IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

Propriétés et vérification

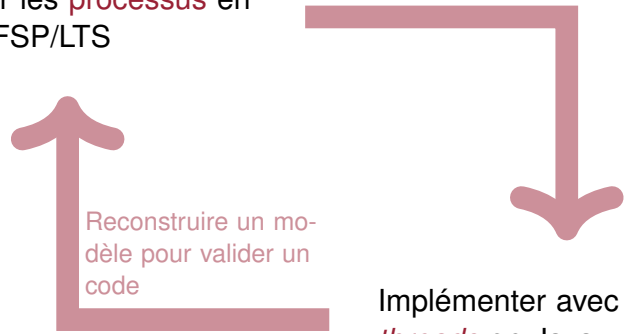
Fabien Dagnat

Concurrence – C5– janvier 2026

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 n'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
    - résolution de famine (on finit par obtenir des ressources demandées)
- On spécifie la propriété (FSP) et on analyse (LTSA)

Plan

Avancement

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 [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 l'éviter, casser une des conditions précédentes
- Sur l'exemple, évitement par
  - acquisition des 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).
```

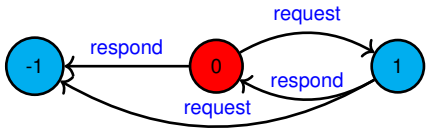
Avancement

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

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



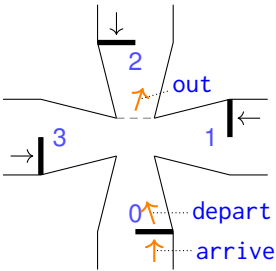
Utilisation des propriétés de sûreté

- 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

$\Rightarrow$ `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 Interblocage

2 Sûreté

3 Vivacité

Progrès

- Vivacité = une action arrivera finalement
  - par ex, le client `c[i]` finira par obtenir le sémaphore
- 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 calcule 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 : $\texttt{term}(P)$
- Calcul des ens. terminaux de (S, A, T, s_0)
  - pour chaque état, ens. des états accessibles (acces)
  - $\forall s \in S, s' \in \texttt{acces}(s) \wedge s \in \texttt{acces}(s') \Rightarrow \exists CFC, \{s, s'\} \subset CFC$
  - $CFC \subset \texttt{term} \Leftrightarrow$ pas de transition sortante de CFC

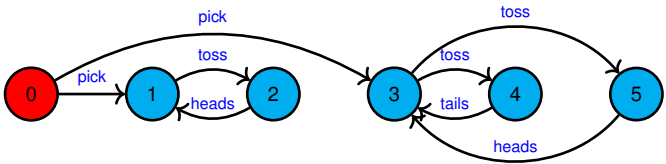
Vérification du progrès

- $\texttt{fireable}(S) = \bigcup_{s \in S} \texttt{fireable}(s)$
- **progress** P = {a1, . . . an} est vrai pour Q ssi
  $\forall S_T \in \texttt{term}(Q), \texttt{fireable}(S_T) \cap \{a1, ...an\} \neq \emptyset$
  - 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}
```

Bilan

Un processus de développement FSP / Java

1. Construction du modèle
   1. identifier les événements ou actions d'entrée / sortie du système
   2. décomposer le système en un ensemble de sous-systèmes
   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)
   4. produire la composition (avec les renommages nécessaires)

Un processus de développement FSP / Java

2. Vérification du modèle et de ses propriétés
   1. valider le modèle élément par élément à partir de l'automate si c'est possible
   2. animer le modèle pour se convaincre de sa validité
   3. vérifier l'absence d'interblocage
   4. spécifier les propriétés de sûreté du système et les vérifier
   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
   1. identifier sa structure
      - les paramètres définissent la forme des constructeurs
      - chaque variable d'état est un attribut
      - chaque action est une méthode
   2. identifier ses états
      - codage des états par des attributs
      - préconditions pour les actions et changement d'état
   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
   4. chaque action est une méthode qu'il faut programmer

Un processus de développement FSP / Java

4. Validation du programme
   1. . . . *as usual* . . .
   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