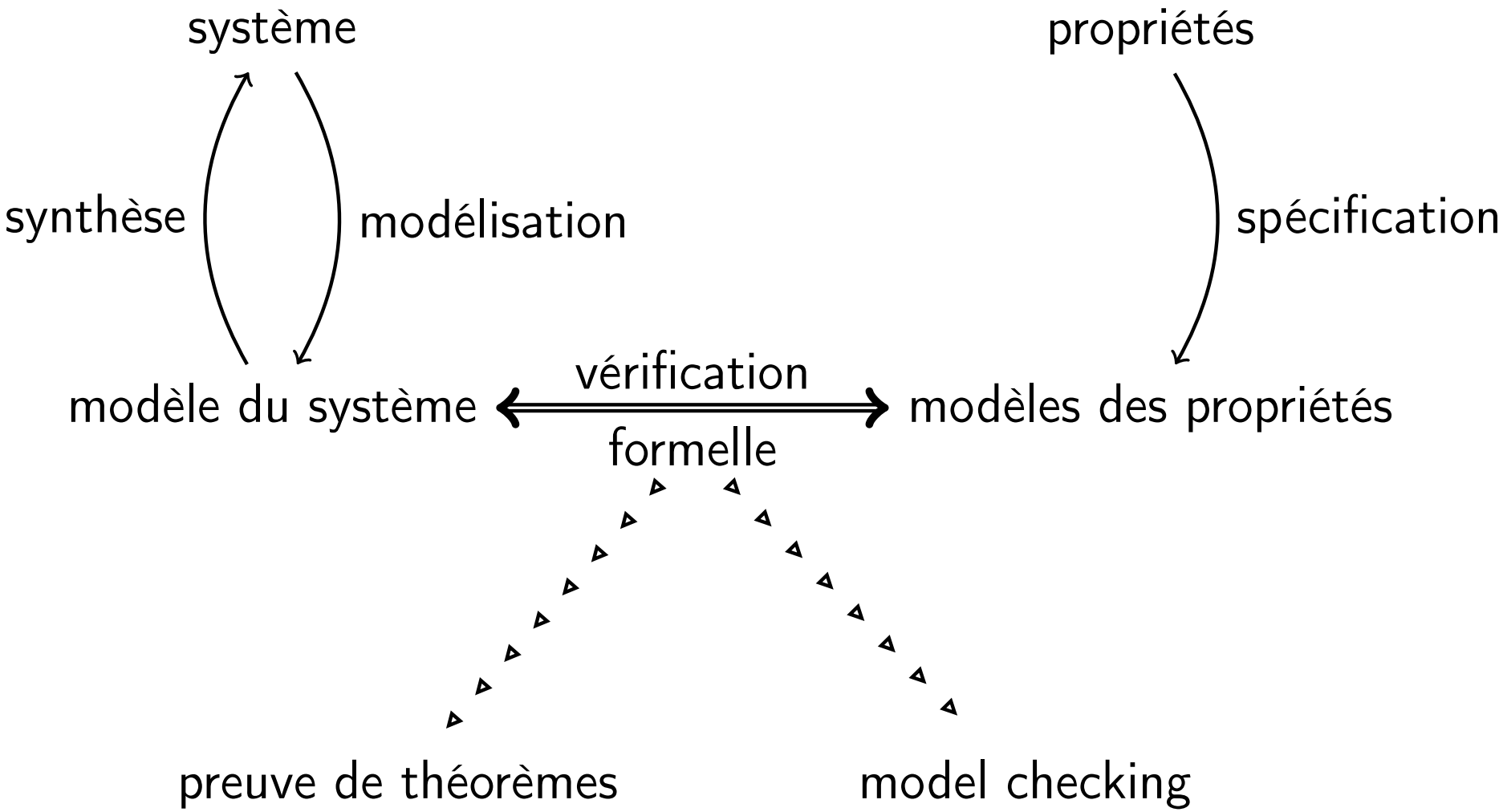
Vérification formelle



Vérification formelle



Vérification formelle



Preuve de théorèmes

1. Décrire le système sous une forme qui permette de raisonner.
2. Prouver les propriétés par déductions logiques.

Peut être manuel, ou plus ou moins automatisé.

Il existe des outils d'aide à l'automatisation des preuves (p.ex. Coq) mais rien n'est entièrement automatique. Difficile d'obtenir un contre-exemple quand le théorème est faux.

Les travaux : développement de systèmes de preuves, étude des puissance d'expression des logiques, etc.

Model checking

Approche automatique de la vérification formelle.

Vérification exhaustive de tous les comportements du modèle.

L'arnaque : le modèle doit être suffisamment abstrait pour que son exploration soit réalisable.

Deux grandes approches :

- ensembliste (symbolique)
- automate (explicite)

Exemple

Exemple : algorithme d'exclusion mutuelle

Processus P (boucle infinie)

1. $req_P \leftarrow 1$
2. $wait(req_Q = 0)$
3. section critique
4. $req_P \leftarrow 0$

Processus Q (boucle infinie)

1. $req_Q \leftarrow 1$
2. $wait(req_P = 0)$
3. section critique
4. $req_Q \leftarrow 0$

Variables globales : req_P et req_Q.
État initial : $P = 1$, $Q = 1$, $req_P = 0$, $req_Q = 0$.

Exemple : algorithme d'exclusion mutuelle

Processus P (boucle infinie)

1. $req_P \leftarrow 1$
2. $wait(req_Q = 0)$
3. section critique
4. $req_P \leftarrow 0$

Processus Q (boucle infinie)

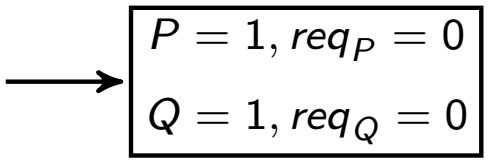
1. $req_Q \leftarrow 1$
2. $wait(req_P = 0)$
3. section critique
4. $req_Q \leftarrow 0$

Variables globales : req_P et req_Q.
État initial : $P = 1$, $Q = 1$, $req_P = 0$, $req_Q = 0$.

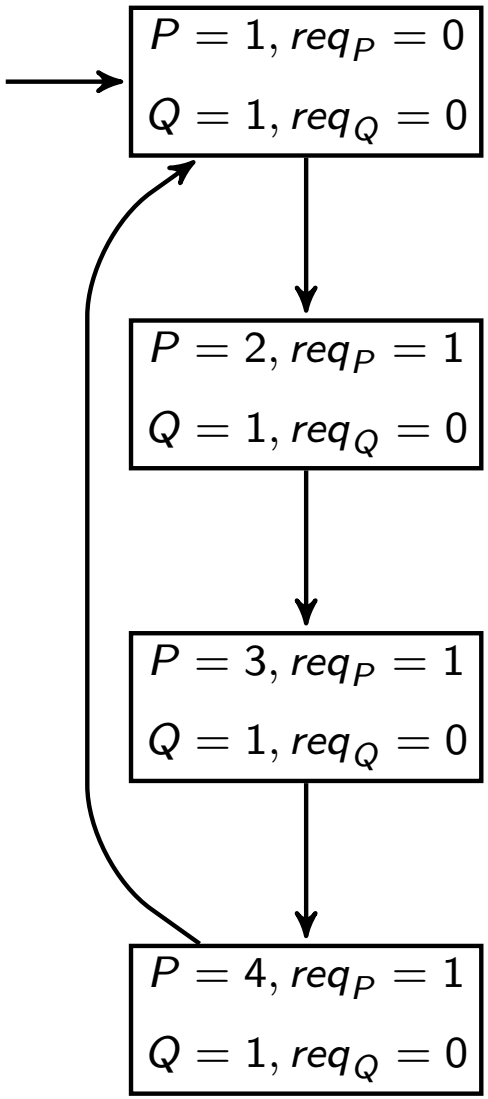
Propriétés à vérifier :

1. À tout moment il y a au plus un processus en section critique.
2. Tout processus demandant l'entrée en section critique finit par y entrer.
3. L'ordre des entrées dans la section critique respecte l'ordre des demandes.

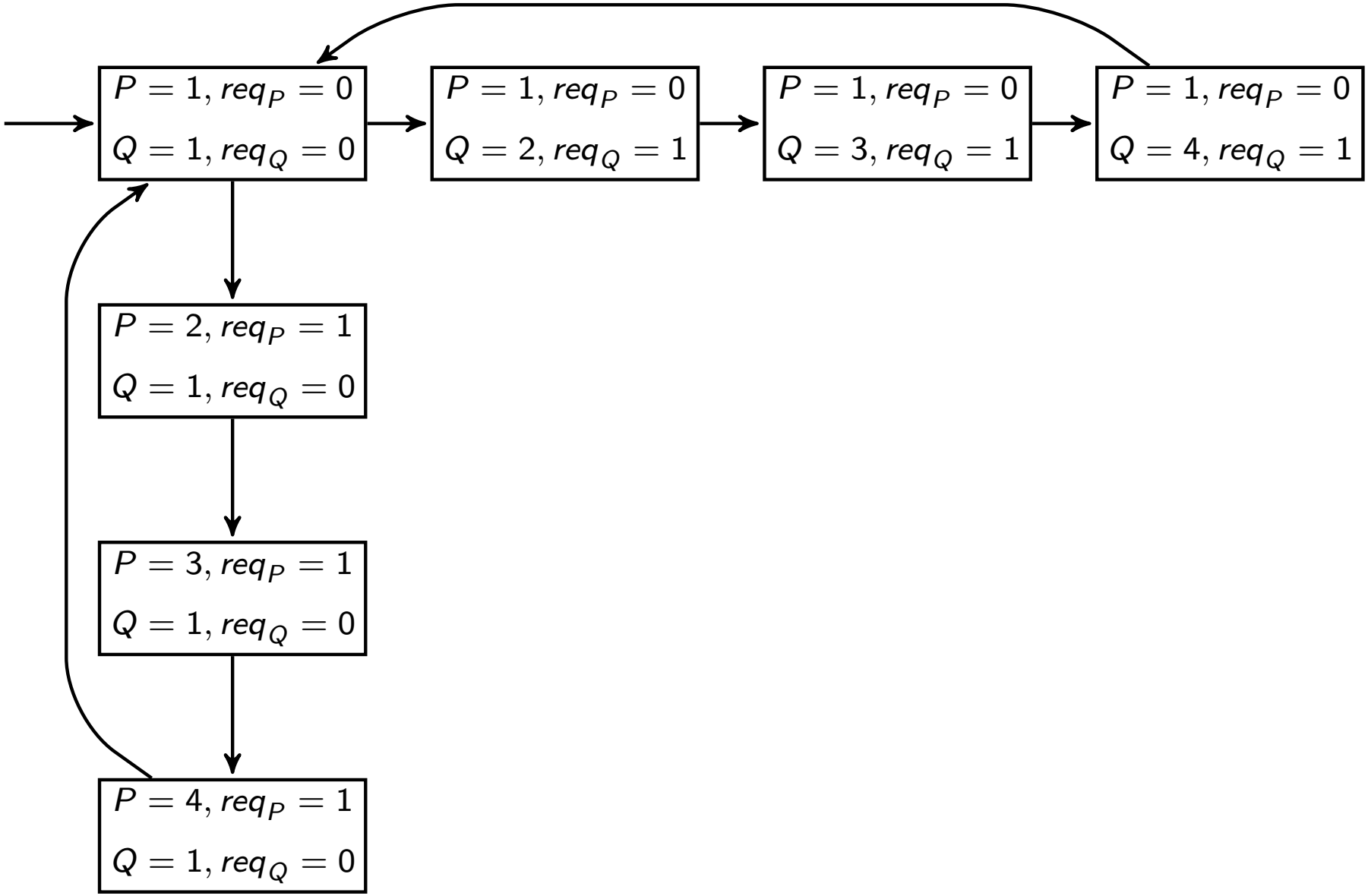
Exemple : espace d'état ou graphe d'accessibilité



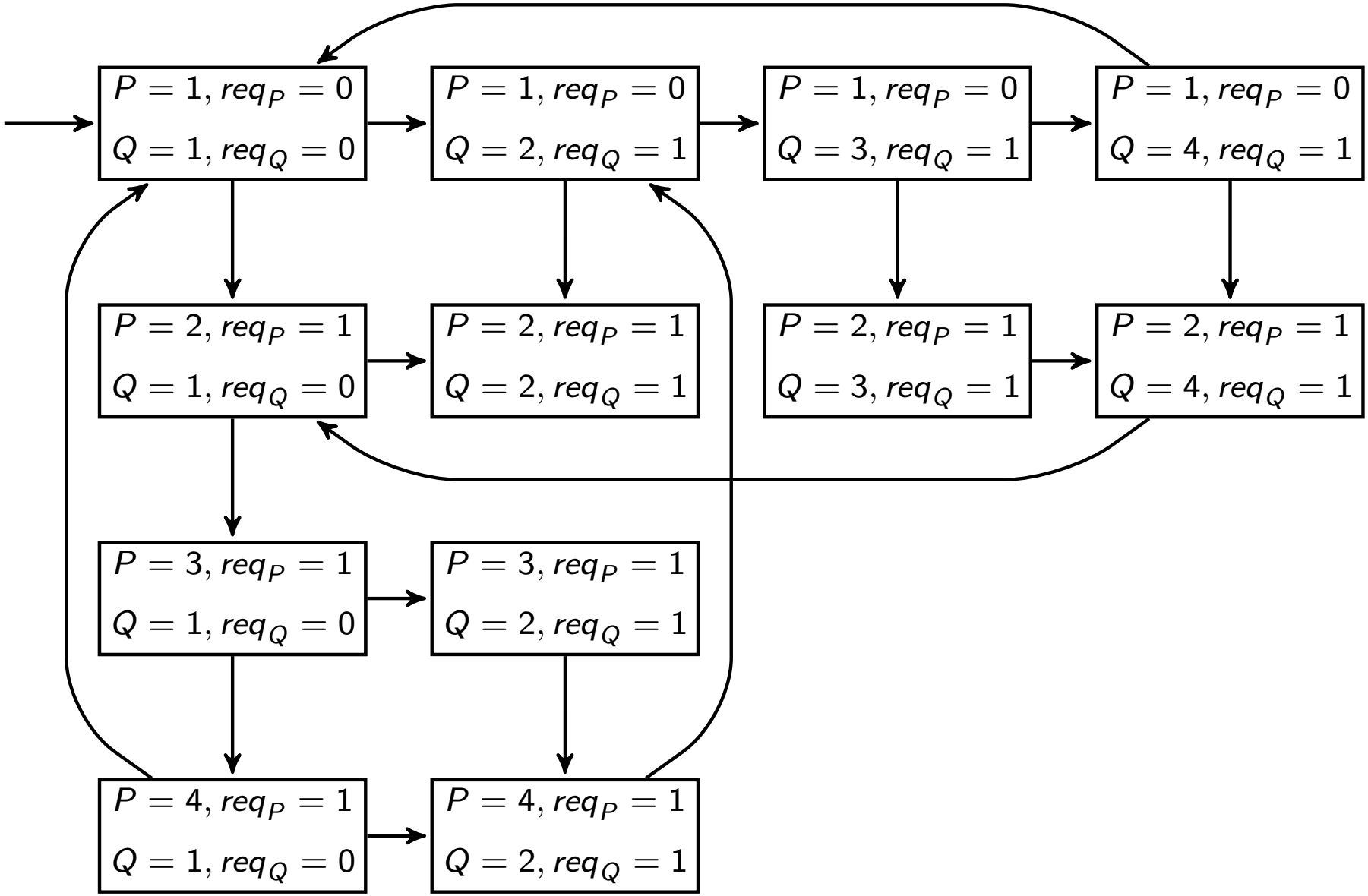
Exemple : espace d'état ou graphe d'accessibilité



Exemple : espace d'état ou graphe d'accessibilité



Exemple : espace d'état ou graphe d'accessibilité



Propriété 1

À tout moment il y a au plus un processus en section critique.

Traduction : dans aucun état on n'a $P = 3$ et $Q = 3$.

C'est vrai.

Pour vérifier cette propriété il suffit de parcourir tout l'espace d'état une fois. On n'a besoin de connaître que l'ensemble des états, pas leurs liens.

Propriété 2

Tout processus demandant l'entrée en section critique finit par y entrer.

Traduction : chaque chemin débutant dans un état accessible tel que $P = 2$ passe par un état où $P = 3$; idem pour $Q = 2$ et $Q = 3$.

C'est faux.

L'état $\boxed{\begin{array}{l} P = 2, req_P = 1 \\ Q = 2, req_Q = 1 \end{array}}$ n'a aucun successeur (c'est un deadlock).

Pour vérifier cette propriété on a besoin de connaître le graphe d'accessibilité (les états seuls ne suffisent pas).

Propriété 3

L'ordre des entrées dans la section critique respecte l'ordre des demandes.

Traduction : chaque chemin débutant dans un état accessible tel que $P = 2 \wedge Q = 1$ ne visite pas d'état vérifiant $Q = 3$ avant passer par un état vérifiant $P = 3$ (+ propriété symétrique pour Q).

C'est faux.

À partir de $\boxed{\begin{array}{l} P = 2, req_P = 1 \\ Q = 1, req_Q = 0 \end{array}}$ il existe un chemin dans lequel $P = 3$ n'est jamais satisfait.

Même type de vérification que la propriété 2.

Propriétés & Machine à états

Types de propriétés

(Lamport'77)

Sûreté Quelque chose de mauvais ne se produit pas.

Vivacité Quelque chose de bon se produira.

Équité Types particuliers de propriétés de vivacité.

Toute propriété est une conjonction d'une propriété de sûreté et d'une propriété de vivacité.

Types de propriétés

(Lamport'77)

Sûreté Quelque chose de mauvais ne se produit pas.
À tout moment il n'y a qu'un processus en section critique.

Vivacité Quelque chose de bon se produira.
Tout processus demandant l'entrée en section critique finit par y entrer.

Équité Types particuliers de propriétés de vivacité.
L'ordre des entrées dans la section critique respecte l'ordre des demandes.

Toute propriété est une conjonction d'une propriété de sûreté et d'une propriété de vivacité.

Comportements

Actions Une étape est une action.
Un comportement est une séquence d'actions.

États Une étape est une paire $\langle s, d \rangle$ d'états.
Un comportement est une séquence $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \ldots$ d'états.

Actions/États Une étape est un triplet $\langle s, \alpha, d \rangle$ où s et d sont des états, α est une action.
Un comportement est une séquence $s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} s_3 \xrightarrow{\alpha_3} \ldots$ d'états et d'actions.

Les comportements peuvent être finis ou infinis.
L'état final d'un comportement fini peut être appelé « deadlock » selon qu'on l'aime ou pas.
Certains comportements infinis (qu'on n'aime pas non plus) sont dit « divergents » (livelock).

Machines à états : génératrices de comportements

Machine à états

Un ensemble d'états $\mathcal{Q}$, des états initiaux $\mathcal{Q}^0$, une relation de transition $\mathcal{R} \subseteq \mathcal{Q} \times \mathcal{Q}$.

Machine à actions/états

Un ensemble d'états $\mathcal{Q}$, des états initiaux $\mathcal{Q}^0$, des actions $\mathcal{T}$, une relation de transition $\mathcal{R} \subseteq \mathcal{Q} \times \mathcal{T} \times \mathcal{Q}$.

Machines à états : génératrices de comportements

Machine à états

Un ensemble d'états $\mathcal{Q}$, des états initiaux $\mathcal{Q}^0$, une relation de transition $\mathcal{R} \subseteq \mathcal{Q} \times \mathcal{Q}$.

Machine à actions/états

Un ensemble d'états $\mathcal{Q}$, des états initiaux $\mathcal{Q}^0$, des actions $\mathcal{T}$, une relation de transition $\mathcal{R} \subseteq \mathcal{Q} \times \mathcal{T} \times \mathcal{Q}$.

$\mathcal{Q}$, $\mathcal{T}$ et $\mathcal{R}$ peuvent être infinis ou non...

On peut ajouter des conditions (de terminaison, d'acceptation, d'équité, ...) pour filtrer les comportements générés.

P.ex. : états terminaux pour n'avoir que des comportements finis.

Tout est (convertible en) machine à (actions/)états

- Automates
  - Automates classiques
  - Automates à pile
  - Machines de Mealy
  - Machines de Moore
  - Machines de Turing (à une ou plusieurs bandes)
  - Automates de Büchi, de Streett, de Rabin, de Muller...
  - Automates cellulaires
- Ordinateurs de von Neumann
- Algorithmes
- Grammaires BNF
- Algèbres de processus
- ...

Machine à états pour un programme (1/2)

```
main() { int f = 1;
         for (int i = 1; i <= 7; ++i) f = i * f;
         std::cout << f << std::endl;
       }
```

```
main() { int f = 1;
         for (int i = 7; 1 < i; --i) f = i * f;
         std::cout << f << std::endl;
       }
```

```
int fact(int i)
      { return (i == 1)? 1 : i * fact(i-1); }
main() { std::cout << fact(7) << std::endl; }
```

Machine à états pour un programme (2/2)

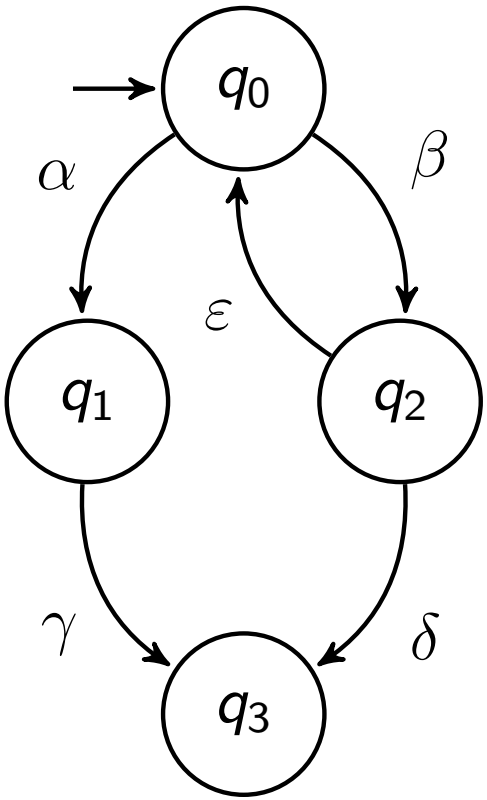
- Quel est le programme le plus différent des autres ?

- Comment décrire ces programmes par une machine à états ?
  Il faut choisir ce qui constitue un pas d'exécution...
  Est-que `f = i * f` représente 1, 2, 3 pas ou plus ?

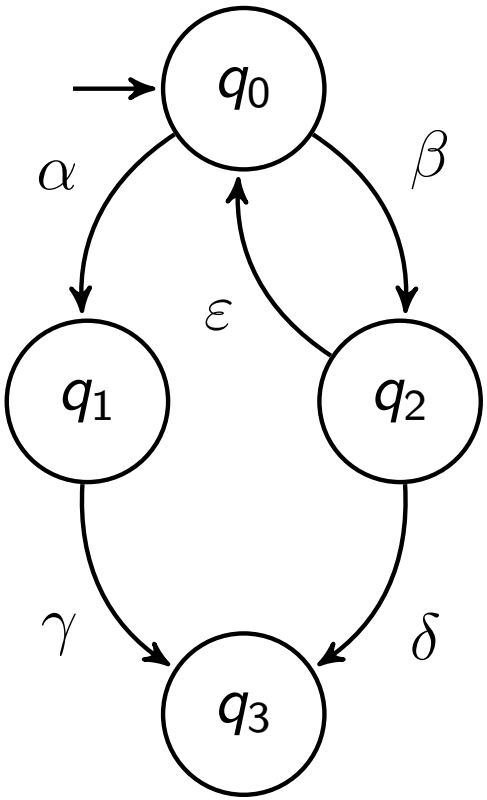
Machine à états pour un programme (2/2)

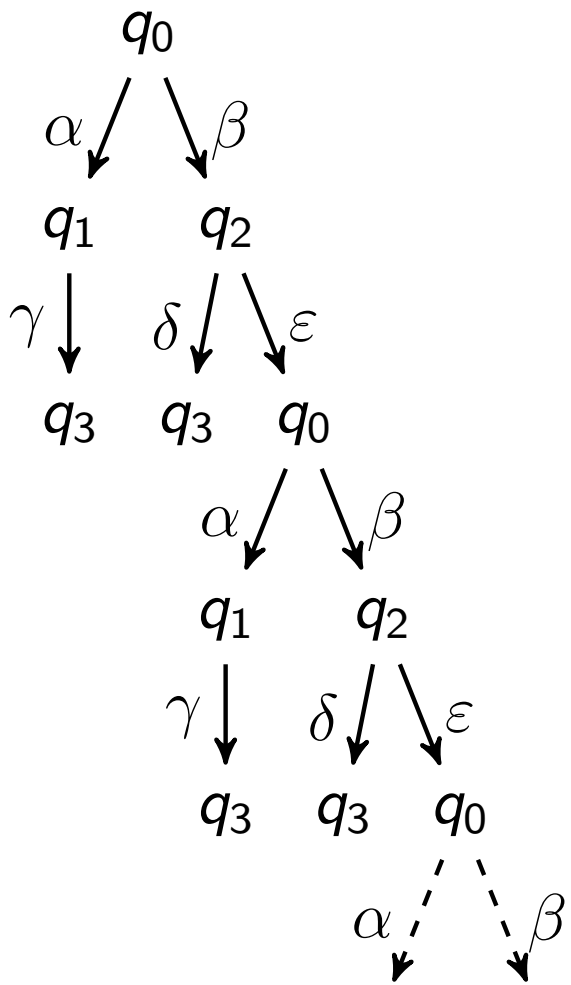
- Quel est le programme le plus différent des autres ?
  - Du point de vue des séquences de multiplication effectuées : 1 et 3 sont identiques.
  - 2 ne donne le même résultat que parce que $*$ est commutatif.
- Comment décrire ces programmes par une machine à états ?
  Il faut choisir ce qui constitue un pas d'exécution...
  Est-que `f = i * f` représente 1, 2, 3 pas ou plus ?
  Tout dépend du niveau d'abstraction auquel on se place.
  Des erreurs dans les programmes concurrents peuvent apparaître si l'on considère que `f = i * f` représente 1 pas alors qu'il se décompose en réalité en plusieurs.
  Pour traduire le C correctement en machine a états on a besoin d'une sémantique opérationnelle du langage qui prenne aussi en compte le modèle de mémoire de la machine.

Temps arborescent ou linéaire (1/2)

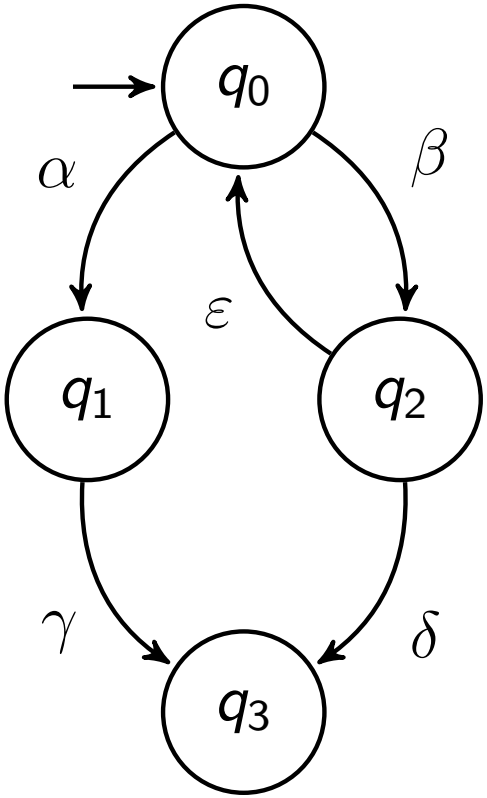


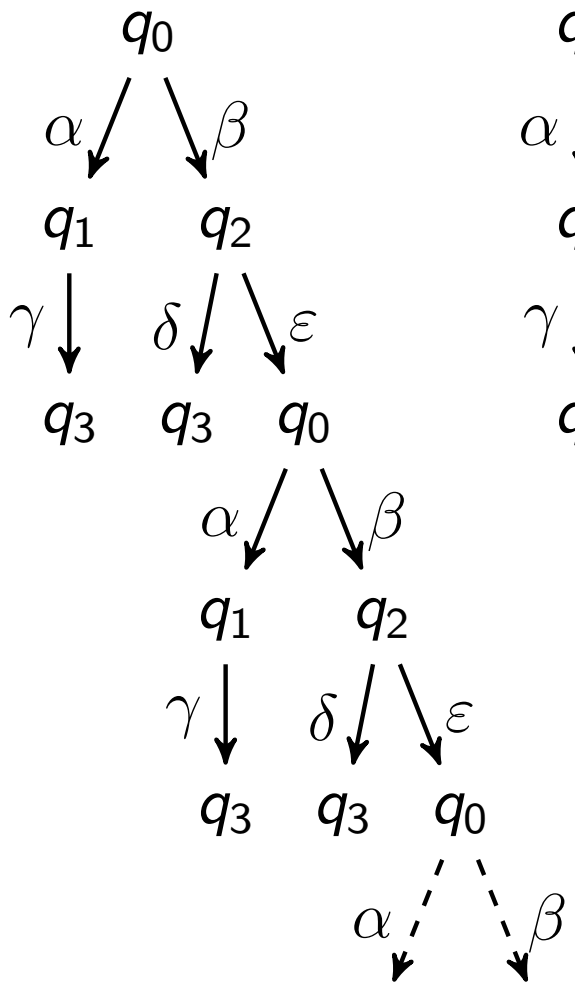
Temps arborescent ou linéaire (1/2)

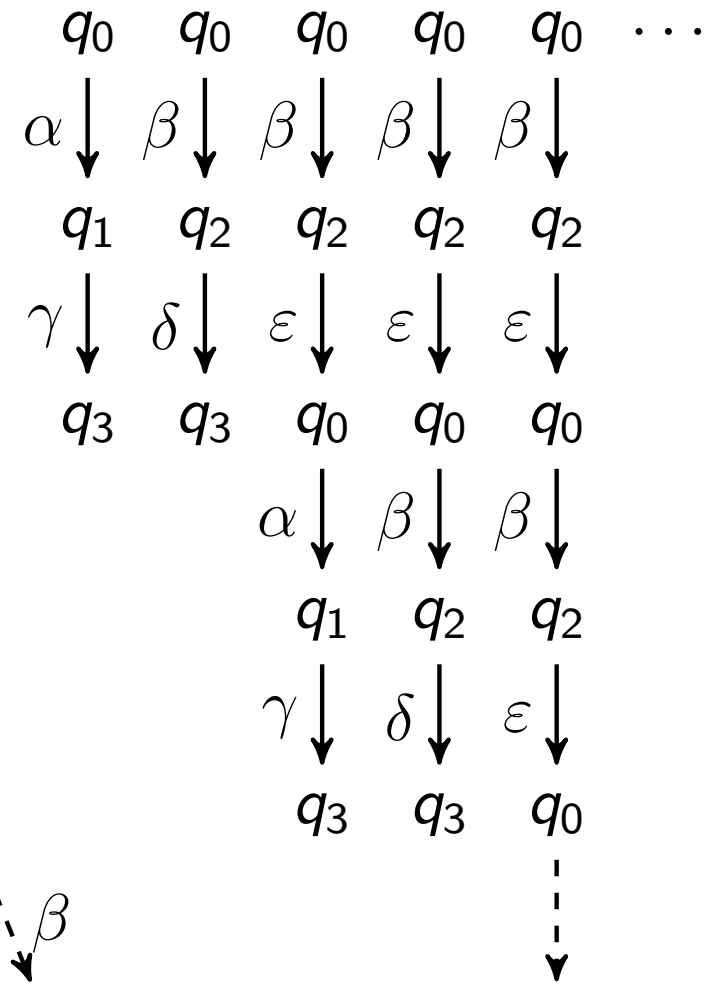




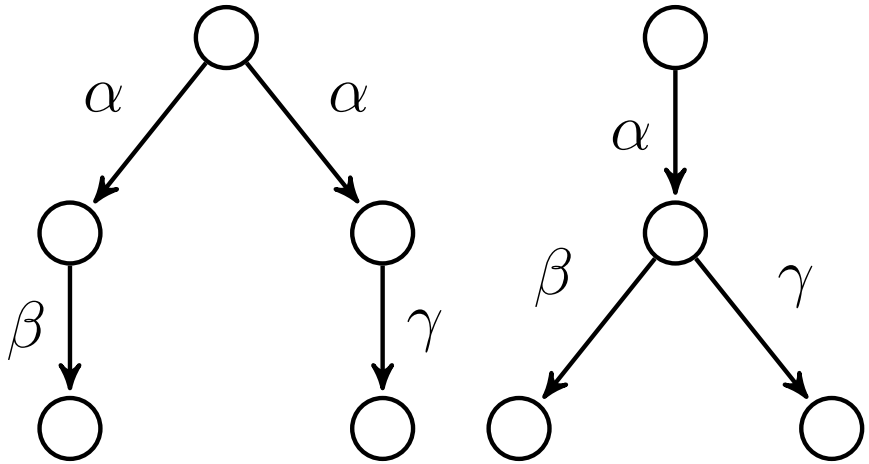
Temps arborescent ou linéaire (1/2)



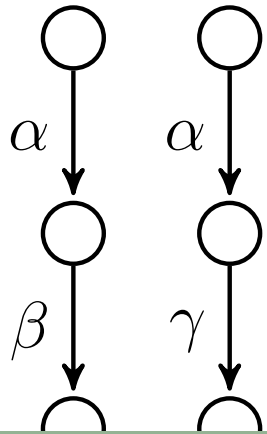




Temps arborescent ou linéaire (2/2)



En logique arborescente, les deux arbres ci-dessus sont générés par deux machines à états différentes. En logique linéaire, les comportements des deux machines ne permettent pas de distinguer les machines. (Ce problème ne se pose que si l'on décide de ne pas distinguer les états.)



Propositions atomiques & notations (1/2)

Variables propositionelles valuées en fonctions des propriétés (observées) du système.

Si $AP = \{p, q\}$, le système peut prendre au plus quatre états (observés) correspondant aux *minterms* sur AP : $p \wedge q$, $p \wedge \neg q$, $\neg p \wedge q$, $\neg p \wedge \neg q$.

Propositions atomiques & notations (1/2)

Variables propositionelles valuées en fonctions des propriétés (observées) du système.

Si $AP = \{p, q\}$, le système peut prendre au plus quatre états (observés) correspondant aux *minterms* sur AP : $p \wedge q$, $p \wedge \neg q$, $\neg p \wedge q$, $\neg p \wedge \neg q$.

$2^{AP} = \{\{p, q\}, \{p\}, \{q\}, \emptyset\}$.

Il existe une bijection entre 2^{AP} et l'ensemble des *minterms* sur AP :

$$\begin{array}{rcll} \{p, q\} & \leftrightarrow & p \wedge q & (\text{ou encore } pq) \\ \{p\} & \leftrightarrow & p \wedge \neg q & (p\bar{q}) \\ \{q\} & \leftrightarrow & \neg p \wedge q & (\bar{p}q) \\ \emptyset & \leftrightarrow & \neg p \wedge \neg q & (\bar{p}\bar{q}) \end{array}$$

Il existe une relation entre $2^{2^{AP}}$ et l'ensemble des formules propositionnelles sur AP. (Est-ce une bijection ?)

Proposition atomiques & notations (1/2)

Si $AP = \{a, b, c\}$, quels sous-ensembles de $2^{2^{AP}}$ représentent les formules suivantes ?

- $\neg c$
- $a \vee \neg b$
- $a \wedge (b \leftrightarrow c)$

Proposition atomiques & notations (1/2)

Si $AP = \{a, b, c\}$, quels sous-ensembles de $2^{2^{AP}}$ représentent les formules suivantes ?

- $\neg c$
  $\{\{a, b\}, \{a\}, \{b\}, \emptyset\}$
- $a \vee \neg b$
  $\{\{a, b, c\}, \{a, b\}, \{a, c\}, \{a\}, \{c\}, \emptyset\}$
- $a \wedge (b \leftrightarrow c)$
  $\{\{a, b, c\}, \{a\}\}$

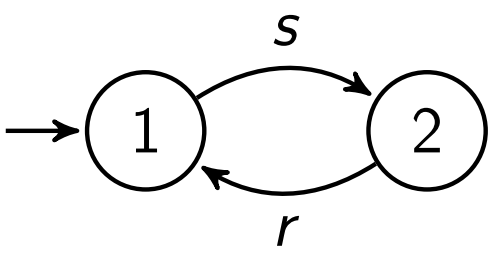
Structures de Kripke

Des machines à état étiquetées par des valuations de toutes les variables propositionelles.

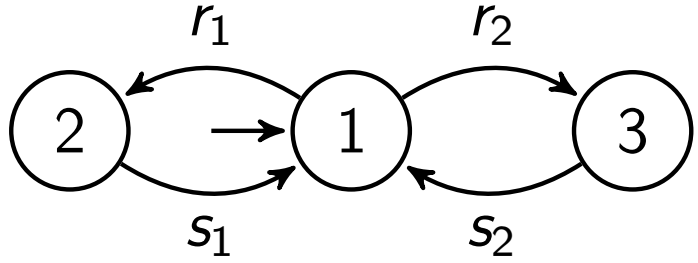
Une structure de Kripke est un quintuplet $K = \langle AP, \mathcal{Q}, q^0, \delta, I \rangle$ où

- AP est une ensemble fini de propositions atomiques,
- $\mathcal{Q}$ est un ensemble fini d'états (les nœuds du graphe),
- $q^0 \in \mathcal{Q}$ est l'état initial,
- $\delta : \mathcal{Q} \mapsto 2^{\mathcal{Q}}$ est une fonction indiquant les états successeurs d'un état,
- $I : \mathcal{Q} \mapsto 2^{AP}$ est une fonction indiquant l'ensemble des propositions atomiques satisfaites dans un état.

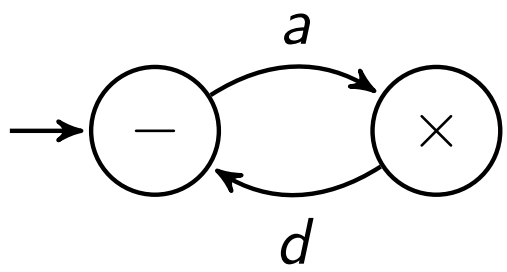
Ex. : clients/serveur par automates synchronisés



Client C

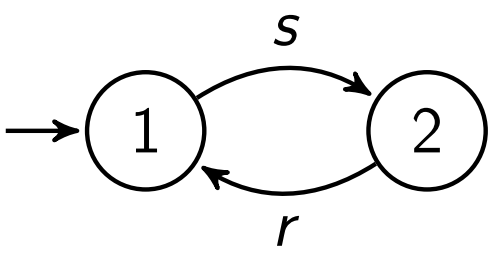


Serveur S

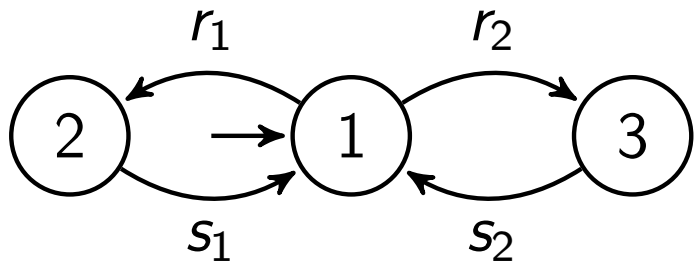


Canal B

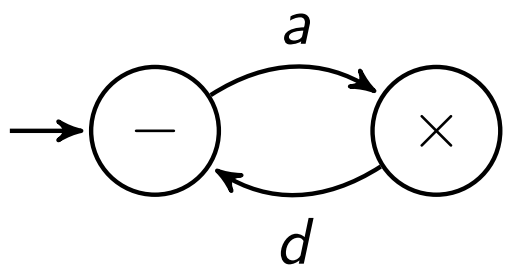
Ex. : clients/serveur par automates synchronisés



Client C



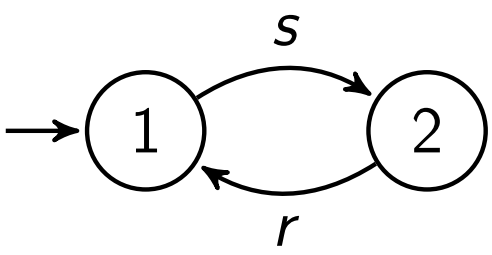
Serveur S



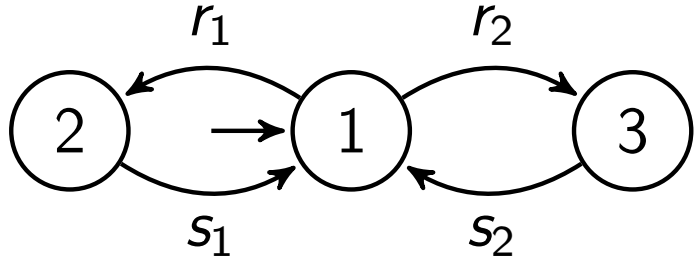
Canal B

1 serveur, 2 clients, 4 canaux

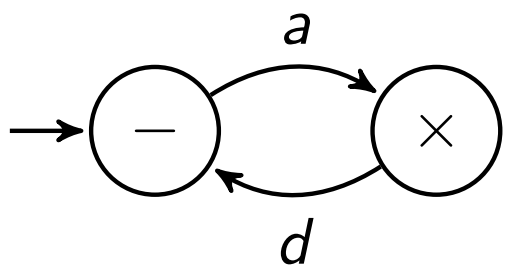
Ex. : clients/serveur par automates synchronisés



Client C



Serveur S



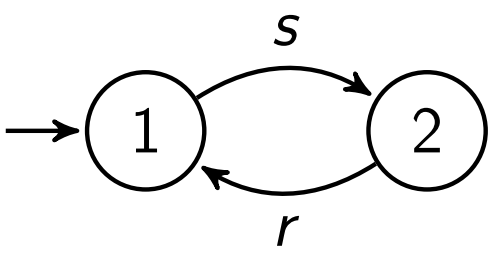
Canal B

1 serveur, 2 clients, 4 canaux

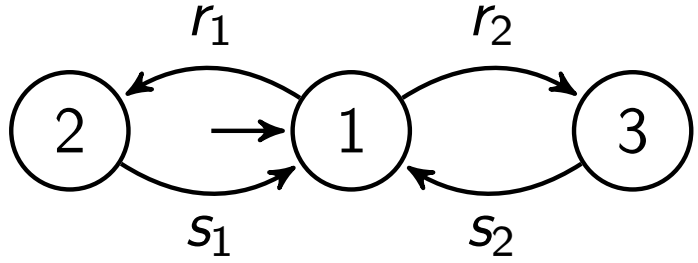
Règles de synchronisation pour le système $\langle C, C, S, B, B, B, B \rangle$:

$$
\begin{array}{ll}
(1) & \langle\ s\ ,\ .\ ,\ .\ ,\ .\ ,\ .\ ,\ .\ ,\ a\ ,\ .\ \rangle \\
(2) & \langle\ .\ ,\ s\ ,\ .\ ,\ .\ ,\ .\ ,\ .\ ,\ .\ ,\ a\ \rangle \\
(3) & \langle\ r\ ,\ .\ ,\ .\ ,\ .\ ,\ d\ ,\ .\ ,\ .\ ,\ .\ \rangle \\
(4) & \langle\ .\ ,\ r\ ,\ .\ ,\ .\ ,\ .\ ,\ d\ ,\ .\ ,\ .\ \rangle \\
(5) & \langle\ .\ ,\ .\ ,\ .\ ,\ r_1\ ,\ .\ ,\ .\ ,\ .\ ,\ d\ ,\ .\ \rangle \\
(6) & \langle\ .\ ,\ .\ ,\ .\ ,\ s_1\ ,\ a\ ,\ .\ ,\ .\ ,\ .\ ,\ .\ \rangle \\
(7) & \langle\ .\ ,\ .\ ,\ .\ ,\ r_2\ ,\ .\ ,\ .\ ,\ .\ ,\ .\ ,\ d\ \rangle \\
(8) & \langle\ .\ ,\ .\ ,\ .\ ,\ s_2\ ,\ .\ ,\ a\ ,\ .\ ,\ .\ ,\ .\ \rangle
\end{array}
$$

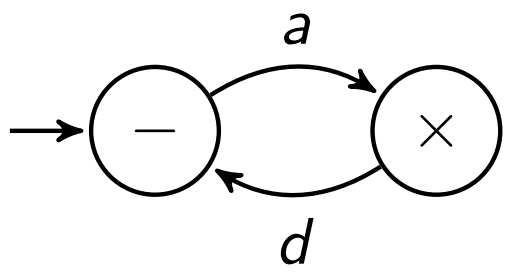
Ex. : clients/serveur par automates synchronisés



Client C



Serveur S



Canal B

1 serveur, 2 clients, 4 canaux

Règles de synchronisation pour le système $\langle C, C, S, B, B, B, B \rangle$:

(1) $\langle\ s\ ,\ .\ ,\ .\ ,\ .\ ,\ .\ ,\ .\ ,\ a\ ,\ .\ \rangle$

(2) $\langle\ .\ ,\ s\ ,\ .\ ,\ .\ ,\ .\ ,\ .\ ,\ .\ ,\ a\ \rangle$

(3) $\langle\ r\ ,\ .\ ,\ .\ ,\ .\ ,\ d\ ,\ .\ ,\ .\ ,\ .\ \rangle$

(4) $\langle\ .\ ,\ r\ ,\ .\ ,\ .\ ,\ .\ ,\ d\ ,\ .\ ,\ .\ \rangle$

(5) $\langle\ .\ ,\ .\ ,\ .\ ,\ r_1\ ,\ .\ ,\ .\ ,\ .\ ,\ d\ ,\ .\ \rangle$

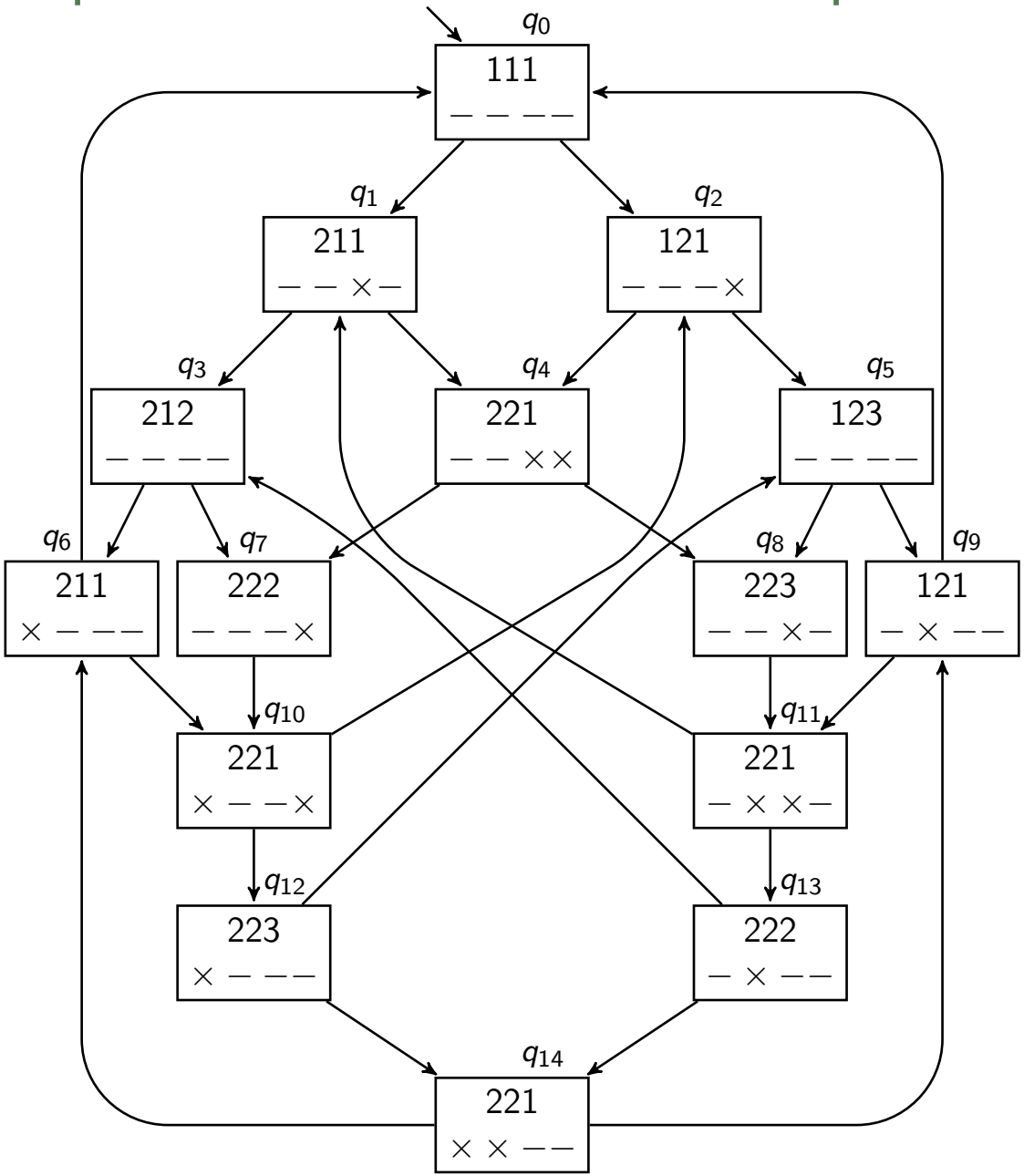
(6) $\langle\ .\ ,\ .\ ,\ .\ ,\ s_1\ ,\ a\ ,\ .\ ,\ .\ ,\ .\ ,\ .\ \rangle$

(7) $\langle\ .\ ,\ .\ ,\ .\ ,\ r_2\ ,\ .\ ,\ .\ ,\ .\ ,\ .\ ,\ d\ \rangle$

(8) $\langle\ .\ ,\ .\ ,\ .\ ,\ s_2\ ,\ .\ ,\ .\ ,\ a\ ,\ .\ ,\ .\ \rangle$

Si un client envoie une requête, recevra-t-il forcément une réponse ?

Espace d'états de l'exemple



Propositions atomiques pour l'exemple

On souhaite exprimer des propriétés concernant les envois et réceptions de messages. $AP = \{r_1, r_2, d_1, d_2\}$ avec :

- r_1 : une réponse est en chemin entre le serveur et le premier client
- r_2 : une réponse est en chemin entre le serveur et le second client
- d_1 : une requête (d pour demande) est en chemin entre le premier client et le serveur
- d_2 : une requête est en chemin entre le second client et le serveur

Comment traduire « Si un client envoie une requête, il recevra forcément une réponse » avec ces propositions atomiques ?

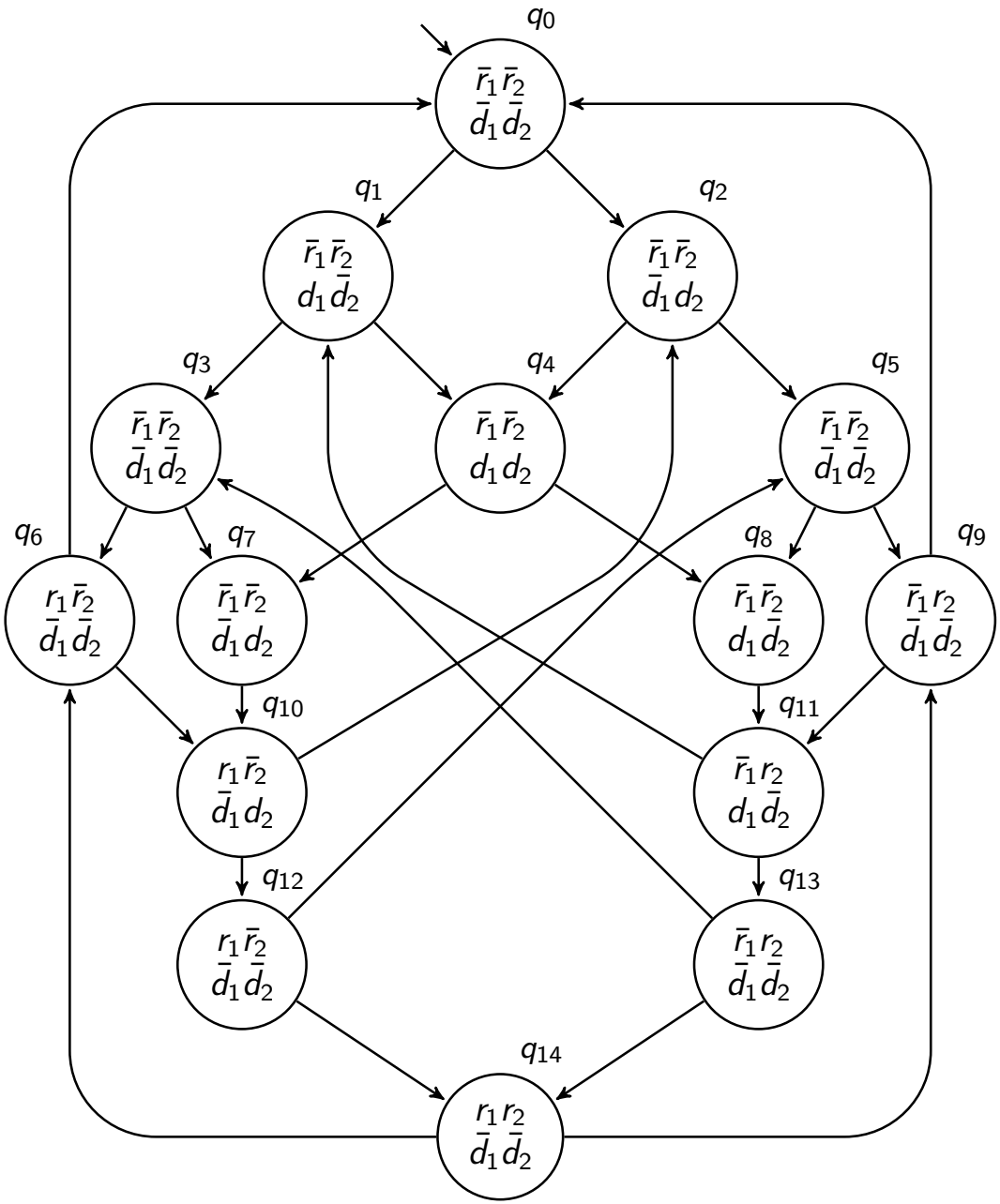
Propositions atomiques pour l'exemple

On souhaite exprimer des propriétés concernant les envois et réceptions de messages. $AP = \{r_1, r_2, d_1, d_2\}$ avec :

- r_1 : une réponse est en chemin entre le serveur et le premier client
- r_2 : une réponse est en chemin entre le serveur et le second client
- d_1 : une requête (d pour demande) est en chemin entre le premier client et le serveur
- d_2 : une requête est en chemin entre le second client et le serveur

Comment traduire « Si un client envoie une requête, il recevra forcément une réponse » avec ces propositions atomiques ?
Pour tout $i \in \{1, 2\}$, si un état vérifie d_i alors dans tous ses futurs possibles il possède un successeur qui vérifie r_i.

Structure de Kripke pour l'exemple

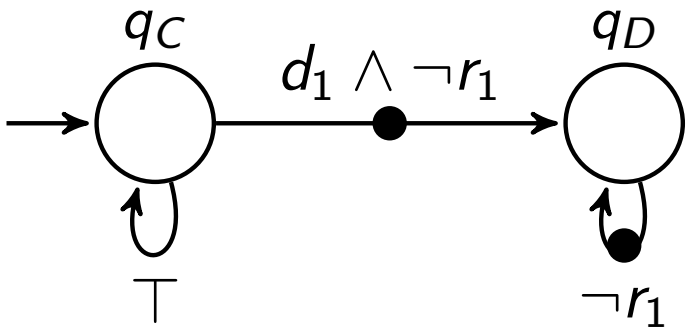


Exploration de la structure de Kripke

Pour tout $i \in \{1, 2\}$, si un état vérifie d_i alors dans tous ses futurs possibles il possède un successeur qui vérifie r_i.

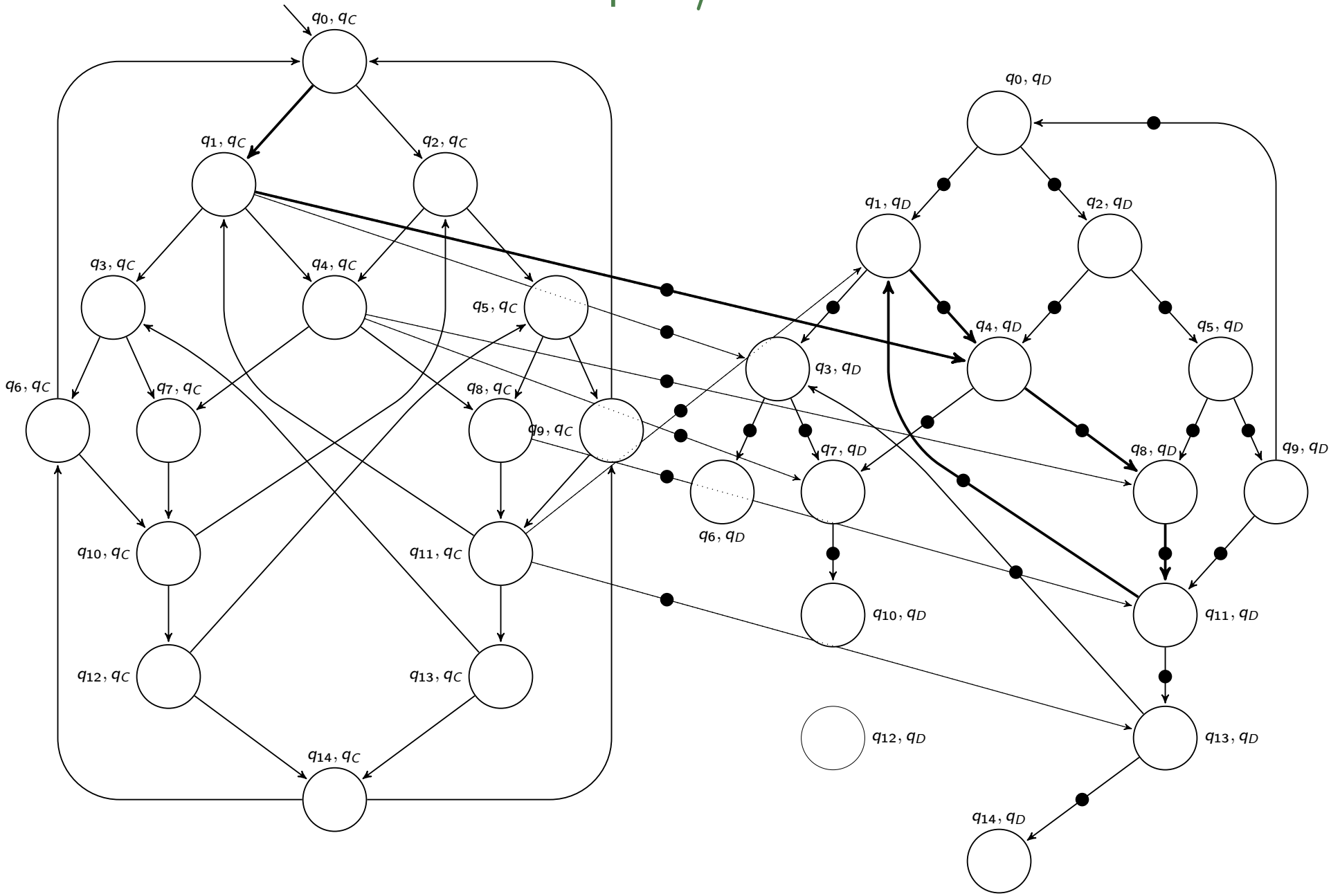
On cherche un contre-exemple, c'est-à-dire un chemin infini qui passe par d_i sans jamais passer par r_i. Par symétrie on peut se limiter à $i = 1$.

Anticipons sur les prochains cours... Le chemin qu'on voudrait reconnaître peut être reconnu par l'automate :



Où les chemins reconnus doivent passer infiniment souvent par des transitions ●.

Produit structure de Kripke/Automate



Vérification de la propriété

Sur le produit on peut alors chercher un cycle étiqueté par $\bullet$ et accessible depuis l'état initial. C'est l'emptiness check dans l'approche automate du model checking.

Autre approche complètement différente (symbolique) :

- On construit l'ensemble E_1 des états de la structure de Kripke qui vérifient $\neg r_1$.
- On construit l'ensemble E_2 des états de E_1 dont les successeurs restent dans E_1.
- On construit l'ensemble E_3 des états de E_2 dont les successeurs restent dans E_2.
- On répète jusqu'à ce que $E_n = E_{n-1}$ ou $E_n = \emptyset$ (point fixe).
- Si $E_n = E_{n-1} \neq \emptyset$ et qu'il contient un état vérifiant d_1 et accessible depuis l'état initial (ce qu'on peut tester aussi par point fixe), on a trouvé un contre-exemple.

Équité

L'équité

Peut désigner deux choses :

- Une propriété que l'on veut vérifier
  - L'ordre des entrées dans la section critique respecte l'ordre des demandes.
- Une propriété que l'on veut supposer pour filtrer les comportements (on parle d'hypothèse d'équité)
  - Le scheduler est équitable : un processus prêt à travailler travaillera après un temps d'attente fini... On doit ignorer les comportement du modèle où ça n'est pas le cas.

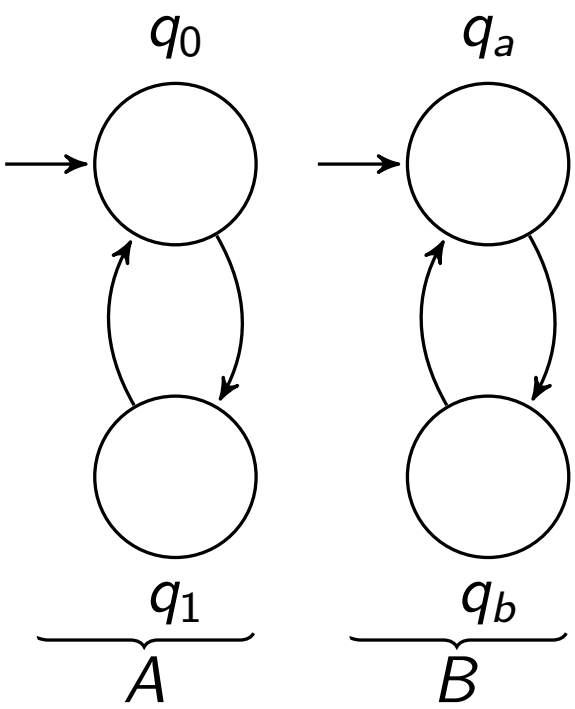
Différent types d'équité

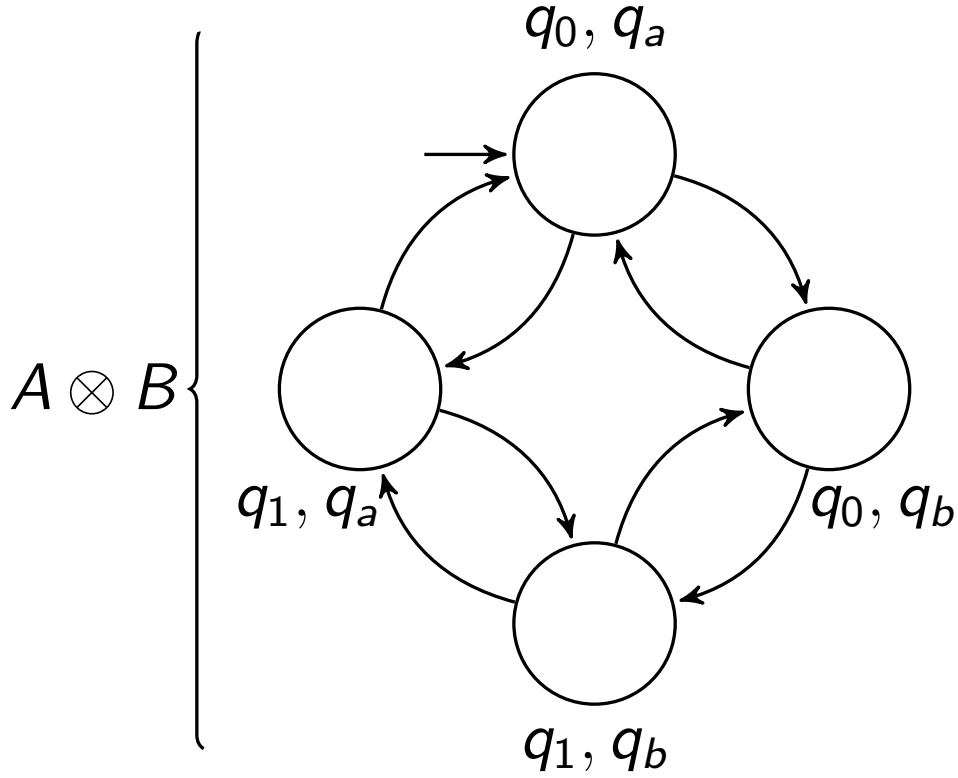
Impartialité (unconditional fairness) Tous les processus s'exécutent un nombre infini de fois.

Justice (weak fairness) Tout processus qui peut toujours s'exécuter à partir d'un moment s'exécutera infiniment souvent.

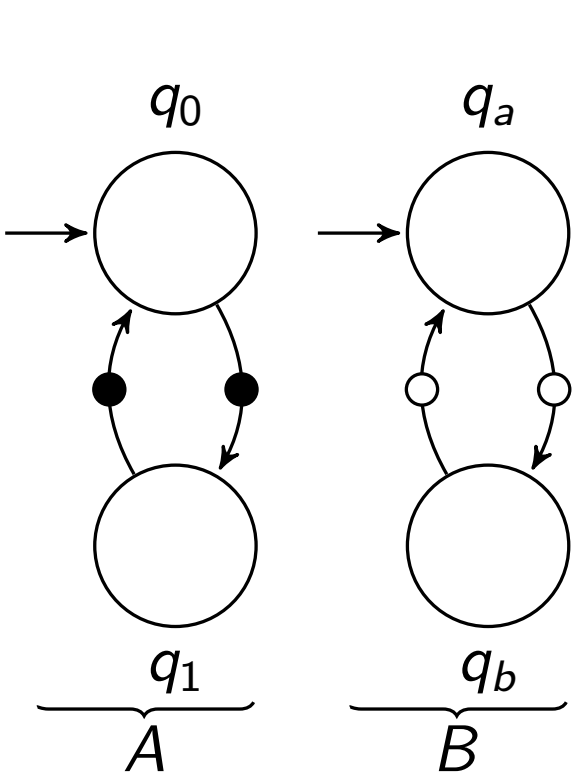
Compassion (strong fairness) Tout processus qui peut s'exécuter infiniment souvent s'exécutera infiniment souvent.

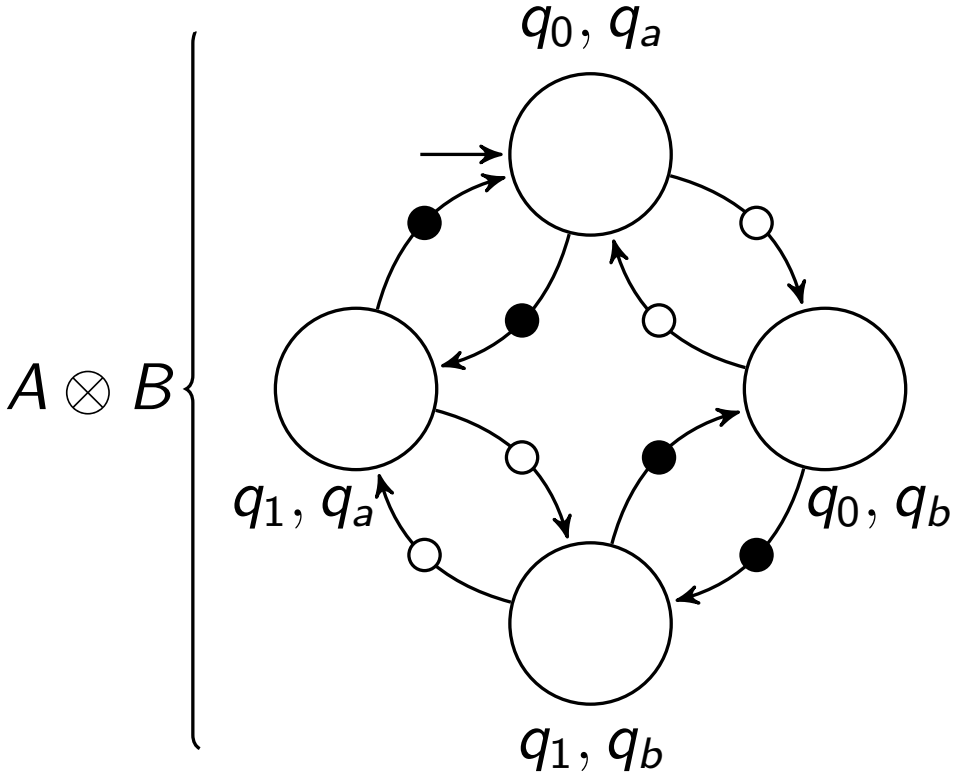
Équité : exemple





Équité : exemple





Conclusion

Le model-checking :

- ne travaille pas directement sur un système mais sur un model (abstrait) de celui-ci.
- Deux approches existent pour obtenir le modèle :
  - soit il est généré depuis le système
  - soit il est concu de manière à pouvoir générer le système
- nécessite la présence de propriétés pour exprimer des comportements que l'on souhaite que le système vérifie