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

Reconstruire un modèle pour valider un code

Implémenter avec des *threads* en Java

Problèmes

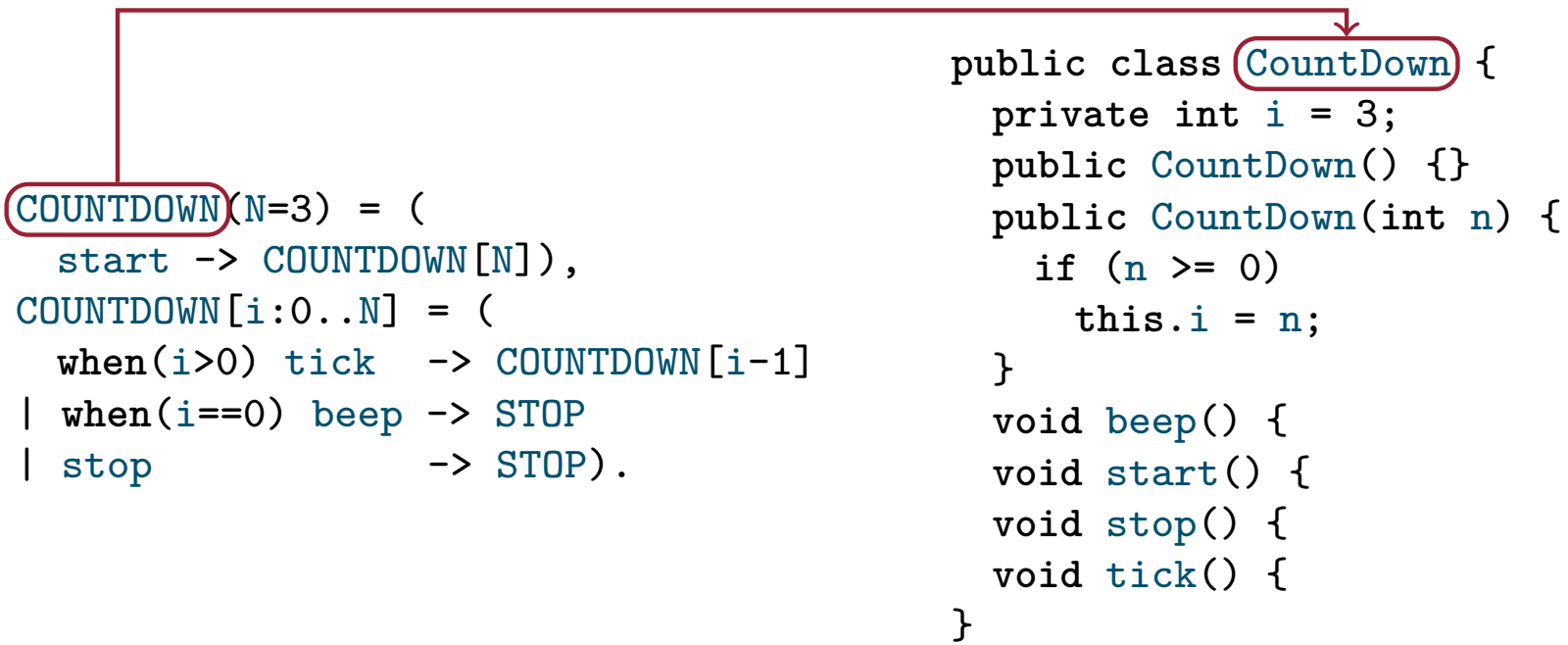
- Sûreté (*safety*)
  - Il ne se passe jamais rien de mauvais
  - On atteint jamais un mauvais état
  - Par exemple
    - pour un comportement séquentiel : l'état final est bon
    - non respect d'une exclusion mutuelle
    - absence d'interblocage (*deadlock*)
- Vivacité (*liveness*)
  - Il finit par 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

Avancement

Correspondance FSP / Java I

- Un processus FSP est une classe

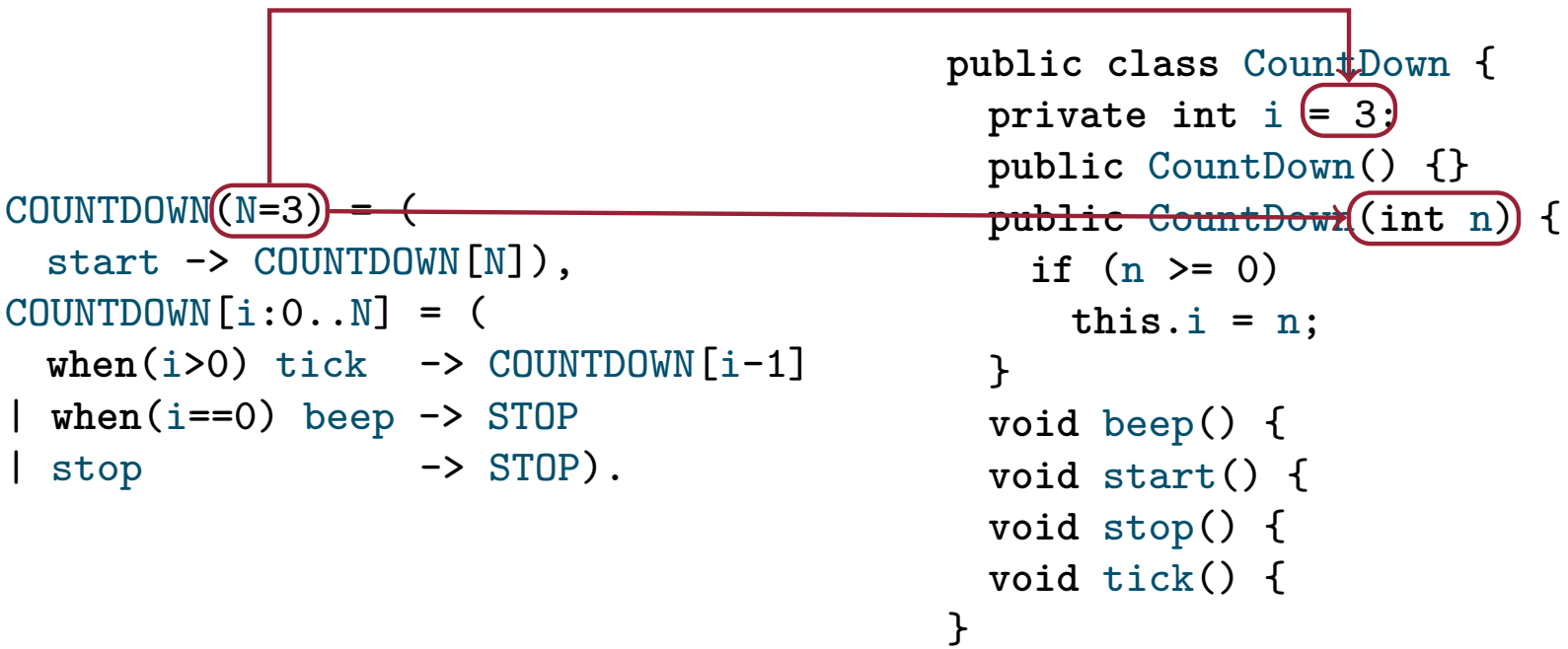


```
COUNTDOWN(N=3) = (
  start -> COUNTDOWN[N]),
COUNTDOWN[i:0..N] = (
  when(i>0) tick  -> COUNTDOWN[i-1]
| when(i==0) beep -> STOP
| stop            -> STOP).
```

```
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 I

- Un processus FSP est une classe
  - il y a toujours un constructeur par défaut
  - chaque paramètre est un paramètre du constructeur



```
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 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

```
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 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 necéssaire
  - 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 necéssaire)
  - 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

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*

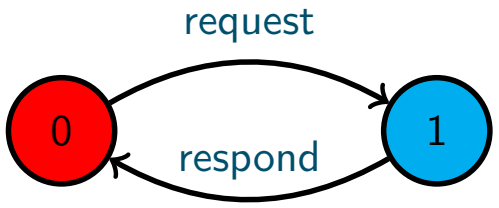
```
RESOURCE = (get -> rel -> RESOURCE).
P = (r1.get -> P1),
P1 = (r2.get -> action -> r1.rel -> r2.rel -> P
    | timeout -> r1.rel -> P).
Q = (r2.get -> r1.get -> action -> r1.rel -> r2.rel -> Q).
||SYS = (p:P || q:Q ||
        {p,q}::r1:RESOURCE ||{p,q}::r2:RESOURCE).
```

Avancement

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).`

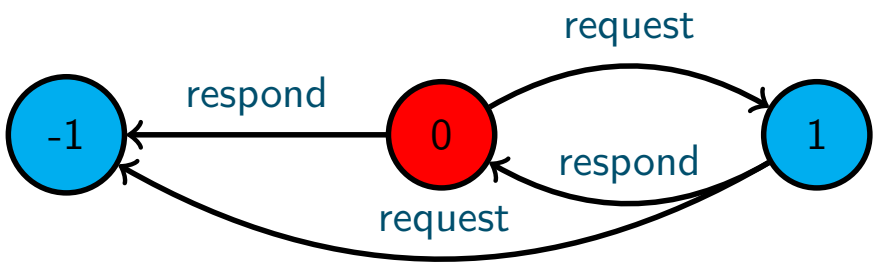
correspond à



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



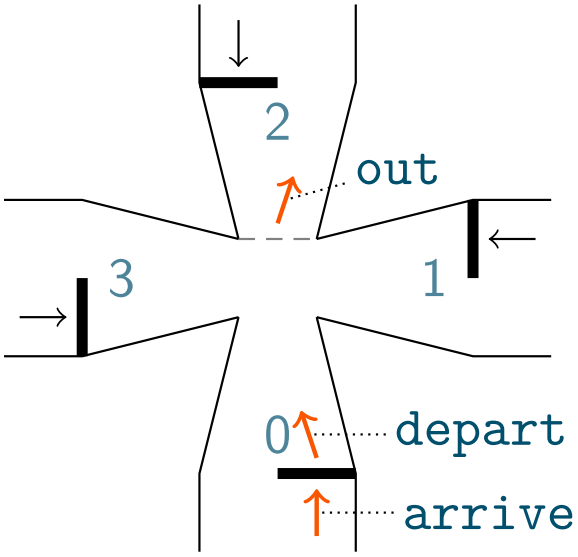
Utilisation des propriétés de sûreté

- 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

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 : $\texttt{term}(P)$
- Calcul des ens. terminaux de (S, A, T, s_0)
  - Pour chaque état, ens. des états accessibles ($\texttt{acces}$)
  - $\forall s \in S, s' \in \texttt{acces}(s) \wedge s \in \texttt{acces}(s') \Rightarrow \exists CFC, \{s, s'\} \subset CFC$
  - $CFC \subset \texttt{term} \Leftrightarrow$ pas de transition sortante de CFC

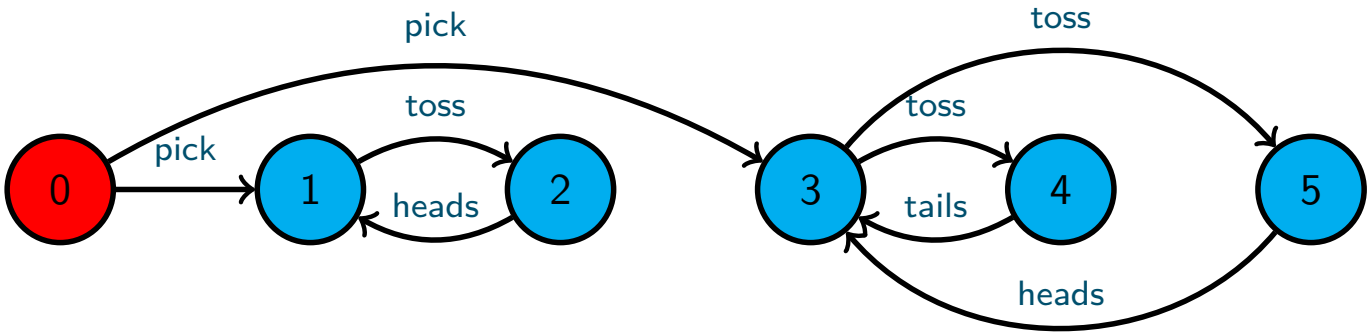
Vérification du progrès

- $\texttt{fireable}(S) = \bigcup_{s \in S} \texttt{fireable}(s)$
- **progress** P = {a1,...an} est vrai pour Q ssi $\forall S_T \in \texttt{term}(Q), \texttt{fireable}(S_T) \cap \{\texttt{a1}, ...\texttt{an}\} \neq \emptyset$
  - 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
   - 1.1 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

Un processus de développement FSP / Java

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