Interface des Programmes d'Application, UE INF302

Master Informatique 1
Université Bordeaux 1
mercredi 20 décembre 2006
Aucun document autorisé, durée 3 heures

Justifier clairement et succinctement vos choix, et décrivez brièvement vos algorithmes lorsque vous le jugez nécessaire.

Toute réponse difficile à comprendre sera considérée comme fausse.

Vous pouvez rajouter du code (méthode, donnée ou classe) non demandé à chaque fois que vous le jugerez nécessaire. Ecrivez entièrement ce code, en particulier dans le cas d'exceptions, de classes abstraites ou d'interfaces.

Vous pouvez toujours considérer une classe ou une méthode demandée dans une question précédente comme disponible, même si vous n'avez pas traité cette question. Il vous est également conseillé de lire le sujet entièrement avant de commencer votre travail.

Si vous souhaitez utiliser une classe (ou une méthode particulière) de l'API Java dont vous ne vous souvenez pas du nom, donnez-lui un nom explicite et précisez-le clairement sur votre copie.

Vous pouvez à chaque fois que vous le jugez compréhensible utiliser des abréviations à la place des noms complets de classe et méthode ou encore '...' pour remplacer du code non modifié.

Les instructions `import` et `package` ne seront pas précisées.

Partie 1 - Automate déterministe

On souhaite réaliser une classe `AutomateDeterministe` pour représenter des automates d'états finis déterministes.
Un *automate d'états fini* est un quintuplet $\{V, Q, I, T, F\}$ où

- V est un ensemble de symboles appelé *alphabet*,
- Q est un ensemble fini appelé *ensemble des états*,
- I est un sous-ensemble de Q appelé *ensemble des états initiaux*,
- T est un sous-ensemble de Q appelé *ensemble des états terminaux*,
- F est un sous-ensemble de $Q \times V \times Q$, appelé *ensemble des transitions*.

Le premier état d'une transition est appelé sa *source*, le second sa *destination*, et le symbole de V son *étiquette*.

Un automate est de plus *déterministe* si et seulement si I est un singleton (un seul état initial) et pour tout état e de Q et symbole s de V, il existe au plus une transition de source e et d'étiquette s.

On représente généralement un automate par un graphe orienté, les états étant représentés par les sommets du graphe, les transactions par les arcs étiquetés par

le symbole correspondant. L'état initial sera désigné par une flèche entrante et les états terminaux par des flèches sortantes. Ainsi, l'automate $A = (V = \{a, b\}, Q = \{1, 2, 3\}, I = \{1\}, T = \{1, 2\}, F = \{(1, a, 1), (1, b, 2), (2, a, 3), (2, b, 3), (3, a, 2), (3, b, 1)\}\}$ est représenté par la figure 1.

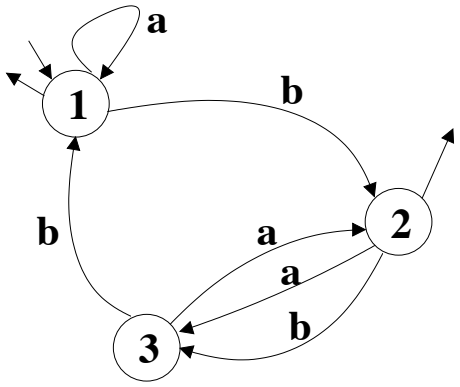

FIG. 1 – Exemple d'automate déterministe

On considère les interfaces `Etat` et `Transition` et la classe `AutomateDeterministe`.

```
public interface Etat {
    public boolean initial();
    public boolean terminal();
}

public interface Transition {
    public Etat source();
    public Etat destination();
    public Object etiquette();
}

public class AutomateDeterministe {

    private Etat etatInitial = null;

    /* Dans la table transitions, a chaque etat e, on associe une table
       dont les valeurs sont les transitions de source e, avec comme cle
       l'etiquette de la transition. */
    private final Map<Etat, Map<Object, Transition>> transitions;

    public AutomateDeterministe(Transition[] transitions) {
        this.transitions = new HashMap<Etat, Map<Object, Transition>>();
```

```
        for (Transition t : transitions) {
            insererEtat(t.source());
            insererEtat(t.destination());
            Map<Object, Transition> map = this.transitions.get(t.source());
            map.put(t.etiquette(), t);
        }
    }

    protected final void insererEtat(Etat e) {
        if (!transitions.containsKey(e)) {
            transitions.put(e, new HashMap<Object, Transition>());
            if (e.initial()) {
                if (etatInitial == null) {
                    etatInitial = e;
                }
            }
        }
    }
}
```

Question 1 Rajouter à la classe `AutomateDeterministe` les méthodes

- `public Etat etatInitial()` qui renvoie l'état initial de l'automate,
- `public Transition transition(Etat source, Object etiquette)` qui renvoie la transition de source `source` et d'étiquette `etiquette` si elle existe, `null` sinon.

Question 2 Rajouter à la classe `AutomateDeterministe` la méthode `public boolean reconnait(Object[] mot)` qui indique si le mot passé en paramètre est reconnu par l'automate. Un mot est dit reconnu par un automate si et seulement si il est possible de passer de l'état initial à un état terminal en utilisant à chaque itération i une transition étiquetée par mot[i]. Par exemple, le mot *abab* est reconnu par l'automate de la figure 1, mais pas le mot *abb*.

Question 3 Modifier la classe `AutomateDeterministe` pour que le constructeur `AutomateDeterministe(Transition[] transitions)` lève une exception

- `TransitionNonDeterministeException` si deux éléments du tableau `transition` ont le même état source et la même étiquette ;
- `EtatInitialNonDeterministeException` s'il existe plusieurs états initiaux parmi les états sources et destination des transitions du tableau `transition` ;
- `EtatInitialInconnuException` s'il n'existe aucun état initial parmi les états sources et destinations des transitions du tableau `transition`.

Remarque : les exceptions peuvent être levées soit directement dans le constructeur, soit dans la méthode `insererEtat(Etat e)`.

Donner également le code de l'exception `TransitionNonDeterministeException`.

Les deux autres classes `EtatInitialNonDeterministeException` et `EtatInitialInconnuException` ne sont pas demandées.

Question 4 Donner le code des classes `TransitionImpl` et `EtatImpl` qui donnent une implémentation respectivement des interfaces `Transition` et `Etat`.

Ces deux classes seront non modifiables, les données nécessaires étant fournies à la construction.

Question 5 On souhaite rendre générique le type des étiquettes des transitions. Par exemple, le nouveau prototype de `etiquette()` dans l'interface `Transition` sera `public T etiquette()`. Donner les classes à modifier et décrire en quelques lignes les modifications à apporter (sans réécrire tout le code).

Les deux parties suivantes ont un même objectif : permettre de rajouter des actions à effectuer lorsque l'on traverse une transition, chacune des deux parties utilisant une technique différente. *On suppose d'autre part qu'il est impossible de modifier le code de la partie 1 !* Ces deux parties sont indépendantes.

Partie 2 - Automate observable

On souhaite créer une classe `AutomateObservable` pour créer un automate qui avertira des observateurs à chaque fois qu'une transition sera traversée.

On rappelle que la méthode `notifyObservers(Object arg)` de la classe `java.util.Observable` n'appelle la méthode `update(Observable o, Object arg)` des observateurs attachés que si la méthode `setChanged()` a été préalablement appelée. Les méthodes de la classe `java.util.Observable` dont vous aurez besoin sont rappelées ci-dessous, ainsi que l'interface `java.util.Observer`.

```
public class Observable {
...
    public void addObserver(Observer o);
    public void notifyObservers(Object arg);
    protected void setChanged();
}

public interface Observer {
   public void update(Observable o, Object arg);
}
```

Question 1 On souhaite qu'un `AutomateObservable` soit un `AutomateDeterministe`. Quel problème cela pose-t-il ?

Pour résoudre ce problème, on va utiliser la solution suivante :

- `AutomateObservable` hérite de `AutomateDeterministe`;
- une instance d'`AutomateObservable` possède comme variable un instance d'une classe qui étend `Observable` et qui rend publique la méthode `setChanged`. Cette variable servira à déléguer le travail dévolu à un observable;
- `AutomateObservable` contient une méthode d'instance `public void ajouterObservateur(Observer o)` qui attache un observateur à l'automate;
- `AutomateObservable` contient une méthode d'instance `public boolean parcours(T[] mot)` qui notifie aux observateurs l'utilisation d'une transition lors du parcours de l'automate pour la reconnaissance du mot `mot`. La transition traversée servira de deuxième paramètre aux méthodes `update` des observateurs notifiés.

Remarque : la méthode `reconnait(T [] mot)` ne sera pas redéfinie pour permettre de tester la reconnaissance d'un mot sans notifier les observateurs.

Le structure du code d'`AutomateObservable` sera donc la suivante :

```
public class AutomateObservable<T> extends AutomateDeterministe<T> {

    public AutomateObservable(Transition<T>[] transitions)
        throws TransitionNonDeterministeException,
        EtatInitialNonDeterministeException, EtatInitialInconnuException {
        ...
    }

    private class ObservableInterne extends Observable {
        public void setChanged() {
            super.setChanged();
        }
    }

    private final ObservableInterne observable = new ObservableInterne();

    public boolean parcours(T[] mot) {...}

    public void ajouterObservateur(Observer o) {...}
}
```

Question 2 Compléter le code d'`AutomateObservable`.

Question 3 Ecrire une classe qui implémente `Observer` et dont le but est d'afficher sur la sortie standard la chaîne de caractère retournée par la méthode `String toString()` appliquée à l'étiquette de la transition traversée.

Partie 3 - Machine d'états finie

On considère l'interface `Action` décrite ci-dessous :

```
public interface Action<T> {
    public void execute(T o);
}
```

Question 1 Ecrire une classe `TransitionAvecAction<T>` qui implémente `Transition<T>` et qui associe une action a à une transition t. t et a seront passées à l'instanciation d'une `TransitionAvecAction<T>`. Les méthodes de `Transition<T>` à implémenter le seront par délégation à t. `TransitionAvecAction<T>` implémentera de plus une méthode `public Etat traverse()` qui appellera la méthode `execute` de a avec l'étiquette de t comme paramètre, avant de renvoyer l'état destination de t.

Question 2 Ecrire une classe `MachineEtatsFinie<T>` dont le constructeur prend en paramètre un tableau de `TransitionAvecAction<T>` et qui implémente une méthode `boolean parcours(T [] mot)` qui parcourt un automate créé à l'aide des transitions en reconnaissant le mot `mot` et qui exécute au passage de chaque transition son action associée.

Afin de pouvoir traiter des mots infinis, on rajoute à la classe `MachineEtatsFinie<T>` une méthode `boolean parcours(Iterator<T> mot)`. L'interface `java.util.Iterator` est rappelée ci-dessous :

```
public interface Iterator<T> {
    public boolean hasNext();
    public T next() throws NoSuchElementException;
    public void remove() throws UnsupportedOperationException;
}
```

Question 3 Rajouter la méthode `boolean parcours(Iterator<T> mot)` à la classe `MachineEtatsFinie<T>`. Réécrire la méthode `boolean parcours(T [] mot)` pour éviter la duplication de code en utilisant l'expression suivante : `Arrays.asList(t).iterator()` qui renvoie un itérateur sur le tableau t.

Question 4 Donner le code nécessaire à l'instanciation d'un `Iterator<String>` correspondant au mot infini $(ab)*$ (suite infinie de "a" et de "b" alternés).

Partie 4 - Conclusion

Question Comparer les deux solutions proposées (utilisation du modèle Observable/Observé et de transitions avec action) en donnant leurs avantages et leurs inconvénients.