TP Process Model

Thierry GOUBIER

UBO, 22/01/2021

Sujet du TP :

- Q1: Expérimenter avec l'environnement de simulation. Essayer toutes les méthodes exemples de la classe TauCExamples, et créez un réseau à vous (en copiant depuis un des exemples déjà présent)
- Q2: Utiliser et décrire ce qu'il se passe avec un réseau sans contraintes (à rythmes désynchronisés) (une tâche émet 3 tokens, une tâche lit 1 token). Donner la taille maximum des FIFOS dans le réseau que vous avez créé
- Q3: Faire le même réseau qu'en Q2, avec une ou des fifos de taille fixe.
- Q4: Écrire une transposition de matrice 3 par 3
  - Écrire un réseau transposant la matrice $\begin{vmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{vmatrix}$ en la matrice $\begin{vmatrix}1 & 4 & 7\\2 & 5 & 8\\3 & 6 & 9\end{vmatrix}$. Ce réseau est composé d'un émetteur (Sender), d'un récepteur (Reader), d'un Split et d'un Join.
- Q5: Écrire un produit matrice vecteur, avec la matrice $\begin{pmatrix}1 & 4 & 7\\2 & 5 & 8\\3 & 6 & 9\end{pmatrix}$ et le vecteur $\begin{pmatrix}5\\2\\3\end{pmatrix}$.=
- Q6: Écrire une multiplication de matrice avec comme matrices $\begin{pmatrix}1 & 2 & 0\\4 & 3 & -1\end{pmatrix}\times\begin{pmatrix}5 & 1\\2 & 3\\3 & 4\end{pmatrix}$ sous la forme d'une grille deux dimensions, une matrice rentrant par le haut de la grille (les colonnes), une matrice rentrant par la gauche de la grille (les lignes).

Méthode :

Vous définirez un package TPProcessModel (Monticello / +Package). Vous sauvegarderez ce package dans le répertoire package-cache de votre Pharo6.1-win répertoire.
Vos réponses au TP seront réalisées sous la forme de méthodes (q1, q2, q3, q4, q5, q6) de la classe TPProcessModel dans votre package TPProcessModel. Vous devrez rendre la dernière version du package (le fichier TP_ProcessModel-PrenomNom.<version>.mcz ou différent suivant la manière dont vous l'aurez nommé).

- Date limite pour rendre le TP : 7/02/2021 à minuit.
- Pénalités de retard : 2 points par jour
- Personne à qui rendre : Erwan Fabiani

Il est aussi possible de rendre un rapport avec un copier/coller des méthodes (q1, q2, q3, q4, q5, q6) qui créent les réseaux pour les différentes questions.

Documentation du Framework

Obtention de tauC-model

Le framework utilisé s'appelle tauC-Model. Il est hébergé sur github, https://github.com/ThierryGoubier/ProcessModel. Une nouvelle image et la dernière version du framework peuvent être installés de la manière suivante :

- Télécharger la version stable de Pharo sur http://www.pharo.org. (pharo launcher), installez là et démarrez le Pharo launcher.
- Dans new, official distributions, choisir de créer une image Pharo 7 (old stable).
- Une fois téléchargée, sélectionner et démarrer cette image (bouton triangle vert - launch)
- Ouvrir un workspace / playground
- Saisir le code suivant et l'exécuter (do it) :
  - Metacello new baseline: 'TauCModel'; repository: 'github://ThierryGoubier/ProcessModel'; load

Structure du code

Les exemples de code sont les méthodes de la classe TauCExamples. Ils s'exécutent en ouvrant un Workspace, et avec un Transcript ouvert, de manière à observer les messages. Dans le workspace, il faut exécuter le code suivant (sélectionner, puis, menu bouton de droite >> Do it) :

TauCExamples join

où 'join' peut être remplacée par une des méthode de la classe TauCExamples.

Il est aussi possible de sélectionner l'intérieur de la deuxième commentaire de la méthode et de faire 'doit'

Il est aussi possible de mettre le curseur dans le pragma <example> et de demander, sur le menu contextuel, 'run example...'

Principes :

- Chaque réseau de processus suppose une instance de TauCSystem comme environnement (ligne 8).
- Un réseau de processus se compose de tâches (lignes 9, 19 et 26)
- Chaque tâche a des ports entrants (lignes 20, 27) et des ports sortants (lignes 10, 21)
- Chaque tâche a au moins un état (ligne 11) ;
  - chaque état comprend trois éléments
    - Un nom (ligne 11) → Une structure d'I/O (ligne 11)
      - Une structure d'I/O est un nom de port, ou une array #(nom de port quantité), ou une array #(structure d'I/O structure d'I/O ….) (ligne 22)
      - La liste des IO doit être dans l'ordre où elles sont déclarées dans les paramètres du bloc (ligne 29 avec les lignes 34 et 37).
    - Un bloc (ligne 12 à 17)
      - Il commence par ses paramètres s'il en a (:i , ligne 12), et ses I/O (:send, ligne 12)
      - Ses I/O sont des tableaux, dans lesquels il peut lire ou écrire (ligne 13). La taille du tableau associé à une IO est défini dans la structure d'IO associée à l'état.
  - Un état peut indiquer quel sera son état suivant avec next : #nom de l'état (ligne 16) ou next : with : (ligne 17) (si cet état a des paramètres)
    - Un état spécial #exit permet de terminer une tâche. Il faut obligatoirement un état #main. Si l'état #main a des paramètres, il faut signaler à la tâche son état initial avec ses paramètres initiaux (ligne 18).
- Les tâches sont connectées via leurs ports, en point à point avec une fifo (lignes 30, 31) avec

le message >>

- Il est possible, via size:, de spécifier la taille maximale de la fifo (sinon elle est infinie). Attention : spécifier la taille des fifos permet de créer des deadlocks. L'environnement émettra une TauCException s'il détecte un deadlock apparent.

- La simulation ouvre son IHM avec exploreInRoassal (ligne 33). le message fullyConnected permet de vérifier que tous les liens entrants et sortants des tâches sont connectés à une autre tâche.
- La simulation avance via des commandes step sur l'environnement. Chaque step exécute l'état désigné des tâches présentes, si, pour chaque tâche les données sont disponibles sur les ports en entrée et si suffisamment d'espace est disponible sur les ports en sortie, conformément à ce que spécifie la liste d'I/O pour l'état de la tâche. Les boutons "step" et "run" dans l'IHM font la même chose.
- Les messages apparaissent sur la console de l'IHM (le Transcript) en envoyant le message tcLog à des chaînes de caractères (lignes 14, 25 et 29).

```
1. sendIdentityReceiveMemory
2.      "A send -> transmit -> receive with a sender which has a memory. "
3.
4.      "TauCExamples sendIdentityReceiveMemory"
5.
6.      <example>
7.      | s sender transmit reader |
8.      s := TauCSystem new.
9.      sender := (s newTask: 'sender')
10.             out: #send;
11.             state: #main -> #send
12.                     do: [ :i :send |
13.                     send at: 1 put: i.
14.                     ('Send ' , i printString) tcLog.
15.                     i < 3
16.                             ifTrue: [ sender next: #main with: i + 1 ]
17.                             ifFalse: [ sender next: #exit ] ].
18.     sender next: #main with: 0.
19.     transmit := (s newTask: 'transmit')
20.             in: #read;
21.             out: #send;
22.             state: #main -> #(#send #read)
23.                     do: [ :send :read |
24.                     send at: 1 put: read first.
25.                     ('Transmitted ' , read first printString) tcLog ].
26.     reader := (s newTask: 'reader')
27.             in: #read;
28.             state: #main -> #read
29.                     do: [ :read | ('Received ' , read printString) tcLog ].
30.     sender send >> transmit read.
31.     transmit send >> reader read.
32.     s fullyConnected.
33.     s exploreInRoassal
34.
```

- Deux tâches spécifiques existent : split et join (ligne 43, s newJoin:).
  - split effectue un 1 vers plusieurs, join effectue plusieurs vers un.

- Ces tâches ont des tableaux de ports qui sont connectés soit éléments par éléments, soit tableu vers tableau (lignes 62, 63) via >>.
- split a un port entrant #input, et un tableau de ports #output en sortie
- join a un tableau de ports entrant #input, et un port sortant #output
- La création d'une de ces tâches indique le nombre d'entreés (Join) / le nombre de sorties (Split) en paramètre du #newJoin:/#newSplit1to: respectivement (ligne 43)
- Ces tâches fonctionnent en round robin, prenant K éléments, et activant leur entrée / leur sortie tour à tour (la première, puis la suivante, … et reboucler sur la première)
  - Par défaut, le nombre K d'éléments est 1, mais il est possible de le spécifier via #newJoin:to1by: / #newSplit1to:by: (le paramètre à by: est le K).

- Enfin, il est souhaitable que le réseau soit entièrement connecté (ligne 65) et on peut simuler le réseau avec la GUI (ligne 66).

```
35.     join
36.             "Example: a join with two senders and one reader. "
37.
38.             "TauCExamples join"
39.
40.             <example>
41.             | s join sender1 sender2 reader |
42.             s := TauCSystem new.
43.             join := s newJoin: 2.
44.             sender1 := (s newTask: 'sender1')
45.                     out: #send;
46.                     state: #main -> #send
47.                             do: [ :send |
48.                             send at: 1 put: 5.
49.                             'S1 send 5' tcLog ].
50.             sender2 := (s newTask: 'sender2')
51.                     out: #send;
52.                     state: #main -> #send
53.                             do: [ :send |
54.                             send at: 1 put: 6.
55.                             'S2 send 6' tcLog ].
56.             reader := (s newTask: 'reader')
57.                     in: #read;
58.                     state: #main -> #(#read 2)
59.                             do: [ :read |
60.                             ('Reader received ' , read first printString , ' '
61.                                     , read second printString) tcLog ].
62.             {sender1 send.
63.             sender2 send} >> join input.
64.             join output >> reader read.
65.             s fullyConnected.
66.             s exploreInRoassal.
67.             ^ s
```