Notes de cours sur le passage au code

INF447

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() {}
```

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

```
  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`.

1.4 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.