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

Notes de cours sur le passage au code

Fondements théoriques du développement des logiciels concurrents

Il y a une correspondance entre modèle FSP et code. Cette correspondance peut être utilisée de deux façons :

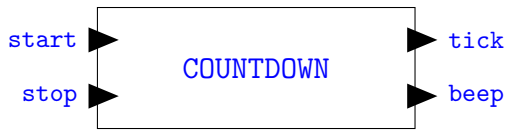
- Après avoir conçu et validé un modèle FSP, on passe au code qui réalise ce modèle.
- Sur un code existant, on peut reconstruire un modèle FSP pour vérifier la correction du programme.

Dans le cadre de notre module, nous n'aborderons que la première correspondance. Nous supposons disposer d'un modèle FSP et nous souhaitons construire un code Java.

Ce document a pour objectif de présenter en détail une méthode systématique pour passer d'un modèle FSP au code Java qui lui correspond. Les différentes étapes de la méthodes sont décrites en les appliquant à un exemple.

1 La structure

Lorsqu'on passe à la réalisation en Java, chaque processus du modèle devient une classe. Ainsi, dans une vue schéma d'architecture, tous les composants deviennent des classes. Toutes leurs actions deviennent des méthodes de ces classes. For example, the following `COUNTDOWN` component has two input actions `start` and `stop` and two output actions `tick` and `beep`.



Ce composant va correspondre à une classe Java avec les quatre méthodes correspondantes. Pour le moment, ces méthodes sont traitées de la même façon, n'ont pas d'argument et ne renvoient rien. Il peut être necéssaire de faire des ajustements de nom pour respecter les conventions de programmation Java. Ainsi, le nom de la classe choisie ici est `CountDown`. Les actions respectant les conventions Java, nous pouvons conserver leurs noms. Il est important de conserver la correspondance entre les noms sous une forme ou sous une autre (ici, nous choisissons un commentaire dans le code Java)..

```
/**
```

```
 * Created by Fabien Dagnat <fabien.dagnat@imt-atlantique.fr> on 10-2017
 *
 * Correspond to the model element COUNTDOWN
 */
public class CountDown {
  void start() {}
  void tick() {}
  void stop() {}
  void beep() {}
}
```

2 La création

Il faut ensuite prévoir la création d'instance de ce composant. En FSP, un composant peut avoir des paramètres d'instanciation, ceux-ci doivent avoir une valeur par défaut. Lors de l'instanciation si aucun paramètre n'est donné, c'est la valeur par défaut qui est utilisée. Par exemple, si le composant ci-dessus a le comportement `COUNTDOWN(N=3) = ...`, il peut être instancié par `COUNTDOWN` (`N` vaut 3) ou par `COUNTDOWN(18)` (`N` vaut 18). Pour conserver un comportement similaire en Java, il va falloir fournir plusieurs constructeurs : le constructeur par défaut (sans argument) et un constructeur par combinaison possible. Le paramètre d'instanciation est constant une fois l'instanciation faite. Nous transformons donc ce paramètre en attribut final. Ici, la valeur par défaut est transformer en une constante pour ne pas apparaître dans le code et donc permettre son changement plus facilement.

Dans la suite, nous ne montrons que les parties du code qui sont impactées par les changements décrits.

```
public class CountDown {
  private static final int N = 3;
  private final int n;
  CountDown() {
    this.n = N;
  }
  CountDown(int n) {
    if (n >= 0)
      this.n = n;
    else
      this.n = N;
  }
}
```

Il est important de noter que le degré de correspondance est sous le contrôle du développeur qui peut choisir de s'aligner ou pas. Ainsi, par exemple, l'ajout de la constante ou la présence d'un constructeur par défaut sont des choix d'ingénierie.

3 Les états

L'étape suivante consiste à faire en sorte que les instances de la classe en cours d'écriture réalisent par leur comportement l'automate. Pour cela, les différents états du composants doivent correspondre à des états de ses objets. Ainsi, chacune des variables d'état d'un de ses états 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).
```

Dans le comportement ci-dessus, il n'y a que « un état » [1], `COUNTDOWN`, qui a une variable d'état `i`. Celle-ci devient une variable d'instance (attribut). D'autres états, peuvent également, si c'est nécessaire, être encodés par des attributs. Dans l'exemple, ci-dessus, il existe deux états non représenté directement par la variable `i` : l'état initial (avant le `start`) et l'état puits final (le `STOP`). Il existe de nombreuses façon de coder ses états. Ici, nous allons utiliser deux booléens. Un qui indiquera si le décompte a commencé (nous ne sommes plus dans l'état initial), `started` et un autre pour l'état final, `stopped`. On arrive ainsi au code suivant.

```
public class CountDown {
  private boolean started = false; // false indicates initial state
  private boolean stopped = false; // true indicates final stopped state
  private int i; // the value encodes the current state (when started and !stopped)
}
```

4 La structure d'automate

L'encodage de l'automate passe par la gestion des changements d'état lors de l'invocation des méthodes. Ainsi, dans le modèle, une action `start` fait sortir de l'état initial. L'action `tick` décrémente `i` et les actions `beep` et `stop` provoque le basculement dans l'état d'arrêt.

Enfin, pour réellement encoder l'automate, il va falloir s'assurer que les méthodes ne sont invoquées que depuis les *bons* états. Pour cela, chacune des méthodes va avoir une précondition qui vérifie l'état courant de l'objet. Si la précondition n'est pas vérifiée plusieurs stratégies sont possibles. Les deux extrêmes sont d'une part ne rien faire ou d'autre part lever une exception. Dans le code ci-dessous, nous avons choisi de lever une exception `WrongState` que nous avons définie.

```
import fsp2java.WrongState;
public class CountDown {
  void start() throws WrongState {
    if (!this.started) {
      this.started = true;
      this.i = n;
      // le code de l'action start
    } else
      throw new WrongState();
  }
```

1. Attention, c'est une construction syntaxique qui correspond à `N`+1 états `COUNTDOWN[0]`, ..., `COUNTDOWN[N]`.

```
  void tick() throws WrongState {
    if (this.started && !this.stopped && this.i > 0) {
      this.i--;
      // le code de l'action tick
    } else
      throw new WrongState();
  }
  void stop() throws WrongState {
    if (this.started && !this.stopped) {
      this.stopped = true;
      // le code de l'action stop
    } else
      throw new WrongState();
  }
  void beep() throws WrongState {
    if (this.started && !this.stopped && this.i == 0) {
      this.stopped = true;
      // le code de l'action beep
    } else
      throw new WrongState();
  }
}
```

5 Les activités

Un processus peut être un objet *actif* qui possède un comportement propre s'exécutant de manière autonome. Cela concrètement signifie qu'il effectue de lui-même des actions (exécute des méthodes). Pour cela, il doit fournir une tâche qui sera exécutée dans un *thread*. Une des façons de le réaliser en Java est de lui faire réaliser l'interface `Runnable` en fournissant la méthode `run` qui décrit alors le comportement actif de l'objet.

Enfin toutes les classes représentant un composant peuvent fournir des méthodes (actions) qui seront appelées par d'autres composants. Ces méthodes font que les objets instances d'une telle classe sont potentiellement partagée. Ils peuvent donc être invoqué depuis plusieurs *thread* et donc simultanément. Il convient donc de protéger l'accès à ses méthodes. La manière la plus simple et systématique étant d'en faire des méthodes synchronisées. Bien sur, si par construction, aucun appel simultané n'est possible cela n'est pas nécessaire.

Ainsi, dans l'exemple, notre compteur est un objet actif car le comportement de comptage (`tick` et `beep`) est interne (il n'est pas déclenché par un objet externe). La classe `CountDown` doit donc réaliser l'interface `Runnable`.

Le contenu de la méthode `run` doit décrire l'enchaînement des actions qui sont autonomes. Dans le code ci-dessous, vous remarquerez que les gardes des actions du modèle se traduisent par des choix dans le corps de la méthode `run`. Ainsi, 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 {
  @Override
```

```
  public void run() {
    try {
      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();
          }
        }
      }
    } catch (WrongState e) {
      System.out.println("Some transition happened in the wrong state!");
    }
  }
}
```

Vous noterez que ce comportement est actif même lorsque la méthode `start` n'a pas été invoqué. Il aurait été possible d'avoir un comportement plus fin. Il aurait été possible de donner accès au *thread* executant le compteur et ainsi la méthode `start` aurait pu démarrer le *thread*. Une autre possibilité aurait été de se bloquer sur un verrou qui serait libéré par la méthode `start` ce qui aurait déclencher le calcul autonome.

6 Le déclenchement des actions

La dernière étape de cette traduction consiste à, pour chaque action, 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. Comme une action en général modifie l'état du processus et donc ses attributs, il est souvent necéssaire de la rendre synchronisée.

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, les actions `start` et `stop` sont des actions « en entrée » alors que les actions `tick` et `beep` sont des actions « en sortie ».

Cela nous amène donc à la version finale suivante.

```
import fsp2java.WrongState;
```

```java
/**
 * Created by Fabien Dagnat <fabien.dagnat@imt-atlantique.fr> on 10-2017
 *
 * Correspond to the model element COUNTDOWN
 */
public class CountDown implements Runnable {
  private static final int N = 3;
  private final int n;
  private boolean started = false; // false indicates initial state
  private boolean stopped = false; // true indicates final stopped state
  private int i; // the value encodes the current state (when started and !stopped)
  CountDown() {
    this.n = N;
  }
  CountDown(int n) {
    if (n >= 0)
      this.n = n;
    else
      this.n = N;
  }
  public synchronized void start() throws WrongState {
    if (!this.started) {
      this.started = true;
      this.i = n;
      System.out.println("start");
    } else
      throw new WrongState();
  }
  private synchronized void tick() throws WrongState {
    if (this.started && !this.stopped && this.i > 0) {
      this.i--;
      System.out.println("tick");
    } else
      throw new WrongState();
  }
  public synchronized void stop() throws WrongState {
    if (this.started && !this.stopped) {
      this.stopped = true;
      System.out.println("stop");
    } else
      throw new WrongState();
  }
  private synchronized void beep() throws WrongState {
    if (this.started && !this.stopped && this.i == 0) {
      this.stopped = true;
      System.out.println("beep");
    } else
      throw new WrongState();
  }
  /* (non-Javadoc)
   * @see java.lang.Runnable#run()
```

```
   */
  @Override
  public void run() {
    try {
      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();
          }
        }
      }
    } catch (WrongState e) {
      System.out.println("Some transition happened in the wrong state!");
    }
  }
}
```

7 Les étapes

Nous avons donc une méthode en six étapes :

1. la structure : identification des classes et de leurs méthodes ;
2. la création : ajout des éléments de créations des instances ;
3. les états : ajout des attributs pour représenter les états ;
4. la structure d'automate : ajout des préconditions aux méthodes ainsi que les transitions ;
5. les activités : ajout des comportements actifs ;
6. le déclenchement des actions : visibilité et protection des méthodes.