Faculté des Sciences Exactes et des Sciences de la Nature et de la Vie
Département d'Informatique
Option: GLSD/ M2
Module : Logique de Réécriture et ses application

MODÉLISATION DES COMMUNICATIONS EN MAUDE

Rappels

- Définition de classe en Maude

```
class Car | brand : String, licencePlate : String, year :
                                        Nat .
```

- Un objet est un terme

```
< "My car" : Car | brand : "Toyota", year : 1991,
                licencePlate : "413 1909" >
```

- Un état dans un système distribué: multi-ensemble d'objets et de messages

- Échange simple de messages :
  - Envoi : ajouter un message à la configuration
  - Réception : supprimer un message

Communication

- Plusieurs types de communications, due au :
  - Plusieurs types de moyens physique de communication
    - PCs connectés à Internet
    - Satellite diffusant (broadcastent) des signaux TV,
    - Rayonnement du fond cosmique
    - Nœuds dans un réseaux sans fils
    - Un système contenant plusieurs types d'unités
  - Modélisation à plusieurs niveaux d'abstraction

Abstraction

- Abstraction : omettre / cacher plusieurs détailles que possible

  - Toutefois, les systèmes sont complexes même sans détailles
  - On a toujours plusieurs niveau d'abstraction souhaitables pour un système donnée.

Quelques Formes de Communication

- Communication synchrone
- Communication asynchrone
  - Réception de message en désordre / en ordre.
  - Unicast/multicast/broadcast
  - Variables partagées
  - Réception de messages Fiables ou non fiables (perte/corruption/duplication de messages).
  - Capacité limitée
  - Temps de transmission
  - …
  - Combinaison des éléments précédents.

- On s’intéresse à la modélisation des formes de communication de haut niveau en Maude
- Exemples très simples pour illustrer les difficultés avec la communication synchrone
  - Comment faire la séparation (dans le modèle de la vie de personnes) avec l’envoi de messages?
- Ensuite: utilisation de ces primitives pour la modélisation de différents protocoles/systèmes,

Maude et communication

- Maude est élégant et flexible : On peut naturellement modéliser :
  - Plusieurs formes de communications différentes ?
  - Sure plusieurs niveaux d'abstraction

Communication Synchrone (I)

- Objets se synchronisent afin de faire des actions en collaboration:

```
crl [engagement] :
  < 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 .
```

Communication Synchrone (II)

- Configuration est une « soupe » d'objets et de messages qui peuvent se rencontre :

```
< "Salim" : Person | age : 29, status : single >
< "Wahid" : Person | age : 27, status : single >
< "Meriem" : Person | age : 27, status : single >
```

Égale à :

```
< "Salim" : Person | age : 29, status : single >
< "Meriem" : Person | age : 27, status : single >
< "Wahid" : Person | age : 27, status : single >
```

Réécrit à

```
< "Salim" : Person | age : 29, status : engaged("Meriem") >
< "Meriem" : Person | age : 27, status : engaged("Salim") >
< "Wahid" : Person | age : 27, status : single >
```

Communication Asynchrone (I)

- Communication entre objets sans synchronisation
  - Lettres, message vocale, e-mail, des transactions bancaires etc.
  - Variables partageables
    - Mettre à jour la même variables partageable (billet d'avion ?)
  - Plusieurs type de messages de communication
  - Réception de messages dans leurs ordre d'envoie ?
    - Si "oui" en ordre, ex: communication Object-to-objet (lien) ou bien all-to-object (mail vocal) ?
  - Perte de messages ? Message à un ou plusieurs récepteurs ?

Communication Asynchrone: désordre en réception

- Messages reçus dans un ordre différent que l'ordre d'envoi

  - Messages échangés sur internet, les lettres aussi
  - Il faut prendre en compte tous les ordres possible de réception
  - Exemple (application): séparation pas échange de messages

Problème de séparation (I)

Initier la séparation :

```
msg separate : Oid -> Msg .

rl [sep] :
  < X : Person | status : married(X') >
=>
  < X : Person | status : separated(X') >
separate(X') .
```

Réception du message:

```
rl [acceptSep] :
separate(X)
  < X : Person | status : married(X') >
=>
  < X : Person | status : separated(X') > .
```

Problème de séparation (II)

maintenant deux séparés peuvent êtres présent à la cour pour devenir divorcés:

```
rl [divorce] :
  < X : Person | status : separated(X') > .
  < X' : Person | status : separated(X) > .

=>
  < X : Person | status : single ) > .
  < X' : Person | status : single ) > .
```

Transportation de messages

Échange de messages : les messages ‘circulent’ due à la nature assoc et comm de _ _:

```
< "Wissem" ... married("Jihad") >
< "Zina" ... single >
< "Jihad" ... married("Wissem") >
```

```
< "Wissem" ... separated("Jihad") >
separate("Jihad")
< "Zina" ... single >
< "Jihad" ... married("Wissem") >
```

≡

```
< "Wissem" ... separated("Jihad") >
< "Zina" ... single >
separate("Jihad")
< "Jihad" ... married("Wissem") >
```

```
< "Wissem" ... separated("Jihad") >
< "Zina" ... single >
< "Jihad" ... separated("Wissem") >
```

Problème de séparation (III)

Ça peut prendre du temps, et éventuellement d'autres choses peuvent avoir lieu:

```
< "Jihad" ... married("Wissem") >
< "Wissem" ... separated("Jihad") >
separate("Jihad")

————→ birthday ————→ birthday ————→

< "Jihad" ... separated("Wissem") >
< "Wissem" ... separated("Jihad") >
```

Les deux décident en même temps de se séparer:

```
< "Jihad" ... married("Wissem") >
< "Wissem" ... married("Jihad") >

  ————→
< "Jihad" ... separated("Wissem") > separate("Jihad")
< "Wissem" ... separated("Jihad") > separate("Wissem")
```

Problème de séparation (IV)

- Il faut prendre en considération que les deux peuvent décider de se séparer:

```
rl [sep2] :

  separate(X) .
  < X : Person | status : separated(X') >
=>
  < X : Person | >
```

- Toujours des problèmes! Les messages peuvent être lus (consommés) en retard

Problème de séparation (V)

- Problème: un message non lu peut détruire le prochain mariage :

```
< "Wissem" ... married("Jihad") >
< "Jihad" ... married("Wissem") >
< "Zina" ... single >
```

```
< "Wissem" ... separated("Jihad") >
separate("Jihad")
< "Jihad" ... married("Wissem") >
separate("Wissem")
< "Zina" ... single >
```


```
< "Wissem" ... single >
separate("Jihad")
< "Jihad" ... single >
separate("Wissem")
< "Zina" ... single >
```

Problème de séparation (V)

```
< "Wissem" ... single >
separate("Jihad")
< "Jihad" ... married ("Zina") >
separate("Wissem")
< "Zina" ... married ("Jihad") >
```

PB : "Jihad“ peut maintenant lire le message separate("Jihad") !

Analyse

- Un message a aussi un émetteur
  - Un message ancien de séparation peut détruire un nouveau mariage entre Jihad et Wissem !
- Le problème peut être découvris en exécutant la spécification (à vous de vérifier)
- Une solution ?

Solution correcte (I)

- Utilisant «separate» comme «acknowledgement»

```
rl [initSep] :
    < X ... married(X') >
  =>
    < X ... waitSep(X') > separate(X') .

rl [accSep] :
  separate(X)
  < X ... married(X') >
  =>
  < X ... separated(X') > separate(X') .

rl [accSep2] :
  separate(X)
  < X ... waitSep(X') >
  =>
  < X ... separated(X') > .
```

Solution correcte (II)

- Solution n'est pas intuitive ?
- Simple de constater que c'est une solution correcte ?
  - Faire des analyses en Maude !
- Ceci est un protocole qui montre aux époux comment se séparer efficacement.

Communication Asynchrone (ordonnée)

On peut avoir :

1. Messages entre la même paire d'objets doivent être reçus dans leurs ordre d'envoi.
2. Tous les message vers un objet donnée doivent être lu dans leurs ordre d'envoi (answering machine).
3. Tous les messages doivent être lus dans leurs ordre d'envoi. On

s'intéresse à 1 :

Comment assurer que les messages sont lus dans leurs ordre d'envoi ?

- Version 1 : chaque message a un numéro de séquence .
- Version 2 : utilisation des liens (Link Objects) entre chaque paires de nœuds (modéliser des liens (canaux)) .

Numéros de séquences (I)

- Émetteur ajoutera un incrément à chaque message envoyé au récepteur :

```
   < A ... | sentNo : (B, N) ... >
=>
   < A ... | sentNo : (B, s N) ... >
m(A, B, s N)
```

- Récepteur lu uniquement le prochain message envoyé par l'émetteur :

```
m(A, B, s M)
   < B ... | recvdNo : (A, M) ... >
=>
   < B ... | recvdNo : (A, s M) ... >
```

Numéros de séquences (II)

- Garder trace à plusieurs numéros de séquences
- Blocage en cas de perte de messages
- Difficile à découvrir la perte de messages
- Besoin d’envoi plus simple et plus modulaire
- Généralement cette techniques n’est pas très utilisée
- Un protocole qui assure la lecture ordonnée dans un moyen de transmission non ordonné (unordered transmission medium)
  - Non pas un modèle de transmission ordonnée

Liens (I)

- Lien (Link): modélise un lien (unidirectionnels /sens unique) entre deux nœuds, représenté par un objet d'une classe,

```
op _to_ : Oid Oid -> Oid [ctor] .
class Link | content : MsgList .
```

- `Content` : est une liste de messages qui voyagent de l'émetteur au récepteur.
  - utilisant `_::_` operateur de concaténation dans la liste
  - «début» de la liste est le premier élément à gauche
  - le nom d'un objet lien doit avoir le format

senderObject to receiverObject

Envoi et réception via des liens

- Un message m de l'objet O à l'objet O′ est envoyé par l'insertion du m dans la fin de la liste du lien O to O′, par l'objet O.

```
rl [send-m] :
   < O : ... >
   < O to O′ : Link | content : L >
   =>
   < O : ... >
   < O to O′ : Link | content : L :: m > .
```

- O' lire le prochain message de O par la suppression du premier élément dans le lien :

```
rl [read-m] :
   < O′ : ... >
   < O to O′ : Link | content : m :: L >
   =>
   < O′ : ... >
   < O to O′ : Link | content : L > .
```

Liens (II)

- Lien (bidirectionnel):peut être modélisé avec deux liens/liaisons unidirectionnels

- Il faut inclure les objets liaisons dans l'état

```
< A : C | ... >
< A to B : Link | content : nil >
< B to A : Link | content : nil >
< B : C' | ... >
```

- On peut avoir des buffers avec capacités limités.

Concurrence dans les Liens

- Un « objet » peut uniquement participer en une seule réécriture à la fois
  - On peut pas lire et écrire dans le lien en même temps,
  - Parfois ce type de concurrence et souhaitable
- Idée :
  - Un lien se compose de deux partie : avant et arrière
  - Insérer un élément en arrière et supprimer celui de l'avant.
  - Si l'objet qui insère en arrière et celui qui supprime en arrière sont différent donc la tâche peut s'effectuée en concurrence.
  - une équation déplace les messages d'arrière en avant.
  - Tant que l'application d'une équation n'est pas un pas de réécriture, on utilises pas des pas de réécriture supplémentaire.

Concurrence sur les Liens (II)

- Équivalence

```
< O to O' : Link | cont : ML :: ML' >
```

=

```
< O to O' : LinkFront | front : ML >
< O to O' : LinkBack | back : ML' >
```

- Déplacer les éléments d'arrière en avant :

```
eq < O to O' : LinkFront | front : ML >
     < O to O' : LinkBack | back : ML' >
=
< O to O' : LinkFront | front : ML:: ML' >
< O to O' : LinkBack | back : nil >  .
```

Concurrence sur les Liens (III)

- Envoi

```
rl [send-M] :
< O ... >
< O to O' : LinkBack | back : ML' >
=>
< O ... >
< O to O' : LinkBack | back : ML' :: M > .
```

- Lecture à partir d'un lien

```
rl [readLink] :
< O' ... >
< O to O' : LinkFront | front : M :: ML
=>
< O' ... >
< O to O' : LinkFront | front : ML >
```

Envoi à plusieurs objets

- Envoi d'un message à plusieurs récepteurs en un seul pas

- Exemples :
  - Envoi des invitations pour fête de mariage comme b-post.
  - Envoi des images TV à tout les inscrits à une chaine du sport
  - Envoi des résultats des Matchs sur SMS a tout les inscrits.

- Appelé Multicast

- Simple pour la communication non ordonnée, difficile pour la communication à base de liens.

Multicast non ordonné (I)

- L'émetteur a une liste des inscrits :

```
class Sender | multicast-group : OidSet, ...


sort OidSet .
subsort Oid < OidSet .
op none : -> OidSet [ctor] .
op _;_ : OidSet OidSet -> OidSet [assoc comm id: none ctor] .
```

- Envoi de message `m(from, to, … )` à plusieurs.
- On doit avoir

```
op m : Oid OidSet ... -> Configuration .
```

- En outre,

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

Multicast non ordonné (II)

- Idée: m(o, o1; o2; . . . ; on, . . .) équivalant à :

```
m(o, o1, ...) m(o, o2, ...) ... m(o, on, ...)
```

- L'équations assurent cette opération de copie.

```
eq m(O, none, ...) = none .

ceq m(O, O' ; OS, ...) =  m(O, O', ...) m(O, OS, ...)
if OS =/= none .
```

- Envoi d'un message m à tout les éléments du groupe Multicast :

```
rl [multicast-m] :
< O : Sender | multicast-group : OS, ... >
=>
< O : Sender | > m(O, OS, ...) .
```

Multicast non ordonné (III)

- inconvénient: deux équations pour chaque type de message !
- Solution : utilisation des «wrappers/ enveloppes» pour les messages :
  - Tous les messages ont la forme `msg m from o to o'` où m est le contenu du message :

```
msg msg_from_to_ : Msg Oid Oid -> Msg [ctor] .
```

  - Un wrappers pour Multi-messages :

```
op multimsg_from_to_ : Msg Oid OidSet -> Configuration .

eq multimsg M from O to none = none .
eq multimsg M from O to O' ; OS =
                              (msg M from O to O')
                              (multimsg M from O to OS) .
```

Broadcast

- Envoyer un message à tout le monde dans le réseau
  - `Ex: Satellites, Radio`
  - Difficile à modéliser avec des messages uniques.

Modélisation de Broadcaste (I)

- L'état « Global » a la forme { configuration }
- Règle de broadcaste :

```
rl [broadcast] :

< O : Node | ... >

=>

< O : Node | ... >

broadCast M from O .
```

Modélisation de Broadcaste (II)

- Le messages Broadcasté est transformé à un simple message dans chaque nœud dans le système :

```
var REST : Configuration .

eq { < O : Node | > (broadCast M from O) REST} =
   { < O : Node | > (broadCast M from O to REST)} .

eq broadCast M from O to (< O' : Node | > REST) =
   < O' : Node | > (msg M from O to O')
   (broadCast M from O to REST)

eq broadCast M from O to (M' REST) =
   M' (broadCast M from O to REST)

eq broadCast M from O to none = none .
```

Modélisation de Broadcaste (III)

- L'émetteur n'a pas besoin de connaître aucun récepteur.

- La concurrence est garantie si on utilise l'équation

```
eq broadcast M from O to
              ((broadcast M' from O') REST) =
            (broadcast M from O to REST)
            (broadcast M' from O') .
```

Perte de Messages

- Les messages peuvent être perdus (sur internet, mail, etc.)
  - Modéliser la perte pour les messages qui nuage dans la configuration et ceux transportés via les liens.
- Les messages peuvent être dupliquées
  - abstraction de renvoyer les messages si un message n'a pas été reçu dans un certain délai.
- Les messages peuvent être corrompus
  - Ce n'est pas fréquent ces jours.
  - Souvent est considéré comme une perte de messages
  - On ignore la corruption de message.

Perte d'un seul Message (I)

Les messages peuvent être perdus (sur internet, mail, etc.)

- Modéliser la perte et la duplication de messages dans une configuration.
- Tentative 1

```
var M : Msg .
rl [lose-msg] : M => none .
rl [dupl-msg] : M => M M .
```

- Chaque message peut être dupliqué ou perdu

- Inconvénient : message dans un objet ou dans un wrapper/enveloppe peut être perdu :

```
< R : Rcvr | msgRead : m1 m2 > => < R : Rcvr | msgRead : m2 >

msg m3 from A to B => msg none from A to B
```

Perte d'un seul Message (II)

- Tentative 2 : uniquement les messages dans les wrappers peuvent être dupliqués ou perdus.

  - Tous les messages qui peuvent être perdu ont la forme msg from_to_ wrapper.

```
var M : Msg . vars O O' : Oid .

rl [lose-msg] : msg M from O to O' => none .

rl [duplicate-msg] :
msg M from O to O' =>
            (msg M from O to O') (msg M from O to O') .
```

- On peut aussi avoir un wrapper spéciale qui indique que le message peut être perdu / dupliqué.

Perte d'un seul Message (III)

- Tentative 3 : un « Requin » nage dans la configuration et qui mange ou duplique les messages dans la configuration.

```
class Destroyer .

rl [destroy] :
M < O : Destroyer | >
=>
< O : Destroyer | > .

rl [duplicate] :
M < O : Destroyer | >
=>
< O : Destroyer | > M M .
```

Perte de messages dans les liens (I)

- On peut définir une sous classe LossyLink pour les liens qui peuvent perdre des messages:

```
class LossyLink .
subclass LossyLink < Link .

vars ML ML' : MsgList . var M : Msg .
vars SOURCE DEST : Oid .

rl [lose-msg] :
< SOURCE to DEST : LossyLink | content : ML :: M :: ML' >
=>
< SOURCE to DEST : LossyLink | content : ML :: ML' > .
```

Duplication de messages dans les liens

```
class DuplLink . subclass DuplLink < Link .

rl [duplMsg] :
< SOURCE to DEST : DuplLink | content : ML :: M :: ML' >
=>
< SOURCE to DEST : DuplLink | content : ML :: M :: M :: ML' > .

class UnrelLink .
subclass UnrelLink < LossyLink DuplLink .
```

- On peut spécifier un système avec des liens « UnrelLink ». On peut aussi le tester avec des objets de type Link, LossyLink, or DuplLink si on veut pas perdre ou dupliquer des messages

Perte de messages dans les liens (II)

- Les messages peuvent être perdus si le lien est plein. On doit rajouter des attributs additionnels : max (capacité du lien) et size (nombre d'élements dans le lien) et on doit modifier l'envoi.

```
crl [send-OK] :
      < O : ... >
      < O to O' : Link | content : L, max : N, size : N' >
=>
      < O : ... >
      < O to O' : Link | content : L :: m , size : N' + 1 >
      if N' < N .

rl [send-full] :
< O : ... >
< O to O' : Link | max : N, size : N >
=>
< O : ... >
< O to O' : Link | > .
```