Université de Biskra
Département d'informatique
Master 2 : GLSD
Logique de réécriture

Objets distribués en Maude

14/11/2023

- Classes et Objets
- Messages
- Création et destruction des objets
- Full Maude : un support de plus pour l'OO.
- Exemple : dîner de philosophes

Classes et Objets

- **Class**
  - «un paterne / modèle »
  - Valeurs / attributs
  - Méthodes
  - *Exemple* : le concept Personne avec les attributs **age** et **situation civile** et des méthodes pour **mariage.**
- **Objet**
  - Instance d'une classe
  - Name/adresse/identificateur d'objet
  - *Exemples* : les objets avec des identificateurs Walid et Warda avec leurs âges et état civile.

Objets en Maude

- Un objet peut être représenté par le terme :

$$< O : C \mid att_1: val_1, ..., att_n: val_n >$$

où :

- O est le nom /référence/ identificateur de l'objet ,
- C est la classe de l'objet O.
- att_1 à att_n sont les attributs de l'objet.
- val_1 à val_n sont les valeurs actuelles de l'objet.

- Exemple :

< "Walid" : Person | age: 63, status: married >

Déclaration des classes

- Supposant qu'on a la sorte Oid d'un identificateur objet.
- Classes, qui ont des instances de la forme :

$$< O : C \mid att_1: val_1, ..., att_n: val_n >$$

Peuvent être déclarées:

```
sorts Oid Object .
op <_: C | att1:_, ..., attn:_> :
Oid s1 ... sn -> Object [ctor] .
```

En maude.

où **si** est la sorte de l'attribut **atti** .

- Exemple :

```
sort Oid Object . subsort String < Oid .
op <_: Person | age:_, status:_> :
Oid Nat Status -> Object [ctor]
```

Configurations

- Un état dans un système distribué peut être modélisé comme un multi-ensemble d'objets et de messages.

```
sorts Object Msg Configuration .
subsort Object Msg < Configuration .
op none : -> Configuration [ctor] .
op___ : Configuration Configuration
-> Configuration [ctor assoc comm id: none] .
```

- Exemple :

```
< "Walid" : Person | age: 63, status: married >
< "Chomsky" : Person | age: 81, status: married >
```

Est un terme de sorte **Configuration**

Méthodes

- Les méthodes sont représentées par des règles de réécriture:

```
var X : Oid . var N : Nat . var S : Status .
rl [birthday] :
   < X : Person | age: N, status: S > =>
   < X : Person | age: s N, status: S > .
```

Par Congruence on a :

```
< "Walid" : Person | age: 63, status: married >
< "Chomsky" : Person | age: 81, status: married >
< "Warda" : Person | age: 23, status: single >
```

Réécrit à :

```
< "Walid" : Person | age: 63, status: married >
< "Chomsky" : Person | age: 82, status: married >
< "Warda" : Person | age: 23, status: single >
```

Concurrence

```
< "Walid" : Person | age: 62, status: married >
< "Chomsky" : Person | age: 80, status: married >
< "Warda" : Person | age: 22, status: single >
```

Peut être réécrit à :

```
< "Walid" : Person | age: 63, status: married >
< "Chomsky" : Person | age: 81, status: married >
< "Warda" : Person | age: 23, status: single >
```

dans un seul pas concurrent (par congruence sur des transitions locales __)

- Dans un système distribué, un objet modélise un **composant** du système distribué.

Plusieurs objets dans une règle

On peut avoir plusieurs objets dans une seule règle :

```
rl [engagement] :
   < X : Person | age: N, status: single >
   < X' : Person | age: N', status: single >
   =>
   < X : Person | age: N, status: engaged(X') >
   < X' : Person | age: N', status: engaged(X) > .
```

- Deux objets dans la “soupe” se trouvent ensemble!
- Souvent appelée **communication asynchrone** (handshake).

Plusieurs objets dans une règle

Exemple

On utilisant la règle de l'engagement :

```
< "Walid" : Person | age: 30, status: single >
< "Wassim" : Person | age: 29, status: single >
< "Warda" : Person | age: 28, status: single >
```

Réécrit à :

```
< "Walid" : Person | age: 30, status: engaged("Warda") >
< "Wassim" : Person | age: 29, status: single >
< « Warda" : Person | age: 28, status: engaged("Walid") >
```

Tant que la configuration est un multi-ensemble avec les attributs assoc et comm .

Messages (I)

Un état est une "soupe" d'objets (Object) et messages (Msg)

- Messages sont des termes de sort Msg .
- Messages sont met dans la "soupe" et ainsi ils peuvent être lus.
- Communication Asynchrone.
- Exemple : séparation

```
op separate : Oid -> Msg [ctor] .

rl [init-separation] :
< X : Person | age: N, status: married(X') >
=>
< X : Person | age: N, status: separated(X') >
separate(X') .
```

Messages (II)

Exemple :

```
< "Walid" : Person | age: 50, status: married(“Warda") >
< "Nassima" : Person | age: 47 status: married(“Wassim") >
< "Wassim" : Person | age: 40, status: married("Nassima") >
< "Warda" : Person | age: 46, status: married("Walid") >
```

Réécrit à :

```
< "Walid" : Person | age: 50, status: married(“Warda") >
< "Nassima" : Person | age: 47 status: married(“Wassim") >
< "Wassim" : Person | age: 40, status: married("Nassima") >
< "Warda" : Person | age: 46, status: separated("Walid") >
Separate("Walid“)
```

Cet état égale à …..

Messages (III)

```
Separate("Walid")
< "Walid" : Person | age: 50, status: married("Warda") >
< "Nassima" : Person | age: 47 status: married("Wassim") >
< "Wassim" : Person | age: 40, status: married("Nassima") >
< "Warda" : Person | age: 46, status: separated("Walid") >
```

Messages (IV)

Lecture (et consommation) du message :

```
rl [sep2] :
  separate(X)
  < X : Person | age: N, status: married(X') >
  =>
  < X : Person | age: N, status: separated(X') >
```

Le message n'existe pas après cette transition : l'état décrit dans le diapositive précèdent devient :

```
< "Walid" : Person | age: 50, status: separated("Warda") >
< "Nassima" : Person | age: 47 status: married("Wassim") >
< "Wassim" : Person | age: 40, status: married("Nassima") >
< "Warda" : Person | age: 46, status: separated("Walid") >
```

Echange de Messages (I)

On a aussi :

```
ops marry? yes no : Oid -> Msg [ctor] .

rl [ask] :
    < X : Person | age: N, status: engaged(X') >
    =>
    < X : Person | age: N, status: engaged(X') >
    marry?(X') .

rl [yes] :
    marry?(X)
    < X : Person | age: N, status: engaged(X') >
    =>
    < X : Person | age: N, status: married(X') >
    yes(X') .
```

Echange de Messages (II)

```
rl [No] :
    marry?(X)
    < X : Person | age: N, status: engaged(X') >
    =>
    < X : Person | age: N, status: Single >
    no(X') .
```

Plus, il faut rajouter les règles pour recevoir les message **yes** et **no** .

Modèle plus élaboré

Un modèle plus adapté au mariage:

```
rl [marriageOK] :
    < X : Person | age: N, status: engaged(X') >
    < X' : Person | age: N', status: engaged(X) >
    =>
    < X : Person | age: N, status: married(X') >
    < X' : Person | age: N', status: married(X) > .

rl [marriageNO] :
    < X : Person | age: N, status: engaged(X') >
    < X' : Person | age: N', status: engaged(X) >
    =>
    < X : Person | age: N, status: single >
    < X' : Person | age: N', status: single > .
```

Création de nouveaux objets : naissance d'un enfant

```
crl [birth] :

      < X : Person | age: N, status: married(X') >
    =>
      < X : Person | age: N, status: married(X') >
      < X + " jr." : Person | age: 0, status: single >
    if N > 15 /\ N < 55 .
```

Exemple

Exemple:

```
< "Walid" : Person | age: 800, status: married("Wissam") >
< "Wissam" : Person | age: 20, status : married("Walid") >
```

```
< "Walid" : Person | age: 800, status: married("Wissam") >
< "Wissam" : Person | age: 20 status : married("Walid") >
< "Wissam jr." : Person | age: 0, status: single >
```

Destruction d'objets

Destruction/suppression d'objets

rl [death] :

```
< X : Person | age: N, status: S > => none .
```

Exemple:

```
< "Moubarek" : Person | age: 83, status: married >
< "ELGUEDHAFI" : Person | age: 69, status: single >
```

```
< "Moubarek" : Person | age: 83, status: married >
```

FULL MAUDE (I)

- **Full Maude** est un prototype d'une interface Orientée Objets en Maude.
- Full maude offre un support syntaxique pour présenter :
  - Classes
  - Sous classes
  - Messages
  - …etc,
- Full Maude commence par lire le fichier **full-maude.maude** soit en donnant la commande **maude full-maude.maude**, ou en ajoutant la ligne **load full-maude.maude** dans votre fichier Maude.

FULL MAUDE (II)

- Tout les Modules du full-maude doivent être inclus dans parenthèses.
- omod … endom pour les modules orientée objets.
- (omod X is … endom)
- (rew ….)
- (search ….)

Classes dans FULL MAUDE

- Importer le module CONFIGURATION dans tous les fichiers omod
- Déclaration de calsse :

```
class C | att1 : s1, ..., attn : sn .
```

- Exemple:

```
class Person | age : Nat, status : Status .
```

- Terme

```
< "Walid" : Person | age : 40, status : single >
```

Sous Classes dans FULL MAUDE

- Sous-classes

```
class B | att'1 : s'1, ..., att'k : s'k .
Subclasse B < C .
```

- sous-calsse **B** hérite tous les attributs et les règles de la supper classe **C** .
- On peut avoir un héritage multiple :

Subclasse B < C D E .

- Remarque : pas de surcharge de règles dans pour les sous-classes.

Messages dans FULL MAUDE

- Les Messages sont déclarés avec le mot clé msg :

```
msg sep : Oid -> Msg .
```

  - Maude assume que le premier argument est le récepteur du message.
  - Avertissement (Warning) si le message n'a pas de paramètre .

Règles dans full-maude

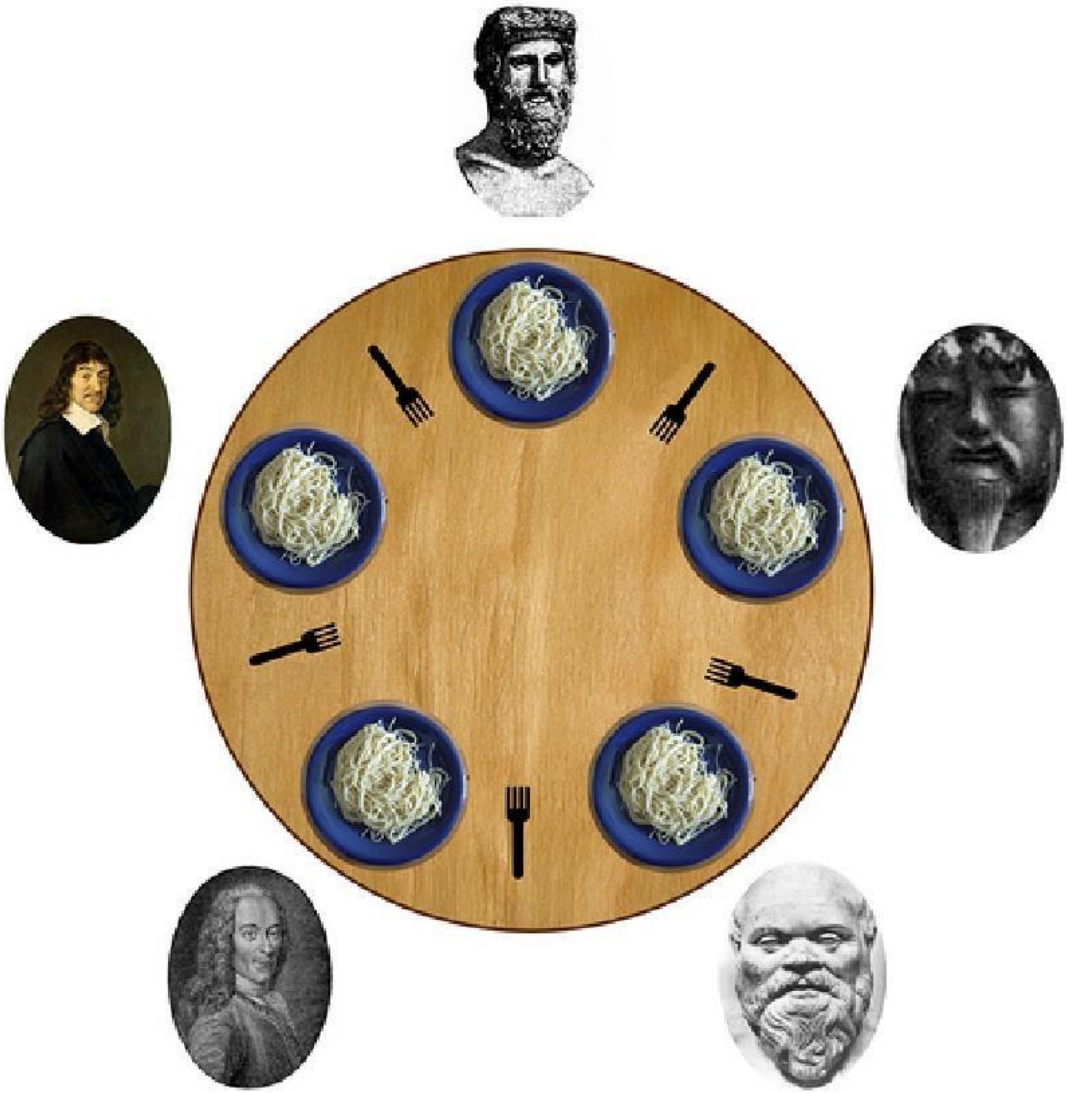
- Les attributs qui ne changent pas de valeurs peuvent être ignorés dans la partie droite de la règle.
- Les attributs qui n'influencent pas sur le nouveau état peuvent être ignorés (cachés) dans la partie gauche.

```
crl [engage] :
     < X : Person | age : N, status : single >
     < X' : Person | age : N', status : single >
   =>
     < X : Person | status : engaged(X') >
     < X' : Person | status : engaged(X) >
   if N > 15 /\ N' > 15 .

rl [death] : < X : Person | > => none .

 rl [birthday] : < X : Person | age : N > =>
                 < X : Person | age : s N > .
```

Dîner de philosophes



Philosophes

- Un Philosophe est modélisé par le terme :

```
< i : Philosopher | state : s, noOfSticks : j, noOfEats : k >
```

- Où i est le nombre de philosophe, j est le nombre de Fourchette que le philosophe détient, et k est le nombre de fois que le philosophe à mangé.

```
class Philosopher | state : State, noOfSticks : Nat,
                    noOfEats : Nat .
subsort Nat < Oid . --- Object names are numbers!
sort State .
ops thinking hungry eating : -> State [ctor] .
```

Modélisation de baguettes

- Une fouchette peut par exemple être modélisée par un message:

```
msg chopstick : Nat -> Msg .
```

- Si le "message" chopstick(3) est dans la configuration, donc la baguette 3 est disponible.
  - Ça peut être vu comme un message pour les philosophes 2 et 3 .
- On peut aussi modéliser une fouchette par un objet.

ÉTAT INITIAL

- L'état initial :

```
op initState : -> Configuration .
eq initState =
   chopstick(1) chopstick(2) chopstick(3)
   chopstick(4) chopstick(5)
   < 1 : Philosopher | state : thinking, noOfSticks : 0,
                       noOfEats : 0 >
   < 2 : Philosopher | state : thinking, noOfSticks : 0,
                       noOfEats : 0 >
   < 3 : Philosopher | state : thinking, noOfSticks : 0,
                       noOfEats : 0 >
   < 4 : Philosopher | state : thinking, noOfSticks : 0,
                       noOfEats : 0 >
   < 5 : Philosopher | state : thinking, noOfSticks : 0,
                       noOfEats : 0 > .
```

RÈGLES (I)

- Un philosophe dans un état de pensé (thinking) devient faim :

```
vars I J K : Nat .


rl [hungry] :
      < I : Philosopher | state : thinking >
    =>
      < I : Philosopher | state : hungry > .
```

RÈGLES (II)

- Un philosophe faim prend sa première fourchette:

```
crl [grabFirst] :
    chopstick(J)
    < I : Philosopher | state : hungry,
                        noOfSticks : 0 >
  =>
  < I : Philosopher | noOfSticks : 1 >
  if I can use stick J .

  op right : Nat -> Nat . --- The "right"chopstick
  eq right(I) = if I == 5 then 1 else I + 1 fi .
  op _can'use'stick_ : Nat Nat -> Bool .
  eq I can use stick J = (I == J) or (J == right(I)) .
```

RÈGLES (III)

- Prendre la 2ème fourchette et commence à manger

```
crl [grabSecond] :
      chopstick(J)
      < I : Philosopher | noOfSticks : 1,
                          noOfEats : K >
      =>
      < I : Philosopher | state : eating,
                          noOfSticks : 2,
                          noOfEats : K + 1 >
      if I can use stick J .
```

RÈGLES (IV)

- Un philosophe satisfait (termine à manger)

```
crl [stopEating] :

      < I : Philosopher |sate : eating >
      =>
      < I : Philosopher | state : thinking,
                          noOfSticks : 0 >
       chopstick(I) chopstick(right(I)) .
```

ANALYSE

- Est il possible que tous meurent de faim ?

```
Maude> (search [1] initState =>! C:Configuration .)

Solution 1
C:Configuration <-
< 1 : Philosopher | noOfEats : 0, noOfSticks : 1,
                    state : hungry >
< 2 : Philosopher | noOfEats : 0, noOfSticks : 1,
                    state : hungry >
< 3 : Philosopher | noOfEats : 0, noOfSticks : 1,
                    state : hungry >
< 4 : Philosopher | noOfEats : 0, noOfSticks : 1,
                    state : hungry >
< 5 : Philosopher | noOfEats : 0, noOfSticks : 1,
                    state : hungry >
```

Chaque philosophe détient une fourchette et il attend pour que la 2ème fourchette soit disponible.

VARIATION

- Prendre les deux fourchette à la fois
  - Est-ce qu'on peut avoir une deadlock (interblocage) ?
  - Est-ce que tout les philosophes auront la possibilité pour manger ? (non famine)

exercice : proposer une solution pour ce modèle.

Merci