Majeure Informatique

Systèmes Distribués

Modélisation des systèmes concurrents et distribués

UV2 MAJ INF 447

Responsable : F. Dagnat

1er semestre 2016

Table des matières

Fiche programme 5

I La modélisation des systèmes concurrents — 7

C 1 : Modélisation de la concurrence 9

Notes de cours sur le langage FSP 19

PC 1-2, TP 1-2 : Système de transitions étiquetées et pratique de FSP 25

II Les propriétés et le passage au code — 27

C 2 : Propriétés et passage au code 29

Notes de cours sur le passage au code 39

PC 3, TP3 : Le repas des philosophes 43

TP 4-5 : Du modèle au code : *Roller coaster* 45

III La modélisation des systèmes répartis — 47

PC 4 : Modélisation de systèmes répartis, Échange de messages et aspects temporels 49

PC 5 : Le kem's 51

Auteurs

Fabien Dagnat, Jean Marie Gilliot, Julien Mallet, Siegfried Rouvrais, Maria-Teresa Segarra, Gwendal Simon

Fiche Programme – Systèmes distribués

Contexte et description

Les systèmes informatiques actuels reposent de plus en plus souvent sur un ensemble de processeurs qui collaborent pour exécuter une même tâche. Les ordinateurs multi-core ou l'Internet en sont des exemples frappants. La conception d'un programme pouvant s'exécuter sur un système distribué n'est pourtant pas une tâche aisée.

La modélisation de systèmes distribués et la conception de programmes concurrents sont au cœur de ce module. Pour l'ingénieur du XXIe siècle, une excellente capacité à concevoir des programmes capables de tirer profit de la multiplicité des processeurs est nécessaire. Ce module vise à apporter quelques bases dans ce domaine.

Nous utilisons un modèle théorique simple de programmation concurrente à base d'automates. L'équipe pédagogique a retenu ce modèle parce qu'il touche la plupart des difficultés majeures de la programmation concurrente tout en restant relativement simple.

Objectifs pédagogiques

Cet axe de l'UV1 de la majeure informatique doit permettre à l'élève de :

— Être capable de modéliser des systèmes concurrents simples en utilisant la notion d'automate (LTS)

— Être capable d'identifier quelques pièges classiques de la programmation concurrente en général (interblocage, état puits, ...)

— Passer d'un modèle LTS à un programme Java et vice-versa

— Assimiler quelques briques fondamentales de la théorie des systèmes distribués

Pré-requis

Un minimum de maitrise du langage Java est nécessaire.

Contenu détaillé

Le module se divise grossièrement en :

— un cours, deux PC et deux TP qui donnent les bases fondamentales de la modélisation des systèmes concurrents en FSP et LTS

— un cours, une PC et deux TP qui permettent de découvrir la notion de propriétés et le passage d'un modèle au code

— deux PC qui visent à introduire la modélisation des systèmes distribués

Un BE de conception FSP fait en salle de TP tient lieu de Contrôle Continu.

Références

[1] Bruce Eckel : *Thinking in Java*. Prentice Hall, 4e édition, 2006. ISBN : 0-13-187248-6.

La bible… Relativement difficile à lire car très exhaustif et aborde les concepts en détail. Il est conseillé de l'utiliser plutôt comme un livre de reférence pour répondre à des questions de Java avancées. L'édition précédente peut être téléchargée depuis le site du livre : `http://mindview.net/Books/TIJ4`.

[2] Jeff Kramer et Jeff Magee : *Concurrency : State Models Java Programs.* Worldwide Series in Computer Science. John Wiley Sons, 2e édition, Avril 2006. ISBN : 978-0470093559.

Une reférence pédagogique majeure beaucoup plus complète que ce qui sera abordé en cours.

[3] Douglas Lea : *Concurrent Programming in Java : Design Principles and Pattern.* The Java Series. Addison Wesley Professional, 2e édition, 1999. ISBN : 0201310090.

[4] Ajay D. Kshemkalyani et Mukesh Singhal : *Distributed Computing : Principles, Algorithms, and Systems.* Cambridge University Press, 1re édition, mai 2008.

Première partie

La modélisation des systèmes concurrents

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

1 Introduction
2 Modéliser les processus
3 FSP, une syntaxe algébrique
4 Composition de processus

Avancement

1 Introduction
2 Modéliser les processus
3 FSP, une syntaxe algébrique
4 Composition de processus

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

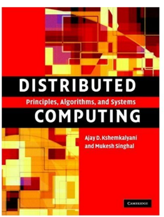
O1 Découvrir et pratiquer la modélisation formelle
- a construire des modèles FSP
- b passer d'un modèle FSP à son LTS et vice et versa
- c composer des LTS
- d savoir décomposer un système à modéliser puis construire son modèle par composition

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

1 Introduction

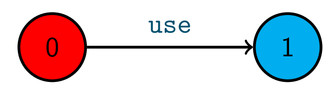
2 Modéliser les processus

3 FSP, une syntaxe algébrique

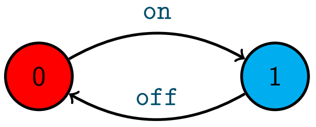
4 Composition de processus

Processus et leur comportement

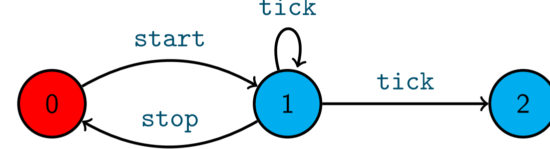
- Suite d'actions avec choix et boucles ⇒ automate
- LTS (*Labelled Transition System*)
- Équipement à usage unique



- Une lampe

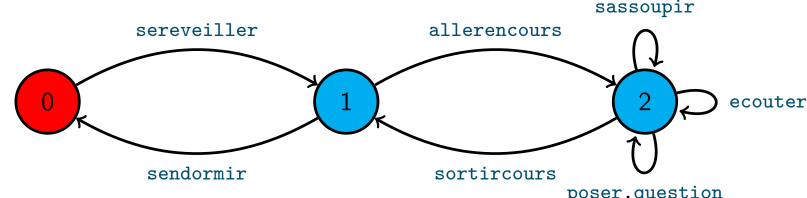


- Une horloge ($\mathcal{H}$)



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
  - ⇒ 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 : `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 $\mathcal{H}$
    - $\texttt{fireable}(0) = ?$ $\texttt{fireable}(1) = ?$ $\texttt{fireable}(2) = ?$
- Un état bloqué : aucune action n'est déclenchable
  - $\texttt{blocked}(\texttt{s}) = (\texttt{fireable}(\texttt{s}) = \{\})$
  - pour $\mathcal{H}$
    - $\texttt{blocked}(0) = ?$ $\texttt{blocked}(1) = ?$ $\texttt{blocked}(2) = ?$

Quelques notions utiles II

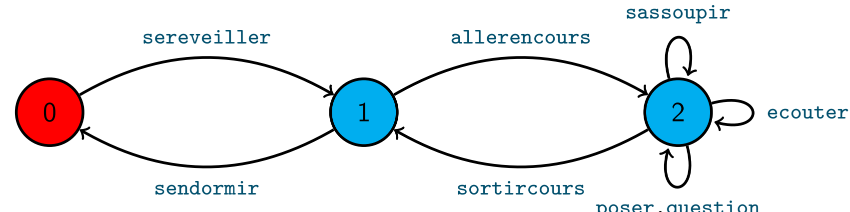
- Les états suivants : `next` : $S \to \mathcal{P}(S)$
  - $\texttt{next}(s_1) = \left\{s_2 \in S \mid \exists a \in A, s_1 \xrightarrow{a} s_2 \in T\right\}$
  - pour $\mathcal{H}$
    - $\texttt{next}(0) = ?$ $\texttt{next}(1) = ?$ $\texttt{next}(2) = ?$
  - $\texttt{blocked}(\texttt{s}) = (\texttt{next}(\texttt{s})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

Quelle est la sémantique d'un LTS ?

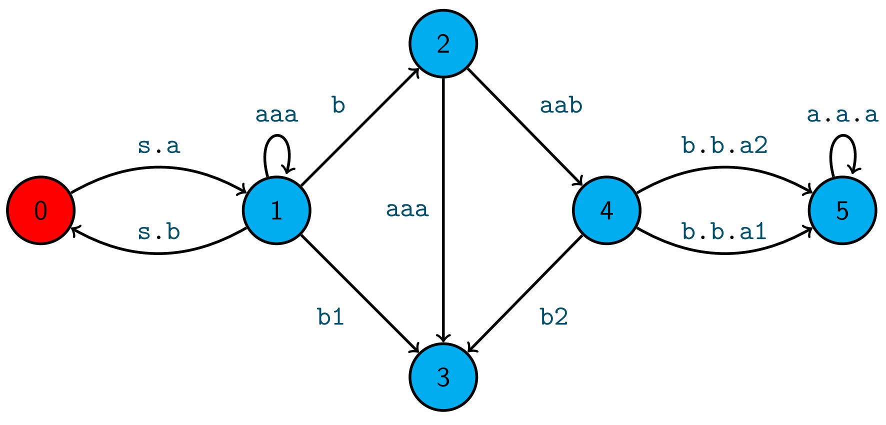
- Comportement d'un LTS = ensemble de ses traces
- Une trace de (S, A, T, i) : suite d'actions $\langle a_n \rangle_{n \in N}$ telle que
  - $N = \begin{cases} \mathbb{N} & \textit{(trace infinie)} \\ [\![0, k]\!] \text{ pour } k \in \mathbb{N} & \textit{(trace finie)} \end{cases}$
  - il existe une 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 : ⟨`start`⟩, ⟨`start`, `stop`⟩, ⟨`start`, `tick`⟩ ou ⟨`start`, `tick`, ..., `tick`, `stop`⟩
  - pas des traces : ⟨`tick`⟩, ⟨`start`, `to`⟩, ⟨`stop`, `start`, `tick`⟩

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

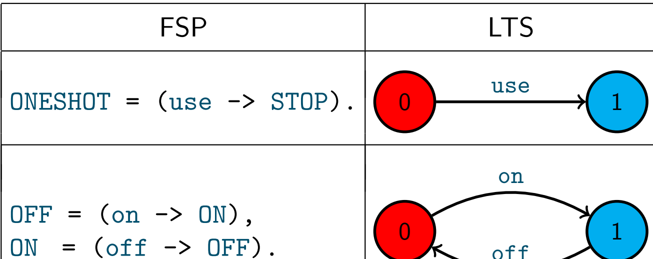
1. Introduction
2. Modéliser les processus
3. FSP, une syntaxe algébrique
4. Composition de processus

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

⇒ 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).`	0 —use→ 1
`OFF = (on -> ON),` `ON = (off -> OFF).`	0 —on→ 1, 1 —off→ 0

- 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).`
	0 1 2 start stop tick tick

`H = (start -> ON | stop -> ERROR),`
`ON = ( stop -> H | tick -> ON`
`| start -> K).`
	-1 0 1 stop start stop tick start

- 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`

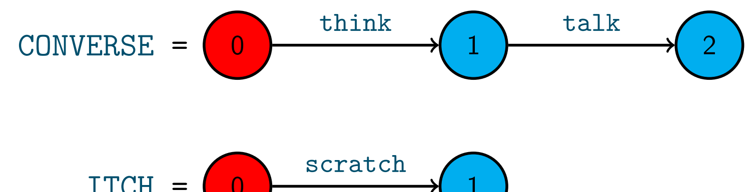
Avancement

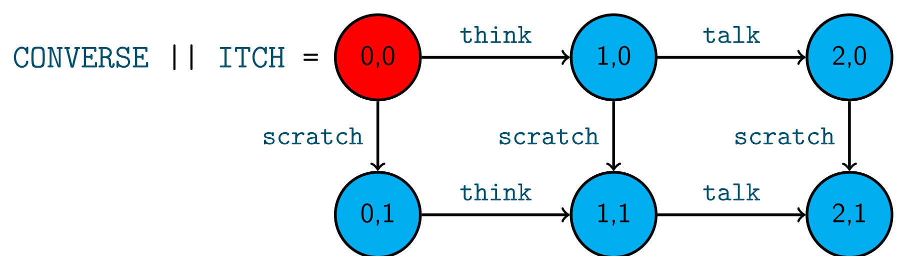
1 Introduction
2 Modéliser les processus
3 FSP, une syntaxe algébrique
4 Composition de processus

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)
  - $S = S_1 \times S_2$ $\quad A = A_1 \cup A_2$ $\quad i = (i_1, i_2)$
  - $T = \{(s_1, s_2) \xrightarrow{a} (s'_1, s'_2) \mid s_1 \xrightarrow{a} s'_1 \wedge s_2 \xrightarrow{a} s'_2 \wedge a \in A_1 \cap A_2 \vee s_1 = s'_1 \wedge s_2 \xrightarrow{a} s'_2 \wedge a \notin A_1 \vee s_2 = s'_2 \wedge s_1 \xrightarrow{a} s'_1 \wedge a \notin A_2\}$
  - en général, on élimine les états non atteignables

Un premier exemple





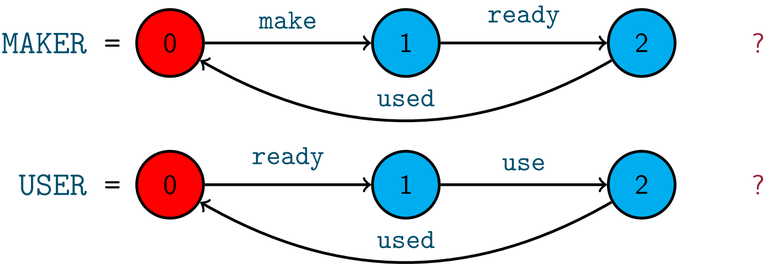
Un second exemple, action partagée

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

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

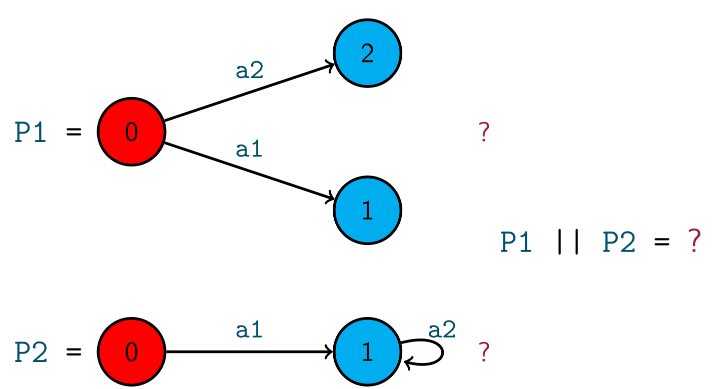
MAKER || USER =

Un exercice, la poignée de main



MAKER || USER = ?

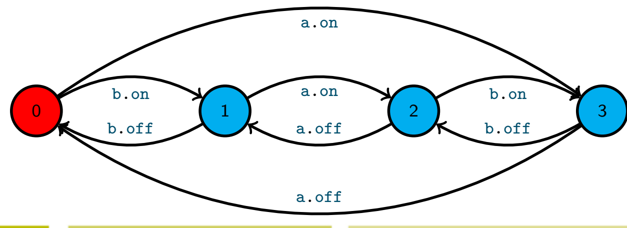
Composition et choix



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

La notion d'instance

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

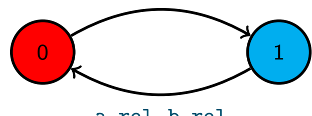


Renommage

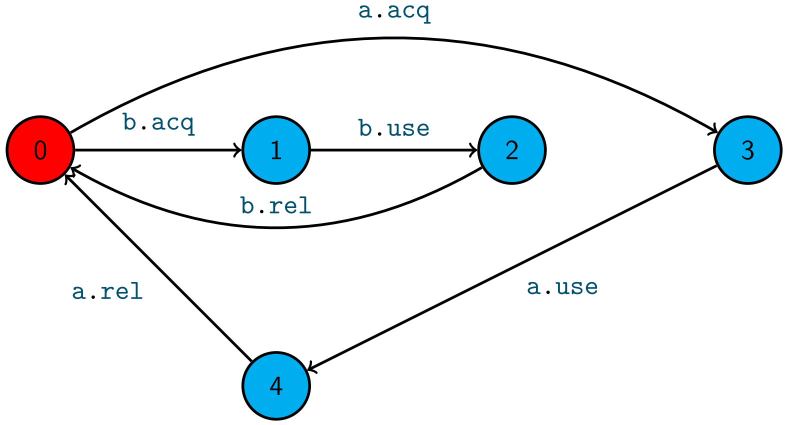
- On peut renommer une (préfixe d') action
  - `P/{remplaçant,remplacé}`
  - `(call -> P)/{request/call}` ≡ `(request -> P)`
  - `(call.a -> call.b -> P)/{request/call}` ≡ `(request.a -> request.b -> P)`
  - `(call.a -> a -> P)/{b/a}` ≡ `(call.a -> b -> P)`
  - `(call.a -> call.b -> P)/{r/call,n/call.a}` ≡ `(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` ?
- 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

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

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

Notes de cours sur le langage FSP

Contenu

Ce document contient une présentation du langage de modélisation FSP. Il couvre les parties essentielles du langage qui sont utilisées durant le module. Vous pouvez également consulter le livre de Jeff Magee et Jeff Kramer référencé dans les transparents de cours.

FSP signifie en anglais *Finite State Processes*, soit processus à état fini. C'est le nom du langage de modélisation de systèmes concurrents que nous allons étudier et utiliser durant la première partie de ce module. Ce langage est à la base d'un outil de vérification LTSA : `http://www.doc.ic.ac.uk/~jnm/book/ltsa/LTSA.html`. Cet outil repose sur la notion de transition étiquetée (LTS – *Labelled Transition System*). À une expression dans le langage FSP peut être associée un automate étendu avec des étiquettes.

Un modèle aura donc deux représentations :

— une vue algébrique : une expression ou *programme* FSP,

— une vue graphique : un automate LTS.

Nous renvoyons au contenu des transparents du cours pour la notion de LTS.

1 Le cœur de FSP

Un terme FSP repose sur trois concepts :

— les `PROCESSUS` : nom en majuscule,

— les `ETATS` : nom en majuscule,

— les `actions` (parfois appelées étiquettes) : suite de noms en minuscule séparés par des points.

Définitions

Un *programme* FSP consiste en la définition d'un ensemble de processus et de compositions. Ces deux formes de définition se font par `=` et se termine par `.`. La définition d'un processus définit le comportement de ce processus sous la forme d'un automate. La définition d'un processus composite se fait par composition de processus dont le comportement a été précédemment défini. Pour être un processus composite, l'identifiant doit être précédé du symbole `||`, elle est présentée dans la sous-section « *La concurrence* ».

La définition du comportement d'un processus sous forme d'automate se fait par un ensemble d'équations mutuellement récursives. Chaque équation définit un état par une expression qui peut utiliser n'importe quel autre état de la définition en cours. Attention, ces états sont locaux et privés et ne sont donc pas visibles pour les autres processus. La définition de chaque état se fait aussi par `=` et est séparée de la suivante par `,`. Ainsi, la forme générale de définition d'un processus sera :

```
NOMDUPROCESSUS = ··· ,
NOMETAT1 = ··· ,
...
NOMETATN = ··· .
```

Pour les processus, la première ligne de la définition contiendra la définition directe de l'état initial. Pour ceux qui sont un peu plus complexes, on privilégiera une définition explicite de tous les états

```
SWITCH = OFF,
OFF = (on -> ON),
ON = (off -> OFF).
```

Figure 1 – Un interrupteur : expression FSP et automate associé

et la première ligne de la définition se contentera de définir l'état initial comme pour la définition de la figure 1 où `OFF` est l'état initial.

Trois états bloqués sont prédéfinis et peuvent être utilisés dans n'importe quelle définition : `END`, `STOP` et `ERROR`. Ce sont des états sans transition qui modélisent un système arrêté. Le premier `END` modélise un système qui s'est bien terminé. Le deuxième `STOP`, un système qui est en état bloqué (*deadlock*). Enfin, le troisième correspond à une erreur pour les outils de vérification de LTSA. Il aura l'étiquette -1 dans les LTS. Toute utilisation d'un nom d'état non connu sera considéré comme correspondant à l'état `ERROR`.

Enfin, un processus peut avoir des paramètres. Ces paramètres doivent avoir une valeur par défaut. Ils peuvent alors être utilisés dans le corps du processus (nous verrons plus loin l'utilité). Ainsi, `NOM(V1=8,V2=1) = ... .` définit un processus qui a les paramètres `V1` et `V2` de valeur par défaut 8 et 2 respectivement. Lorsqu'un processus paramétré est utilisé, on peut donner des valeurs à ces paramètres. Si l'on en donne pas, les paramètres prennent les valeurs par défaut. Ainsi, `NOM` considère `V1` valant 8 et `V2` valant 2 et `NOM(2,1)` considère `V1` valant 2 et `V2` valant 1. Attention, il faut fournir ou bien aucun paramètre ou bien tous les paramètres necéssaires.

Actions et Choix

Les processus peuvent réaliser des actions suivant la syntaxe suivante : `action -> PROCESSUS`. Ainsi, le processus `SWITCH` de la figure 1 (déjà vu en cours) modélise un interrupteur qui a deux états `OFF` et `ON`. Son état initial est `OFF` et il pourra exécuter infiniement et successivement les actions `on` et `off`.

Une trace (d'exécution) d'un processus consiste en un enchaînement d'actions qu'il peut réaliser durant son exécution. Le processus `SWITCH` peut suivre la trace suivante :

```
on -> off -> on -> off -> on -> off -> ...
```

Les définitions mutuellement récursives peuvent être dépliées en substituant un état par sa définition. Ainsi, l'exemple précédent peut être simplifié par dépliage en :

```
SWITCH = (on -> off -> SWITCH).
```

L'instruction `|` permet de décrire des alternatives : le processus peut avoir plusieurs comportements possibles. Remarquons qu'un modèle FSP abstrait le choix [1]. Le processus `(x -> P | y -> Q)` commence par une action `x` **ou** une action `y`, son comportement est ensuite soit `P` (si `x` est l'action qui a été exécutée) soit `Q` (si c'est `y` qui a été exécutée). Attention, uniquement une des deux actions est exécutée. Notons enfin, que le choix peut être *non deterministe* si un processus peut depuis un état transiter vers deux états par la même action. Par exemple, `(x -> P | x -> Q)` est non deterministe si `P` et `Q` sont différents. On parle de non determinisme car rien ne permet de determiner si le processus va aller vers un état `P` ou un état `Q`.

1. C'est-à-dire que l'on ne peut pas savoir quelle branche sera choisie. Dans le cadre d'un modèle cela permet de ne pas se poser la question du choix mais juste de spécifier les différentes évolutions possibles.

La concurrence

Jusqu'à présent nous avons modélisé des systèmes à un seul fil de calcul, nous allons donc ajouter la concurrence.

La composition

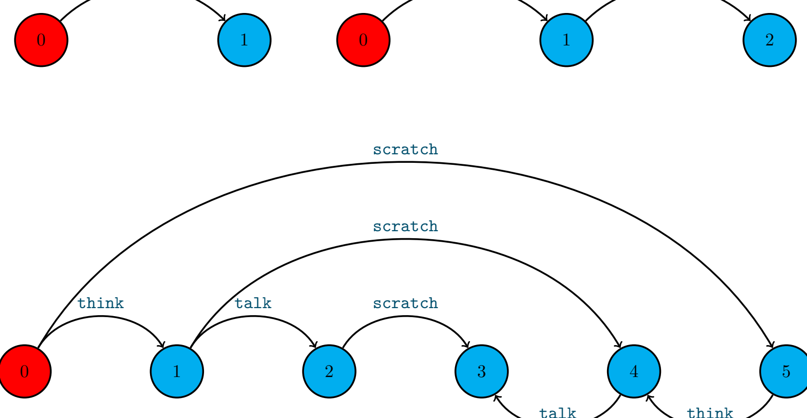
Pour indiquer que deux processus s'exécutent de manière concurrente FSP offre l'opérateur parallèle `||` (qui est commutatif, associatif, dont `END` et `STOP` sont des éléments neutres [2] et `ERROR` est un élément absorbant [3]). La définition d'un processus composite repose sur la mise en parallèle de plusieurs sous-processus et se note `||COMPOSITE = (P || Q).`. La sémantique de cet opérateur correspond à autoriser tous les entrelacements possibles d'exécution entre les sous-processus. Ainsi, si nous disposons de deux processus `ITCH` (démangeaison) et `CONVERSE` :

```
ITCH = (scratch -> STOP).
CONVERSE = (think -> talk -> STOP).
||CONVERSE_ITCH = (ITCH || CONVERSE).
```

Les trois traces possibles sont :

1. `think -> talk -> scratch`
2. `think -> scratch -> talk`
3. `scratch -> think -> talk`

Et leurs automates respectifs sont :



Dans l'automate composé les états sont obtenus par couplage de la manière suivante :
- 0 correspond à 0 dans `ITCH` et 0 dans `CONVERSE`
- 1 correspond à (0,1)
- 2 correspond à (0,2)
- 3 correspond à (1,2)
- 4 correspond à (1,1)
- 5 correspond à (1,0)

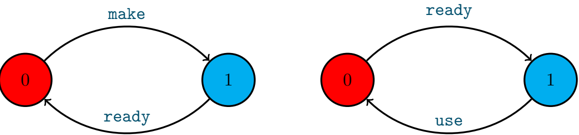
2. `P || END = P || STOP = P`
3. `P || ERROR = ERROR`

La synchronisation

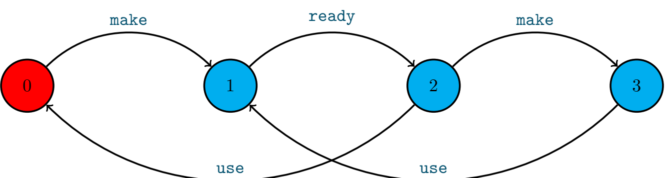
Afin de modéliser des systèmes concurrents, il faut également offrir la possibilité de synchroniser deux processus. Pour cela, en FSP, les processus à synchroniser partagent des actions. La sémantique de ce partage est la suivante : une action partagée entre plusieurs processus ne peut avoir lieu que si tous les processus en cours d'exécution la contenant sont capables de l'exécuter (simultanément).

Soit les deux processus `MAKER` et `USER` :

```
MAKER = (make -> ready -> MAKER).
USER = (ready -> use -> USER).
```



Leur composition parallèle : `||MAKER_USER = (MAKER || USER).` conduit à l'automate :



La sémantique de la composition

La sémantique formelle de la composition peut être donnée par la définition du LTS d'une composition de deux processus P_1 et P_2 de LTS (S_j, A_j, T_j, i_j) est (S, A, T, i) avec :
- $S = S_1 \times S_2$
- $A = A_1 \cup A_2$
- $T = \{(s_1, s_2) \xrightarrow{a} (s'_1, s'_2) \mid (s_1 = s'_1 \land s_2 \xrightarrow{a} s'_2 \land a \notin A_1) \lor (s_2 = s'_2 \land s_1 \xrightarrow{a} s'_1 \land a \notin A_2) \lor (s_1 \xrightarrow{a} s'_1 \land s_2 \xrightarrow{a} s'_2 \land a \in A_1 \cap A_2)\}$
- $i = (i_1, i_2)$

Il faut noter que cette sémantique implique que lors de sa composition avec d'autres processus, le choix d'un processus peut être dirigé. Ainsi, `P1 = (a1 -> a3 -> STOP | a2 -> a4 -> STOP).` composé avec `P2 = (a1 -> a2 -> STOP).` inhibe le choix `a2` depuis l'état initial puisque `P2` le contient mais ne l'accepte pas dans son état initial. Plus généralement, `P1 || P2` est équivalent à `a1 -> a3 -> STOP`. Plus loin, on verra une autre façon de guider le choix avec un processus `P2` plus simple.

Il existe deux algorithmes pour calculer le composé de deux processus :

- Soit de proche en proche depuis l'état initial en determinant les actions déclenchables,
- Soit en calculant l'ensemble de tous les états possibles et celui de toutes les transitions possibles ; il faut alors ensuite éliminer les états non accessible et les transitions inutiles.

Instances

Lors de la mise en concurrence de processus, il est possible de nommer un processus de façon à pouvoir avoir plusieurs instances du même processus (plusieurs éléments qui ont le même comportement que le processus). Par exemple, on peut mettre en parallèle deux interrupteurs :

```
||TWO_SWITCH = (a:SWITCH || b:SWITCH).
```

Les actions d'une instance sont alors préfixées par son nom. Ainsi, les actions de `a:SWITCH` sont `a.on` et `a.off`.

Les instances peuvent être mises dans un tableau, il y a alors deux syntaxes possibles :

```
||SWITCHES(N=3) = (forall[i:1..N] s[i]:SWITCH).
||SWITCHES(N=3) = (s[i:1..N]:SWITCH).
```

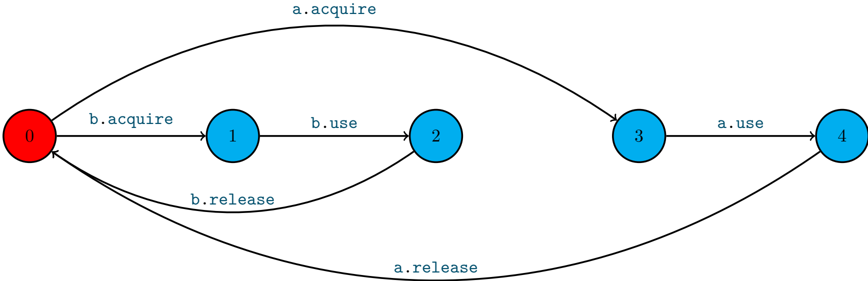
La différence entre les deux écritures est que la première permet de partager un indice entre plusieurs expression (par exemple `forall[i:1..N] (s[i]:SWITCH || t[i]:TOTO)`).

On peut ajouter un ensemble de préfixes sous la forme d'un ensemble `{a,b}` (resp. d'un tableau `a[i:1..N]`) à toutes les actions d'un processus `P` par `{a,b}::P`. Ainsi, chaque transition `n -> X` de `P` est remplacée par `{a,b}.n -> X` (resp. `a[i:1..N].n -> X`).

Par exemple, deux utilisateurs partageant une ressource peuvent être modélisés par :

```
RESSOURCE = (acquire -> release -> RESSOURCE).
USER = (acquire -> use -> release -> USER).
||RESSOURCE_SHARE = (a:USER || b:USER || {a,b}::RESSOURCE).
```

qui conduit à l'automate suivant :



Renommage

La synchronisation de deux processus se fait en partageant une action. Or, ces processus peuvent avoir été développés par ailleurs sans penser à cette future synchronisation. Dans ce cas, il est possible de spécifier que lors de leur composition deux actions de noms différents sont considérées comme identiques (et sont donc synchronisables). Ainsi :

```
CLIENT = (call -> wait -> continue -> CLIENT).
SERVER = (request -> service -> reply -> SERVER).
||CLIENT_SERVER = (CLIENT || SERVER) / {call/request, reply/wait}.
```

synchronise le client et le serveur sur les deux couples d'actions `call/request` et `reply/wait`.

Plus généralement, on peut renommer uniquement un préfixe d'action et réaliser des renommages multiples :

- `(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).`
- `P/{{a,b}/{x,y}}` ≡ `P/{a/x, a/y, b/x, b/y}`

2 Du FSP un peu plus complexe

Le langage FSP présenté jusqu'à présent ne permet pas la construction de processus plus sophistiqués et donc plus complexes. L'objectif de cette section est de vous présenter les extensions qui permettent de produire des modèles plus sophistiqués.

Indexation d'état et garde

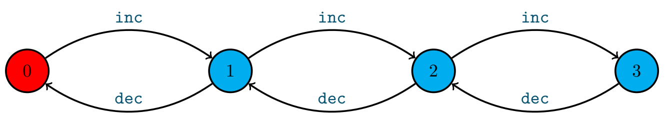
Il est possible d'associer des informations à un état. Cela se fait en utilisant la notion de variable d'état. En FSP, les variables d'état sont des entiers et il faut définir l'intervalle borné dans laquelle chacune d'elle varie. Ces bornes permettent une exploration des différentes valeurs possibles par l'outil de vérification[4]. Ainsi, `COUNT[i:0..N] =` *exp* définit un état `COUNT` ayant une variable d'état `i` (qui peut être utilisée dans *exp*). Un tel état est considéré par l'outil comme `N`+1 états `COUNT[0] =` exp_0, …, `COUNT[N] =` exp_N où exp_v correspond à l'expression *exp* dans laquelle on remplace toutes les occurrences de `i` par la valeur v.

Toute référence à l'un de ces états doit préciser la valeur de la variable d'état. Ainsi, si l'on veut transiter par l'action `x` vers `COUNT[5]`, il faut utiliser `x -> COUNT[5]`. On peut également utiliser une variable d'état définie par l'état courant lors d'une transition ainsi, dans un état ayant une variable d'état `j`, on peut faire `x -> COUNT[j]`.

Les variables d'état peuvent aussi être utilisées dans des gardes permettant de conditionner l'exécution d'action. La syntaxe pour une action gardée est `(`**when** *cond* `x -> P)` et signifie que si la condition *cond* est vérifiée l'action `x` est possible sinon `x` ne peut pas s'exécuter. La syntaxe des gardes correspond à la syntaxe des expressions Java. Ainsi, on peut définir le processus suivant :

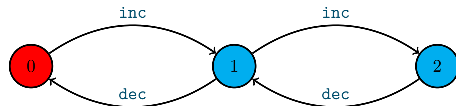
```
COUNT(N=3) = COUNT[0],
COUNT[i:0..N] = (when (i < N) inc -> COUNT[i+1]
               | when (i > 0) dec -> COUNT[i-1]).
```

Alors, le processus `COUNT` correspond à l'automate suivant (`N` vaut 3, il y a donc 4 états) :



4. En effet, celui-ci pratique une exploration dite exhaustive et doit donc connaître de manière exacte l'automate.

Alors que `COUNT(2)` correspond à l'automate suivant :



Pour avoir plusieurs variables d'état, il suffit d'enchaîner les crochets, ainsi `P[x:I1][y:I2]` est un état qui définit deux variables d'état qui sont nommés `x` et `y` et évoluent respectivement dans les intervalles `I1` et `I2`.

Extension d'alphabet

La construction suivante de FSP que nous allons aborder est l'extension d'alphabet. C'est une construction un peu délicate puisqu'elle n'a pas d'effet réellement visible sur un processus mais son sens n'apparaît que lors de la composition avec d'autres processus.

L'intérêt de l'extension d'alphabet est de permettre d'interdire lors de la composition l'exécution de certaines actions. Ainsi, le processus `P2 = (a1 -> STOP) + {a2}.` est un processus d'alphabet {`a1`, `a2`} dans lequel l'action `a2` ne peut jamais être exécutée. Ainsi, lors d'une composition, il interdira toute exécution de cette action.

Par exemple, composé avec `P1 = (a1 -> a3 -> STOP | a2 -> a4 -> STOP).`, l'extension d'alphabet va interdire l'exécution du second choix et conduit donc à `a1 -> a3 -> STOP`.

Actions indexées

Les actions peuvent être indexées par des entiers. Cette syntaxe fournit des raccourcis syntaxiques intéressants comme par exemple, la possibilité de décrire un choix non déterministe :

```
BUFFER = (in[i:0..2] -> out[i] -> BUFFER).
```

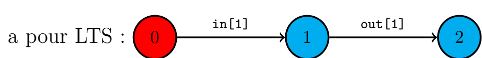
Ce processus est équivalent à :

```
BUFFER = (in[0] -> out[0] -> BUFFER
        | in[1] -> out[1] -> BUFFER
        | in[2] -> out[2] -> BUFFER).
```

où le nom des actions peuvent maintenant comporter des index ainsi les actions `in[0]` et `in[1]` sont des actions différentes.

Les index d'action peuvent être utilisés pour simuler la synchronisation d'une donnée entre deux processus. Pour cela, il suffit de combiner cette construction avec l'extension d'alphabet. Par exemple :

```
BUFFER = (in[i:0..2] -> out[i] -> BUFFER).
P =(in[1] -> STOP) + {in[i:0..2]}.
||C = (BUFFER || P).
```

a pour LTS : 

Voici un exemple, un peu plus complet d'action indexée et d'extension d'alphabet. La spécification ci-dessous conduit à l'automate de la figure 2.

```
A(N=3) = (a[N] -> synchro[N] -> A).
B = (synchro[n:1..3] -> b[n] -> B).
||ESSAI=(B||A(2)).
```

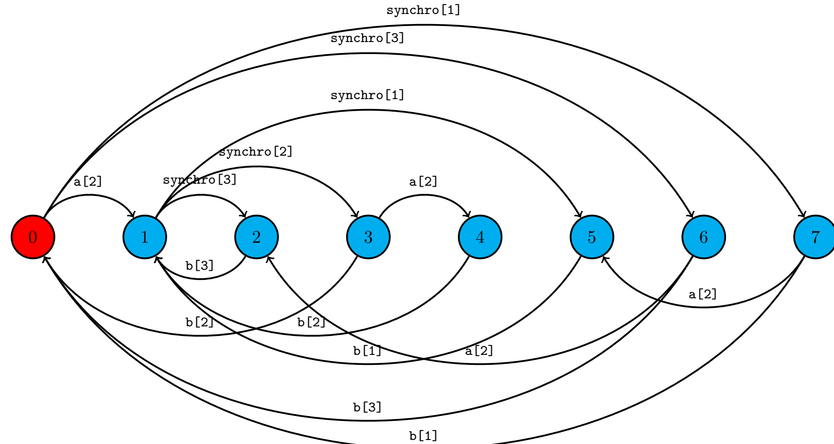



FIGURE 2 – La composition sans extension d'alphabet

Les actions `synchro[1]` et `synchro[3]` ne se produisent pas dans `A(2)`, elles peuvent donc se produire à tout moment dans sa composition avec `B`. Cela peut être un problème, si le but de la spécification était d'indiquer que lors de la synchronisation, `B` récupérait la valeur de synchro de `A`. Dans ce cas, on ne veut pas des actions `synchro[1]` et `synchro[3]`. Pour cela, on utilise l'extension d'alphabet `+` qui permet de les bloquer lors de la composition.

Par exemple, le précédent processus `A` peut être étendu pour indiquer qu'il peut produire des actions `synchro` avec les paramètres de 1 à 3 :

```
A(N=3) = (a[N] -> synchro[N] -> A) + {synchro[1..3]}.
```

L'automate de la composition est alors modifié (voir figure 3) et les actions de `synchro` avec les paramètres 1 et 3 ne peuvent plus se produire. Et donc, `B` *reçoit* bien 2 (l'action `b[2]` se produit).

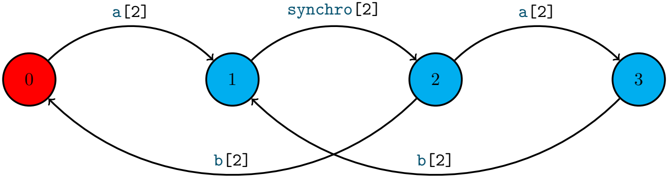


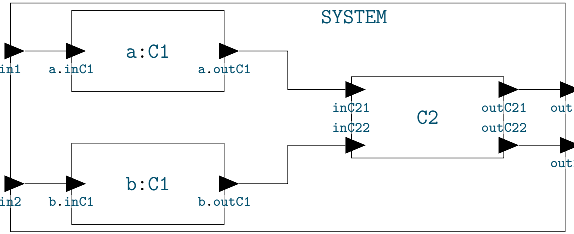
FIGURE 3 – La composition avec extension d'alphabet

3 Une méthode de conception

Lors de la conception du modèle d'un processus complexe, il convient de suivre une méthode pour garantir la production d'un modèle de qualité.

1. Identifier les événements ou actions d'entrée / sortie du système. Ici, le système est vu comme une boîte noire (on ignore l'intérieur) et il s'agit de lister toutes les interactions possibles avec le système. Le résultat est une liste d'action d'entrée et une liste d'action de sortie. On peut aller jusqu'à produire un diagramme de bloc de la forme suivante :

2. Décomposer le système en un ensemble de sous-systèmes. Chaque sous-système est modélisé sous la forme d'une instance de processus FSP. Chacune de ces instances correspond à un *type* de sous-système qui sera réalisé par un processus FSP.
3. Pour chaque type de sous-système :
   — s'il est simple, décrire en FSP son comportement (enchaînement des actions) de manière incrémentale. Ici, il s'agit donc de produire un processus (simple) FSP.
   — s'il est complexe, réappliquer la décomposition (étape 1 à 4). Au résultat, on obtient un processus composite.
4. Produire la composition avec les instanciations et renommages nécessaires. Pour aider à la production de cette composition, on peut construire un diagramme de la forme suivante :



qui correspond à la composition :

```
||SYSTEM = (a:C1 || b:C1 || C2)/{
    in1/a.inC1, in2/b.inC1,
    inC21/a.outC1, inC22/b.outC1,
    out1/outC21, out2/outC22}
).
```

Il serait possible de cacher les actions internes (ici `inC21` et `inC22`) avec les constructions FSP `\` et `@`[5].

Durant l'application de cette méthode, il convient de commencer à valider au plus tôt le modèle en observant son automate lorsque c'est possible et en l'animant pour se convaincre de sa validité.

Enfin, il convient de réaliser les étapes de 1 à 4 sur des versions simplifiées et ajouter des fonctionnalités au fur et à mesure.

5. Cacher en FSP consiste à transformer un ensemble d'actions en l'action silencieuse notée τ. Ces actions ne peuvent alors plus être observées de l'extérieur du processus même si elles ont lieu. `P\S` cache les actions de S alors que `P@S` cache toutes les actions de `P` ne figurant pas dans `S`.

PC1-2/TP1-2 – Système de transitions étiquetées et pratique de FSP

Objectifs

L'objectif de ces quatre séances est de découvrir par la pratique les bases d'une modélisation formelle de la concurrence qu'est FSP. Nous allons :
- apprendre à modéliser un système simple,
- travailler la syntaxe de base de FSP,
- approfondir la correspondance entre les automates et la syntaxe FSP,
- composer des systèmes simples pour décrire des systèmes plus complexes.

Tout au long des séances, vous êtes invités à utiliser les notes de cours sur FSP. Les deux premières sections utilisent les notions présentées dans le premier cours et résumées dans la première section des notes de cours. La troisième section nécessite la lecture de la deuxième section des notes de cours.

Il est fortement conseillé de faire tous les exercices même s'ils n'ont pas pu être abordés en séance par manque de temps. En cas de problème, vous pouvez poser des questions sur le forum, contacter les enseignants, utiliser le monitorat informatique ou consulter la correction sur moodle.

1 La découverte de FSP (PC1)

Exercice 1 (*Feu tricolore*)

Modéliser le comportement d'un feu tricolore en fournissant :
- un programme FSP,
- l'automate associé,
- une trace.

Exercice 2 (*Machine à café*)

Modélisez le comportement d'une machine à café qui a deux boutons : un qui permet d'obtenir un café et l'autre un thé.

Exercice 3 (*Une bombe simple*)

Modéliser le comportement d'une bombe qui explose après 3 tics ou qui peut être désamorcée.

Exercice 4 (*Modélisation d'un processus de production*)

Pour produire un produit Y, trois robots sont utilisés (R1, R2 et R). Ils génèrent respectivement les composants Y1, Y2 et le produit final Y. La production suit les règles suivantes :
- R1 génère un composant Y1 et il attend que le composant soit utilisé avant d'en générer un autre,
- R2 génère le composant Y2,

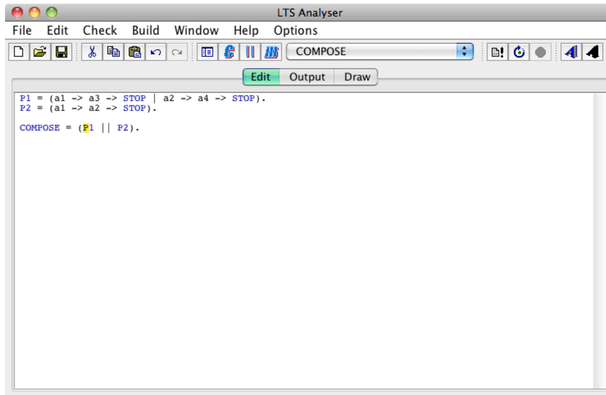


Figure 1 – L'interface graphique de l'outil LTSA.

- R2 a la possibilité de produire deux fois le composant avant d'attendre l'utilisation de sa production,
- R prend les composants Y1 et Y2 et les assemble pour générer le produit Y.

Inspiré de Michel Riveill.

Exercice 5 (*Sémantique de FSP*)

Donner le LTS des constructions FSP suivantes en fonction du LTS de leurs sous-éléments :
- `STOP`,
- `ERROR`,
- `a -> P`
- et `a1 -> P1 | a2 -> P2`.

2 Prise en main de l'outil LTSA (TP1)

LTSA est un outil de vérification de modèles développé à l'Imperial College de Londres, disponible de plusieurs manières :
- en tapant `ltsa.sh` dans un terminal,
- en le récupérant : `http://www.doc.ic.ac.uk/~jnm/book/ltsa/download.html` ou dans l'espace Moodle de ce module :
  - le fichier téléchargé est une archive zip, il faut le décompresser grâce à la commande `unzip ltsa.zip -d ./ltsa` ;
  - lancer l'archive java (extension `.jar`) grâce à la commande `java -jar ltsa.jar`.

L'interface de LTSA est représentée dans la figure 1.

Les étapes de développement d'un modèle dans LTSA sont :

1. Édition du texte dans l'onglet d'édition (accessible via le bouton Edit) pour insérer (p.ex. par copier/coller) une spécification formelle FSP.
2. Vérification syntaxique de votre spécification (bouton Parse,). Le résultat des différentes opérations peut être consulté dans l'onglet Output.

3. Compilation pour créer les automates des processus (bouton Compile,). Vous remarquerez que la compilation lance automatiquement la vérification syntaxique si vous ne l'avez pas faite.
4. Visualisation des automates générés automatiquement dans l'onglet Draw. Des boutons de redimensionnement (et) permettent de zoomer/dézoomer certaines parties.
5. Composition éventuelle à l'aide du bouton Compose (||) permettant de générer un unique LTS correspondant au produit des automates des différents processus.
6. Analyses de propriétés (par exemple Safety, Progress, Reachability) sont disponibles dans le menu Check. En cas de détection de problèmes fonctionnels dans une spécification, une trace est fournie. Dans un premier temps, nous n'utiliserons pas ces fonctionnalités.
7. Exécution du modèle avec l'action Run du menu Check, qui ouvre une fenêtre Animator. La fenêtre permet de retracer visuellement, que ce soit en exécution totale ou en pas à pas (bouton Step). Dans le deuxième cas, à chaque pas, vous sélectionnez une action à réaliser. Vous pouvez voir en instantané sur les automates des processus les actions étiquetées qui passent en rouge.

Vous allez pouvoir prendre en main cet outil au travers des exercices qui suivent.

Exercice 6 (*Radio FM*)

Une radio portable FM a trois boutons. Un bouton `on`/`off` qui allume et éteint l'appareil. Le choix des fréquences radios s'effectue au travers de 2 boutons : `scan` et `reset`. Quand on allume la radio ou que l'on appuie sur le bouton `reset`, la radio est ramenée sur la fréquence de base de la bande FM (88MHz). Quand on appuie sur le bouton `scan`, la radio recherche la prochaine fréquence d'émission en montant en fréquence jusqu'à s'ajuster sur une station ou à atteindre la fin de la bande FM. Si la radio est déjà ajustée et que l'on appuie à nouveau sur le bouton `scan`, le même processus est appliqué jusqu'à atteindre la prochaine station sur la bande FM.

▷ **Question 6.1 :**
En utilisant l'alphabet {`on`, `off`, `scan` `reset`, `ajuster`, `fin`}, modéliser la radio FM sous forme d'un processus FSP.

▷ **Question 6.2 :**
Construire un radio réveil en composant ce processus avec une horloge et visualiser le comportement résultant.

Exercice 7 (*Le télérupteur*)

Un télérupteur est un type de relais électromécanique commandé par des impulsions électriques. Il permet l'alimentation d'un circuit (par exemple l'éclairage d'un escalier) par plusieurs boutons poussoirs. Ce dispositif de commande d'éclairage remplace avantageusement le va-et-vient car il peut comporter un nombre illimité de boutons de commande.

Un télérupteur a un fonctionnement similaire à une « bascule ». Une impulsion (pression sur un bouton poussoir) déclenche le télérupteur, ce dernier ferme le circuit jusqu'à ce qu'une nouvelle impulsion ouvre à nouveau le circuit.

▷ **Question 7.1 :**
Modéliser le système télérupteur composé d'un bouton poussoir et d'un relais basculant dans les positions `on`/`off`.

▷ **Question 7.2 :**
Connecter plusieurs boutons poussoirs au même télérupteur.

3 Vers des plus gros modèles (PC2)

Exercice 8 (*La bombe*)

Modéliser le comportement d'une bombe qui explose après N tics ou qui peut être désamorcée.

Exercice 9 (*Additionneur*)

▷ **Question 9.1 :**
Modéliser le comportement d'un processus additionneur.

▷ **Question 9.2 :**
Utiliser cet additionneur pour calculer la somme de 1 et 2.

Exercice 10 (*Retour sur la machine à café*)

Maintenant, la machine à café se voit enrichie des caractéristiques suivantes :
- le prix d'une boisson est 30 centimes,
- les pièces acceptées sont 10, 20 et 50 centimes,
- la machine rend la monnaie.

▷ **Modéliser un tel système en FSP.**

Exercice 11 (*Des utilisateurs pour la machine à café*)

Ajouter quelques utilisateurs à la machine à café. Par exemple :
- Toto qui n'a que des pièces de 50
- Tutu qui utilise uniquement des pièces de 10
- Tata qui utilise alternativement des pièces de 10 et de 20

4 Mise en pratique (TP2)

Exercice 12 (*La centrale de réservation*)

Modéliser une centrale de réservation de siège de spectacle. Cette centrale est reliée à un ensemble de terminaux de réservation. Sur un de ces terminaux, un employé peut consulter l'état d'un siège qui peut être libre ou occupé. Un client peut réserver un siège qui est libre, l'employé entre alors le numéro du siège et imprime un ticket. Un siège ne peut être réservé deux fois et le système doit garantir cette propriété. Inspiré de [M&K] exercice 4.3.

Deuxième partie

Les propriétés et le passage au code

Propriétés et passage au code

F. Dagnat
Majeure Informatique – INF447 – C2
1er semestre 2015

Liens modèle code

Modéliser les processus en utilisant FSP/LTS

Après avoir validé un modèle, le réaliser

Implémenter avec des *threads* en Java

Reconstruire un modèle pour valider un code

Problèmes

- Sûreté (*safety*)
  - Il ne se passe jamais rien de mauvais
  - On atteint jamais un mauvais état
  - Par exemple
    - pour un comportement séquentiel : l'état final est bon
    - non respect d'une exclusion mutuelle
    - absence d'interblocage (*deadlock*)
- Vivacité (*liveness*)
  - Il finit par se passer quelque chose de bien
  - On finit par arriver dans un bon état
  - Par exemple
    - pour un comportement séquentiel : la terminaison
    - de famine (on finit par obtenir des ressources demandées)
- On spécifie la propriété (FSP) et on analyse (LTSA)

Plan

1. Du modèle au code
2. Interblocage
3. Sûreté
4. Vivacité

Avancement

1. Du modèle au code
2. Interblocage
3. Sûreté
4. Vivacité

Correspondance FSP / Java I

- Un processus FSP est une classe
  - il y a toujours un constructeur par défaut
  - chaque paramètre est un paramètre du constructeur
  - toute variable d'état devient un attribut
  - chaque action est une méthode

```
COUNTDOWN(N=3) = (
  start -> COUNTDOWN[N]),
COUNTDOWN[i:0..N] = (
  when(i>0) tick  -> COUNTDOWN[i-1]
| when(i==0) beep -> STOP
| stop            -> STOP).
```

```
public class CountDown {
  private int i = 3;
  public CountDown() {}
  public CountDown(int n) {
    if (n >= 0)
      this.i = n;
  }
  void beep() {
  void start() {
  void stop() {
  void tick() {
}
```

Correspondance FSP / Java II

- Les états d'un processus
  - sont codés sous la forme d'attributs
  - ajoute des préconditions aux actions et changement d'état

```
COUNTDOWN(N=3) = (
  start -> COUNTDOWN[N]),
COUNTDOWN[i:0..N] = (
  when(i>0) tick  ->
    COUNTDOWN[i-1]
| when(i==0) beep ->
    STOP
| stop            ->
    STOP).
```

```
public class CountDown {
  private boolean started;
  private boolean stopped;
  void beep() {
    if (this.started && this.i == 0 && !this.stopped) {
      this.stopped = true;
  void start() {
    if (!this.started) {
      this.started = true;
  void stop() {
    if (this.started && !this.stopped) {
      this.stopped = true;
  void tick() {
    if (this.started && this.i > 0 && !this.stopped) {
  }
}
```


Correspondance FSP / Java III

- Un processus FSP est
  - actif : sa classe est une tâche (`Runnable`)
    - il faut définir son comportement (méthode `run`)
  - passif : c'est un moniteur
    - ses méthodes sont synchronisées si nécessaire
  - les deux
- Pour chaque action, il faut determiner si
  - c'est une action d'entrée, elle est alors appelée et la méthode correspondante doit être publique et synchronisée (si nécessaire)
  - c'est une action de sortie, elle n'est invoquée que par le processus et la méthode peut être privée

Correspondance FSP / Java IV

```
public class CountDown implements Runnable {
  public void run() {
    while (!this.stopped) {
      try {
        Thread.sleep(5);
      } catch (InterruptedException e) {
        return;
      }
      if (this.started) {
        if (this.i > 0) {
          tick();
        } else if (this.i == 0) {
          beep();
        }
      }
    }
  }
```

```
public synchronized void start() {
public synchronized void stop() {
private void beep() {
private void tick() {
public static void main(String[] args) {
  CountDown cd = new CountDown(10);
  new Thread(cd).start();
  cd.start();
  try {
    Thread.sleep(75);
  } catch (InterruptedException e) {}
  cd.stop();
}
```

Un second exemple

- Un contrôleur pour un parking
- Il interdit l'entrée aux voitures si le parking est plein

```
CARPARK(N=4) = SPACES[N],
SPACES[i:0..N] = (
  when(i>0) arrive -> SPACES[i-1]
| when(i<N) depart -> SPACES[i+1]).
ARRIVALS  = (arrive -> ARRIVALS).
DEPARTURES = (depart -> DEPARTURES).
||CARPARK = (ARRIVALS || CARPARK(4) || DEPARTURES).
```

- contient
  - deux processus actifs `ARRIVALS` et `DEPARTURES`
  - un processus passif `CARPARK`
  - `arrive` et `depart` sont des méthodes d'entrée de `CarPark`

Les classes Arrivals et Departures

```
1  public class Arrivals implements Runnable {
2    CarPark carpark;
3    Arrivals(CarPark c) {
4      this.carpark = c;
5    }
6    public void run() {
7      while (true) {
8        this.carpark.arrive();
9      }
10   }
11 }
```

```
1  public class Departures implements Runnable {
2    CarPark carpark;
3    Departures(CarPark c) {
4      this.carpark = c;
5    }
6    public void run() {
7      while (true) {
8        this.carpark.depart();
9      }
10   }
11 }
```

La classe CarPark

```
1  public class CarPark {
2    private final int capacity;
3    private final DisplayedNumber nc;
4    public synchronized void arrive() {
5      while (this.nc.getValue() == 0)
6        try {
7          wait();
8        } catch (InterruptedException e) {}
9      this.nc.decrValue();
10     notifyAll();
11   }
12   public synchronized void depart() {
13     while (this.nc.getValue() == this.capacity)
14       try {
15         wait();
16       } catch (InterruptedException e) {}
17     this.nc.incrValue();
18     notifyAll();
19   }
20 }
```

Avancement

1 Du modèle au code

2 Interblocage

3 Sûreté

4 Vivacité

Interblocage (*deadlock*)

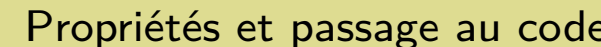
- Le système (ou une de ses parties) ne peut plus progresser car aucune action n'est possible
  - Ex : un croisement à 4 feux avec 4 voitures
- En LTS, le système atteint un état bloqué
  - `A = (north -> (south -> A | north -> STOP)).`
- LTSA fournit une trace qui mène à cet état bloqué
- Cas difficile, quand apparaît lors de la composition
  - ex du croisement ci-dessus
- Si on veut un état bloqué sans interblocage, il faut utiliser `END` (considéré comme une bonne fin)

Un exemple

- Deux processus `P` et `Q` partagent deux ressources `r1` et `r2` qu'ils peuvent prendre `get` puis relâcher `rel`

```
RESOURCE = (get -> rel -> RESOURCE).
P = (r1.get -> r2.get -> action -> r1.rel -> r2.rel -> P).
Q = (r2.get -> r1.get -> action -> r1.rel -> r2.rel -> Q).
||SYS = (p:P || q:Q ||
        {p,q}::r1:RESOURCE ||{p,q}::r2:RESOURCE).
```

- Trace vers le *deadlock* ?
- Comment l'éviter ?



Interblocage (*deadlock*)

- 4 conditions nécessaires et suffisantes d'interblocage d'un ensemble de processus [1]
  1. ils partagent des ressources en exclusion mutuelle
  2. ces ressources sont acquises de manière incrémentale
  3. une ressource ne peut être libérée que par son détenteur
  4. Un cycle (de processus) d'attente existe
- Pour éviter casser une des conditions précédentes
- Sur l'exemple, évitement par
  - Acquérir les ressources dans le même ordre
  - *Timeout*
  - ...

1. E. G. Coffman, M. Elphick, and A. Shoshani. System Deadlocks. ACM Comput. Surv. (1971). http://doi.acm.org/10.1145/356586.356588

Retour à l'exemple

- Acquérir les ressources dans le même ordre

```
RESOURCE = (get -> rel -> RESOURCE).
P = (r1.get -> r2.get -> action -> r1.rel -> r2.rel -> P).
Q = (r1.get -> r2.get -> action -> r1.rel -> r2.rel -> Q).
||SYS = (p:P || q:Q ||
        {p,q}::r1:RESOURCE ||{p,q}::r2:RESOURCE).
```

- *Timeout*

```
RESOURCE = (get -> rel -> RESOURCE).
P = (r1.get -> P1),
P1 = (r2.get -> action -> r1.rel -> r2.rel -> P
    | timeout -> r1.rel -> P).
Q = (r2.get -> r1.get -> action -> r1.rel -> r2.rel -> Q).
||SYS = (p:P || q:Q ||
        {p,q}::r1:RESOURCE ||{p,q}::r2:RESOURCE).
```

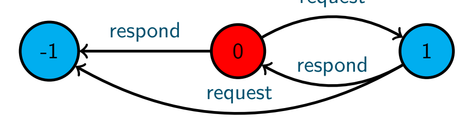
Avancement

1 Du modèle au code
2 Interblocage
3 Sûreté
4 Vivacité

Sûreté et propriétés

- Plus généralement, mauvais état = état `ERROR`
  - soit apparaît dans le modèle
  - soit on veut le faire apparaître
- Comment ajouter ces transitions vers `ERROR` ?
  - en modifiant le LTS
  - en le composant avec un LTS spécial qui spécifie les cas d'erreur
- En FSP, LTS de propriété
  - toute transition non prévue provoque une erreur
  - ex : `property P = (request -> respond -> P).`

correspond à



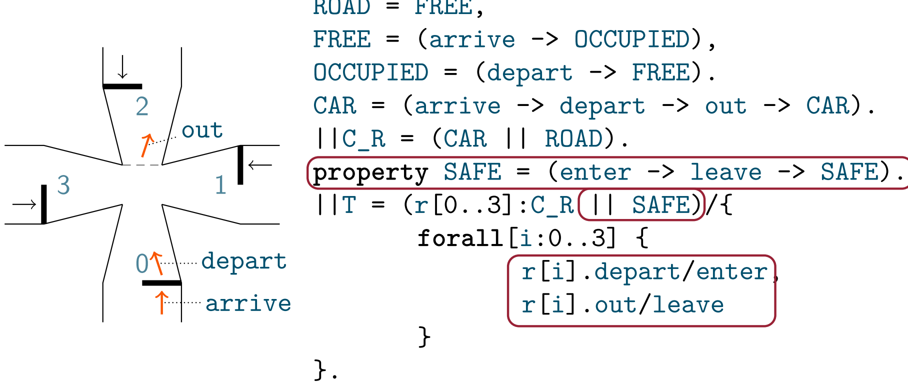
Utilisation des propriétés de sûreté

- Dans une propriété toutes les actions de l'alphabet sont déclenchables dans tous les états
  - soit parce qu'une transition existe
  - soit c'est une transition vers erreur

⇒ `P||S` transite vers `ERROR` dès que `S` exécute une action de A(`P`) au « mauvais moment »

- Ainsi, si `ERROR` n'est pas atteignable dans `P||S`, `S` est correct vis-à-vis de `P`
- Attention
  - `P` doit accepter tous les comportements corrects
  - il faut que `P` soit déterministe (pas de choix non déterministe)

Un exemple : un croisement




```
ROAD = FREE,
FREE = (arrive -> OCCUPIED),
OCCUPIED = (depart -> FREE).
CAR = (arrive -> depart -> out -> CAR).
||C_R = (CAR || ROAD).
property SAFE = (enter -> leave -> SAFE).
||T = (r[0..3]:C_R(|| SAFE))/{
        forall[i:0..3] {
                r[i].depart/enter,
                r[i].out/leave
        }
}.
```

```
Trace to property violation in SAFE: r.0.arrive
r.0.depart r.1.arrive r.1.depart
```

Autres exemples

- Spécifier qu'une action ne doit jamais arriver
  - l'ajouter à l'alphabet de la propriété
  - ex : `property NOA = STOP + {a}.`
- Exclusion mutuelle

```
range NBC = 0..2
SEMAPHORE(N=0) = SEMA[N],
SEMA[n:0..3] = (v -> SEMA[n+1] | when(n>0) p -> SEMA[n-1]).
CLIENT = (mutex.p -> enter -> exit -> mutex.v -> CLIENT).
property MUTEX = (c[i:NBC].enter -> c[i].exit -> MUTEX).
||SYS = (c[NBC]:CLIENT||c[NBC]::mutex:SEMAPHORE(1)||MUTEX).
```

  - Ok dans LTSA
  - Et si on initialise les sémaphores à 2 ?

Avancement

1. Du modèle au code
2. Interblocage
3. Sûreté
4. Vivacité

Progrès

- Vivacité = une action arrivera finalement
  - par ex, le client `c[i]` finira par obtenir le semaphore
- En général, nécessite une syntaxe pour parler de suites d'états (logique temporelle)
- Une version simplifiée, le progrès
  - `progress P = {a1,...an}`
  - sens : dans une exécution infinie, au moins une des actions `a1`, ..., `an` est exécutée un nombre infini de fois
  - à partir d'un état, on calcul les futures actions possibles
  - suppose des choix équitables
    - si un choix s'exécute un nombre infini de fois, chacune de ses transitions doit être exécutée un nombre infini de fois
- Contraire de la famine

Ensemble terminal d'états

- $S_T \subset S$ est terminal pour un LTS (S, A, T, s_0) ssi
  1. S_T est fortement connexe suivant T
     $\forall s_0, s_{n+1} \in S_T, \exists a_0, \ldots, a_n \in A, \exists s_1, \ldots, s_n \in S_T, \forall i \in [\![0, n]\!]$,
     $s_i \xrightarrow{a_{i+1}} s_{i+1} \in T$
  2. S_T est stable par T (pas de transition sortante)
     $\forall s \in S_T, s \xrightarrow{a} s' \in T \Rightarrow s' \in S_T$
- Une sorte d'ensemble *puits*
- Ensemble des ens. terminaux d'états : $\mathtt{term}(P)$
- Calcul des ens. terminaux de (S, A, T, s_0)
  - Pour chaque état, ens. des états accessibles (acces)
  - $\forall s \in S, s' \in \mathtt{acces}(s) \wedge s \in \mathtt{acces}(s') \Rightarrow \exists CFC, \{s, s'\} \subset CFC$
  - $CFC \subset \mathtt{term} \Leftrightarrow$ pas de transition sortante de CFC

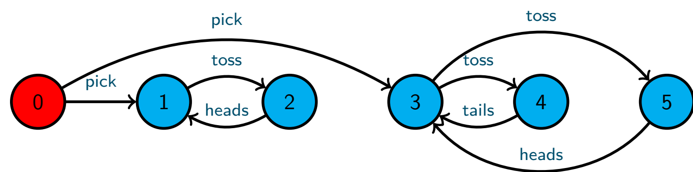
Vérification du progrès

- $\mathtt{fireable}(S) = \bigcup_{s \in S} \mathtt{fireable}(s)$
- `progress P = {a1,...an}` est vrai pour Q ssi
  $\forall S_T \in \mathtt{term}(Q), \mathtt{fireable}(S_T) \cap \{a1, ...an\} \neq \emptyset$
  - En effet, un LTS a un nombre fini d'états,
  - Donc, tout état visité infiniment l'est en boucle
  - Donc, il est déclenchable dans un des ens. terminaux d'états
- Par défaut vérifié pour tout l'alphabet
- Le défaut implique toutes les propriétés de progrès

Un exemple

- Deux pièces (une vraie et une fausse)

```
TWOCOIN = (pick -> COIN | pick -> TRICK),
TRICK = (toss -> heads -> TRICK),
COIN = (toss -> heads -> COIN | toss -> tails -> COIN).
```



- term(TWOCOIN) ?
- Quelles sont les propriétés de progrès vraies ?

```
progress HEADS = {heads}
progress TAILS = {tails}
progress HEADSorTAILS = {heads,tails}
```


La notion de priorité

- Les actions peuvent avoir des niveaux de priorité (pour résoudre une partie des problèmes de vivacité)
- Faible < normal < forte
- Lors d'un choix seuls les actions de priorité forte sont conservées
- Deux syntaxes
  - Priorité forte : `P << {a1,...,an}`
  - Priorité faible : `P >> {a1,...,an}`

```
P = (p -> P | q -> q -> P | r -> q -> P).
||SYS1 = (P) >> {q}.            = (p -> P | r -> q -> P)
||SYS2 = (P) << {q}.            = (q  -> q -> P)
```

Avancement

Bilan

Un processus de développement FSP / Java

1. Construction du modèle
   - 1.1 identifier les événements ou actions d'entrée / sortie du système
   - 1.2 décomposer le système en un ensemble de sous-systèmes
   - 1.3 pour chaque sous-système
     - s'il est simple, décrire en FSP son comportement (enchaînement des actions) de manière incrémentale
     - s'il est complexe, réappliquer la décomposition (1.1 à 1.3)
   - 1.4 produire la composition (avec les renommages nécessaires)

Un processus de développement FSP / Java

2. Vérification du modèle et de ses propriétés
   - 2.1 valider le modèle élément par élément à partir de l'automate si c'est possible
   - 2.2 animer le modèle pour se convaincre de sa validité
   - 2.3 vérifier l'absence d'interblocage
   - 2.4 spécifier les propriétés de sûreté du système et les vérifier
   - 2.5 spécifier les propriétés de vivacité du système et les vérifier si le défaut n'est pas vrai

Un processus de développement FSP / Java

3. Passage du modèle au code, pour chaque processus
   - 3.1 identifier sa structure
     - les paramètres définissent la forme des constructeurs
     - chaque variable d'état est un attribut
     - chaque action est une méthode
   - 3.2 identifier ses états
     - codage des états par des attributs
     - préconditions pour les actions et changement d'état
   - 3.3 identifier sa forme : actif, passif ou les deux
     - s'il est actif, il faut qu'il réalise `Runnable`
     - s'il est passif, c'est un moniteur, identifier ses conditions
   - 3.4 chaque action est une méthode qu'il faut programmer

4. Validation du programme
   4.1 . . . *as usual* . . .
   4.2 on peut essayer de reproduire les traces du modèle

Un processus de développement FSP / Java

1. Construction du modèle
2. Vérification du modèle et de ses propriétés
3. Passage du modèle au code, pour chaque processus
4. Validation du programme

- Ne pas le faire de manière linéaire
- Il faut réaliser les étapes de 1 à 4 sur des versions simplifiées et ajouter des fonctionnalités au fur et à mesure

Notes de cours sur le passage au code

Contenu

Ce document contient une présentation du passage d'un modèle FSP au code Java qui correspond. Il couvre les points abordés dans le deuxième cours et les illustrent par des exemples.

Comme vu en cours, il y a une correspondance entre modèle FSP et code. Cette correspondance peut être utilisée de deux façons :

- Après avoir conçu et validé un modèle FSP, on passe au code qui réalise ce modèle.
- Sur un code existant, on peut reconstruire un modèle FSP pour vérifier la correction du programme.

1 Du modèle au code

1.1 La structure

Lorsqu'on passe à la réalisation en Java, chaque processus du modèle devient une classe. Chacune des variables d'état d'un de ses états devient un attribut et chacune de ces actions devient une méthode. Les paramètres du processus deviennent les paramètres de son constructeur. Enfin, un constructeur par défaut doit toujours être fourni [1].

Ainsi, par exemple, le modèle suivant donne le squelette de code qui suit.

```
COUNTDOWN(N=3) = (start->COUNTDOWN[N]),
COUNTDOWN[i:0..N] = ( when(i>0) tick -> COUNTDOWN[i-1]
                    | when(i==0) beep -> STOP
                    | stop            -> STOP).
```

```
public class CountDown {
  private int i = 3;
  public CountDown() {}
  public CountDown(int n) {
    if (n >= 0)
      this.i = n;
  void beep() {
  }
  void start() {
  }
  void stop() {
  }
  void tick() {
  }
}
```

1.2 Les états

Les différents états sont, si nécessaire, encodés également par des attributs. Enfin, si une action n'est possible que dans un certain état, son corps vérifie que l'on est dans le bon état. Les gardes des actions sont également transformées en préconditions.

```
public class CountDown {
  private int i = 3;
  private boolean started;
  private boolean stopped;
  public CountDown() {}
  public CountDown(int n) {
    if (n >= 0)
      this.i = n;
  }
  void beep() {
    if (this.started && this.i == 0 && !this.stopped) {
      this.stopped = true;
      // le code de l'action beep
    }
  }
  void start() {
    if (!this.started) {
      this.started = true;
      // le code de l'action start
    }
  }
  void stop() {
    if (this.started && !this.stopped) {
      this.stopped = true;
      // le code de l'action stop
    }
  }
  void tick() {
    if (this.started && this.i > 0 && !this.stopped) {
      this.i--;
      // le code de l'action tick
    }
  }
}
```

1.3 Objet actif ou passif

Un processus peut être un objet :

- *actif* qui possède un comportement propre s'exécutant de manière autonome ;
- *passif*, c'est une donnée partagée qui peut être appelée par plusieurs autres processus ;
- hybride qui possède les deux formes simultanément.

En terme de réalisation sous forme de code :

- un objet actif devient une tâche d'un *thread*, sa classe doit donc réaliser l'interface `Runnable` et fournir le comportement de l'objet par la méthode `run` ;
- un objet passif est un moniteur (au sens de Java), ses méthodes doivent être synchronisées (si elles sont appelées par plusieurs autres processus) ;

Ainsi, dans l'exemple, notre compteur est un objet actif et doit donc réaliser l'interface `Runnable`.

1. Cette exigence n'est nécessaire que si on veut maintenir une correspondance exacte entre le code et le modèle.

1.4 Le déclenchement des actions

Pour chaque action, il faut identifier le ou les processus déclencheurs.

Une action prévue pour être appelée de l'extérieur par un autre processus ou par un autre programme est dite action « en entrée ». Une telle action doit être publique et elle doit être synchronisée si plusieurs sources peuvent l'invoquer ou si elle doit être exécutée en exclusion mutuelle.

Une action « en sortie » est uniquement appelée depuis la classe du processus, soit dans la méthode `run`, soit depuis une autre méthode. Elle est donc privée et doit être synchronisée si c'est nécessaire.

Si une action est synchronisée dans le modèle FSP (partagée entre plusieurs processus), un seul des processus doit véritablement fournir l'action « en entrée ». Les autres processus doivent lors du déclenchement de leur version de l'action invoquer cette action d'entrée.

Dans le compteur, il n'y a qu'un seul processus et donc aucune action partagée. Les actions `start` et `stop` sont des actions « en entrée » alors que les actions `tick` et `beep` sont des actions « en sortie ».

On pourrait ainsi obtenir le code ci-dessous. Dans ce code, vous remarquerez que les gardes des actions se traduisent par des choix dans le corps de la méthode `run`. Plus généralement, la méthode `run` des objets actifs doit assurer le comportement. Ici, tant que le compteur n'est pas stoppé, il va régulièrement faire le bon nombre de `tick` s'il est démarré. Lorsque le décompte est terminé, il `beep`.

```
public class CountDown implements Runnable {
  private int i = 3;
  private boolean started;
  private boolean stopped;
  public CountDown(int n) {
    if (n >= 0)
      this.i = n;
  }
  @Override
  public void run() {
    while (!this.stopped) {
      try {
        Thread.sleep(5);
      } catch (InterruptedException e) {
        return;
      }
      if (this.started) {
        if (this.i > 0) {
          tick();
        } else if (this.i == 0) {
          beep();
        }
      }
    }
  }
  public synchronized void start() {
    if (!this.started) {
      this.started = true;
      // le code de l'action start
      System.out.println("start");
    }
  }
  public synchronized void stop() {
    if (this.started && !this.stopped) {
      this.stopped = true;
      // le code de l'action stop
      System.out.println("stop");
    }
  }
  private void beep() {
    if (this.started && this.i == 0 && !this.stopped) {
      this.stopped = true;
      // le code de l'action beep
      System.out.println("beep");
    }
  }
  private void tick() {
    if (this.started && this.i > 0 && !this.stopped) {
      this.i--;
      // le code de l'action tick
      System.out.println("tick (" + this.i + ")");
    }
  }
  public static void main(String[] args) {
    CountDown cd = new CountDown(10);
    new Thread(cd).start();
    cd.start();
    try {
      Thread.sleep(75);
    } catch (InterruptedException e) {}
    cd.stop();
  }
}
```

2 Un autre exemple

Soit le modèle FSP suivant :

```
CARPARK(N=4) = SPACES[N],
SPACES[i:0..N] = (
  when(i>0) arrive -> SPACES[i-1]
| when(i<N) depart -> SPACES[i+1]).
ARRIVALS  = (arrive -> ARRIVALS).
DEPARTURES = (depart -> DEPARTURES).
||CARPARK = (ARRIVALS || CARPARK(4) || DEPARTURES).
```

Il contient trois processus dont deux sont des objets actifs `ARRIVALS` et `DEPARTURES` et le dernier, `CARPARK` est un objet passif dont les méthodes doivent être synchronisées.

La classe Java correspondant au processus `ARRIVALS` est :

```
public class Arrivals implements Runnable {
  CarPark carpark;
  Arrivals(CarPark c) {
    this.carpark = c;
  }
  public void run() {
```

```
    while (true) {
      DisplayedThread.rotate(330);
      this.carpark.arrive();
      DisplayedThread.rotate(30);
    }
  }
}
```

Ainsi, le comportement récursif est codé sous la forme d'une boucle infinie dans le corps de la méthode `run`. Ici, le comportement du processus est très simple, il effectue une suite d'actions `arrive`. Cette action est ici une méthode de l'objet partagé `CarPark`. Le processus `DEPARTURES` est similaire mais appelle la méthode `depart`.

L'objet partagé `CarPark` a le code suivant :

```
public class CarPark {
  private final int capacity;
  private final DisplayedNumber nc;
  public CarPark(int n, DisplayedNumber nc) {
    this.capacity = n;
    this.nc = nc;
    nc.setValue(n);
  }
  public synchronized void arrive() {
    while (this.nc.getValue() == 0)
      try {
        wait();
      } catch (InterruptedException e) {}
    this.nc.decrValue();
    notifyAll();
  }
  public synchronized void depart() {
    while (this.nc.getValue() == this.capacity)
      try {
        wait();
      } catch (InterruptedException e) {}
    this.nc.incrValue();
    notifyAll();
  }
}
```

Chaque action provoquant une synchronisation avec d'autres processus et modifiant l'état de l'objet doit être `synchronized`. Dans le code ci-dessus, un affichage est géré, en plus, pour pouvoir suivre l'évolution du nombre de voitures dans le parking.

Il est important de remarquer que le contenu des méthodes correspondant à des actions est de la responsabilité du développeur. Ici, le développeur a fait le choix de rendre bloquante les méthodes en utilisant une boucle d'attente comme vue dans l'UV1.

PC3-TP3 – Le repas des philosophes

Objectifs

Au cours de ces deux séances, nous allons découvrir par la pratique la notion d'interblocage (ou « étreinte fatale »), en particulier :

— modéliser un problème célèbre en FSP,
— tenter de trouver des solutions aussi élégantes que possible.

Puis, nous allons sur cet exemple découvrir le passage d'un modèle au code Java correspondant.

Le repas des philosophes

Un groupe de philosophes se retrouvent pour un repas. Ils sont assis autour d'une table ronde, devant un bon plat de *spaghettis*. Une fourchette est placée entre chaque paire d'assiettes, ainsi, un philosophe a toujours une fourchette à sa gauche et une fourchette à sa droite. La configuration de la table est représentée en figure 1.

Une fois assis, un philosophe a besoin de deux fourchettes pour manger, puis il relâche ces deux fourchettes dès que lui prend une irrépressible envie de réfléchir.

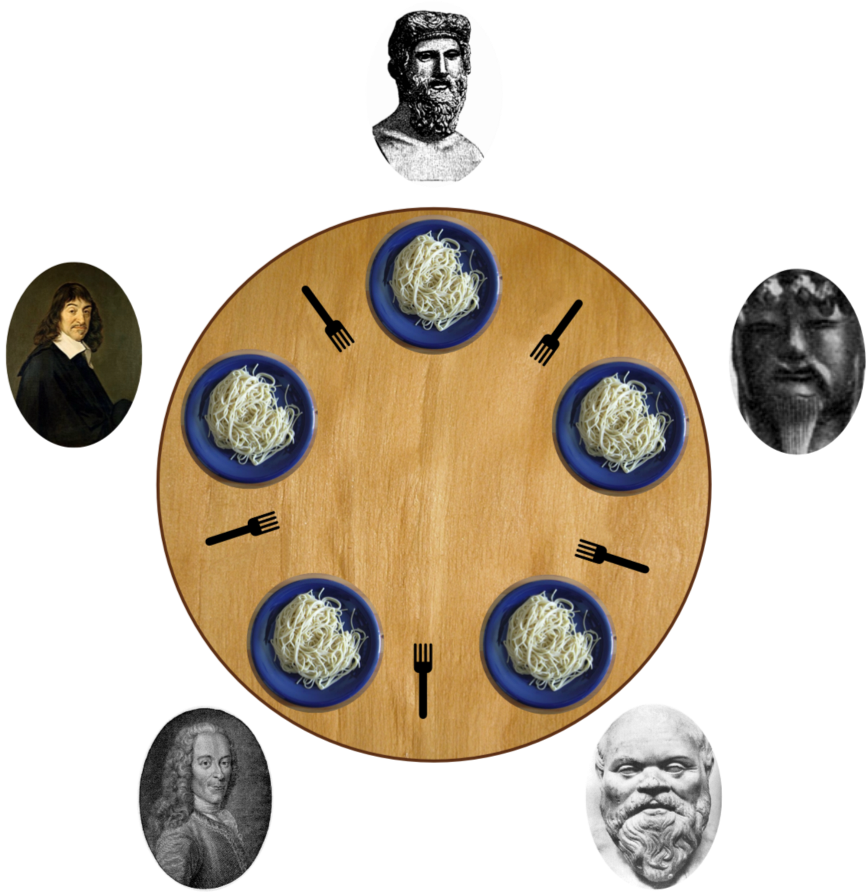


FIGURE 1 – repas des philosophes

Diner
+ NUMPHILS : int
+ Diner()
+ eatTime() : int
+ thinkTime() : int

PhiloView
+ THINKING : int
+ HUNGRY : int
+ GOTRIGHT : int
+ EATING : int
+ GOTLEFT : int
+ PhiloView()
+ setFork(int,boolean)
+ setPhil(int,int)

FIGURE 2 – Diagramme de classes fournies.

Exercice 1 (*Un modèle pour les philosophes*)

▷ **Question 1.1 :**
Proposer dans un premier temps une modélisation « intuitive » de ce repas, puis déterminer une exécution conduisant à une situation de *deadlock*.

▷ **Question 1.2 :**
Dans un second temps, modifier votre modèle afin qu'aucune situation d'interblocage ne puisse survenir.

Les philosophes en Java

Avant de réaliser cet exercice, il sera utile de relire les notes de cours sur le passage d'un modèle FSP au code Java qui lui correspond.

Pour faciliter la visualisation de la tablée de philosophes, vous trouverez sur moodle, un projet eclipse contenant deux classes pour construire une interface graphique à notre dîner. Le diagramme de classe est présenté figure 2. Une classe `Diner` permet d'initialiser l'application et son interface graphique. Elle est incomplète car il manque l'initialisation des fourchettes et des philosophes. Une classe `PhiloView` est en charge de l'affichage des philosophes et des fourchettes suivant leur état. Pour cela, elle définit les états des philosophes (définis par des constantes `THINKING`, ...). Une image spécifique est associée à chaque état. Pour gérer la mise à jour de l'interface graphique, vous devrez utiliser les méthodes `void setFork(int id, boolean taken)` et `void setPhil(int id, int s)` qui prennent toutes deux en paramètre le numéro de l'élément à mettre à jour (fourchette ou philosophe) ainsi que son nouvel état.

Remarque :

Attention, la classe `PhiloView` impose une numérotation spécifique des fourchettes. Un philosophe de numéro `i` a la fourchette de numéro `i` à sa droite et la fourchette de numéro `i-1` à sa gauche.

Exercice 2 (*Les philosophes en Java*)

▷ **Reprendre l'exercice précédent et proposer une réalisation en Java.**

TP4-5 – Du modèle au code : *Roller coaster*

Objectifs

L'objectif de ces deux TP est de produire des modèles un peu plus complexes et mieux comprendre le lien entre un modèle et le programme Java correspondant.

1 Le cadre de travail

La commune de Plouzané souhaite se munir d'un « grand huit » pour la saison touristique. Le commercial en charge de la vente des manèges propose un système de contrôle extensible en terme de nombre de wagons et de nombre de places à bord des wagons. Le logiciel assurant le contrôle est sous licence GPL et peut donc être modifié à volonté. Le commercial garantit qu'il n'y a jamais eu de problème sur la première et unique version de ce manège qui a été vendue sans facture à l'appui. La commune ne souhaite pourtant pas rester sur cette intime conviction du vendeur.

Dans son cahier des charges, elle souhaite mettre en œuvre la version du manège avec deux wagons. Le premier wagon peut accueillir deux passagers, le second trois passagers. Les deux wagons sont indépendants sur le circuit, c'est-à-dire, ils ne sont pas attachés. En revanche, ils ne peuvent pas se doubler. Les passagers sont en attente d'un wagon sur une plate-forme qui peut accueillir neuf personnes au maximum. Lorsqu'un wagon arrive sur la plate-forme, il attend qu'il y ait suffisamment de passagers présents pour les charger et partir (c'est-à-dire deux ou trois passagers suivant le wagon). Un seul des deux wagons peut être en attente sur la plate-forme d'accès à un instant donné (il existe une unique rampe d'accès pour les passagers).

2 Les exercices

Il s'agit de construire une version simplifiée du modèle du système présenté ci-dessus. Notre système sera composé de trois processus :

1. Le premier `CAR` modélise le comportement d'une voiture.
2. Le deuxième `PLATFORM` modélise le comportement de la plate-forme.
3. Le troisième `PASSENGERS` modélise le comportement des clients souhaitant accéder à la plate-forme.

Exercice 1 (*Un seul wagon à un passager*)

Dans ce premier exercice, nous commençons par travailler avec un seul wagon qui ne peut accueillir qu'un seul passager. On suppose qu'une voiture peut effectuer les actions suivantes :

- `arrive` pour modéliser l'arrivée de la voiture sur la plate-forme de chargement des passagers,
- `requestPassenger` pour indiquer au système que la voiture est prête à embarquer des passagers,
- `getPassenger` pour permettre la montée à bord du passager,
- `depart` pour indiquer le départ de la voiture de la plate-forme.

▷ **Question 1.1 :**
Proposer un modèle de la voiture `CAR` en FSP.

(a) Un exemple d'affichage de *thread*

(b) L'affichage voulu pour l'exercice 1

La plate-forme peut contenir un nombre maximal de neuf passagers. Son contrôleur peut réaliser les actions `newPassenger` lorsqu'un nouveau passager candidat se présente, `requestPassenger` et `getPassenger` pour contrôler l'accès des passagers aux voitures. Initialement, on supposera que la plate-forme est vide.

▷ **Question 1.2 :**
Proposer un modèle de la plate-forme `PLATFORM` et de `PASSENGERS` en FSP.

▷ **Question 1.3 :**
Proposer un modèle complet du système.

▷ **Question 1.4 :**
Proposer une première version simple du code Java du système.

Pour faciliter la visualisation de l'application, nous vous proposons de réutiliser un code fourni qui permet de suivre l'affichage de valeur et l'évolution de *thread* en les animant. Un code exemple est sur moodle. Son rendu graphique est en figure 1(a). La partie supérieure affiche une valeur entière (ici la valeur de l'attribut `val` de la classe `Exemple`), la partie centrale affiche la progression du *thread* sous la forme d'un secteur angulaire qui progresse (ici en violet) et en partie inférieure, un bouton pour démarrer et un pour suspendre le *thread*.

▷ **Question 1.5 :**
En vous inspirant de l'exemple fourni, proposer une version de votre code Java du système de façon à ce qu'il suive l'affichage de la figure 1(b).

Exercice 2 (*avec plusieurs voitures*)

▷ **Question 2.1 :**
Modifier le modèle de l'exercice précédent pour qu'il supporte les deux voitures en même temps.

Un de vos camarades a un modèle dans lequel il y a deux voitures `a` et `b`. Dans son modèle, la trace suivante est accessible :

`a.arrive`, `a.requestPassenger`, `newPassenger`, `a.getPassenger`, `newPassenger`, `b.arrive`, `b.requestPassenger`, `b.getPassenger`, `b.depart`, `a.depart` ...

▷ **Question 2.2 :**
Cette trace vous semble-t-elle acceptable ? Pourquoi ?

Exercice 3 (*version finale de la spécification*)

▷ **Question 3.1 :**
Généraliser votre solution précédente pour fournir un modèle (correct [1]) répondant au problème décrit dans la première section (c'est-à-dire, les voitures peuvent embarquer plus d'un passager).

▷ **Question 3.2 :**
Modifier votre code Java pour qu'il réalise votre nouveau modèle, l'exécuter et vérifier si vous obtenez des comportements non souhaitables.

Exercice 4

L'objectif de cette question est d'éliminer le problème décrit par la trace de la question 2.2.

Une solution possible consiste à obliger la trace à ne contenir que des suites de la forme `arrive` puis `depart` puis `arrive` puis `depart` . . .

▷ **Modifier votre modèle et vérifier à l'aide de l'animateur que les wagons ne peuvent plus être en même temps au niveau de la plate-forme. Modifier ensuite le code pour mettre en œuvre cette solution.**

3 Pour aller plus loin

Exercice 5 (*Optimisation du temps d'attente des passagers*)

Pour le moment, nos wagons ne partent que s'ils sont pleins. Afin d'éviter une longue attente inutile aux passagers, on permet qu'un wagon parte même s'il n'est pas plein.

▷ **Modifier le modèle et le code pour satisfaire à cette nouvelle exigence.**

Exercice 6 (*Remplissage des wagons au fur et à mesure de l'arrivée de passagers*)

On abandonne l'idée précédente et on choisit de remplir totalement les wagons en permettant aux passagers de s'installer dans le wagon pendant que celui-ci est immobilisé dans l'attente de passagers.

▷ **Modifier le modèle et le code pour satisfaire à cette nouvelle exigence.**

1. Sans *deadlock* par exemple.

Troisième partie

La modélisation des systèmes répartis

PC4 – Modélisation de systèmes répartis

Échange de messages et aspects temporels

La modélisation FSP permet de décrire le fonctionnement de processus, ainsi que l'entrelacement de leurs exécutions. Mais, parfois, les processus peuvent avoir des vitesses d'exécution très différentes, par exemple lorsqu'ils sont situés sur des machines différentes. Pour se synchroniser, les processus doivent également utiliser des canaux de communications qui sont plus ou moins performants. Nous allons nous intéresser maintenant aux problèmes que peuvent soulever ces contraintes.

La première des choses pour construire un modèle d'un système réparti est la donnée d'un ensemble de processus $p_1, \dots, p_n$. Ces processus modélisent des exécutions sur n machines indépendantes. Chaque processus peut lui même être complexe et se décomposer en un ensemble de sous-processus concurrents (au sens vu dans la première partie du module).

Ensuite, dans le cadre d'un système réparti, la synchronisation d'actions de FSP qui impose leur exécution simultanée n'est plus réaliste car on ne peut assurer une synchronisation parfaite entre deux machines. La communication doit donc devenir *asynchrone*. Pour cela, nous introduisons deux primitives atomiques (c'est-à-dire indivisibles) :

- $send(m)$ *to* i pour décrire l'envoi d'un message m au processus p_i
- $rec(m)$ pour décrire la reception d'un message m d'un autre processus

Remarquez qu'ici le message n'est pas une donnée qui est transportée d'un processus à un autre mais un identifiant permettant de relier un *rec* effectué par un processus au *send* qui lui correspond dans le processus envoyeur.

Ces deux actions primitives s'ajoutent au modèle FSP existant. Nous parlerons d'*événements*[1] en faisant une distinction entre les événements *internes*, qui correspondent à une séquence d'opérations exécutées par le processus (*cf.* une action dans la modélisation FSP), et les événements *de communication*, qui sont relatifs aux sus-dites primitives (*send* et *rec*).

Afin de formaliser le concept de communication, nous introduisons maintenant quelques notations. Nous considérons n processus indépendants $p_1, \dots, p_n$. Lors d'une exécution d'un processus, nous notons e_i^x le x-ième événement produit par le processus p_i. Une trace d'exécution du processus p_i est donc :

$$e_i^1 \; e_i^2 \; \dots \; e_i^x \; e_i^{x+1} \dots$$

Exercice 1 (*Événements et relations causales*)

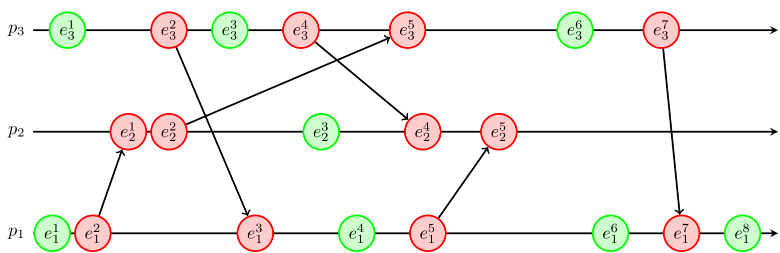
On définit alors la relation suivante dite de *causalité* : «*un événement e_i^x précède un événement e_j^y*». Elle s'écrit $e_i^x \xrightarrow{ev} e_j^y$.

Un événement e_i^x précède un événement e_j^y si et seulement si une parmi les trois conditions suivantes est vraie :

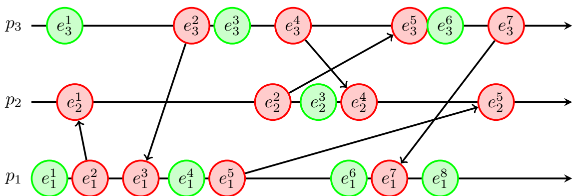
1. les événements se déroulent au sein d'un même processus
2. les deux événements concernent le même message m
3. une transition par un événement indépendant existe

▷ **Question 1.1 :**
Dans le but de vérifier que vous n'avez pas survolé, hagards, les quelques lignes précédentes, veuillez définir formellement ces trois conditions (attention, c'est trivial).

Voici un petit exemple d'une exécution d'un système distribué :



▷ **Question 1.2 :**
Quelle différence faites-vous entre ce diagramme et la réprésentation ci-dessous ?



Un *chemin causal* est défini comme étant une suite d'événements consécutifs au sens de la relation $\xrightarrow{ev}$. Par extension, le *passé causal* (respectivement *futur causal*) d'un événement e est l'ensemble des événements situés avant (respectivement après) e dans l'ensemble des chemins causaux.

▷ **Question 1.3 :**
Dans le diagramme précédent, quel est le passé causal de e_2^2 ? Quel est son futur causal ?

Soit F un ensemble de n événements donnés, un événement par processus. Une *coupe* est l'ensemble des événements antérieurs aux événements de F dans chaque processus. On appelle *frontière* cet ensemble F. Par exemple dans le diagramme précédent, une coupe C_1 est définie par sa frontière $\{e_1^3, e_2^1, e_3^4\}$ et la coupe C_2 est définie par $\{e_1^3, e_2^1, e_3^5\}$[2].

▷ **Question 1.4 :**
Quelle différence fondamentale voyez-vous entre les coupes C_1 et C_2 ? Définissez formellement une coupe *cohérente*.

Avez-vous une formulation qui permettrait à un enfant de cinq ans de comprendre ce qu'est une coupe cohérente sur un diagramme de temps comme celui représenté ci-dessus ?

Exercice 2 (*Cohérence d'états et exécutions*)

Dans ce contexte, à quoi ressemblent nos bons vieux états de FSP ? Nous utilisons ici une notation simplifiée où σ_i^x représente l'état dans lequel se trouve le processus i immédiatement après l'événement e_i^x. Naturellement, le processus est dans cet état jusqu'à l'événement e_i^{x+1}.

On peut définir une relation causale entre les états par la transition $\xrightarrow{state}$. Ainsi, nous avons :

$$e_i^{x+1} \xrightarrow{ev} e_j^y \Longleftrightarrow \sigma_i^x \xrightarrow{state} \sigma_j^y$$

1. En FSP, on employait plutôt le terme d'action mais dans le cadre de cette séance, nous emploierons le vocabulaire standard des systèmes répartis : événement.

2. Un événement à la frontière d'une coupe est considéré comme étant inclus dans la coupe.

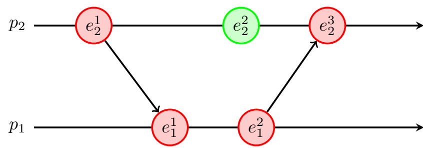
Un *état global* du système est un ensemble d'états locaux, par exemple $\Sigma = \{\sigma_1^{x_1}, \ldots, \sigma_n^{x_n}\}$.

▷ **Question 2.1 :**
Qu'est-ce qu'un état global cohérent ? Qu'est-ce que cela signifie concrètement dans un système distribué ?

On peut maintenant revenir au concept de *trace* qui a été vu dans le modèle FSP. On parle également d'*exécution*.

▷ **Question 2.2 :**
Qu'est-ce qu'une exécution cohérente ?

Partons d'un diagramme relativement simple.



▷ **Question 2.3 :**
Déterminez l'ensemble des exécutions cohérentes du système et tentez de le représenter.

▷ **Question 2.4 :**
Comment pourrait-on utiliser la composition usuelle de FSP (||) de processus pour modéliser un système réparti ?

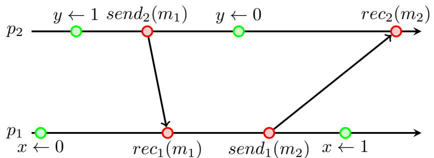
Exercice 3 (*Propriétés*)

On dit qu'un état global Σ_2 est *atteignable* à partir d'un état global Σ_1 s'il existe une exécution cohérente qui permet de passer de Σ_1 à Σ_2. Un état global peut vérifier une propriété donnée, ou pas (on parle également de prédicat).

▷ **Question 3.1 :**
Qu'est-ce qu'une propriété globale stable ? Donnez quelques exemples de propriétés globales stables.

Définissez la *vivacité* et la *sûreté* d'un système distribué.

Un ultime petit exercice. Voici un diagramme.



▷ **Question 3.2 :**
Dans ce système, quelle est la caractéristique de la propriété : $x = 0 \ \wedge \ y = 0$.

PC5 – Le kem's

Préambule

Le *kem's* est un jeu de cartes qui se joue entre équipes de deux joueurs. Les membres d'une même équipe doivent convenir préalablement d'un signe qui permettra à un joueur d'avertir son partenaire. Entre le moment où un joueur réalise ce signe et le moment où son partenaire aperçoit le signe, il peut se passer un certain temps.

Le jeu se base sur des échanges fréquents de cartes. Lorsqu'un joueur possède quatre cartes identiques, il peut réaliser le signe. Dès que son partenaire aperçoit le signe, il peut alors s'écrier, triomphant, **KEMS**. En revanche, un adversaire qui suspecte un joueur de posséder quatre cartes identiques peut s'écrier **CONTRE-KEMS** [1] et remporter la partie. Le joueur qui possède quatre cartes identiques est donc dans une situation risquée :

- s'il n'échange pas de cartes, il augmente les risques d'être suspecté
- s'il échange une carte alors que son partenaire n'a pas dit KEMS, il n'aura plus quatre cartes identiques

La situation extatique correspond à deux joueurs qui possèdent quatre cartes identiques et qui ont aperçu le signe réalisé par le partenaire. Ils peuvent alors s'écrier en chœur **DOUBLE-KEMS**.

Exercice 1 (*LTSA*)

Les événements d'un jeu de kem's peuvent être simplifiés ainsi :

- `draw` : un échange de cartes qui permet d'obtenir quatre cartes identiques
- `break` : un échange de cartes qui casse un jeu de quatre cartes identiques
- `sendKems` : réalise le signe
- `recKems` : aperçoit le signe
- `kems` : s'écrit KEMS
- `doubleKems` : s'écrit DOUBLE-KEMS

▷ **Représentez l'automate LTSA d'une équipe de deux joueurs de kem's**

Exercice 2 (*Analyse d'une partie*)

Vous venez de jouer une partie de kem's et vous souhaitez analyser votre partie. Vous avez pris soin de noter l'ordre des événements tels que vous les avez perçus :

`g1.draw`, `g1.sendKems`, `g1.recKems`, `g1.break`

▷ **Question 2.1 :**
Votre partenaire et vous avez-vous été dans une situation où :

1. **vous auriez pu annoncer un KEMS gagnant**
2. **il aurait pu annoncer un KEMS gagnant**
3. **vous auriez pu annoncer un double KEMS**
4. **un des deux joueurs annonce un KEMS alors que son partenaire n'a pas quatre cartes identiques**

1. Pour simplifier, nous ne modéliserons pas le CONTRE-KEMS.

Justifiez par une trace d'exécution.

Votre partenaire vous donne sa vision des événements :

`g2.draw`, `g2.recKems`, `g2.sendKems`, `g2.break`

▷ **Question 2.2 :**
Représentez le système sous forme de diagramme.

Vous souhaitez maintenant analyser formellement votre partie.

▷ **Question 2.3 :**
Formalisez les états locaux et l'état global de manière à répondre aux questions suivantes :

1. **un des joueurs a-t-il été forcément dans une position d'annoncer un KEMS gagnant ?**
2. **un des joueurs a-t-il été dans une position d'annoncer un KEMS alors que son partenaire n'avait pas quatre cartes identiques ?**
3. **était-il possible d'annoncer un DOUBLE-KEMS ?**

Exercice 3 (*Extension à* 3 *joueurs*)

La version *hardcore* du jeu de kem's se joue avec des équipes de n joueurs.

Tous les joueurs doivent avoir aperçu le signe émis par un joueur pour s'écrier KEMS X dans lequel X est l'identifiant du joueur qui a quatre cartes identiques. (pour l'instant, la coordination des n joueurs ayant aperçu le signe leur permettant de crier KEMS X simultanément n'est pas l'objet de cette Petite Classe).

▷ **Question 3.1 :**
Proposez une nouvelle définition pour les événements de communication.

▷ **Question 3.2 :**
Nous considérons $n = 3$. Analysez le système dont la trace d'exécution est la suivante.

g1 : `g1.draw`, `g1.sendKems`, `g1.recKems2`, `g1.break`, `g1.recKems3`

g2 : `g2.draw`, `g2.sendKems`, `g2.recKems3`, `g2.recKems1`

g3 : `g3.recKems2`, `g3.draw`, `g3.sendKems`, `g3.recKems1`

www.telecom-bretagne.eu

Campus de Brest
Technopôle Brest-Iroise
CS 83818
29238 Brest Cedex 3
France
Tél : + 33 (0)2 29 00 11 11
Fax : + 33 (0)2 29 00 10 00

Campus de Rennes
2, rue de la Châtaigneraie
CS 17607
35576 Cesson Sévigné Cedex
France
Tél : + 33 (0)2 99 12 70 00
Fax : + 33 (0)2 99 12 70 19

Campus de Toulouse
10, avenue Édouard Belin
BP 44004
31028 Toulouse Cedex 04
France
Tél : + 33 (0)5 61 33 83 65
Fax : + 33 (0)5 61 33 83 75