

Université d'Evry Val d'Essonne, Master Informatique et Systèmes
Conception et Programmation d'Applications Réparties
TD 1 : Processus Légers

Tarek Melliti & Pascal Poizat

2011-2012

Exercice - 1 *Synchronisation de workflow*

Soient les trois processus suivants :

- P1=A ;D ;F
- P2=B ;X ;E
- P3=C ;G

Nous voulons réaliser le Workflow représenté par la figure 1.

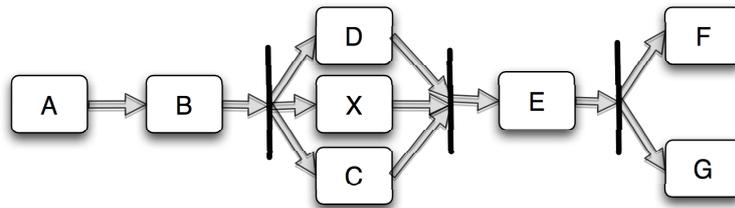


FIGURE 1 – Le workflow des trois processus

Les manières de réaliser un tel workflow sont nombreuses, nous favorisons ici la solution à base de contrôleur vue en cours. Voici le contrôleur proposé :

```
public class JobController {
    boolean done=false;

    synchronized public void jobDone(){
        done=true;
        this.notifyAll();}

    synchronized public void isJobDone(){
        if(!done)
            try {
                wait();
            } catch (InterruptedException e) {e.printStackTrace();}
    }
}
```

1- Étant données trois instances de processus, respectivement de type P1, P2 et P3, de combien d'instances de contrôleurs avez-vous besoin pour réaliser le workflow ? Expliquez.

2- Donnez le code des trois classes P1, P2 et P3 permettant de réaliser le workflow. Pour les actions, par

exemple pour A, utilisez `System.out.println("A")`.

3- Écrivez une `main` permettant de lancer les instances des trois processus.

Exercice - 2 Les TupleSpaces

Linda est un langage de coordination où plusieurs processus interagissent via une mémoire partagée appelée **TupleSpace**. Un **TupleSpace** est un multi-ensemble de tuples (un ensemble où la duplication de tuples est permise). Un tuple est un vecteur de valeurs, par exemple $\langle 25, "toto" \rangle$. Les processus interagissent en effectuant trois actions :

- **Écriture** : notée $ECR(t)$ avec t un tuple. Cette action permet d'ajouter le tuple t dans le **TupleSpace**.
- **Extraction** : L'action d'extraction se fait en utilisant des patrons de tuple appelés "templates" et notés \hat{t} . Les templates sont des tuples composés de valeurs et de variables et jouent le rôle de requêtes. L'action d'extraction, $EXT(\hat{t})$ renvoie un tuple t du **TupleSpace** qui "match" le template et le supprime du **TupleSpace**. Le *matching* se fait selon les règles suivantes (i) il faut que le tuple ait une taille égale à celle du template, et (ii) il faut que chaque couple de valeurs partageant la même position dans le tuple et dans le template soient égaux.
- **Lecture** : L'action de lecture, $LEC(\hat{t})$, est similaire à celle de l'extraction à la différence près que le tuple qui a *matché* le template n'est pas supprimé du **TupleSpace** par cette action.

t	\hat{t}	<i>match?</i>
$\langle "a" \rangle$	$\langle x_1, x_2 \rangle$	Faux
$\langle "a" \rangle$	$\langle "a" \rangle$	Vrai
$\langle "a", 3 \rangle$	$\langle x_1, x_2 \rangle$	Vrai ($x_1 = "a"$ et $x_2 = 3$)
$\langle "a", 3 \rangle$	$\langle "b", x \rangle$	Faux
$\langle "a", 3 \rangle$	$\langle "a", x \rangle$	Vrai ($x = 3$)

TABLE 1 – Exemples de fonctionnement du *matching*

Les actions de lecture et d'extraction sont des actions bloquantes. Le processus exécutant une telle action sur un **TupleSpace** qui ne contient aucun tuple satisfaisant le template reste bloqué jusqu'à ce qu'un tuple satisfaisant le template soit ajouté. Bien évidemment, si un tuple qui vient d'être ajouté satisfait plusieurs processus bloqués sur une action de lecture ou d'extraction, alors un seul de ces processus obtiendra le tuple et les autres resteront bloqués.

On désigne par P_i les processus. Chaque processus est une suite d'actions d'écriture, d'extraction et/ou de lecture (ici désignés par des a). La grammaire des processus est la suivante :

$$P ::= a.P \mid \mathbf{0} \text{ où } \mathbf{0} \text{ désigne le processus terminé.}$$

On désigne par $T = \{t_1 \dots t_n\}$ le **TupleSpace**. Un programme **Linda** L est composé d'un **TupleSpace** et d'un ensemble de processus s'exécutant en parallèle. L est noté $L = (T, P_1 \parallel \dots \parallel P_n)$ où \parallel représente le constructeur de parallélisme.

$R1 = a.P \xrightarrow{a} P$
$R2 = \frac{P_i \xrightarrow{ECR(t)} P'_i}{(T, P_1 \parallel \dots \parallel P_i \parallel \dots) \xrightarrow{ECR_i(t)} (T' = T \cup \{t\}, P_1 \parallel \dots \parallel P'_i \parallel \dots)}$
$R3 = \frac{P_i \xrightarrow{EXT(\hat{t})} P'_i \wedge \exists t \in T, match(t, \hat{t}) = vrai}{(T, P_1 \parallel \dots \parallel P_i \parallel \dots) \xrightarrow{EXT_i(\hat{t})} (T' = T \setminus \{t\}, P_1 \parallel \dots \parallel P'_i \parallel \dots)}$
$R4 = \frac{P_i \xrightarrow{LEC(\hat{t})} P'_i \wedge \exists t \in T, match(t, \hat{t}) = vrai}{(T, P_1 \parallel \dots \parallel P_i \parallel \dots) \xrightarrow{LEC_i(\hat{t})} (T, P_1 \parallel \dots \parallel P'_i \parallel \dots)}$

TABLE 2 – Sémantique opérationnelle de **Linda**

- Le tableau 2 représente l'ensemble des règles de la sémantique opérationnelle d'un programme **Linda** :
- La règle $R1$ spécifie que si un processus est défini par une action a suivie d'un autre processus P , alors ce processus peut exécuter a et se réduire à P .

- La règle *R2* spécifie que si l'un des processus du programme **Linda** peut écrire un tuple alors tout le programme **Linda** peut exécuter une action d'écriture indiquée par le numéro du processus pour se transformer en un autre programme **Linda** où le **TupleSpace** est augmenté par le tuple écrit et où le processus qui a exécuté l'action est remplacé par sa réduction.
- La règle *R3* spécifie qu'à chaque fois qu'un processus veut extraire un tuple satisfaisant un template et que le **TupleSpace** contient un tuple qui satisfait le template demandé alors le programme **Linda** exécute une action d'extraction indiquée par le numéro du processus pour se transformer en un autre programme **Linda** où le tuple *matché* est supprimé du **TupleSpace** et le processus est remplacé par sa réduction.
- La règle *R4* spécifie qu'à chaque fois qu'un processus veut lire un tuple satisfaisant le template et que le **TupleSpace** contient un tuple qui satisfait cette template alors le programme **Linda** exécute une action de lecture indiquée par le numéro du processus pour se transformer en un autre programme **Linda** où le **TupleSpace** reste inchangé et le processus concerné est remplacé par sa réduction.

Soit le programme **Linda** $L = (\{t_1\}, P_1 || P_2 || P_3)$ avec :

- $P_1 = EXT(\hat{t}_1).ECR(t_3).0$
- $P_2 = EXT(\hat{t}_1).0$
- $P_3 = EXT(\hat{t}_3).ECR(t_1).0$

1- Dessinez le système de transition des états du programme (en appliquant les règles) en indiquant à chaque état du programme la valeur du **TupleSpace**. Notez bien que pour cette question $match(t_i, \hat{t}_j)$ est *vrai* si $i = j$ et *faux* sinon.

2- Est-ce que L se bloque ? Caractériser le blocage d'un programme **Linda** dans le cas général.

Linda sous java

On veut maintenant programmer le **TupleSpace** en java, **JTupleSpace**. Pour simplifier, les **Tuples** et **Templates** seront des vecteurs d'objets Java ce qui nous permet d'utiliser la classe **Vector** pour les représenter et les méthodes $equals(Object o)$, héritée de la class **Object** pour comparer un à un leurs contenus. Les processus seront représentés par des **Threads**. Attention, le **TupleSpace** étant un objet partagé par un ensemble de processus, il est sujet à des accès concurrents.

3- Indiquez les propriétés de sûreté que le fonctionnement de la classe **JTupleSpace** doit garantir.

Il est clair que le fonctionnement d'un programme **Linda** peut être assimilé à un problème de type *producteurs-consommateurs* à la différence que les consommateurs ne peuvent pas consommer n'importe quel produit (ici tuple) mais seulement les produits qui satisfont une condition (ici le matching avec le template).

4- La figure 2 contient le code incomplet de la classe **JTupleSpace**. La méthode *match* est fournie. Complétez le code des méthodes *ECR*, *EXT* et *LEC*.

5- On veut utiliser le **TupleSpace** pour programmer le problème des philosophes. Le **TupleSpace** est la table et elle contient au départ n tuples représentant les n fourchettes (chaque fourchette est représentée par un objet **Integer** portant le numéro de la fourchette). Programmez la classe **Philosophe** dont le comportement serait de prendre la fourchette qui porte son numéro moins 1, ensuite de prendre la fourchette qui porte son numéro modulo n (n est la taille du problème, i.e le nombre des philosophes) avant de les libérer.

6- En utilisant les classes **JTupleSpace**, **Philosophe** et **Integer** écrivez une *main* pour un problème de taille trois ($n = 3$).

```
2 import java.util.*;
3 public class JTupleSpace {
4     HashSet contenu=new HashSet();
5     public synchronized Vector EXT(Vector v)
6     { /* à compléter... */ }
7
8     public synchronized Vector LEC(Vector v)
9     { /* à compléter... */ }
10
11    public synchronized void ECR(Vector v)
12    { /* à compléter... */ }
13
14    public Vector match(Vector template)
15    {Iterator it=contenu.iterator();
16      Vector res=null;
17      while (it.hasNext() && res==null)
18      {Vector v=(Vector) it.next();
19        if (v.size()==template.size())
20        {res=v;
21          for (int i=0;i<template.size();i++)
22          {if(template.get(i)!=null && !v.get(i).equals(template.get(i)))
23            {res=null;break;}
24          }
25        }
26      }
27      return res;
28    }
29 }
```

FIGURE 2 – Code incomplet de la classe **JTupleSpace**