Notes de cours sur le langage FSP

INF447

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 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).
```

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

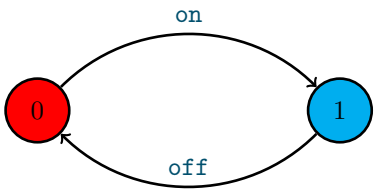


FIGURE 1 – Un interrupteur : expression FSP et automate associé

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`.

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` :

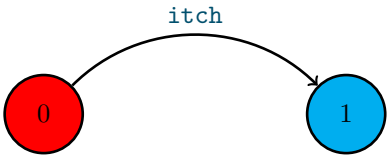
```
ITCH = (scratch -> STOP).
CONVERSE = (think -> talk -> STOP).
||CONVERSE_ITCH = (ITCH || CONVERSE).
```

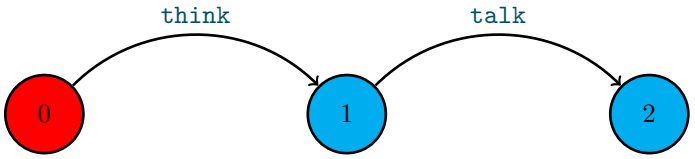
Les trois traces possibles sont :

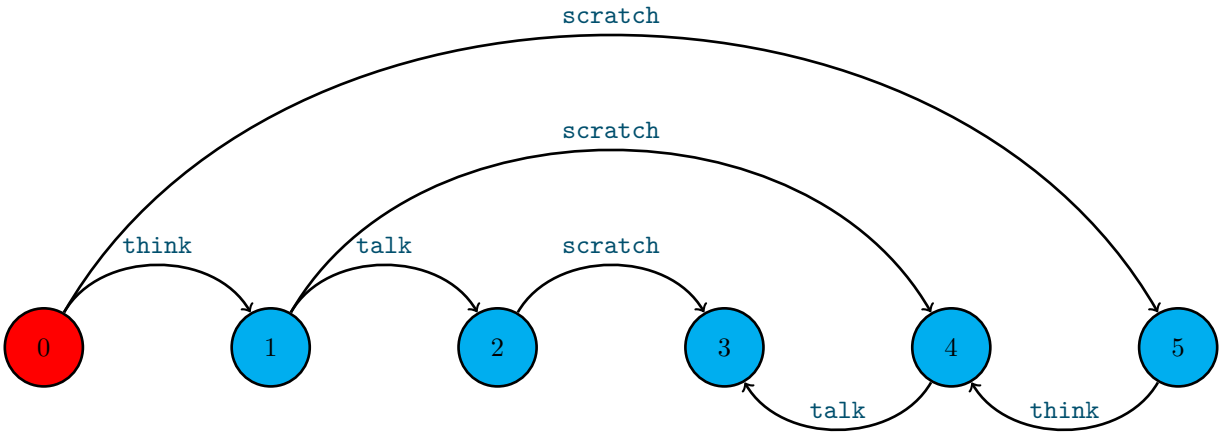
1. `think -> talk -> scratch`
2. `think -> scratch -> talk`
3. `scratch -> think -> talk`

Et leurs automates respectifs sont :

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







Dans l'automate composé les états sont obtenus par couplage de la manière suivante :

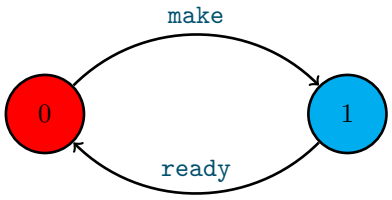
- 0 correspond à 0 dans `ITCH` et 0 dans `CONVERSE`
- 1 correspond à (0,1)
- 2 correspond à (0,2)
- 3 correspond à (1,2)
- 4 correspond à (1,1)
- 5 correspond à (1,0)

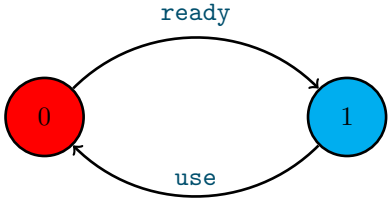
La synchronisation

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).

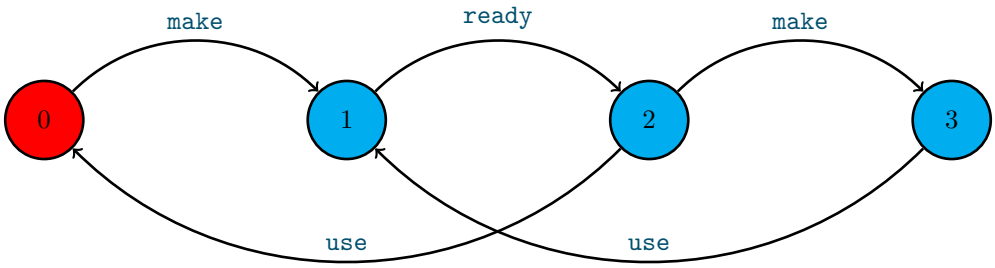
Soit les deux processus `MAKER` et `USER` :

```
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 :

- $S = S_1 \times S_2$
- $A = A_1 \cup A_2$
- $T = \{(s_1, s_2) \xrightarrow{a} (s'_1, s'_2) \mid (s_1 = s'_1 \wedge s_2 \xrightarrow{a} s'_2 \wedge a \notin A_1) \vee (s_2 = s'_2 \wedge s_1 \xrightarrow{a} s'_1 \wedge a \notin A_2) \vee (s_1 \xrightarrow{a} s'_1 \wedge s_2 \xrightarrow{a} s'_2 \wedge a \in A_1 \cap A_2)\}$
- $i = (i_1, i_2)$

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)`).

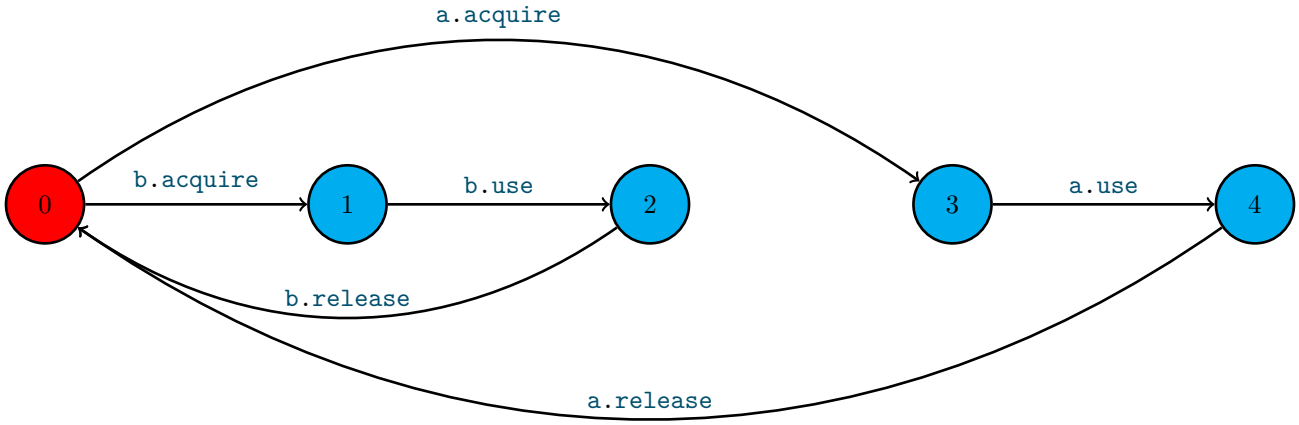
On peut ajouter un ensemble de préfixes sous la forme d'un ensemble `{a,b}` (resp. d'un tableau `a[i:1..N]`) à toutes les actions d'un processus `P` par `{a,b}::P`. Ainsi, chaque transition `n -> X`

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

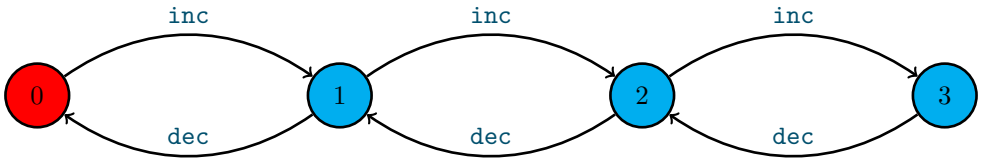
Il est possible d'associer des informations à un état. Cela se fait en utilisant la notion de variable d'état. En FSP, les variables d'état sont des entiers et il faut définir l'intervalle borné dans laquelle chacune d'elle varie. Ces bornes permettent une exploration des différentes valeurs possibles 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]`.

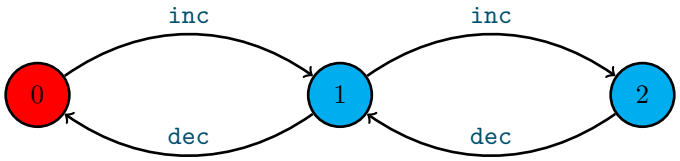
Les variables d'état peuvent aussi être utilisées dans des gardes permettant de conditionner l'exécution d'action. La syntaxe pour une action gardée est `(`**`when`** *cond* `x -> P)` et signifie que si la condition *cond* est vérifiée l'action `x` est possible sinon `x` ne peut pas s'exécuter. La syntaxe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



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.

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

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).
```

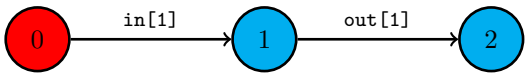
Ce processus est équivalent à :

```
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)).
```

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).

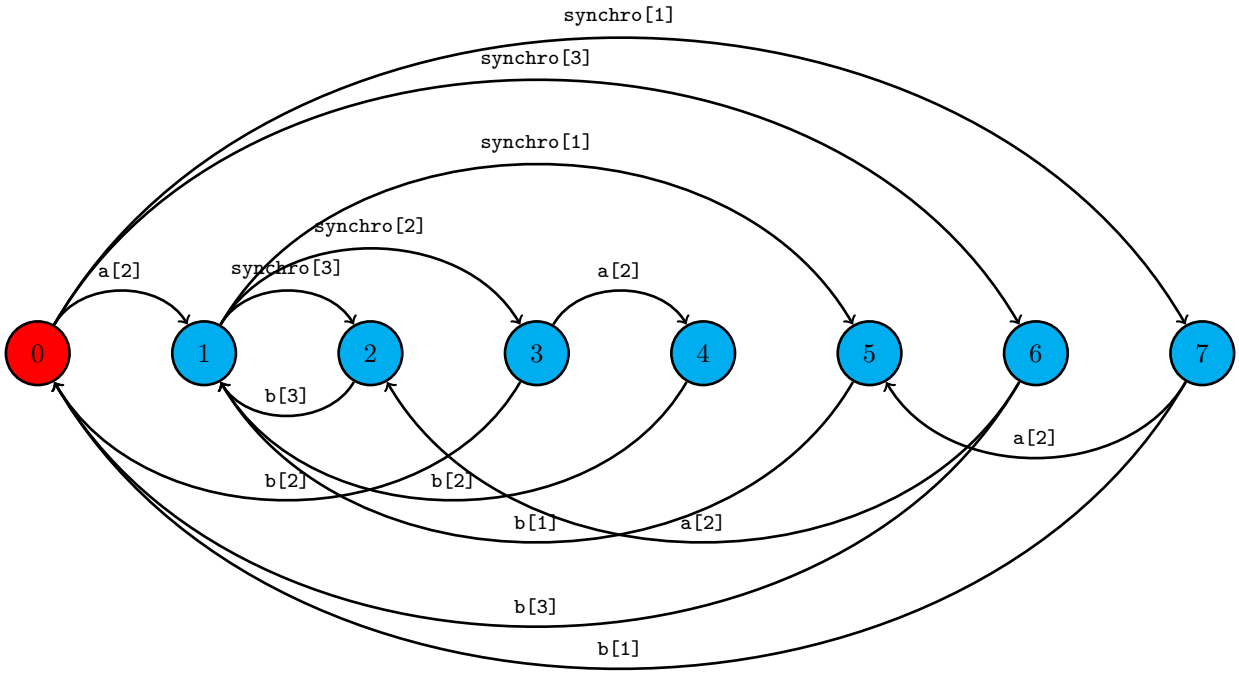


FIGURE 2 – La composition sans extension d'alphabet

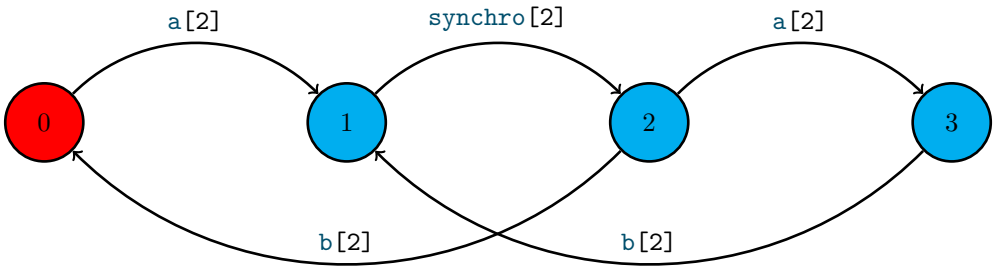


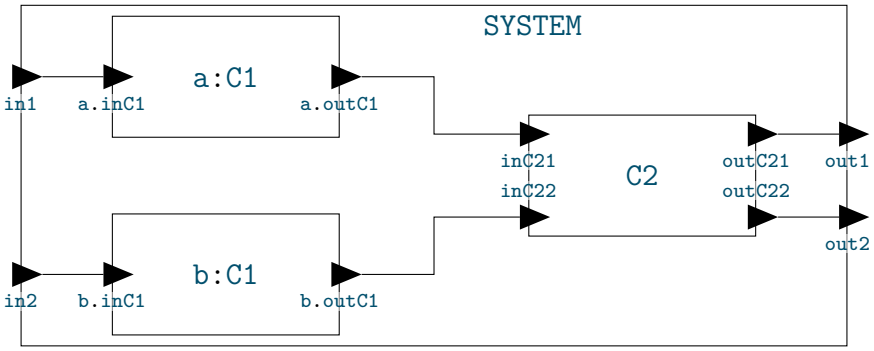
FIGURE 3 – La composition avec extension d'alphabet

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.

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`.

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