Modélisation de la concurrence

F. Dagnat

Majeure Informatique – INF447 – C1

1er semestre 2015

L'organisation d'INF447

1. Introduction à la modélisation de la concurrence
   - C1, PC1, TP1 les bases de FSP
2. Des modèles plus complexes
   - PC2, TP2 des modèles indexés
3. Les propriétés d'intérêt
   - C2, PC3
4. Du modèle au code
   - C2, TP3-4
5. La modélisation des systèmes distribués
   - PC4-5

- Un BE évalué sur la production de modèles FSP

Plan

Avancement

Rappel UV1

- Un programme concurrent contient plusieurs fils d'activités (*threads*) qui s'exécutent simultanément
- Un *thread* est une suite d'instructions qui s'exécutent en séquence
- L'exécution d'un prog conc consiste en l'entrelacements des instructions de ses *threads*
- Le choix de l'ordre d'exécution est inconnu, il est non déterministe
- Pour comprendre, il faut imaginer tous les ordres
  - 10 *threads* de 10 instructions $\Rightarrow 10^{10}$ cas possibles !

Le principe : modéliser

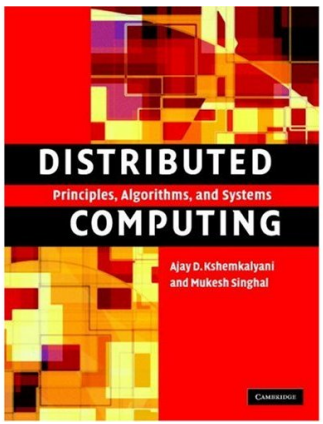
- Modèle : représentation simplifiée d'un système
- Pour concevoir et valider une conception
  - animer le modèle pour visualiser le comportement
  - vérifier mécaniquement des propriétés
- Un modèle, lequel ?
  - en UV1, UML (diag de séquence, d'état ou d'activité)
  - dans ce cours, des modèles
    - sous forme de LTS (*Labelled Transition System*) des automates
    - défini en FSP (*Finite State Processes*)
- Modèles formels pour vérifier mathématiquement des propriétés
  - dans ce cours, outil LTSA *model checking*

Objectifs du module

- O1 Découvrir et pratiquer la modélisation formelle
  - a construire des modèles FSP
  - b passer d'un modèle FSP à son LTS et vice et versa
  - c composer des LTS
  - d savoir décomposer un système à modéliser puis construire son modèle par composition
- O2 Mieux comprendre les difficultés de la concurrence et de la repartition
  - a citer et expliquer les propriétés importantes
  - b spécifier et vérifier les propriétés de sûreté et de vivacité avec LTSA
- O3 Faire le lien entre un modèle et un programme
  - a passer d'un modèle FSP à un programme Java
  - b connaître et expliquer la méthode

Bibliographie

Jeff Magee & Jeff Kramer,
Concurrency: State Models & Java Programs, 2nd ed,
Wiley, `http://www.doc.ic.ac.uk/~jnm/book`
Chapitre 1 à 8 principalement

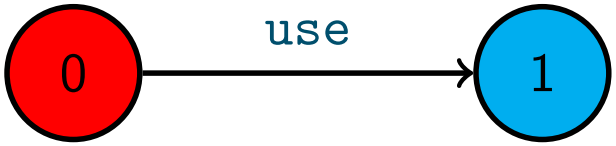


Ajay D. Kshemkalyani & Mukesh Singhal,
Distributed Computing Principles, Algorithms, and Systems,
Cambridge University Press
Chapitre 1 à 4 principalement

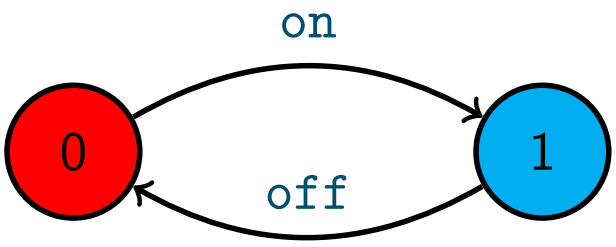
Avancement

Processus et leur comportement

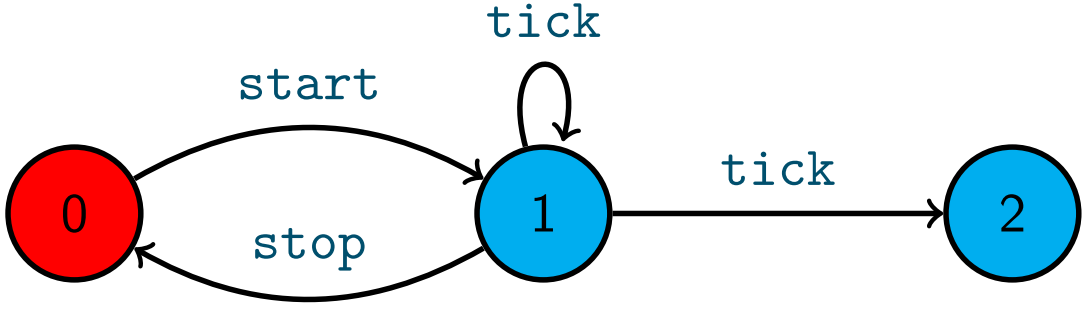
- Suite d'actions avec choix et boucles $\Rightarrow$ automate
- LTS (*Labelled Transition System*)
- Équipement à usage unique



- Une lampe



- Une horloge ($\mathcal{H}$)

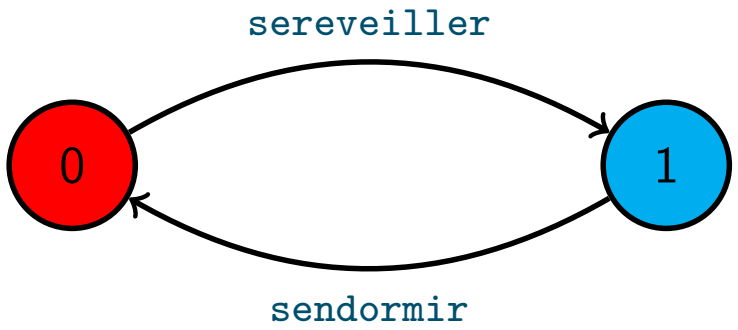


Modéliser la journée d'un élève

- Il faut trouver
  - les états (dont l'initial), les actions et les transitions
  - certains pensent mieux *état*, d'autres *action*
- Qu'est-ce qu'une action / un état ?
  - un *truc* atomique observable à l'extérieur du processus
  - $\Rightarrow$ de nombreux modèles possibles

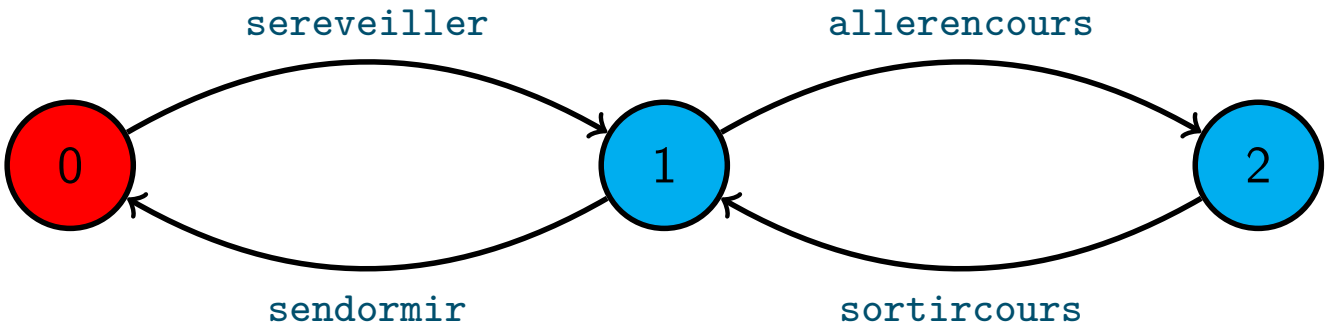
Modéliser la journée d'un élève

- Il faut trouver
  - les états (dont l'initial), les actions et les transitions
  - certains pensent mieux *état*, d'autres *action*
- Qu'est-ce qu'une action / un état ?
  - un *truc* atomique observable à l'extérieur du processus
  - $\Rightarrow$ de nombreux modèles possibles



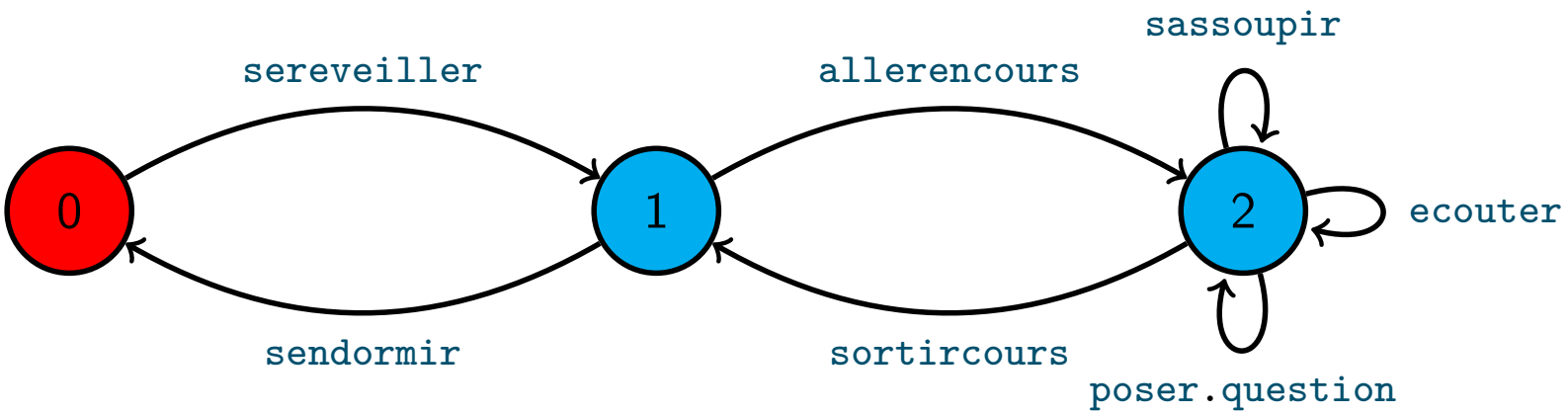
Modéliser la journée d'un élève

- Il faut trouver
  - les états (dont l'initial), les actions et les transitions
  - certains pensent mieux *état*, d'autres *action*
- Qu'est-ce qu'une action / un état ?
  - un *truc* atomique observable à l'extérieur du processus
  - $\Rightarrow$ de nombreux modèles possibles



Modéliser la journée d'un élève

- Il faut trouver
  - les états (dont l'initial), les actions et les transitions
  - certains pensent mieux *état*, d'autres *action*
- Qu'est-ce qu'une action / un état ?
  - un *truc* atomique observable à l'extérieur du processus
  - $\Rightarrow$ de nombreux modèles possibles



- Remarques : évènement vs action ; entrée vs sortie

Qu'est-ce qu'un LTS ?

- Un LTS est (S, A, T, s_0) où
  - S est l'ensemble de ses états (ici, ce sont des entiers)
  - A est l'alphabet, l'ensemble de ses actions, une action est une suite de mots[1] séparés par des points
  - T est sa relation de transition, ses éléments sont des triplets (s_1, a, s_2) notés $s_1 \xrightarrow{a} s_2$, $T \subseteq S \times A \times S$
  - s_0 est son état initial, $s_0 \in S$
- Pour $\mathcal{H}$
  - $S = \{0, 1, 2\}$
  - $A = \{\texttt{start}, \texttt{tick}, \texttt{stop}\}$
  - $T = \{0 \xrightarrow{\texttt{start}} 1, 1 \xrightarrow{\texttt{stop}} 0, 1 \xrightarrow{\texttt{tick}} 1, 1 \xrightarrow{\texttt{tick}} 2\}$
  - $s_0 = 0$

1. commence par une minuscule

Quelques notions utiles

- Les actions déclenchables : $\texttt{fireable} : S \to \mathcal{P}(A)$
  - $\texttt{fireable}(s) = \left\{ a \in A \mid \exists s' \in S, s \xrightarrow{a} s' \in T \right\}$
  - pour $\mathcal{H}$
    - $\texttt{fireable}(0) = ?$ $\texttt{fireable}(1) = ?$ $\texttt{fireable}(2) = ?$

Quelques notions utiles

- Les actions déclenchables : $\texttt{fireable} : S \to \mathcal{P}(A)$
  - $\texttt{fireable}(s) = \left\{ a \in A \mid \exists s' \in S, s \xrightarrow{a} s' \in T \right\}$
  - pour $\mathcal{H}$
    - $\texttt{fireable}(0) = \{\texttt{start}\}$
      $\texttt{fireable}(2) = \{\}$
      $\texttt{fireable}(1) = \{\texttt{tick}, \texttt{stop}\}$

Quelques notions utiles

- Les actions déclenchables : $\texttt{fireable} : S \to \mathcal{P}(A)$
  - $\texttt{fireable}(s) = \left\{ a \in A \mid \exists s' \in S, s \xrightarrow{a} s' \in T \right\}$
  - pour $\mathcal{H}$
    - $\texttt{fireable}(0) = \{\texttt{start}\}$ $\quad\quad$ $\texttt{fireable}(1) = \{\texttt{tick}, \texttt{stop}\}$
      $\texttt{fireable}(2) = \{\}$
- Un état bloqué : aucune action n'est déclenchable
  - $\texttt{blocked}(\texttt{s}) = (\texttt{fireable}(\texttt{s}) = \{\})$
  - pour $\mathcal{H}$
    - $\texttt{blocked}(0) = ?$ $\quad\quad$ $\texttt{blocked}(1) = ?$ $\quad\quad$ $\texttt{blocked}(2) = ?$

Quelques notions utiles

- Les actions déclenchables : $\texttt{fireable} : S \to \mathcal{P}(A)$
  - $\texttt{fireable}(s) = \left\{ a \in A \mid \exists s' \in S, s \xrightarrow{a} s' \in T \right\}$
  - pour $\mathcal{H}$
    - $\texttt{fireable}(0) = \{\texttt{start}\}$ $\quad$ $\texttt{fireable}(1) = \{\texttt{tick}, \texttt{stop}\}$
      $\texttt{fireable}(2) = \{\}$
- Un état bloqué : aucune action n'est déclenchable
  - $\texttt{blocked}(\texttt{s}) = (\texttt{fireable}(\texttt{s}) = \{\})$
  - pour $\mathcal{H}$
    - $\texttt{blocked}(0) = \texttt{false}$ $\quad$ $\texttt{blocked}(1) = \texttt{false}$ $\quad$ $\texttt{blocked}(2) = \texttt{true}$

Quelques notions utiles II

- Les états suivants : $\texttt{next} : S \rightarrow \mathcal{P}(S)$
  - $\texttt{next}(s_1) = \left\{ s_2 \in S \mid \exists a \in A, s_1 \xrightarrow{a} s_2 \in T \right\}$
  - pour $\mathcal{H}$
    - $\texttt{next}(0) = ?$ $\texttt{next}(1) = ?$ $\texttt{next}(2) = ?$
  - $\texttt{blocked(s)} = (\texttt{next(s)} = \{\})$

Quelques notions utiles II

- Les états suivants : $\texttt{next} : S \rightarrow \mathcal{P}(S)$
  - $\texttt{next}(s_1) = \left\{ s_2 \in S \mid \exists a \in A, s_1 \xrightarrow{a} s_2 \in T \right\}$
  - pour $\mathcal{H}$
    - $\texttt{next}(0) = \{1\}$ $\texttt{next}(1) = \{0, 1, 2\}$ $\texttt{next}(2) = \{\}$
  - $\texttt{blocked}(\texttt{s}) = (\texttt{next}(\texttt{s}) = \{\})$

Quelques notions utiles II

- Les états suivants : $\texttt{next} : S \to \mathcal{P}(S)$
  - $\texttt{next}(s_1) = \left\{ s_2 \in S \mid \exists a \in A, s_1 \xrightarrow{a} s_2 \in T \right\}$
  - pour $\mathcal{H}$
    - $\texttt{next}(0) = \{1\}$ $\texttt{next}(1) = \{0, 1, 2\}$ $\texttt{next}(2) = \{\}$
  - $\texttt{blocked}(\texttt{s}) = (\texttt{next}(\texttt{s}) = \{\})$
- Un état puits : soit bloqué soit dans lequel toutes les actions déclenchables bouclent
  - $\texttt{sink}(\texttt{s}) = (\texttt{next}(\texttt{s}) \subseteq \{\texttt{s}\})$
  - pour $\mathcal{H}$
    - $\texttt{sink}(0) = ?$ $\texttt{sink}(1) = ?$ $\texttt{sink}(2) = ?$

Quelques notions utiles II

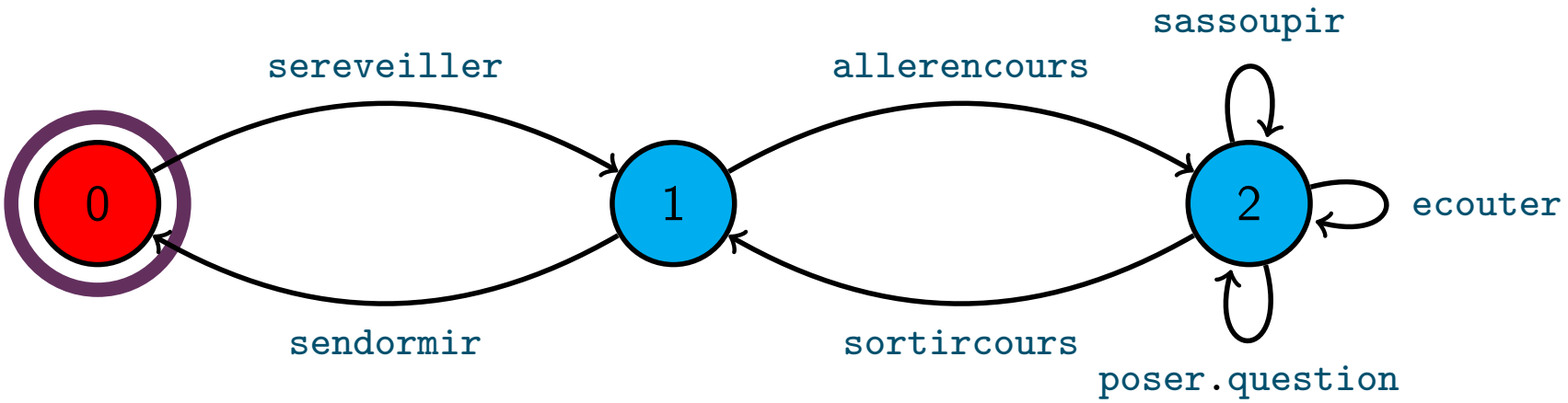
- Les états suivants : $\texttt{next} : S \to \mathcal{P}(S)$
  - $\texttt{next}(s_1) = \left\{ s_2 \in S \mid \exists a \in A, s_1 \xrightarrow{a} s_2 \in T \right\}$
  - pour $\mathcal{H}$
    - $\texttt{next}(0) = \{1\}$ $\texttt{next}(1) = \{0, 1, 2\}$ $\texttt{next}(2) = \{\}$
  - $\texttt{blocked}(\texttt{s}) = (\texttt{next}(\texttt{s}) = \{\})$
- Un état puits : soit bloqué soit dans lequel toutes les actions déclenchables bouclent
  - $\texttt{sink}(\texttt{s}) = (\texttt{next}(\texttt{s}) \subseteq \{\texttt{s}\})$
  - pour $\mathcal{H}$
    - $\texttt{sink}(0) = \texttt{false}$ $\texttt{sink}(1) = \texttt{false}$ $\texttt{sink}(2) = \texttt{true}$

Quelle est la sémantique d'un LTS ?

- Comportement d'un LTS = ensemble de ses traces
- Une trace de (S, A, T, i) : suite d'actions $\langle a_n \rangle_{n \in N}$ telle que
  - $N = \begin{cases} \mathbb{N} & \textit{(trace infinie)} \\ [\![0, k]\!] \quad \text{pour } k \in \mathbb{N} & \textit{(trace finie)} \end{cases}$
  - il existe une suite d'états de S $\langle s_n \rangle_{n \in N}$ avec $s_0 = i$ et $\forall n \in N, s_n \xrightarrow{a_n} s_{n+1} \in T$
- pour $\mathcal{H}$
  - sont des traces : $\langle$`start`$\rangle$, $\langle$`start`, `stop`$\rangle$, $\langle$`start`, `tick`$\rangle$ ou $\langle$`start`, `tick`, ..., `tick`, `stop`$\rangle$
  - pas des traces : $\langle$`tick`$\rangle$, $\langle$`start`, `to`$\rangle$, $\langle$`stop`, `start`, `tick`$\rangle$

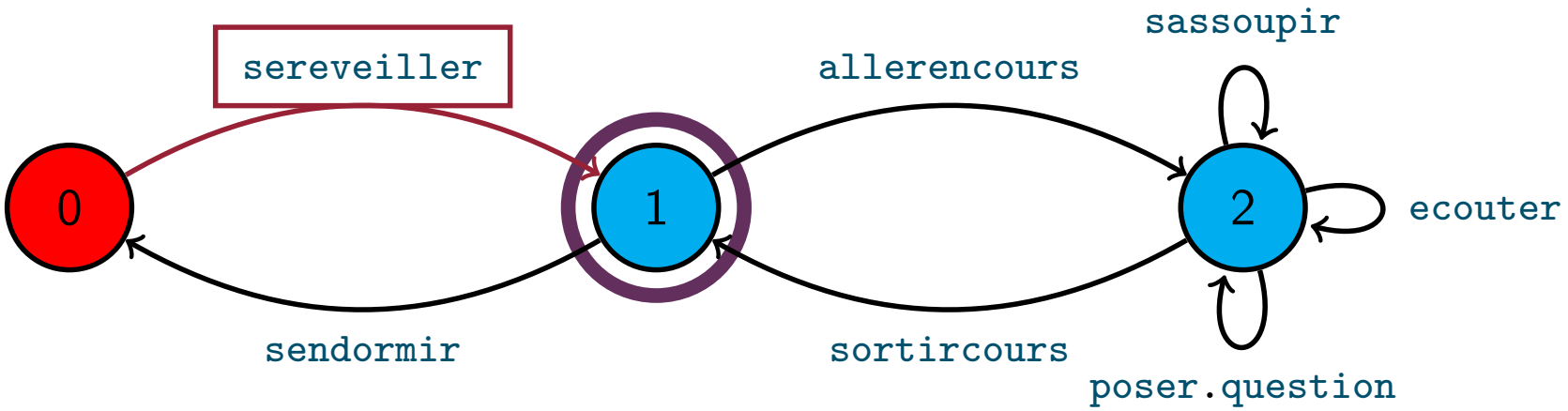
Animer un modèle

- Suivre une trace, c'est animer un modèle
- Une trace



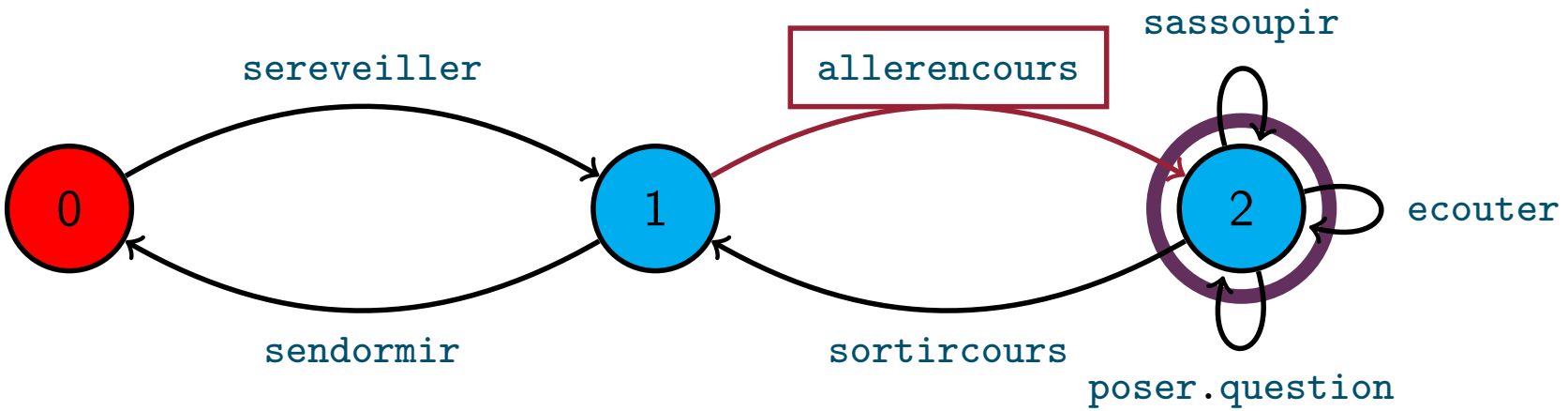
Animer un modèle

- Suivre une trace, c'est animer un modèle
- Une trace
  - `sereveiller`



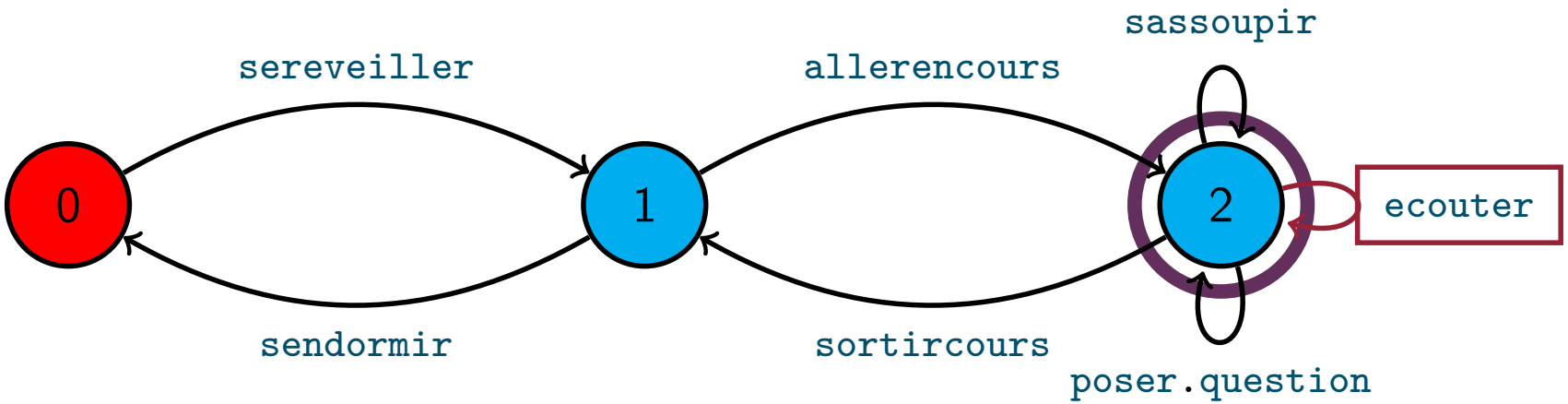
Animer un modèle

- Suivre une trace, c'est animer un modèle
- Une trace
  - `sereveiller`, `allerencours`



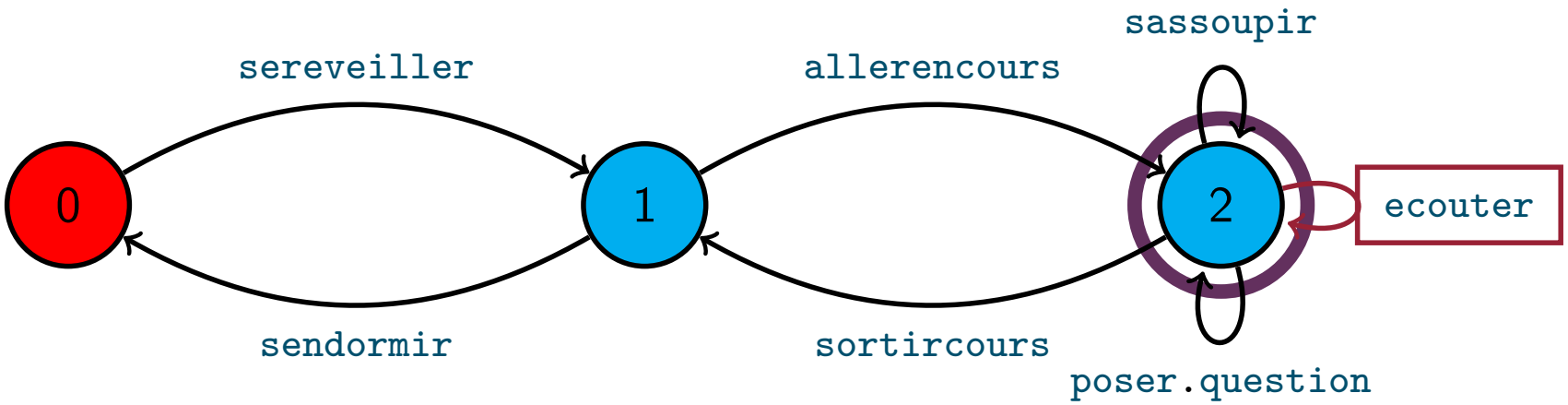
Animer un modèle

- Suivre une trace, c'est animer un modèle
- Une trace
  - `sereveiller`, `allerencours`, `ecouter`



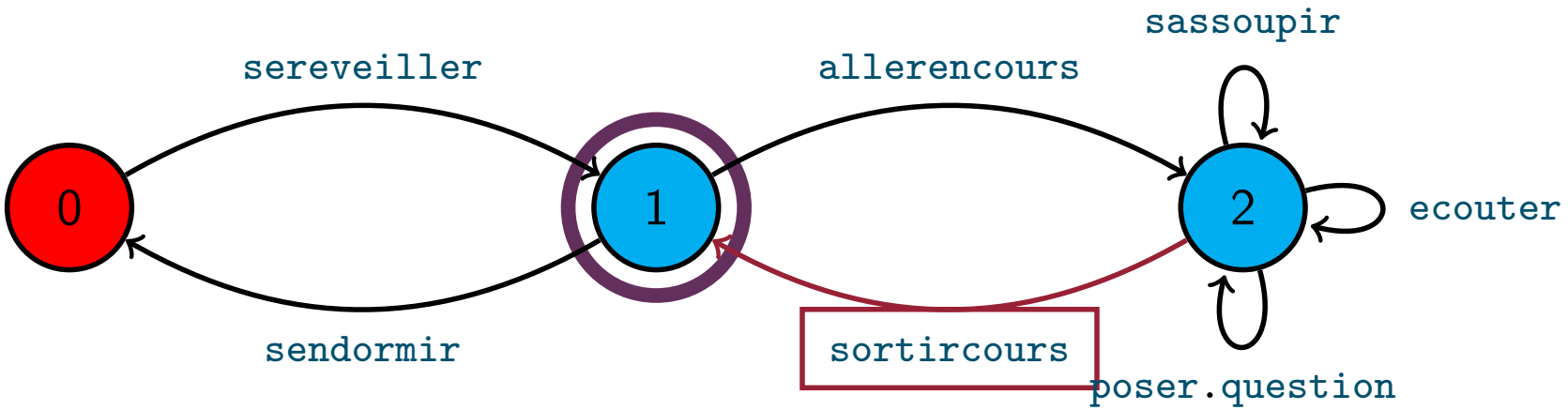
Animer un modèle

- Suivre une trace, c'est animer un modèle
- Une trace
  - `sereveiller`, `allerencours`, `ecouter`, `ecouter`



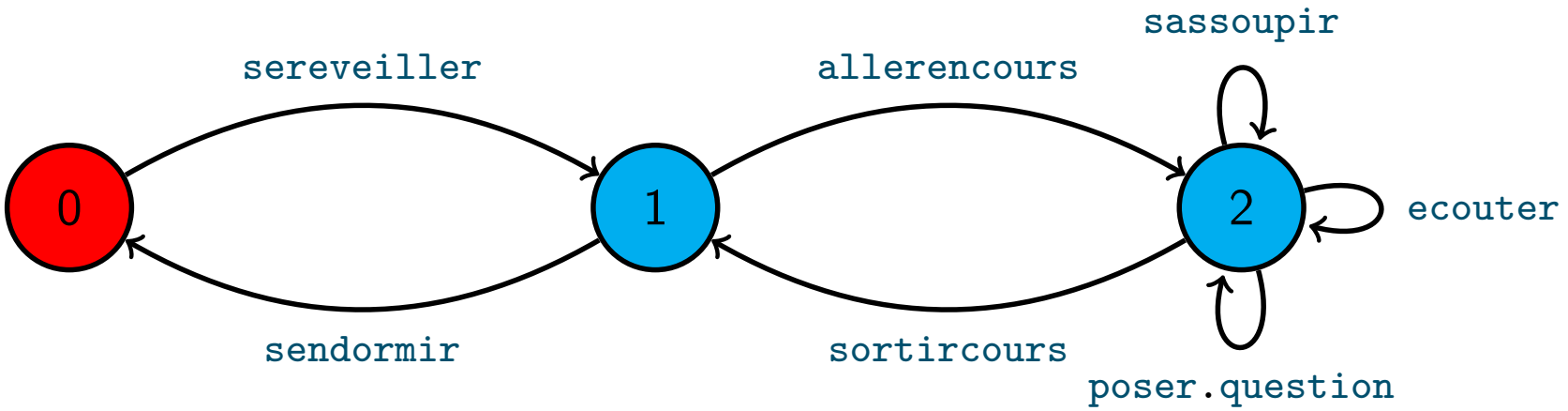
Animer un modèle

- Suivre une trace, c'est animer un modèle
- Une trace
  - `sereveiller`, `allerencours`, `ecouter`, `ecouter`, `sortircours`

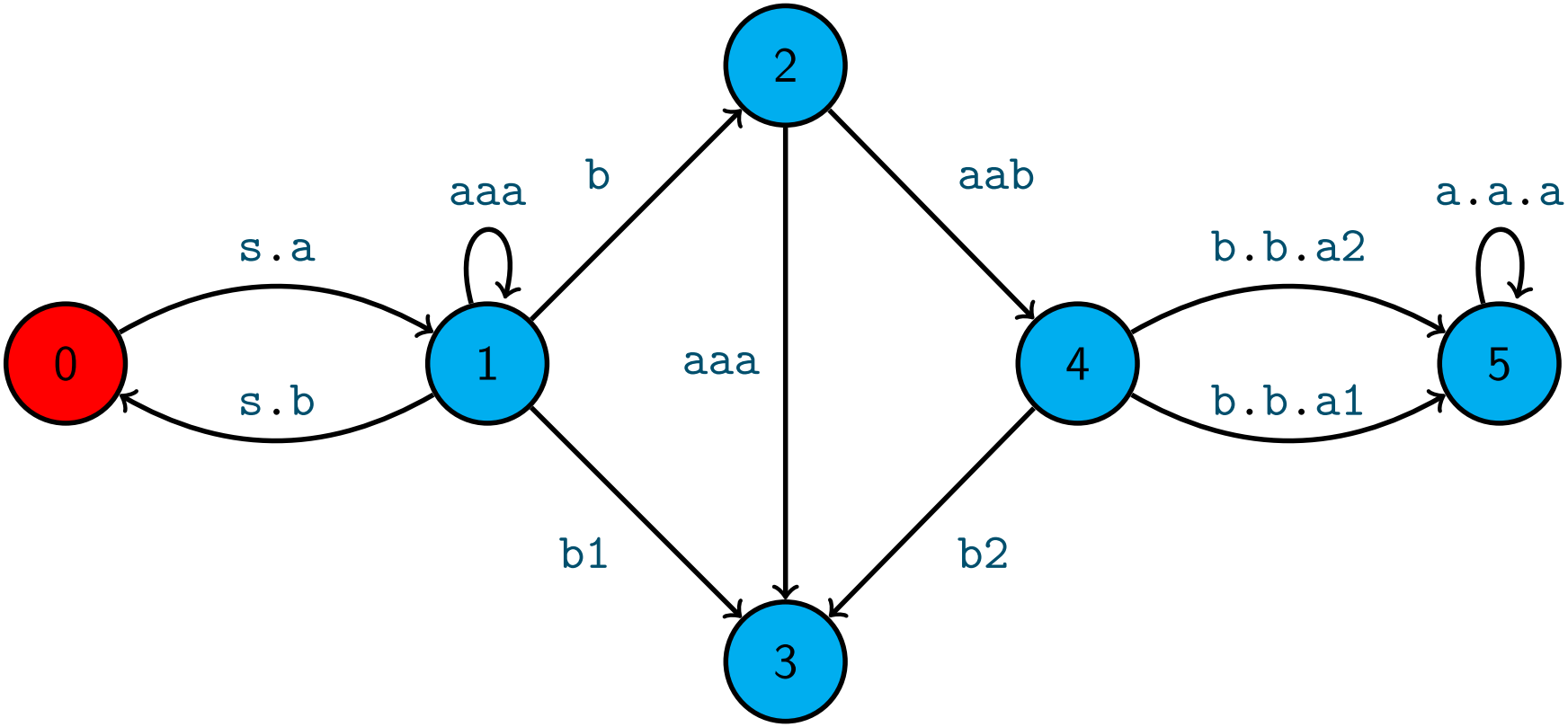


Animer un modèle

- Suivre une trace, c'est animer un modèle
- Une trace
  - `sereveiller`, `allerencours`, `ecouter`, `ecouter` `sortircours`, ...



Un exercice



- Des traces ?
- Des états bloqués ou puits ?

Avancement

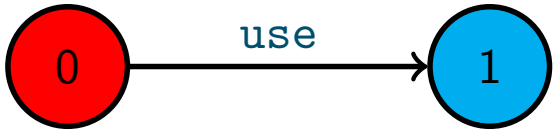
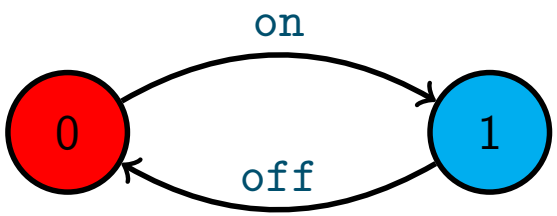
Et pour un processus plus complexe ?

- Si le processus contient
  - beaucoup d'états
  - beaucoup de transitions
  - un grand nombre de transitions de ou à partir d'un état
- La représentation graphique devient difficile à construire et à manipuler

$\Rightarrow$ FSP, un langage

- pour décrire des processus
- est interprété comme un automate (LTS)
- équivalence entre un programme FSP et son LTS

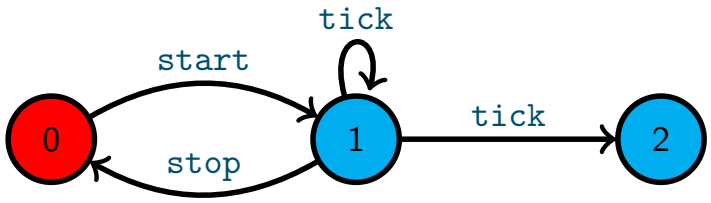
Des premiers exemples de FSP I

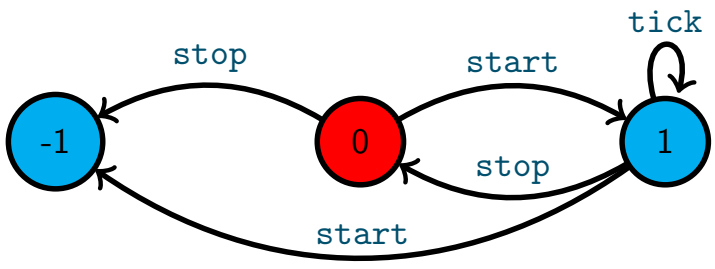
FSP	LTS
`ONESHOT = (use -> STOP).`	
`OFF = (on -> ON),` `ON = (off -> OFF).`	

- Processus en majuscule, seul nom public, définition terminée par ".", `STOP` état bloqué prédéfini
- Actions en minuscule et transition par "->"
- Définitions *récursives* d'états en majuscule, séparées par ","

Des premiers exemples de FSP II

FSP	LTS

`H = (start -> ON),`
`ON = ( stop -> H | tick -> ON`
`  | tick -> STOP).`
	

`H = (start -> ON | stop -> ERROR),`
`ON = ( stop -> H | tick -> ON`
`  | start -> K).`
	

- Choix par "|"
- État prédéfini `ERROR` bloqué (-1 dans LTS)
- État non défini est égal à `ERROR`

Un exercice

- Communication non fiable
  - Modéliser un canal de communication qui accepte des actions `in`
  - Si une faute se produit, il ne génère pas de sortie, sinon il génère une action `out`

Un exercice

- Communication non fiable
  - Modéliser un canal de communication qui accepte des actions `in`
  - Si une faute se produit, il ne génère pas de sortie, sinon il génère une action `out`
- Utiliser le choix (non déterminisme)
  - `CHAN = (in -> CHAN | in -> OUT),`
    `OUT = (out -> CHAN).`

  ou de manière équivalente

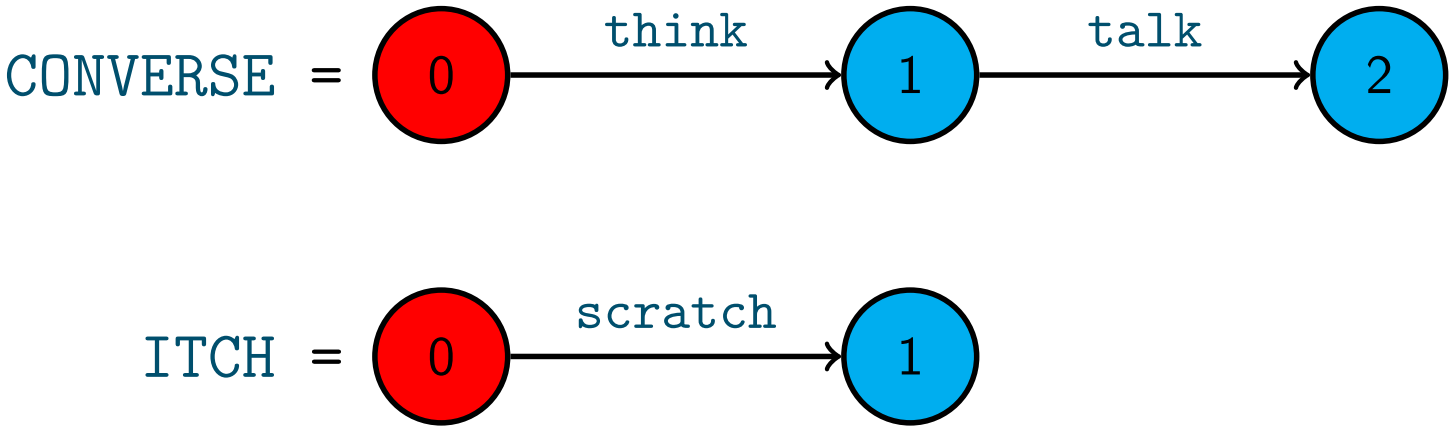
  - ```
    CHAN = ( in -> CHAN
           | in -> out -> CHAN).
    ```
    ```

Avancement

Composition parallèle

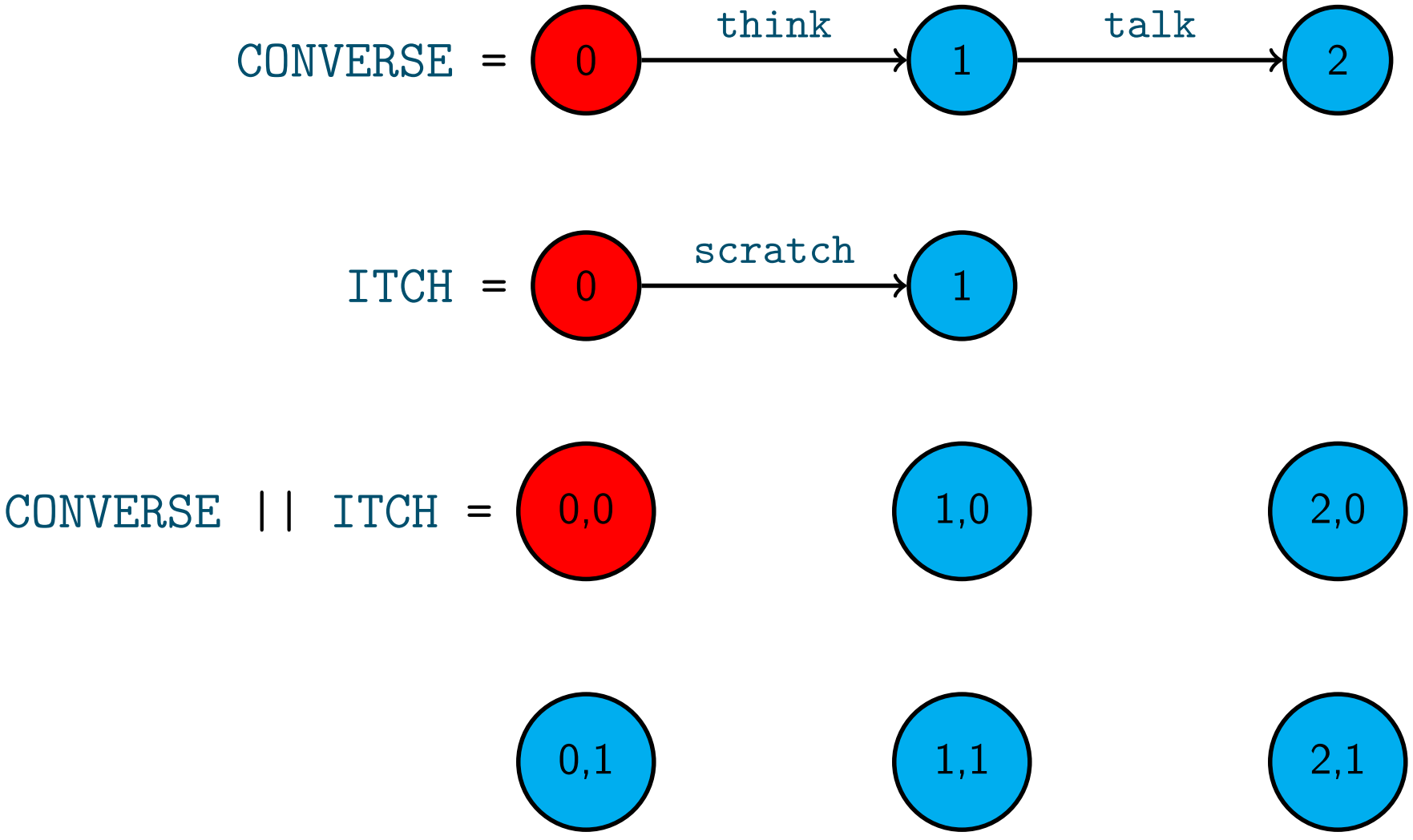
- `||NOMCOMPOSITION = (P1 || ... || Pn)`
  - commutatif et associatif,
  - élément neutre `STOP`, élément absorbant `ERROR`
- Sémantique intuitive (composition asynchrone)
  - exécution simultanée des actions de même nom
  - 2 actions indépendantes ne s'exécutent pas simultanément
  - tous les entrelacements possibles des actions indépendantes de tous les processus
- Sémantique formelle (LTS de la composition)
  - $S = S_1 \times S_2 \qquad A = A_1 \cup A_2 \qquad i = (i_1, i_2)$
  - $T = \{(s_1, s_2) \xrightarrow{a} (s'_1, s'_2) \mid s_1 \xrightarrow{a} s'_1 \wedge s_2 \xrightarrow{a} s'_2 \wedge a \in A_1 \cap A_2 \vee s_1 = s'_1 \wedge s_2 \xrightarrow{a} s'_2 \wedge a \notin A_1 \vee s_2 = s'_2 \wedge s_1 \xrightarrow{a} s'_1 \wedge a \notin A_2\}$
  - en général, on élimine les états non atteignables

Un premier exemple

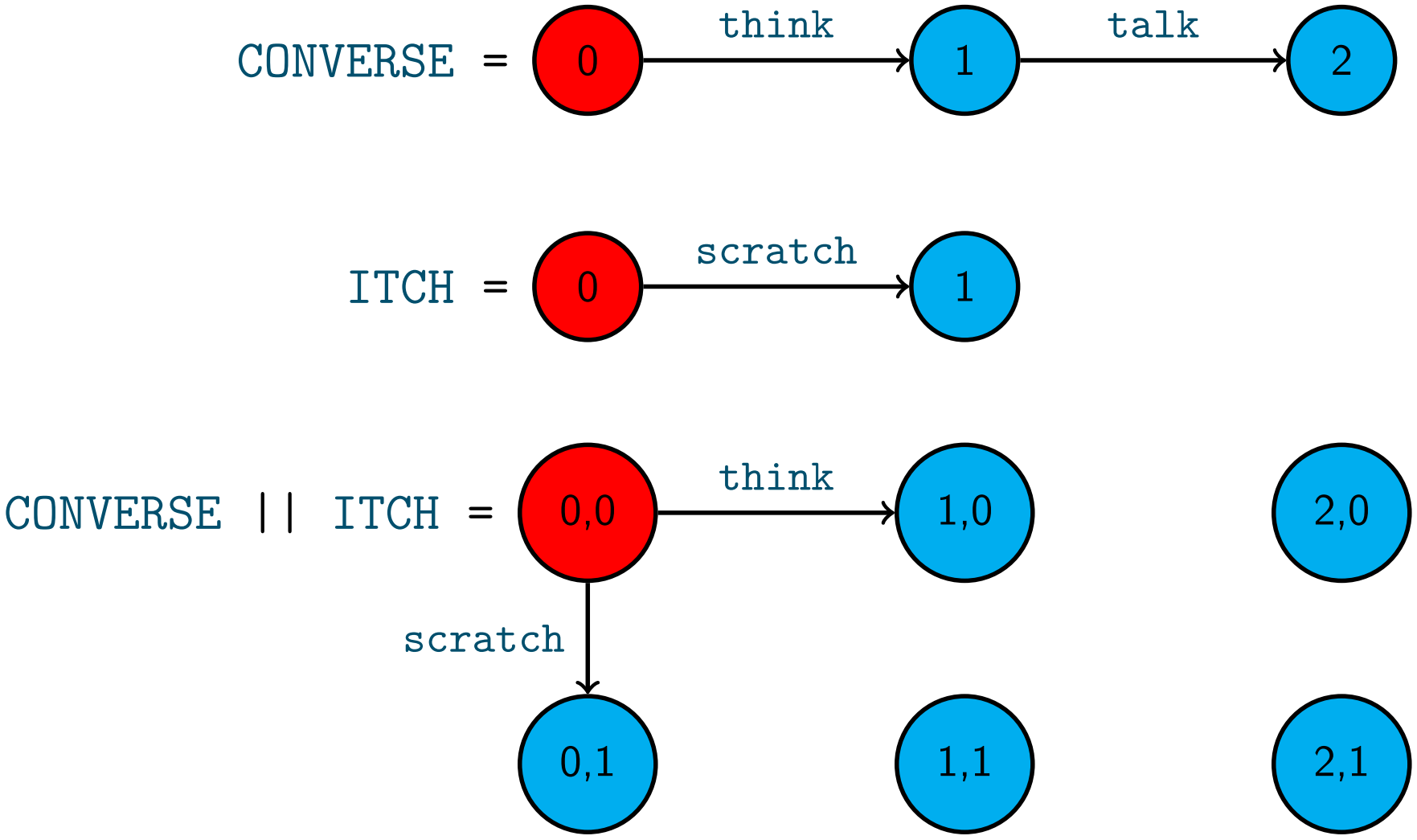


CONVERSE || ITCH =

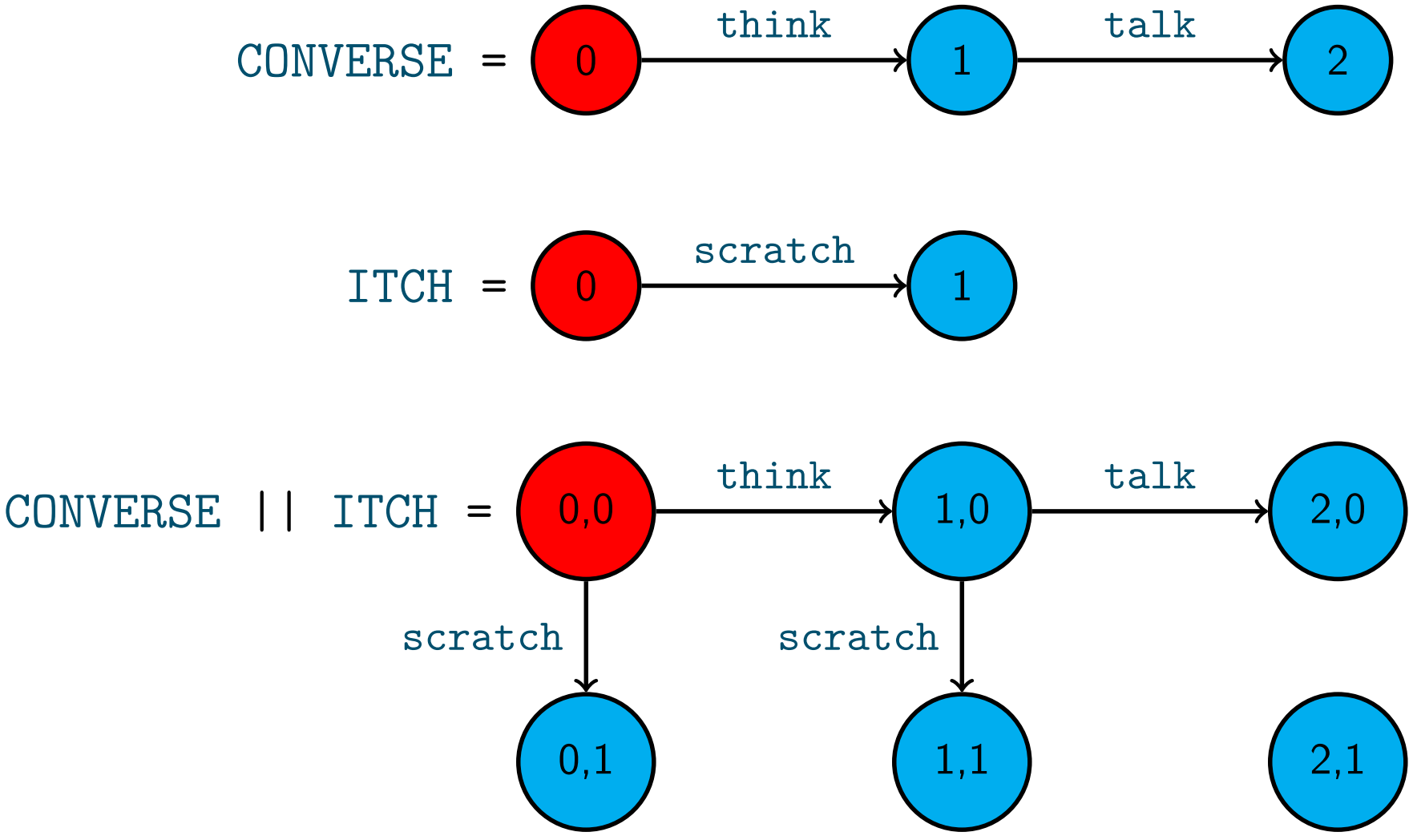
Un premier exemple



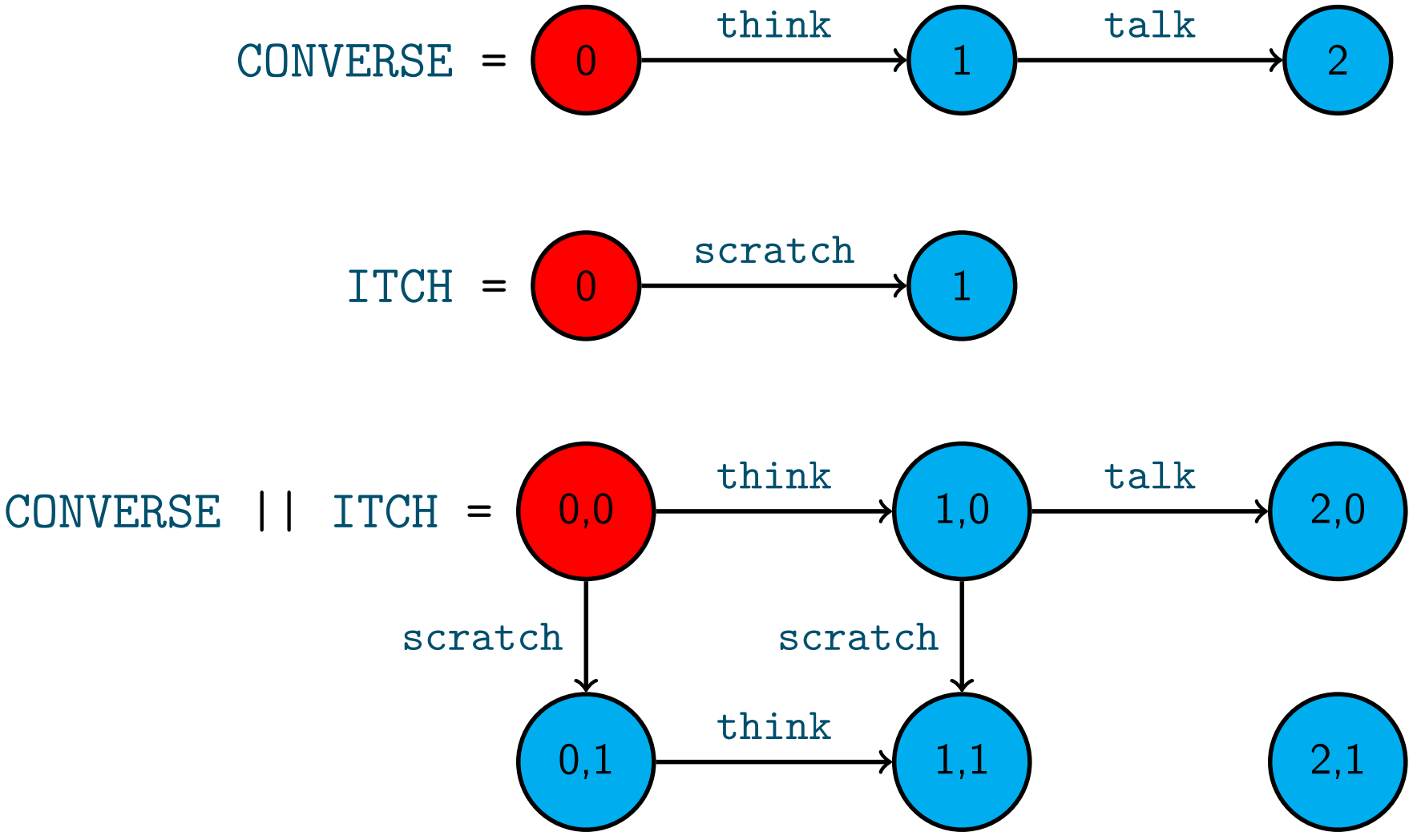
Un premier exemple



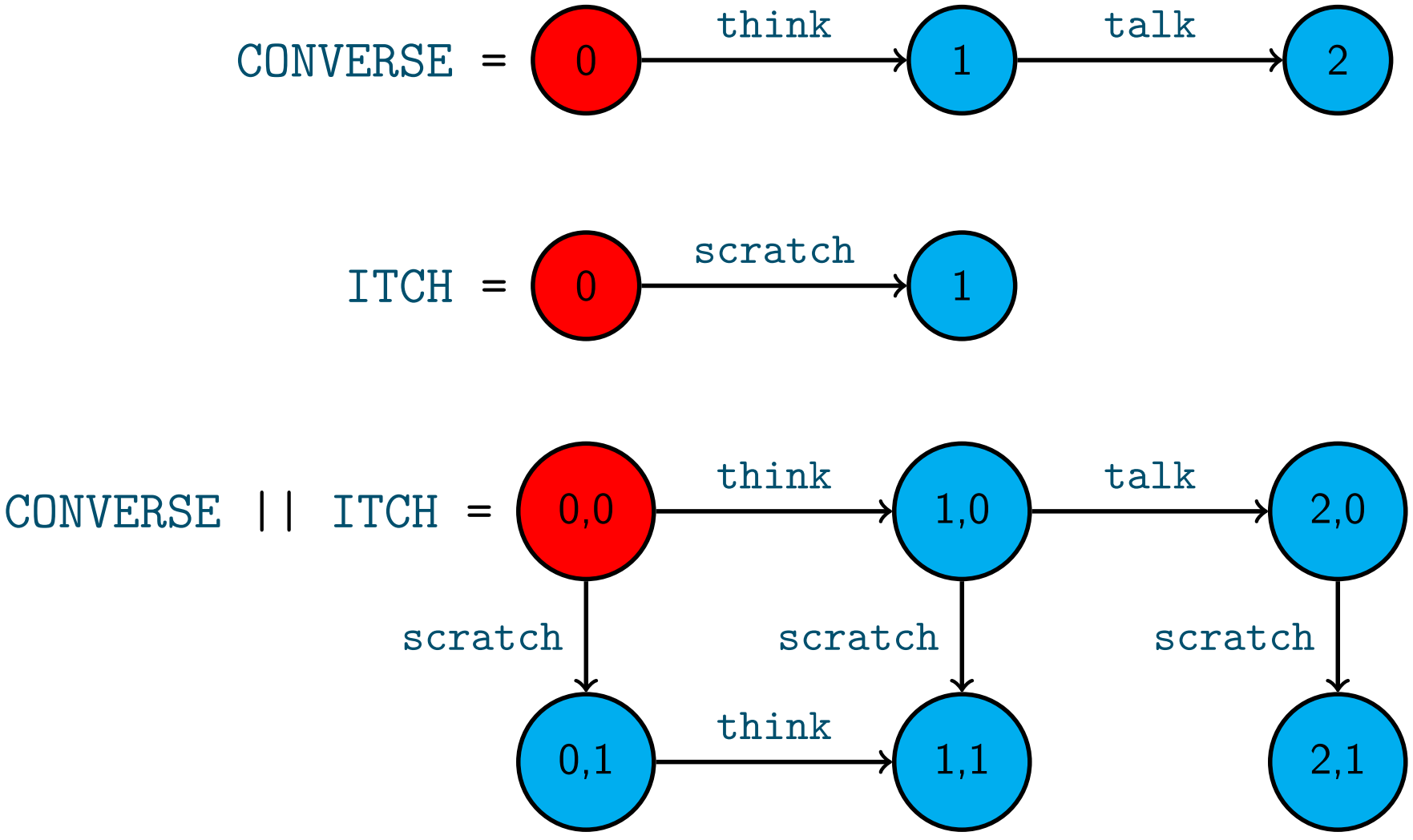
Un premier exemple



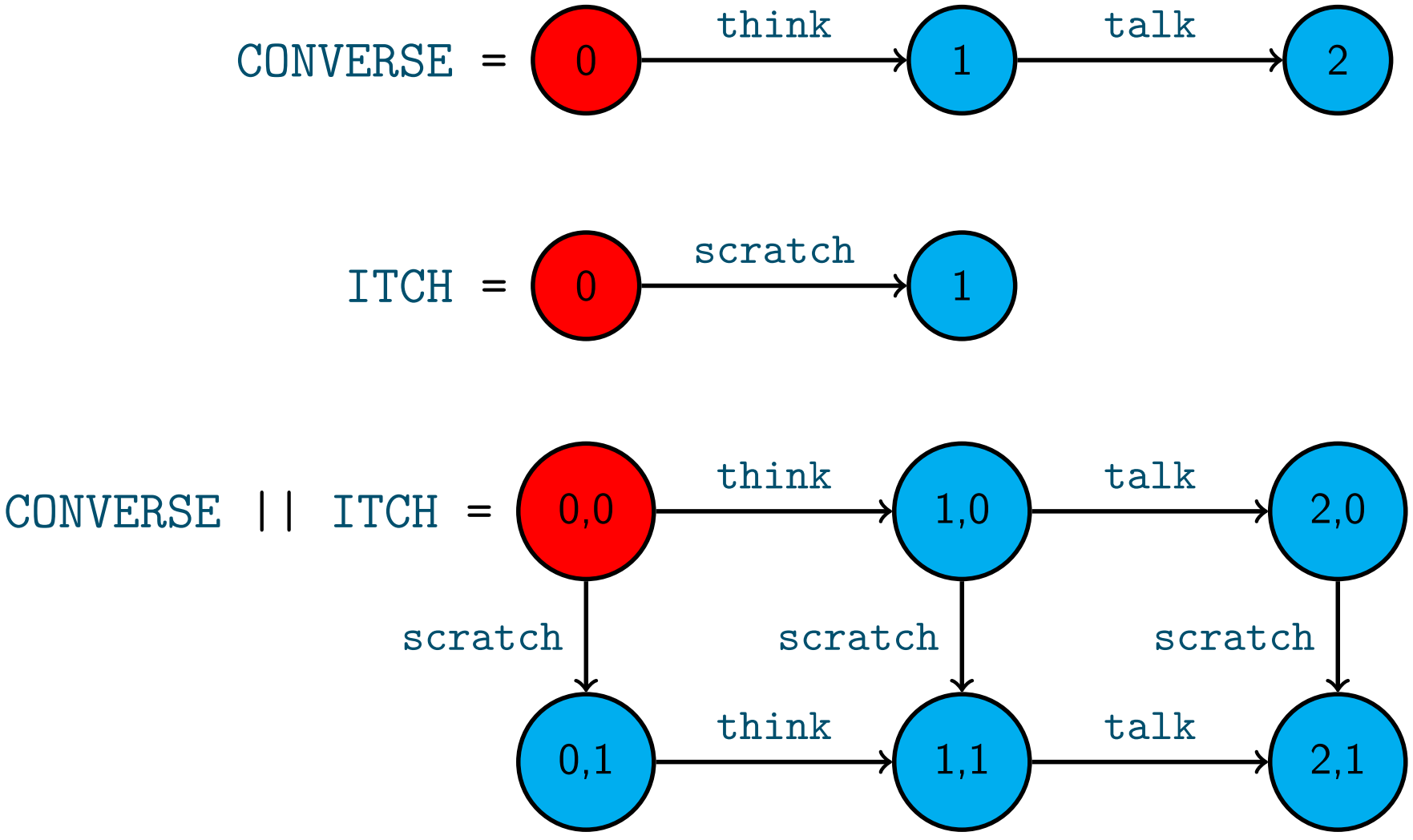
Un premier exemple



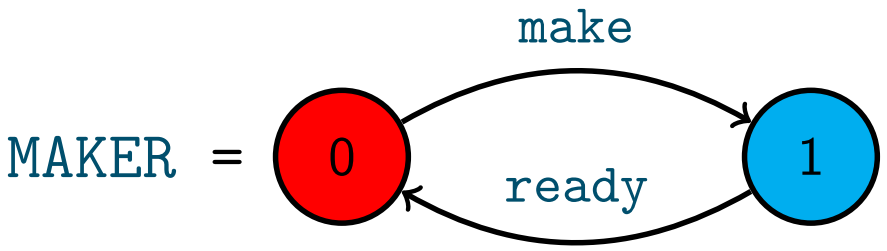
Un premier exemple

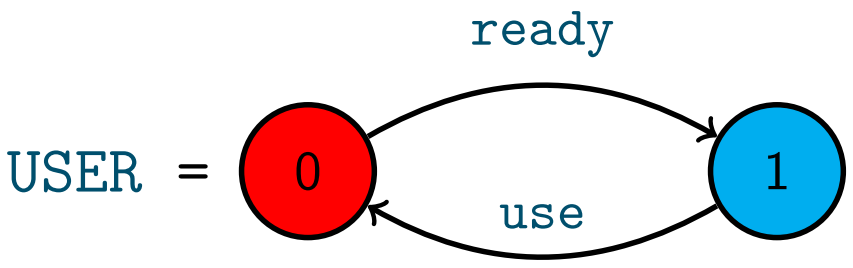


Un premier exemple



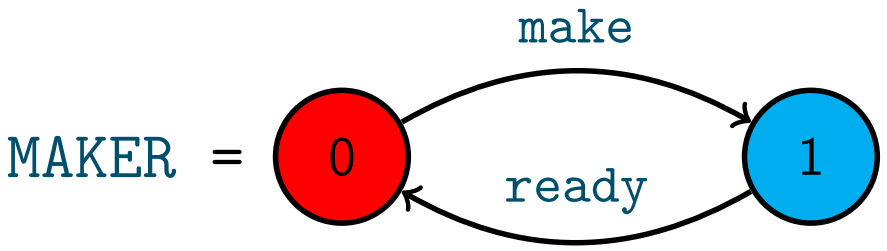
Un second exemple, action partagée



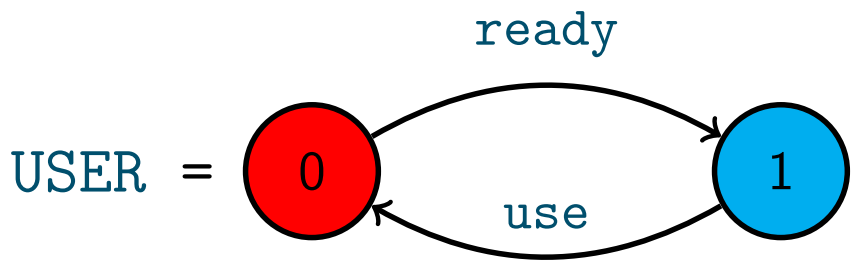


MAKER || USER =

Un second exemple, action partagée



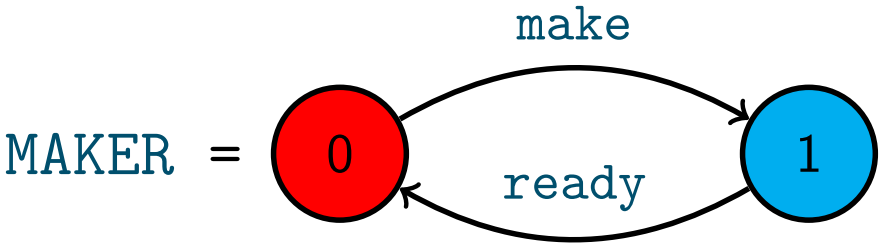
(make -> ready -> MAKER).



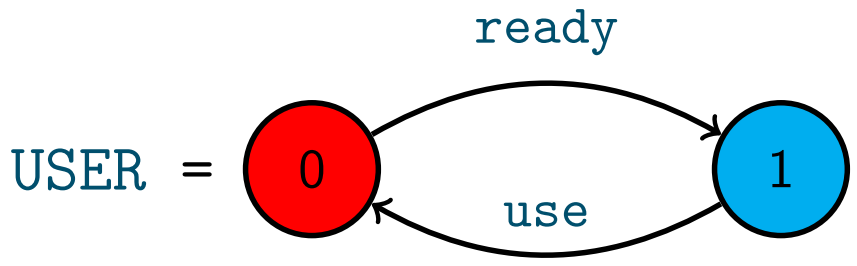
(ready -> use -> USER).

MAKER || USER =

Un second exemple, action partagée



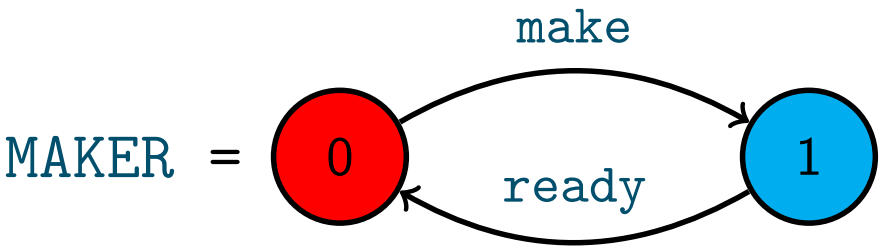

```
(make -> ready -> MAKER).
```



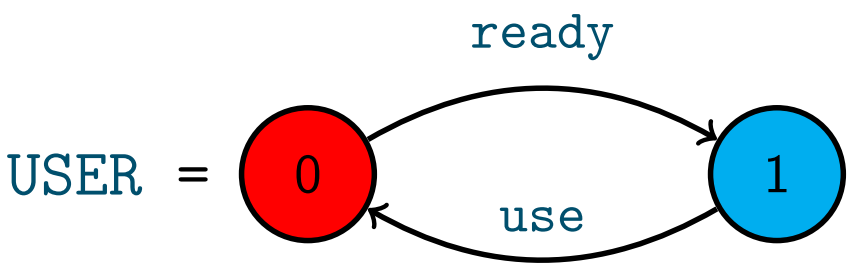

```
(ready -> use -> USER).
```

```
MAKER || USER =
```

Un second exemple, action partagée

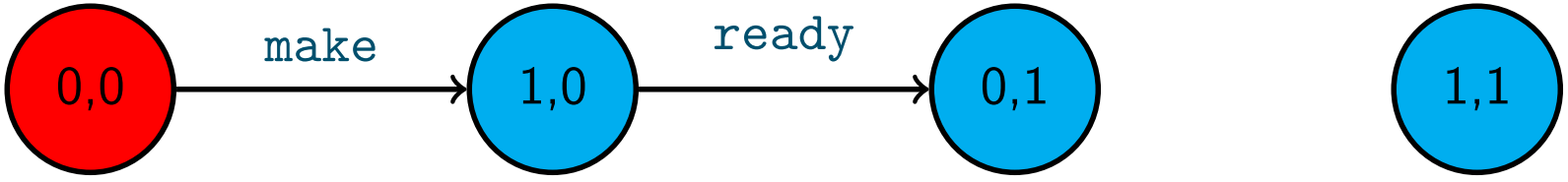



```
(make -> ready -> MAKER).
```

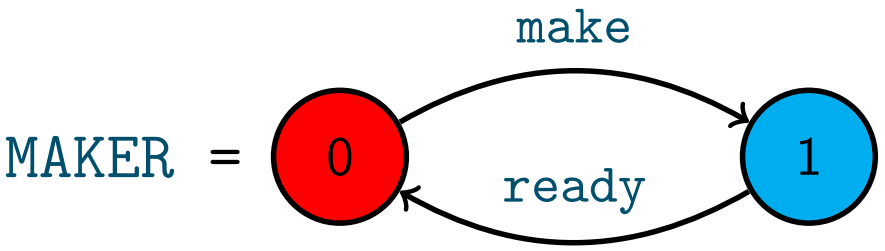



```
(ready -> use -> USER).
```

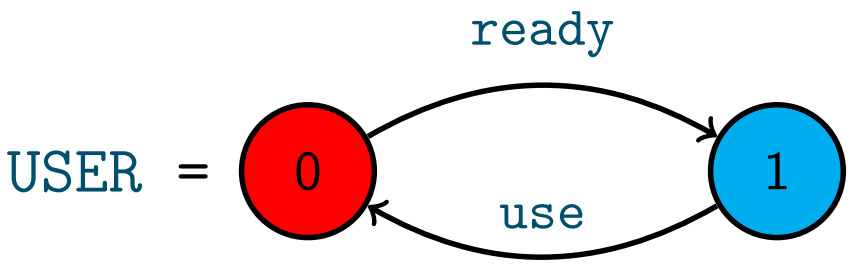
```
MAKER || USER =
```



Un second exemple, action partagée

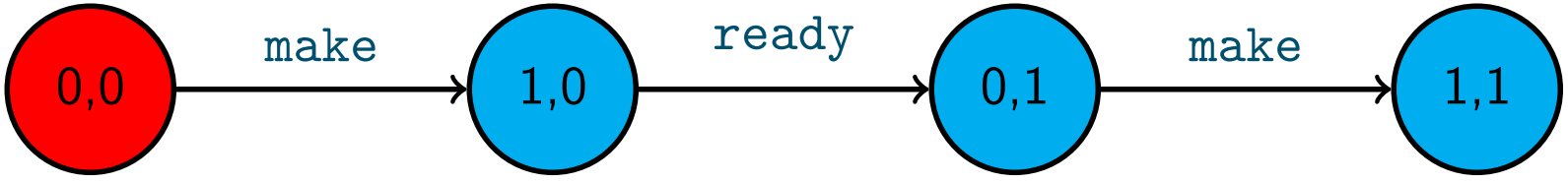



```
(make -> ready -> MAKER).
```

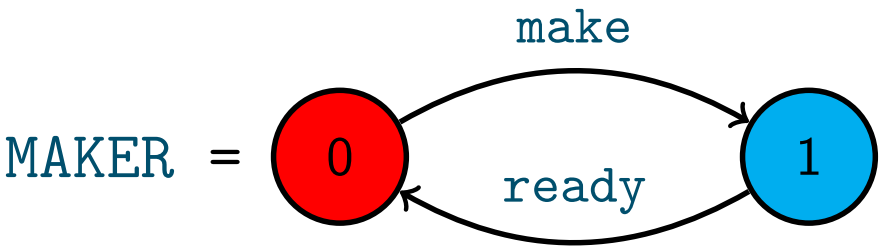



```
(ready -> use -> USER).
```

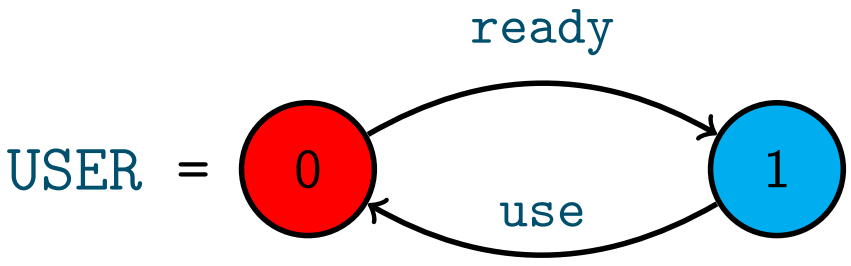
```
MAKER || USER =
```



Un second exemple, action partagée



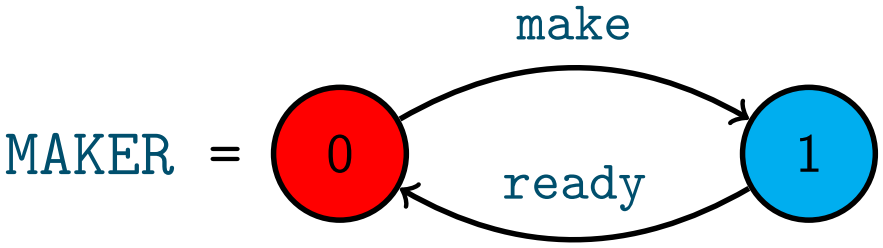

```
(make -> ready -> MAKER).
```



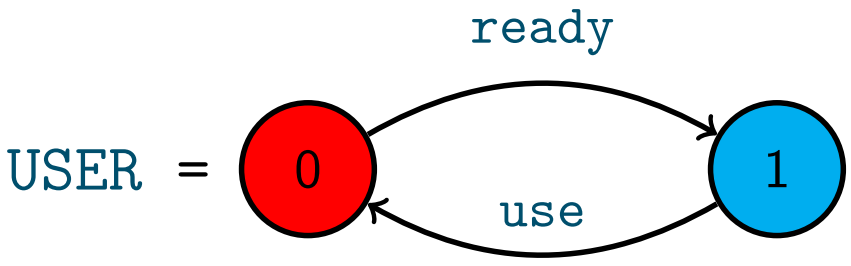

```
(ready -> use -> USER).
```

```
MAKER || USER =
```

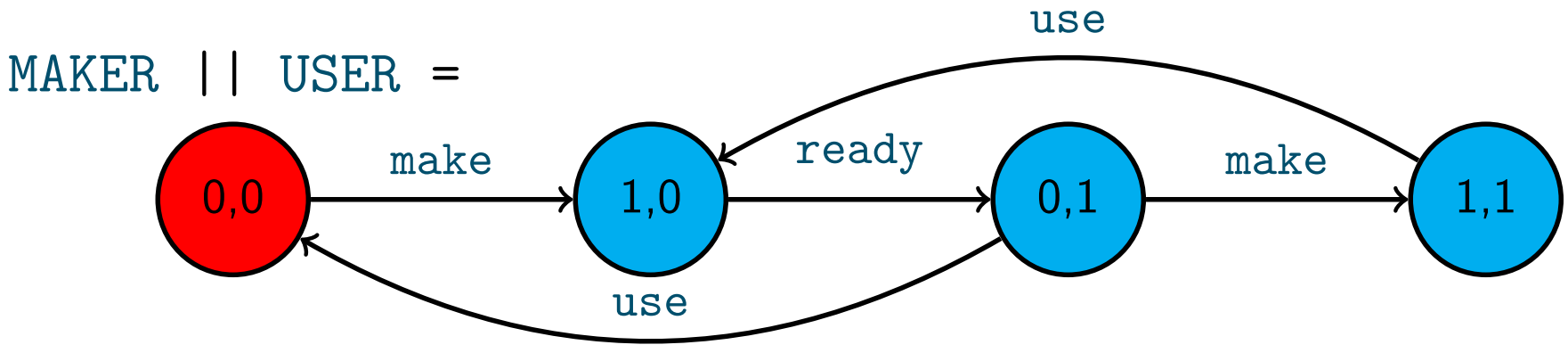
Un second exemple, action partagée



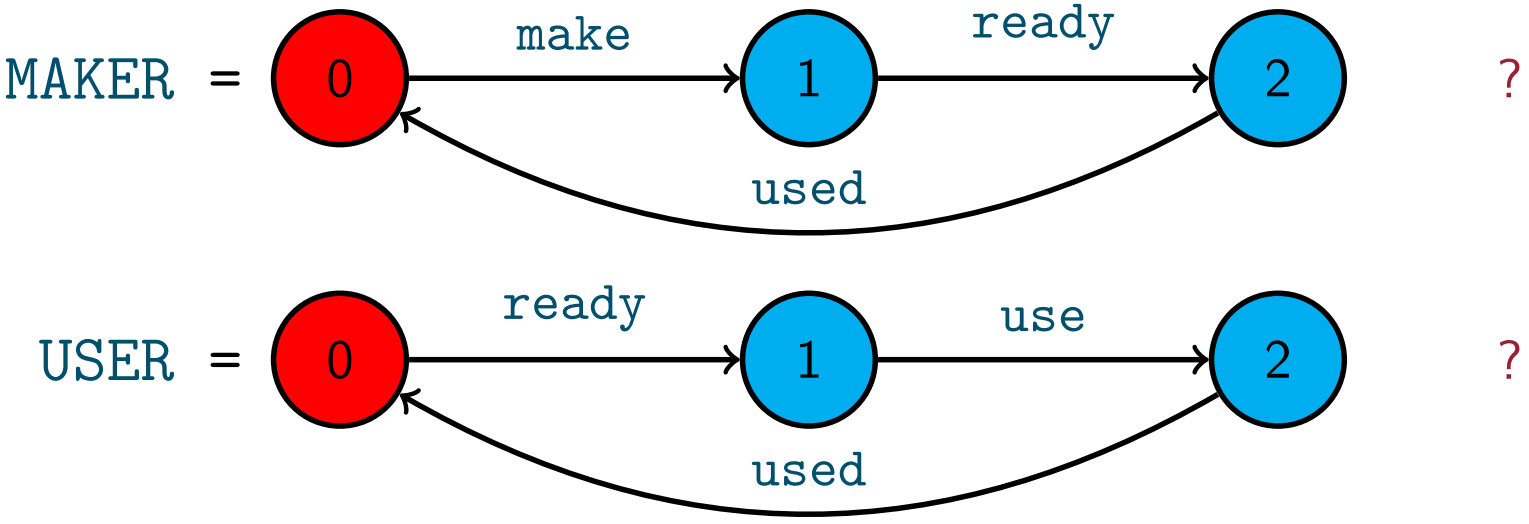
(make -> ready -> MAKER).



(ready -> use -> USER).

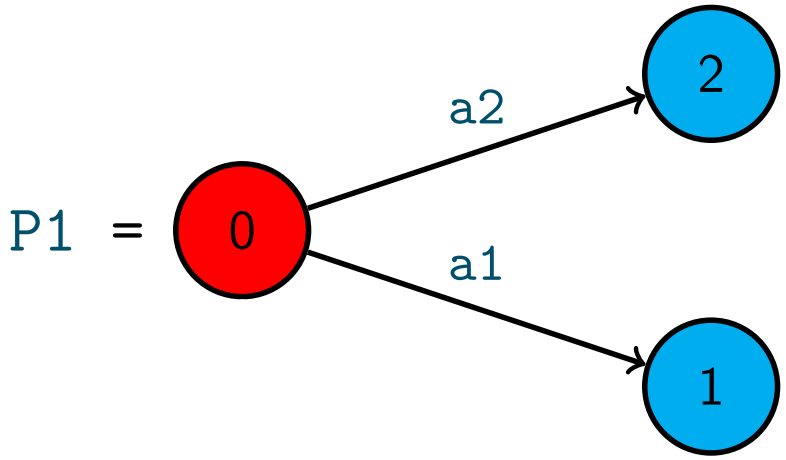


Un exercice, la poignée de main



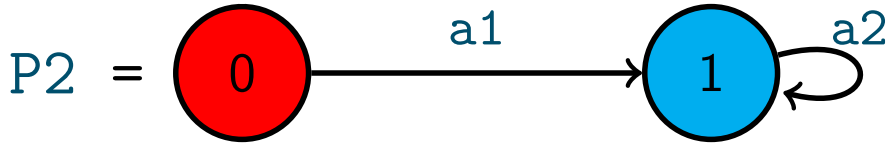
MAKER || USER = ?

Composition et choix



?

P1 || P2 = ?



?

- La synchronisation permet de sélectionner de l'extérieur une branche d'un choix

La notion d'instance

- Soit `SWITCH = (on -> off -> SWITCH).`
- `SWITCH || SWITCH` $\equiv$?

La notion d'instance

- Soit `SWITCH = (on -> off -> SWITCH).`
- `SWITCH || SWITCH` $\equiv$ `SWITCH`

La notion d'instance

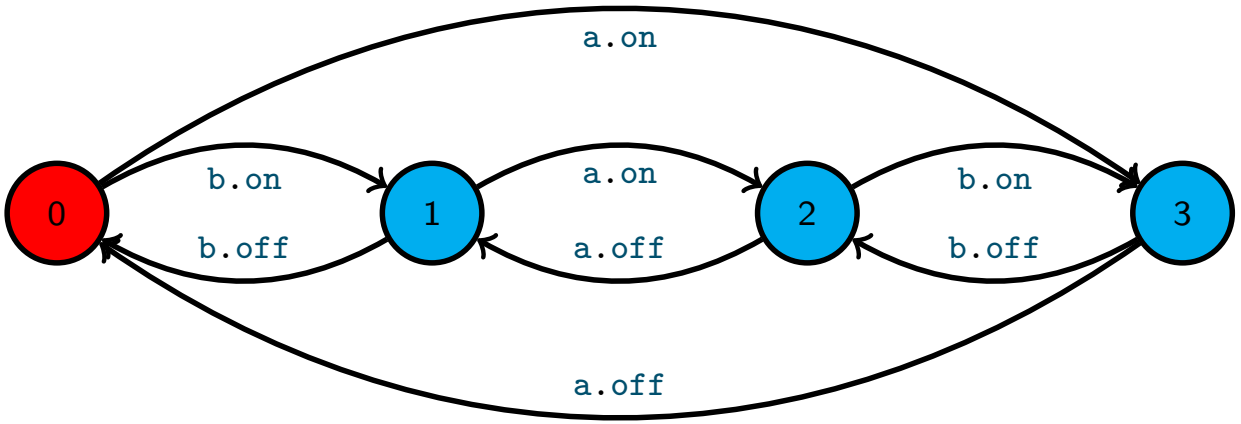
- Soit `SWITCH = (on -> off -> SWITCH).`
- `SWITCH || SWITCH` $\equiv$ `SWITCH`
- Comment modéliser deux interrupteurs ?

La notion d'instance

- Soit `SWITCH = (on -> off -> SWITCH)`.
- `SWITCH || SWITCH` $\equiv$ `SWITCH`
- Comment modéliser deux interrupteurs ?
- Notion d'instance, `a:SWITCH`
  - Ajoute `a` en préfixe des actions de `SWITCH`
  - $\equiv$ `ASW` avec `ASW = (a.on -> a.off -> ASW)`

La notion d'instance

- Soit `SWITCH = (on -> off -> SWITCH)`.
- `SWITCH || SWITCH` $\equiv$ `SWITCH`
- Comment modéliser deux interrupteurs ?
- Notion d'instance, `a:SWITCH`
  - Ajoute `a` en préfixe des actions de `SWITCH`
  - $\equiv$ `ASW` avec `ASW = (a.on -> a.off -> ASW)`
- LTS de `a:SWITCH || b:SWITCH` ?



Renommage

- On peut renommer une (préfixe d') action
  - `P/{remplaçant,remplacé}`
  - `(call -> P)/{request/call}` ≡ `(request -> P)`
  - `(call.a -> call.b -> P)/{request/call}` ≡ `(request.a -> request.b -> P)`
  - `(call.a -> a -> P)/{b/a}` ≡ `(call.a -> b -> P)`
  - `(call.a -> call.b -> P)/{r/call,n/call.a}` ≡ `(n -> r.b -> P)`
- Renommages multiples
  - `P/{{a,b}/{x,y}}` ≡ `P/{a/x, a/y, b/x, b/y}`
- Utile principalement pour composer des processus

Interaction avec de multiples instances

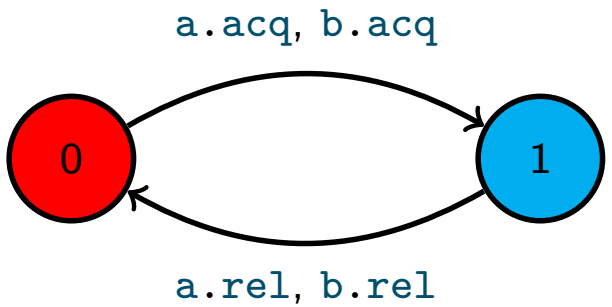
- Utilisateur : `USER = (acq -> use -> rel -> USER).`
- Ressource : `RESOURCE = (acq -> rel -> RESOURCE).`
- Comment avoir plusieurs utilisateurs pour une ressource ?
- `a:USER || b:USER || RESOURCE` ?

Interaction avec de multiples instances

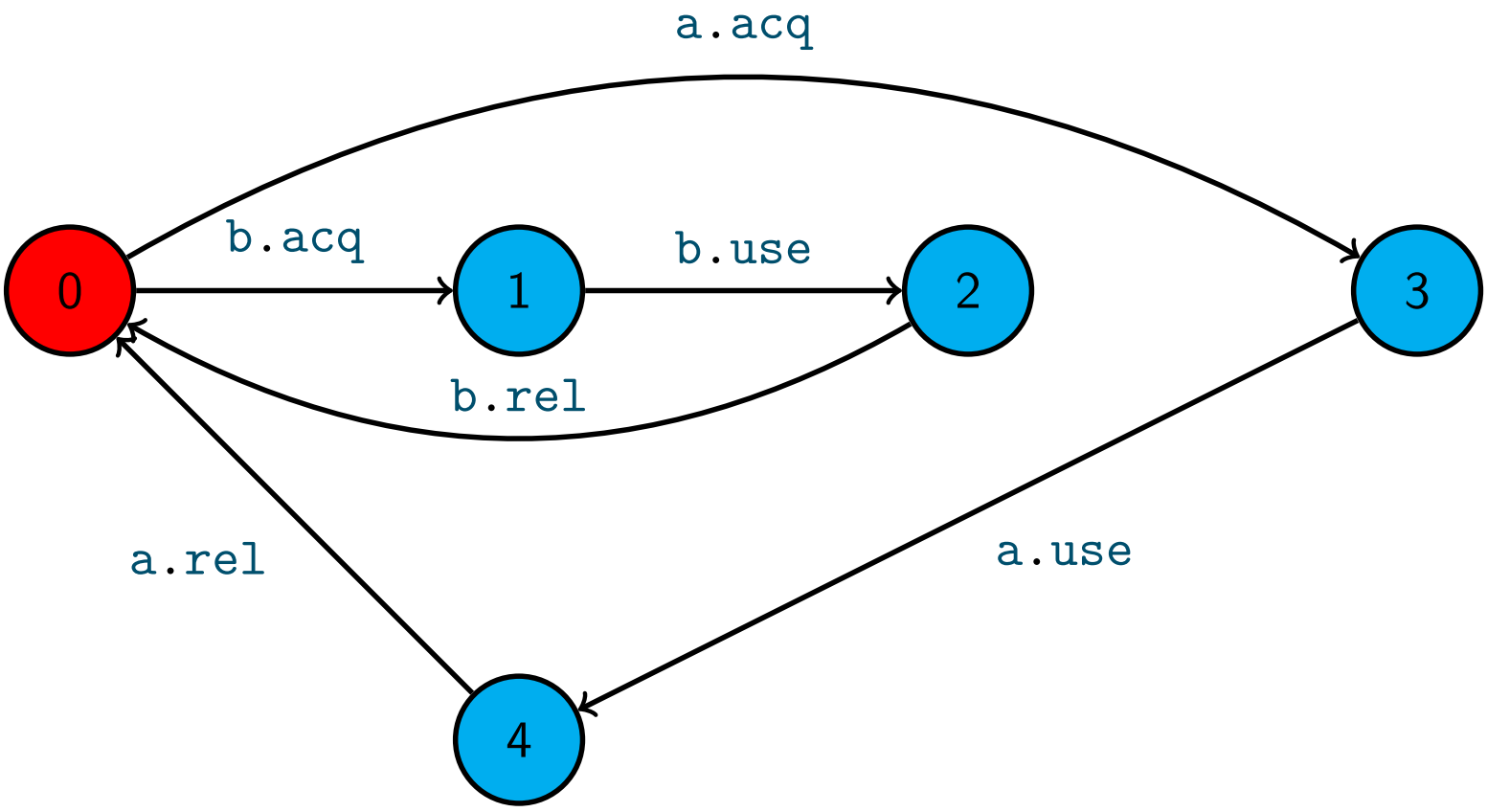
- Utilisateur : `USER = (acq -> use -> rel -> USER).`
- Ressource : `RESOURCE = (acq -> rel -> RESOURCE).`
- Comment avoir plusieurs utilisateurs pour une ressource ?
- `a:USER || b:USER || RESOURCE` ?
- Non, il faut du renommage

Interaction avec de multiples instances

- Utilisateur : `USER = (acq -> use -> rel -> USER).`
- Ressource : `RESOURCE = (acq -> rel -> RESOURCE).`
- Comment avoir plusieurs utilisateurs pour une ressource ?
- `a:USER || b:USER || RESOURCE` ?
- Non, il faut du renommage
- `a:USER || b:USER || {a,b}::RESOURCE`



Résultat



- Comment le modèle assure-t-il que l'utilisateur qui acquiert la ressource est celui qui la libère ?

Conclusion

- Notion de LTS (automate)
- Syntaxe de base pour définir des processus (FSP)
- Sémantique de ce cœur

$$P ::= \texttt{STOP} \mid \texttt{ERROR} \mid X \mid (a \texttt{ -> } P) \mid P_1 \texttt{ | } P_2 \mid P_1 \texttt{ || } P_2$$

- Il faut pratiquer
  - Construire des modèles
  - Les valider en les animant (LTSA)