**Partie V: Systèmes réactifs et concurrents**

Un système est dit *réactif* ...

- lorsqu'il maintient une interaction avec son environnement
- lorsque son comportement est dirigé par les événements ("*event-driven*")

Un système est dit *fonctionnel* (ou *transformationel*)

- lorsqu'il produit à partir de données d'entrée un ensemble de sorties puis *termine* son exécution

Exemples:

- Transformat.: Traitement en lots, compilateurs, etc.
- Réactifs: réseaux, OS, contrôle de procédés, systèmes embarqués, interfaces personnes–machines, etc.

Module `Spec =>` module réactif: l'activation d'un `MESSAGE` est lié à un événement (*réactif*)

Un système est dit *concurrent* lorsque son comportement est déterminé par *l'interaction* de plusieurs processus qui coopèrent et communiquent entre eux

## 15 Les transactions `Spec`

Rôle d'une transaction `Spec` = Spécifier l'ordre dans lequel les divers messages reçus par un module pourront être acceptés et traités

Permet de spécifier non pas *un* message donné mais plutôt *les diverses séquences possibles de messages pouvant être traités par le module*

Spécification de transaction ≈ expression régulière

### 15.1 Le problème du producteur et du consommateur avec tampon partagé

Problème classique de la programmation concurrente = un processus producteur et un processus consommateur échangent de l'information par l'intermédiaire d'un *tampon partagé*

- Le producteur génère certaines informations et les ajoute dans le tampon
- Le consommateur retire l'information du tampon et la traite
- Les deux processus travaillent de façon concurrente, à des vitesses différentes

#### 15.1.1 Première version du tampon partagé: Version séquentielle

```
MACHINE tampon{ t: type, taille: nat SUCH THAT taille > 0 }
  STATE( contenu: sequence{t} )
  INVARIANT length(contenu) <= taille
  INITIALLY contenu = []

  MESSAGE ajouter( x: t )
    WHEN length(contenu) < taille
      TRANSITION contenu = *contenu || [x]
    OTHERWISE
      REPLY EXCEPTION tampon_plein

  MESSAGE retirer
    WHEN length(contenu) > 0
       REPLY( x: t ) WHERE x = contenu[1]
       TRANSITION contenu = *contenu[2..length(*contenu)]
     OTHERWISE
       REPLY EXCEPTION tampon_vide
END
```

Problème si le producteur travaille beaucoup plus rapidement que le consommateur =

- le tampon va devenir rapidement plein
- une exception sera générée à chaque ajout

Problème similaire si c'est le consommateur qui travaille beaucoup plus rapidement

#### 15.1.2 Deuxième version du tampon partagé: Avec `TRANSACTION`s

```
MACHINE tampon{ t: type, taille: nat SUCH THAT taille > 0 }
  STATE( contenu: sequence{t} )
  INVARIANT length(contenu) <= taille
  INITIALLY contenu = []

  MESSAGE ajouter( x: t )
    TRANSITION contenu = *contenu || [x]

  MESSAGE retirer
     REPLY( x: t ) WHERE x = contenu[1]
     TRANSITION contenu = *contenu[2..length(*contenu)]

  TRANSACTION ajout_ou_retrait =
     IF WHEN length(contenu) < taille -> ajouter
     |  WHEN length(contenu) > 0      -> retirer
     FI
END
```

Clause `TRANSACTION ajout_ou_retrait`: indique qu'on ne pourra recevoir un message `ajouter` qu'à la condition que le tampon ne soit pas plein.

Idem pour `retirer`, mais avec un tampon vide.

### 15.1.3 Troisième version du tampon partagé: Un tampon de taille variable

```
MACHINE tampon{ t: type }
  STATE( taille: nat, contenu: sequence{t} )
  INVARIANT length(contenu) <= taille
  INITIALLY taille = 0

  MESSAGE definir_taille( n: nat SUCH THAT n > 0 )
    TRANSITION taille = n

  MESSAGE ajouter( x: t )
    TRANSITION contenu = *contenu || [x]

  MESSAGE retirer
    REPLY( x: t ) WHERE x = contenu[1]
    TRANSITION contenu = *contenu[2..length(*contenu)]

  MESSAGE liberer
    TRANSITION taille = 0

  TRANSACTION
    utilisation_tampon =
      definir_taille;
      DO  WHEN length(contenu) < taille -> ajouter
      |   WHEN length(contenu) > 0      -> retirer
      OD;
      IF WHEN length(contenu) = 0 -> liberer FI
END
```

Autre version du tampon partagé:

- La taille peut varier de façon dynamique (message `definir_taille`)
- La taille ne peut être modifiée que si `liberer` a été appelé
- `liberer` ne peut être traitée que si le tampon est vide

## 15.2 Forme générale d'une spécification de TRANSACTION

- Action de base
  = nom de message ou de transaction
- Action avec garde
  = `WHEN` *condition* `->` *action*
  Note: La garde peut référer aux arguments ou à l'état
- Séquence d'actions
  = actions séparées par ";"
- Choix
  = `IF choix1 | ... | choixk FI`
- Répétition
  = `DO choix1 | ... | choixk OD`
- Combinaison parallèle
  = groupe d'actions sans séparateur

## 15.3 Une machine distributrice de biscuits et muffins

```
MACHINE distributrice_biscuits_muffins
  IMPORT montant FROM montant

  STATE( montant_recu: montant )
  INVARIANT 0.0 <= montant_recu <= 1.00
  INITIALLY montant_recu = 0.0

  MESSAGE piece25
    TRANSITION montant_recu = *montant_recu + 0.25

  MESSAGE piece100
    TRANSITION montant_recu = *montant_recu + 1.00

  MESSAGE bouton_biscuit
    SEND envoyer_biscuit TO distributeur
    SEND retourner_monnaie( monnaie: montant ) TO distributeur_monnaie
      WHERE monnaie = *montant_recu - 0.75
    TRANSITION montant_recu = 0.0

  MESSAGE bouton_muffin
    SEND envoyer_muffin TO distributeur
    TRANSITION montant_recu = 0.0

  TRANSACTION
    achat_biscuit_muffin =
      IF  IF piece25; piece25; piece25          | piece100 FI; bouton_biscuit
      |   IF piece25; piece25; piece25; piece25 | piece100 FI; bouton_muffin
      FI
END
```

Machine distributrice

- Biscuits = 0.75$; muffins = 1.00$
- Accepte uniquement les pièces de 0.25$ et 1.00$
- Rend la monnaie
- N'accepte aucune pièce après 1.00$