Partie V

Systèmes réactifs et concurrents

Introduction

Un système est dit *réactif* lorsqu'il maintient une interaction constante avec son environnement et lorsque son comportement est dirigé par les événements ("*event-driven*"). Les événements sont liés soit à des stimuli internes ou externes, soit à des contraintes liées à l'écoulement du temps. L'objectif d'un système réactif n'est donc pas de produire un ultime résultat final mais bien d'interagir avec son environnement.

Par opposition, un système est dit *fonctionnel* ou *transformationel* (en anglais, "*functional*" ou "*transformational*") lorsqu'il produit à partir de données d'entrée un ensemble de sorties puis *termine* son exécution. De tels systèmes sont aussi dit "*input-output driven*" (plutôt que "*event-driven*").

Les systèmes de traitement en lots ("*batch*") sont des systèmes fonctionnels. Par contre, des réseaux de télécommunication, des systèmes d'exploitation, des systèmes de contrôle de procédés, des systèmes embarqués, des interfaces personnes–machines, etc., sont plutôt des systèmes réactifs.

Dans le cadre de l'étude de la notation `Spec`, nous avons déjà examiné de nombreux exemples de systèmes réactifs. En fait, la notion `Spec` de `MESSAGE` est liée directement à la notion d'événement (voir Chap. 8, section 8.5): la réception d'un message envoyé par l'environnement ou par un autre module représente, dans la terminologie de `Spec`, un événement *réactif*.

Un système est dit *concurrent* lorsque le comportement du système dans son ensemble est déterminé par l'interaction de plusieurs processus qui coopèrent et communiquent entre eux (soit par des variables partagées, soit par des messages), donc qui travaillent ensemble "*en même temps*" pour atteindre un objectif global commun.

La notion de système concurrent est implicitement présente dans les notions `Spec` de modules communiquant par l'intermédiaire de `MESSAGE`s: un système est généralement formé de plusieurs machines ou objets (représentant divers sous-systèmes) qui échangent entre eux des messages dans le but de réaliser le comportement global du système. Toutefois, comme on le verra dans le prochain chapitre (Chap. 15), certains concepts `Spec` supplémentaires peuvent être utiles pour mieux définir le comportement de systèmes véritablement concurrents. Plus précisément, au Chap. 15, nous introduirons la notion de *transaction atomique*, notion qui permettra de *contraindre* l'ordre dans lequel un module pourra *accepter* et traiter les divers messages qu'il peut recevoir.

La notion de `TRANSACTION` en `Spec` permettra, comme on le verra bientôt, de spécifier non pas uniquement les propriétés d'une opération donnée (spécification d'un message avec pré-/post-condition), mais plutôt les diverses séquences possibles de messages pouvant être traités par un module. En `Spec`, ces séquences de messages sont décrites à l'aide d'expressions régulières augmentées de gardes (conditions logiques). Au Chap. 16, on étudiera une notation complètement différente pour la description et spécification des séquences possibles d'événements associés à un système réactif et concurrent. Plus précisément, nous étudierons le langage `LOTOS`, un langage de style fonctionnel (un changement d'état sera modélisé par un appel récursif créant un processus distinct avec un nouvel état) permettant la description du comportement d'un système réactif et concurrent à l'aide d'*expressions de comportement*, lesquelles expressions indiquent de façon explicite les actions *observables* pouvant être effectuées par un processus.

Par la suite, au Chap. 17, nous verrons comment décrire, non pas les séquences d'actions possibles, mais plutôt les propriétés de ces séquences. Comme il s'agira de propriétés portant sur des événements survenant à des instants différents, on utilisera alors une forme différente de logique appelée *logique temporelle*.

Chapitre 15

Les transactions Spec

15.1 Le problème du producteur–consommateur avec tampon partagé

Un problème classique de programmation concurrente est celui du producteur–consommateur. La version que nous allons examiner est celle où deux processus concurrents communiquent entre eux par l'intermédiaire d'un tampon partagé. L'un de ces processus, le producteur, produit certaines informations et les ajoute dans le tampon. L'autre processus, le consommateur, extrait ces informations du tampon et les traite. [1] Le tampon, quant à lui, sert d'intermédiaire de communication entre les deux processus, s'assurant de transmettre les informations — du producteur vers le consommateur — dans l'ordre où elles sont reçues (*FIFO*).

15.1.1 Première version du tampon partagé: Version séquentielle

Une première version d'un tel tampon FIFO est présentée à la figure 15.1. Nous allons tenter de voir pourquoi ce tampon, valable dans un modèle purement séquentiel, serait inacceptable dans le cas d'un système avec processus concurrents.

Le premier point à souligner est que, dans un système concurrent, il n'est pas possible de connaître ou fixer la vitesse à laquelle les divers processus vont s'exécuter. Par exemple, dans le cas du producteur et du consommateur, il est tout à fait possible que le producteur soit parfois plus rapide, parfois plus lent que le consommateur; ceci signifie que le tampon pourrait devenir plein ou, au contraire, être vide. Que se passe-t-il dans le cas de la machine `tampon` de la figure 15.1 si le producteur tente d'ajouter un élément alors que le tampon est plein? Dans ce cas, une exception sera signalée; cette exception devra alors être traitée par le producteur, lequel devra récupérer de cette exception et devra essayer ensuite à nouveau de transmettre son information au tampon. Or, si le consommateur n'a toujours retiré aucun élément du tampon, une exception sera signalée à nouveau. Si le consommateur est plus lent que le producteur, le processus producteur travaillera donc inutilement, essayant constamment d'ajouter un élément et récupérant de l'exception jusqu'à ce que le consommateur ait enfin retiré un élément du tampon.

Lorsqu'une situation semblable se produit, c'est-à-dire tampon plein pour le producteur (resp. tampon vide pour le consommateur), il serait certainement préférable que le tampon refuse de traiter tout message d'ajout tant et aussi longtemps que le tampon reste plein (resp. refuse tout message de retrait tant que le tampon est vide).

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

En `Spec`, la notion de transaction atomique peut être utilisée pour spécifier l'ordre dans lequel les divers messages envoyés à un module peuvent être acceptés. De plus, il est aussi possible de rendre *conditionnel* l'acceptation de certains messages, en fonction de l'état interne de la machine ou de l'objet.

La figure 15.2 présente une nouvelle version d'une machine réalisant un tampon partagé servant de communication entre un producteur et un consommateur. Le premier point à souligner est l'absence de pré-conditions (clauses `WHEN`) associées aux messages et, donc, l'absence de réponses `EXCEPTION`. Par contre, on remarque l'ajout d'une clause `TRANSACTION`, nommée `ajout_ou_retrait`, après la spécification des messages. Dans l'exemple du

[1] Dans les exemples qui suivent, nous laisserons de coté les détails liés à la production et au traitement de ces informations. L'accent sera plutôt mis sur la communication entre les deux processus par l'intermédiaire du tampon.

```
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
```

Figure 15.1: Version séquentielle d'un tampon FIFO pour la communication entre un producteur et un consommateur

```
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
```

Figure 15.2: Version concurrente d'un tampon FIFO avec `TRANSACTION`s

```
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
```

Figure 15.3: Version concurrente d'un tampon de taille variable

tampon, cette clause indique qu'il ne sera possible de recevoir et traiter un message `ajouter` qu'à la condition que le tampon ne soit pas plein; inversement, un message `retirer` ne pourra être reçu et traité que si le tampon n'est pas vide.

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

Les transactions peuvent aussi être utilisées pour définir des contraintes plus générales sur l'ordre dans lequel divers messages peuvent être traités. Par exemple, supposons, toujours dans le problème du producteur–consommateur, que l'on désire définir un tampon partagé, mais cette fois de taille pouvant varier selon les besoins du client. La spécification d'un tel tampon est présentée à la figure 15.3. La taille du tampon est spécifiée, pour une série d'utilisations par un client donné, à l'aide du message `definir_taille`. Une fois ce message reçu et traité, on peut alors effectuer une série d'ajouts et retraits: ceci est indiqué par "`DO ... OD`", qui dénote la réception et traitement de *0 ou plusieurs* messages `ajouter` ou `retirer`. La taille du tampon peut par la suite être modifiée en envoyant tout d'abord le message `liberer`, message qui ne sera accepté que si le contenu du tampon est vide. Ceci complétera alors une `utilisation_tampon` par un client; une autre série d'utilisations, par un même client ou par un autre, pourra alors être amorcée en indiquant la taille requise (avec `definir_taille`) et en repétant la suite des actions possibles.

15.2 Forme générale d'une spécification de `TRANSACTION`

De façon générale, les transactions atomiques sont définies en termes des éléments suivants:

- Actions de base: Une action de base est soit un nom de message, soit un nom de transaction. S'il s'agit d'un message d'exception, le mot clé `EXCEPTION` doit être indiqué.

- Action avec garde: Une action peut être précédée d'une *garde*, c'est-à-dire, d'une expression booléenne indiquant à quelle condition le message peut être accepté. Une telle garde est notée par "`WHEN condition ->`". La condition peut référer à l'état (machine) ou au modèle (type).
- Séquence d'actions: Une séquence d'actions est spécifiée par une suite d'actions (simples ou complexes) séparées par ";". Par exemple:

  `definir_taille ; DO ... OD; IF ... FI`.
- Choix: La construction `IF ... FI` permet de spécifier un choix, chaque alternative étant séparée par "`|`". Un choix indique qu'exactement une seule des alternatives indiquées pourra survenir.
- Répétition: Une répétition est spécifiée à l'aide de `DO ... OD`. Comme dans le cas du choix avec `IF`, il est aussi possible de spécifier diverses alternatives séparées par "`|`".
- Combinaison parallèle: Un groupe d'actions pouvant être acceptées ou exécutées en parallèle (donc dans n'importe quel ordre) est indiquée par un groupe d'actions sans séparateur (ni ";", ni "`|`").

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
```

Figure 15.4: Machine distributrice de biscuits et muffins

La figure 15.4 présente une spécification d'une machine distributrice à biscuits (0.75$) et muffins (1.00$). La machine accepte uniquement les pièces de 0.25$ et 1.00$ (pièce indiquée par `piece25` ou `piece100`) et, dans le cas des biscuits payés avec une pièce de 1.00$, rend la monnaie. De plus, elle n'accepte jamais aucune pièce après qu'un montant de 1.00$ ait été inséré. Toutes les contraintes sur l'ordre possible de réception des messages sont indiquées avec la spécification de transaction, y compris la contrainte voulant qu'il ne soit pas possible d'insérer plus de 1.00$ dans la machine. Évidemment, il s'agit d'un version très simple d'une telle machine distributrice, puisqu'il n'existe aucune possibilité d'annuler ou interrompre la transaction en cours.

Exercices

15.1.

```
MACHINE trouver_max
  STATE( vals: set{integer} )
  INVARIANT true
  INITIALLY vals = {}

  MESSAGE ajouter_valeur( v: integer )
    TRANSITION
      vals = *vals U {v}

  MESSAGE obtenir_max
    REPLY( m: integer )
      WHERE m = MAX( x:integer SUCH THAT x IN vals :: x )
    TRANSITION
      vals = {}
END
```

Figure 15.5: Machine pour trouver le maximum parmi une série de nombres entiers

La figure 15.5 décrit une machine qui permet de recevoir une série de valeurs entières (via `ajouter_valeur`) et qui permet ensuite d'obtenir le maximum parmi les valeurs transmises (via `obtenir_max`).

15.1.1. Définissez une transaction qui va assurer que `obtenir_max` ne pourra être appelé que si au moins une valeur a été transmise, tout en permettant d'en ajouter autant que désiré.

15.1.2. Supposons que l'on désire modifier l'interface de la machine de façon à pouvoir, en tout temps, réinitialiser la machine sans devoir obtenir le maximum parmi les valeurs déjà transmises. Pour ce faire, on ajoute le message suivant:

```
  MESSAGE reinitialiser
    TRANSITION vals = {}
```

Définissez une transaction qui permetra d'obtenir le même comportement que dans l'exercice précédent pour `obtenir_max` tout en permettant d'annuler la transaction en cours via `reinitialiser`.

Chapitre 16

Le langage LOTOS

16.1 Aperçu du langage LOTOS

Le langage LOTOS — *Language Of Temporal Ordering Specification* — est l'une des deux techniques de spécifications formelles développées par l'ISO (*International Organization for Standardization*) pour la spécification de systèmes distribués et de protocoles de communication (particulièrement pour le modèle OSI = *Open Systems Interconnection*). [1]

Une norme internationale ISO pour LOTOS existe depuis la fin des années 80 [BB87]. Toutefois, les concepts à la base du langage sont plus anciens, puisqu'ils ont été inspirés des travaux de R. Milner sur CCS [Mil80, Mil89] (*Calculus of Communicating Systems*) et de C.A.R Hoare sur CSP [Hoa85] (*Communicating Sequential Processes*).

Le langage LOTOS dans son ensemble est constitué de deux parties:

1. une notation servant à la description de processus concurrents (aspect contrôle);
2. une notation utilisée pour la spécification des types de données manipulés par ces processus (aspect données).

Dans ce chapitre, nous allons examiner uniquement la première notation (spécification des processus); l'autre notation, tout à fait indépendante de la première, est inspirée du langage de spécifications algébriques ACT ONE [dMRV92] dont l'esprit est très semblable à celui du langage Larch [GH93] (niveau LSL) présenté dans un autre chapitre (Chap. 14).

LOTOS est un langage de spécification de processus réactifs et concurrents. Un processus est défini par son comportement observable, plus précisément, par les actions et communications qu'il peut effectuer. Ces diverses actions et communications sont décrites par des expressions formées de constantes processus (par ex., STOP, EXIT) et d'opérateurs permettant de combiner des actions et des processus pour créer de nouveaux processus (";", "[]", "|||", etc.). À cause de cette façon de construire des expressions définissant des processus — appelées des "expressions de comportement" — et parce qu'on peut, comme en mathématique, établir diverses équivalences entre ces expressions, on parle alors d'*algèbres de processus*.

Un aspect important de LOTOS — héritage de CCS et CSP — est que, contrairement à Spec où l'envoi et traitement de messages se fait de façon asynchrone, les communications entre processus en LOTOS se font toujours de façon synchrone, par l'intermédiaire de *rendez-vous*. On verra un peu plus loin comment exprimer de telles communications entre processus.

Avant d'examiner le langage lui-même à l'aide d'exemples, mentionnons que de nombreux outils ont été développés pour supporter différents aspects de l'écriture et vérification de spécifications LOTOS [GLO91, FGK+92]. Ainsi, outre des outils d'analyse syntaxique et de vérification de types, des outils de simulation et d'exécution ont été développés. Des compilateurs ont aussi été réalisés, permettant de produire du code (en langage C) à partir des descriptions LOTOS. Finalement, quelques outils de vérification ont aussi été développés, par exemple, vérification de propriétés des transitions, équivalence entre processus, correspondance avec réseaux de Petri.

[1] L'autre technique est Estelle — *Extended State Transition Language* — langage basé sur l'utilisation de machines et systèmes de transition, où les actions sont décrites en Pascal.

16.2 Les principaux concepts de LOTOS

Une notion fondamentale pour comprendre la spécification de processus à l'aide du langage LOTOS est celle d'*expression de comportement*. Une telle expression vise à décrire le comportement d'un *processus* en décrivant de façon explicite les *actions* que le processus peut exécuter (les messages qu'il peut reçevoir et envoyer).

16.2.1 Processus inactif

L'expression de comportement la plus simple est l'expression (la constante) STOP. Cette expression représente un processus bloqué, inactif, donc un processus qui ne peut exécuter aucune action. On verra plus loin que le comportement de STOP est différent de celui de la constante EXIT, laquelle dénote un processus qui termine normalement, avec succès, son exécution.

16.2.2 Séquence d'actions et choix

En LOTOS, la notion d'action est un concept primitif, atomique, non-défini: une action est simplement dénotée par un identificateur et n'est associée à aucune interprétation.

Soient `a, b, c` des actions possibles pour un processus, des messages pouvant être reçus ou envoyés par le processus. Soit alors l'expression (de comportement) suivante:

```
a; a; c; b; STOP
```

Cette expression représente le comportement d'un processus qui peut exécuter la séquence suivante d'actions: tout d'abord l'action `a`, suivie d'une autre action `a`, suivie d'une action `c`, suivie d'une action `b`. Ensuite, le processus devient bloqué, donc inactif (`STOP`).

Le nom d'une action peut évidemment être un nom significatfi plutôt qu'une simple lettre. Par exemple, supposons qu'on désire modéliser une forme de sémaphore servant à protéger une région critique. Une séquence d'actions possibles pour un tel sémaphore pourrait être la suivante:

```
prendreSemaphore; relacherSemaphore; ...
```

Un opérateur de base pour construire des expressions de comportements intéressantes est l'opérateur de choix "`[]`", lequel permet de choisir entre plusieurs actions possibles. Par exemple, soit l'expression de comportement suivante:

```
a; b; STOP [] c; d; STOP
```

Cette expression décrit le comportement d'un processus qui peut faire soit la séquence d'actions "`a; b; STOP`", soit la séquence "`c; d; STOP`". En l'absence de toute influence extérieure, le choix entre les deux alternatives sera fait de façon *non-déterministe*.

Bien que l'opérateur `[]` soit un opérateur binaire, il est possible, parce que cet opérateur est à la fois commutatif et associatif, de définir des expressions combinant plusieurs alternatives. Par exemple, on verra à la section 16.7 une version LOTOS d'un système modélisant une banque. Chaque interaction possible avec la banque (ajout d'un client, emprunt, retrait) serat modélisée par une action distincte; dans son état initial, le système serat alors prêt à effectuer n'importe laquelle de ces actions. L'expression décrivant le comportement de la banque, au niveau supérieur, aura donc l'allure suivante:

```
AjouterClient ...
[]
Solde ...
[]
Crediter ...
[]
Debiter ...
```

Comme l'illustre l'exemple précédent, le choix non-déterministe associé à l'opérateur `[]` peut, très souvent, être déterminé par l'environnement: ceci survient lorsqu'un processus `p` décrit par une expression de comportement contenant un choix est mis en présence d'autres processus avec lesquels `p` interagit (par l'intermédiaire de l'opérateur de synchronisation que nous verrons plus loin). Dans l'exemple de la banque, on doit donc comprendre

que le choix de l'une des alternatives va dépendre de ce que désireront faire les usagers de la banque (commis, client): le processus associé à la banque *choisirera* donc l'alternative associée au choix de l'usager.

Voyons un autre exemple illustrant la notion de choix. Soient les deux expressions de comportement suivantes:

(1) `a; b; STOP [] a; c; STOP`

(2) `a; (b; STOP [] c; STOP)`

Dans le deuxième expression, des parenthèses sont utilisées pour bien indiquer que l'action `a` doit tout d'abord être effectuée avant que l'expression de comportement qui suit le soit. Intuitivement, la différence entre ces deux expressions est que, dans la deuxième, l'action `a` est tout d'abord exécutée et, *ensuite*, un choix est fait entre les actions `b` et `c`; par contre, dans la première expression (1), le choix de l'une des deux branches est fait *avant* l'exécution de l'action `a`. Cette différence a un impact majeur lorsque des processus dont le comportement est défini par ces expressions sont mis en présence d'autres processus. On verra plus loin (section 16.5.3) dans quel contexte on peut effectivement distinguer ces deux expressions de comportement. Mentionnons simplement que, de façon intuitive, on peut réussir à distinguer ces deux comportements parce que, dans le premier cas, le choix de l'alternative droite pour l'action `a` a pour effet de rendre *impossible* l'exécution de l'action `b`; par contre, cette possibilité est toujours présente dans le deuxième cas après l'exécution de `a`, puisque le choix n'est pas encore fait.

16.2.3 Déclaration de processus et création d'instances

Dans tous les langages de programmation, il est possible de déclarer une procédure ou fonction en indiquant son nom, le type de ses arguments et le type de son résultat; on parle alors d'*abstractions procédurales*. Un mécanisme d'abstraction semblable est aussi disponible en `LOTOS` pour définir des processus.

Par exemple, soit la définition suivante d'un processus:

```
PROCESS Proc[a1, a2]: EXIT :=
  a1; a2; a1; EXIT
ENDPROC
```

Cette expression `LOTOS` définit un processus appelé `Proc` et pouvant effectuer les actions `a1` ou `a2` — de façon générale, l'ensemble des actions pouvant être effectuées par un processus est indiqué par les actions apparaissant entre "`[...]`". Il faut souligner que ces actions sont en fait des paramètres formels (génériques) du processus; une instance du processus devra donc être créée en indiquant les actions effectives devant être effectuées par le processus. Le mot-clé `EXIT` — apparaissant entre "`:`" et "`:=`" — indique la fonctionalité du processus `Proc`, à savoir qu'il s'agit d'un processus qui termine sans retourner aucun résultat.

Une fois un tel processus déclaré, il est alors possible de l'appeler — plus précisément, d'en créer une instance, comme on le ferait pour un appel de procédure. Par exemple, soit l'expression de comportement suivante:

```
x; z; Proc[a, b]
```

Les actions effectuées par cette expression de comportement seraient alors les suivantes:

```
x; z; a; b; a
```

16.2.4 Processus récursifs

Un processus réactif possède très souvent un comportement cyclique, sans fin: à partir d'un état initial, le processus effectue un certain nombre d'actions, en réponse à une demande de l'environnement, puis revient à son état initial. En `LOTOS`, de tels processus sont spécifiés à l'aide de processus récursifs. Voyons un premier exemple:

```
PROCESS Semaphore[p, v]: NOEXIT :=
  p; v; Semaphore[p, v]
ENDPROC
```

Cette expression `LOTOS` définit le processus infini (`NOEXIT`) appelé `Semaphore` qui peut effectuer les actions `p` ou `v`. Initialement, ce processus ne peut effectuer que l'action `p` (prise de contrôle du sémaphore). Une fois cette action effectuée, le processus peut ensuite effectuer l'action `v` (libération du sémaphore), auquel cas le processus retourne alors dans son état initial grâce à l'appel récursif (instanciation récursive du processus) `Semaphore[p, v]`.

Un exemple encore plus simple de processus cyclique infini est le suivant, qui modélise une horloge générant une suite sans fin de `tic`s:

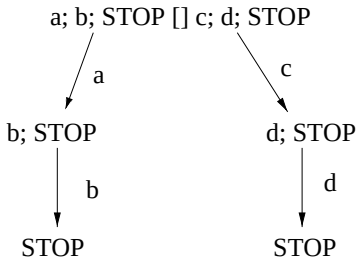

Figure 16.1: Un diagramme de transitions

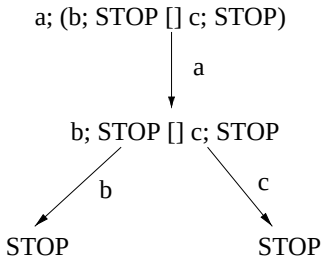

Figure 16.2: Un autre diagramme de transitions

```
PROCESS Horloge[tic]: NOEXIT :=
  tic; Horloge[tic]
ENDPROC
```

16.2.5 Processus et diagrammes de transition

Avant d'examiner d'autres exemples, il est intéressant de souligner les liens qui existent entre des processus décrits par des expressions de comportements et des diagrammes de transitions d'état (*state transition diagrams*).

Un processus peut être inteprété comme une entité qui, suite à des actions, effectue des changements d'état. On a donc la correspondance suivante, que nous allons illustrer graphiquement dans les paragraphes qui viennent:

Expression de comportement	$\leftrightarrow$	État
Action	$\leftrightarrow$	Transition

Par exemple, soit l'expression de comportement suivante:

```
a; b; STOP [] c; d; STOP
```

Une telle expression peut être représentée graphiquement par le diagramme de transition présenté à la figure 16.1, où les transitions sont annotées par des actions alors que les états sont annotés par des expressions de comportement

Par contre, soit l'expression suivante:

```
a; ( b; STOP [] c; STOP )
```

Cette expression serait alors représentée par le diagramme de transition présentée à la figure 16.2.

Cette équivalence entre processus et diagramme de transitions reste aussi valide dans le cas d'expressions définissant des processus de façon récursive. Soit le processus défini comme suit:

```
PROCESS P[a, b, c]: NOEXIT :=
  a; c; a; P[a, b, c]
  []
  b; a; c; P[a, b, c]
ENDPROC
```

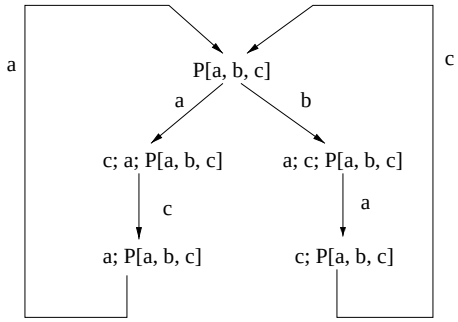

Figure 16.3: Un diagramme cyclique de transitions

Graphiquement, ce processus pourrait être représenté comme indiqué à la figure 16.3. Une définition récursive de processus comme celle de `p` correspond donc à un diagramme d'état cyclique: le nombre d'états est alors fini; les séquences d'actions (de transitions) peuvent, par contre, être *infinies*. Ceci est tout à fait naturel pour des processus réactifs, car on considère que de tels processus maintiennent une interaction constante avec leur environnement. Contrairement à un *algorithme* classique qui doit, après un temps fini, produire une réponse, un processus réactif peut donc ne jamais terminer son exécution (d'où l'utilisation de `NOEXIT`, expliqué plus en détail à la Section 16.4.1).

Dans les pages qui suivent, on verra d'autres exemples plus complexes de processus infinis et cycliques décrits à l'aide d'expressions de comportement `LOTOS`.

16.2.6 Exemples de machines distributrices

Machine distributrice de biscuits et muffins

La figure 16.4 présente une description `LOTOS` d'une machine distributrice de biscuits et muffins. Les caractéristiques de cette machine sont les suivantes:

- La machine accepte des pièces de 0.25$ et de 1.00$;
- Un muffin coûte 1.00$ alors qu'un biscuit coûte 0.75$;
- Si une pièce de 1.00$ est insérée et qu'un biscuit est sélectionné, la monnaie (0.25$) est rendue;
- Une fois qu'une pièce est insérée, il n'est pas possible d'annuler la transaction.

Les éléments importants à souligner dans la spécification de la figure 16.4 sont les suivants:

- La machine distributrice est définie comme un processus (`PROCESS`) possédant sept (7) ports de communication, sept points d'interaction — on peut aussi dire que ce processus peut effectuer sept actions différentes. Ces ports, qui sont présentés comme des arguments formels (entre "[...]"), devront être instanciés lors de l'utilisation effective du processus et représentent des actions pouvant ête exécutées par le processus, des interactions avec l'environnement ou avec d'autres processus. [2]

  Dans cet exemple, l'interprétation de ces ports, de ces actions, du point de vue de la machine distributrice, est la suivante:

  - `p25`/`p100`: Insertion/réception d'une pièce de 0.25$ ou 1.00$.
  - `b_bisc` et `b_muff`: Sélection du bouton pour un biscuit ou muffin.
  - `bisc` et `muff`: Livraison d'un biscuit ou muffin.
  - `m25`: Envoi d'une pièce de 0.25$ pour rendre la monnaie.

[2] Une analogie permettant de comprendre le rôle de ces paramètres qui indiquent les ports, les actions possibles du processus est de les considérer comme des paramètres *génériques*, au sens vu précédemment dans le cadre de la notation `Spec`: c'est l'instanciation de ces paramètres avec des valeurs effectives qui permet de créer une nouvelle instance du processus, lequel peut alors effectuer les actions indiquées en argument.

```
PROCESS Distributrice
        [p25, p100, b_bisc, b_muff, bisc, muff, m25]: NOEXIT :=
  p100;
    (b_muff;
       muff;
       Distributrice[p25, p100, b_bisc, b_muff, bisc, muff, m25]
     []
     b_bisc;
       bisc;
       m25;
       Distributrice[p25, p100, b_bisc, b_muff, bisc, muff, m25]
    )

  []

  p25;
    p25;
    p25;
    (b_bisc;
       bisc;
       Distributrice[p25, p100, b_bisc, b_muff, bisc, muff, m25]
     []
     p25;
       b_muff;
       muff;
       Distributrice[p25, p100, b_bisc, b_muff, bisc, muff, m25]
    )
ENDPROC
```

Figure 16.4: Machine distributrice de biscuits et muffins

```
PROCESS Client_muffin[p25, p100, b_muff, muff]: EXIT :=
  I; p100; b_muff; muff; EXIT
  []
  I; p25; p25; p25; p25; b_muff; muff; EXIT
ENDPROC
```

Figure 16.5: Un client qui désire un muffin

- Le mot-clé `NOEXIT`, apparaissant entre ":" et ":=", indique la *fonctionalité* du processus. Dans ce cas-ci, cette fonctionalité indique que le processus ne se termine jamais et ne retourne aucun résultat. Plus de détails sur les notions de fonctionalité et de terminaison des processus seront présentés à la Section 16.4.1.
- Le symbole ":=", qui suit la présentation des ports et la fonctionalité, indique le début de la description du comportement du processus `Distributrice` à l'aide d'une expression de comportement. Cette expression comporte diverses alternatives, lesquelles sont séparées par le symbole "`[]`". Cet opérateur, tel que mentionné précédemment, indique un choix entre deux ou plusieurs suites d'actions possibles. Quant aux suites d'actions, elles sont indiquées par des actions séparées par le symbole ";". [3]

  Dans cet exemple, la première action possible de `Distributrice` est `p100` ou `p25`. Dans le premier cas (insertion d'une pièce de 1.00$: `p100`), cela conduit à permettre deux alternatives possibles: la sélection du bouton pour un muffin (`b_muffin`) ou la sélection du bouton pour un biscuit (`b_biscuit`). Selon l'alternative choisie, qui dépendra du choix fait *par l'environnement* (c'est-à-dire, l'usager, qu'on modélisera un peu plus loin) l'item approprié sera livré (`muff` ou `bisc`) et, dans le cas d'un biscuit, la monnaie sera retournée (`m25`). Dans le deuxième cas (insertion de 0.25$ comme premier événement, donc, alternative du bas: `p25`), la machine va alors accepter une série de pièces de 0.25$ suivie de la sélection appropriée d'un item grâce au bouton associé.

  Il est à noter que, dans cette expression de comportement, lorsque 4 pièces de 0.25$ ont été insérées, il n'est alors plus possible de sélectionner un biscuit: une telle sélection ne peut se faire qu'après l'insertion d'une première pièce de 1.00$ ou après l'insertion de 3 pièces de 0.25$.
- Dernier point à souligner, le comportement de `Distributrice` est défini de façon récursive: après avoir livré l'item sélectionné et, si nécessaire, après avoir rendu la monnaie, le processus retourne ensuite dans son état initial par l'intermédiaire de l'appel récursif `Distributrice[p25, ..., m25]`, ce qui peut aussi être interprété comme la création d'une nouvelle instance du processus `Distributrice`. Cette création d'une nouvelle instance est indiquée par l'apparition du nom de processus `Distributrice` suivi de l'indication de ports à utiliser (dans ce cas-ci, les mêmes que ceux indiqués en paramètres formels).

Pour montrer les interactions possibles entre processus, nous allons maintenant définir un processus modélisant un client. Ce client, qui désire obtenir un muffin, va fouiller dans sa poche pour trouver la première pièce disponible; on suppose, pour simplifier, que le client a suffisamment de pièces pour payer et obtenir son muffin.

Une description `LOTOS` d'un tel client est présentée à la figure 16.5. Pour modéliser le fait que le client ne sait pas, *a priori*, s'il trouvera une pièce de 0.25$ ou 1.00$ au fond de sa poche, une action *invisible* est utilisée. Une telle action, notée par `I`, représente une action silencieuse et invisible, action toujours prête à s'effectuer et qui se fait de façon *interne* et spontanée, donc sans interaction avec d'autres processus (on utilise donc `I` pour une action dite `I`nterne ou `I`nvisible).

Pour bien comprendre le comportement du client, il faut savoir que lorsque plusieurs alternatives débutent par une action prête à s'effectuer, un choix *non-déterministe* est fait parmi ces diverses action et *une seule* d'entre elles s'effectue. Or, dans ce cas-ci, les deux alternatives de la construction de choix (`[]`) débutent par une transition interne, donc toutes les deux sont prêtes à s'effectuer. Un choix non-déterministe est donc fait entre ces deux alternatives, lequel choix n'est pas influencé par l'environnement, modélisant ainsi le choix non-déterministe de la première pièce trouvée par le client au fond de sa poche.

Lorsque la première pièce ainsi obtenue est une pièce de 1.00$, le client insère alors cette pièce dans la machine (`p100`), sélectionne le bouton pour un muffin (`b_muffin`), retire le muffin de la machine (`muff`) et ensuite, du point de vue de la machine distributrice, disparaît, donc termine son exécution (`EXIT`).

Dans le cas où la première pièce trouvée est un 0.25$, le client doit ensuite ne retirer et insérer dans la machine que des pièces de 0.25$, car seules ces pièces peuvent être acceptées lorsqu'un premier 0.25$ a été inséré et reçu par la machine. Après l'insertion de ces quatre pièces, le client, comme dans le cas précédent, presse le bouton de sélection approprié, retire son muffin et termine son interaction avec la machine.

À partir de ces deux processus, le comportement global d'un système machine distributrice–client peut maintenant être décrit à l'aide d'une expression de comportement mettant en présence les deux processus et leur permettant d'interagir:

```
Client_muffin[p25, p100, b_muff, muff]
  |[p25, p100, b_muff, muff]|
```

[3] Plus précisément, l'opérateur ";" doit toujours être *précédé* d'une action *simple* (avec ou sans échange d'information, avec ou sans prédicat de sélection) et non d'une expression générale de comportement.

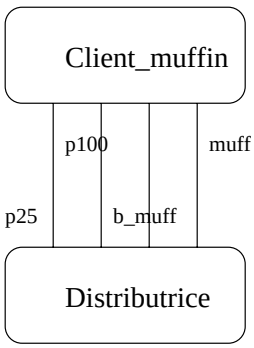

Figure 16.6: Un client qui utilise la machine pour acheter un muffin

```
Distributrice[p25, p100, b_bisc, b_muff, bisc, muff, m25]
```

Ici, les deux processus sont mis en présence l'un de l'autre et se synchronisent par l'intermédiaire des ports, des actions indiquées entre "`|[...]|`". De façon générale, ces ports représentent les canaux de communication utilisés par les processus pour échanger des informations (signaux de synchronisation ou, comme on le verra plus loin, information de divers types). Dans le cas présent, il s'agit simplement d'indiquer les actions qui devront être exécutées de façon synchrone par chacun des processus. Graphiquement, de telles synchronisations et communications entre processus peuvent être représentées telles qu'indiquées à la figure 16.6.

Une expression de synchronisation entraîne que chacun des processus ne peut effectuer l'une des actions indiquées par l'opérateur "`|[...]|`" que lorsque *l'autre* processus est lui aussi prêt à effectuer la même action; lorsque c'est bien le cas, les deux processus effectuent alors cette action en même temps, effectuant alors une transition les conduisant dans leur état résultant respectif. Par exemple, si le processus `Client_muffin` effectue la transition interne associée à la première alternative (la première pièce trouvée au fond de sa poche vaut 1.00$), les deux processus seront alors tous deux prêts à effectuer l'action `p100`, action qu'ils effectueront en même temps.

Il est important de souligner le fait que même si les deux processus exécutent la même action au même moment, l'interprétation de cette action est différente selon le processus qui l'effectue. Par exemple, dans le cas de l'action `p100`, on a les interprétations suivantes:

- `Client_muffin`: L'action `p100` est interprétée comme l'insertion *par le client* de la pièce dans la machine;
- `Distributrice`: Ici, l'action `p100` dénote la réception de la pièce par la machine et la transition associée indique alors que la pièce a bien été acceptée.

Les actions utilisées comme point de synchronisation entre processus représentent donc *le même événement*, mais de points de vue différents selon le participant impliqué dans l'interaction.

Soulignons que, de façon générale pour une unique action arbitraire `a`, un processus de la forme `e1 |[a]| e2` ne peut exécuter l'action `a` que si les processus `e1` *et* `e2` sont prêts, au même moment, à effectuer l'action `a`. Si un seul des deux processus est prêt à faire l'action, alors l'action *ne peut pas* s'effectuer. Pour une présentation plus formelle des transitions possibles pour cet opérateur ainsi que pour les autres opérateurs, voir la Section 16.5.

Machine distributrice avec bouton d'annulation

La figure 16.7 présente un autre exemple de machine distributrice. La machine permet d'obtenir du bouillon (0.50$) ou du café (0.75$). À chaque étape, le client peut annuler la transaction en appuyant sur la bouton `b_annul`. Dans le système présenté, la machine interagit avec un client qui n'est intéressé que par du café; toutefois, à chaque étape le client peut décider, de façon non-déterministe, qu'il désire annuler la transaction en cours (par exemple, s'il s'aperçoit qu'il ne lui reste pas suffisamment de pièces de 0.25$), ce qui est modélisé, comme dans le client de l'exemple précédent, par des transitions internes `I` préfixant chacune des alternatives. Lorsque le client annule la transaction, il récupère alors sa monnaie par l'action `m25`, une pour chacune des pièce de 0.25$ qu'il a insérée.

```
PROCESS Machine_cafe
        [p25, b_cafe, cafe, b_bouillon, bouillon, m25, b_annul]: NOEXIT :=
  p25;
  (b_annul;
     m25;
     Machine_cafe[p25, b_cafe, cafe, b_bouillon, bouillon, m25, b_annul]
   []
   p25;
     (b_annul;
        m25;
        m25;
        Machine_cafe[p25, b_cafe, cafe, b_bouillon, bouillon, m25, b_annul]
      []
       b_bouillon;
         bouillon;
         Machine_cafe[p25, b_cafe, cafe, b_bouillon, bouillon, m25, b_annul]
      []
      p25;
        (b_annul;
           m25;
           m25;
           m25;
           Machine_cafe[p25, b_cafe, cafe, b_bouillon, bouillon, m25, b_annul]
         []
         b_cafe;
           cafe;
           Machine_cafe[p25, b_cafe, cafe, b_bouillon, bouillon, m25, b_annul]
        )
     )
  )
ENDPROC

PROCESS Client_cafe[p25, b_cafe, cafe, m25, b_annul]: NOEXIT :=
  p25;
  (I; b_annul; m25; STOP
   []
   I; p25;
   (I; b_annul; m25; m25; STOP
    []
    I; p25;
    (I; b_annul; m25; m25; m25; STOP
     []
     I; b_cafe; cafe; STOP
    )
   )
  )
ENDPROC

PROCESS Machine_cafe_Client
        [p25, b_cafe, cafe, b_bouillon, bouillon, m25, b_annul]: NOEXIT :=
  Machine_cafe[p25, b_cafe, cafe, b_bouillon, bouillon, m25, b_annul]
    |[p25, b_cafe, cafe, m25, b_annul]|
  Client_cafe[p25, b_cafe, cafe, m25, b_annul]
ENDPROC
```

Figure 16.7: Machine distributrice de café (avec bouton d'annlulation) et interaction avec un client

```
SPECIFICATION Producteur_Consommateur: NOEXIT
BEHAVIOUR
  HIDE ajouter, retirer IN
    Producteur[ajouter]
      |[ajouter]|
    Tampon[ajouter, retirer]
      |[retirer]|
    Consommateur[retirer]

WHERE
  PROCESS Producteur[ajouter]: NOEXIT :=
    ajouter; Producteur[ajouter]
  ENDPROC

  PROCESS Consommateur[retirer]: NOEXIT :=
    retirer; Consommateur[retirer]
  ENDPROC

  PROCESS Tampon[ajouter, retirer]: NOEXIT :=
    ajouter; retirer; Tampon[ajouter, retirer]
  ENDPROC
ENDSPEC
```

Figure 16.8: Producteur–consommateur avec tampon et alternance stricte

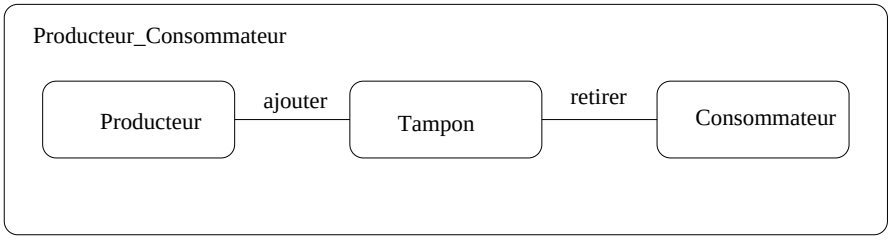

Figure 16.9: Les processus et leurs liens de communication pour le problème du producteur et consommateur avec tampon

16.2.7 Problème du producteur–consommateur (premières versions)

Producteur–consommateur avec alternance stricte

Le prochain exemple (figure 16.8) est une version simplifiée du problème du producteur et consommateur communiquant par l'intermédiaire d'un tampon partagé. Graphiquement, ces processus peuvent être représentés comme à la figure 16.9.

Dans cette version, aucune information n'est véritablement communiquée au tampon. Ce dernier sert uniquement à assurer une alternance stricte entre le producteur qui ajoute — et qui ne peut le faire que lorsque le tampon est lui-même prêt à effectuer la transition `ajouter` — et le consommateur qui retire — lorsque le tampon est prêt à effectuer `retirer`. Le processus `Tampon` se comporte donc comme un tampon de taille 1, assurant que le producteur ne puisse ajouter de nouvel élément en plus de celui déjà présent tant que le consommateur n'a pas retiré, et inversement pour le consommateur qui ne peut retirer un élément tant qu'il n'a pas été ajouté par le producteur.

Il est à remarquer que le système dans son ensemble est défini par un module introduisant une `SPECIFICATION`. Dans un tel cas, le comportement de l'ensemble est donné par l'expression de comportement qui suit le mot-clé `BEHAVIOUR`; les processus auxiliaires utilisés dans cette spécification sont ensuite définis dans la partie qui suit le

```
SPECIFICATION Producteur_Consommateur: NOEXIT

BEHAVIOUR
  HIDE ajouter, retirer IN
    Producteur[ajouter]
      |[ajouter]|
    Tampon[ajouter, retirer]
      |[retirer]|
    Consommateur[retirer]

WHERE
  PROCESS Producteur[ajouter]: NOEXIT :=
    ajouter ?x: nat; Producteur[ajouter]
  ENDPROC

  PROCESS Consommateur[retirer]: NOEXIT :=
    retirer ?y: nat; Consommateur[retirer]
  ENDPROC

  PROCESS Tampon[ajouter, retirer]: NOEXIT :=
    ajouter ?z: nat; retirer !z; Tampon[ajouter, retirer]
  ENDPROC
ENDSPEC
```

Figure 16.10: Producteur–consommateur avec tampon, transmission d'information et alternance stricte

mot-clé `WHERE`.

Dans cet exemple, une autre nouvelle construction est introduite: `HIDE ... IN`. Cette construction permet d'introduire des identificateurs (des ports, des actions) qui sont *locaux* à la spécification ou expression de comportement. Ces ports sont locaux en ce sens qu'ils sont définis pour utilisation par l'expression de comportement mais sont *cachés* des observateurs externes; du point de vue d'un tel observateur, l'exécution d'une action cachée ne peut pas être distinguée d'une transition interne (transition `I`), donc est invisible.

Soulignons que cette spécification n'indique *aucun* port lui permettant d'interagir avec l'environnement et ne fait qu'utiliser des ports dissimulés, ce qui est tout à fait inhabituel; en ce sens, cet exemple n'est pas vraiment réaliste et ne sert qu'à illustrer les constructions de base de `LOTOS`.

Producteur–consommateur avec tampon de taille 1 (bis)

Évidemment, il serait plus intéressant de pouvoir modéliser la transmission réelle d'information entre le producteur et le consommateur par l'intermédiaire du tampon. Ceci est illustré à la figure 16.10.

Dans cet exemple, la production d'une valeur est modélisée dans le processus `Producteur` par l'action "`ajouter ?x: nat`". De façon générale, une action suivie d'une expression "`?x: type`" indique la *réception* d'une valeur du type indiqué et son association (comme pour un paramètre formel) à l'identificateur `x`. Toutefois, dans ce cas-ci, l'action "`ajouter ?x: nat`" est synchronisée avec une autre action "`ajouter ?x: nat`" (dans le `Tampon`); ceci sert alors à modéliser la production *non-déterministe* d'une valeur du type indiqué et l'association de cette valeur à chacune des variables (`x` dans les deux cas). Dans le processus `Tampon`, cette valeur est ensuite transmise au consommateur par l'intermédiaire de la communication (synchrone) entre le "`retirer ?y: nat`" du `Consommateur` et le "`retirer !z`" du `Tampon`, cette dernière expression indiquant l'envoi d'une valeur.

Une expression de la forme "`!x`" est appelée une *déclaration de valeur*; par contre, une expression "`?x: type`" est appelée une *déclaration de variable*. Comme on vient de le voir, lorsqu'une communication s'effectue entre un processus déclarant [4] une valeur et un autre déclarant une variable du type approprié, une transmission d'information s'effectue alors d'un processus vers l'autre. Lorsque les deux processus qui se synchronisent utilisent une déclaration de variable, on modélise alors la génération non-déterministe d'une valeur. Finalement, si les deux processus utilisent une déclaration (offre) de valeur, alors la même valeur doit être offerte pour que la synchronisation ait lieu.

[4] Le terme *production de valeur* serait plus approprié. Le terme *offre de valeur* est aussi parfois employé.

```
SPECIFICATION Producteur_Consommateur: NOEXIT

BEHAVIOUR
  HIDE echanger IN
    Producteur[echanger]
      |[echanger]|
    Consommateur[echanger]

WHERE
  PROCESS Producteur[echanger]: NOEXIT :=
    echanger ?x: nat; Producteur[echanger]
  ENDPROC

  PROCESS Consommateur[echanger]: NOEXIT :=
    echanger ?y: nat; Consommateur[echanger]
  ENDPROC
ENDSPEC
```

Figure 16.11: Version sans tampon du problème du producteur et consommateur avec alternance stricte

```
SPECIFICATION Max3[inx1, inx2, inx3, outmax]: NOEXIT

BEHAVIOUR
  HIDE max12 IN
    Max2[inx1, inx2, max12]  |[max12]|  Max2[max12, inx3, outmax]

WHERE
  PROCESS Max2[inx, iny, outmax]: NOEXIT :=
    inx ?x: nat;
    iny ?y: nat;
    ( [x <= y] -> outmax !y; Max2[inx, iny, outmax]
      []
      [x >= y] -> outmax !x; Max2[inx, iny, outmax]
    )
  ENDPROC
ENDSPEC
```

Figure 16.12: Processus pour calcul du maximum de 3 nombres

Il est à noter que dans l'exemple du producteur interagissant avec un consommateur par l'intermédiaire du processus `Tampon` de taille 1, on aurait pu plus simplement faire interagir directement le producteur et le consommateur sans passer par l'intermédiaire d'un tampon: puisque le producteur ne peut plus faire d'ajout tant que le consommateur n'a pas retiré la valeur du tampon, il serait aussi bien de lui passer directement l'information, quitte à attendre lorsque le consommateur n'est pas prêt à recevoir (et symétriquement pour le consommateur). On pourrait donc définir un système équivalent sans tampon tel que présenté à la figure 16.11.

Dans un exemple ultérieur, nous allons présenter une version plus générale de communication entre producteur et consommateur par l'intermédiaire d'un tampon (de taille supérieure à 1). Auparavant, il nous faut toutefois introduire la notion de transitions avec gardes, ainsi que la notion de processus avec paramètres. Pour illustrer ces nouveaux concepts, nous allons examiner, dans un premier temps, deux exemples plus simples que celui du tampon partagé.

16.2.8 Calcul du maximum de 3 nombres

La figure 16.12 présente une spécification d'un processus qui reçoit trois (3) nombres naturels en entrée (sur les ports `inx1, inx2, inx3`) et qui produit en sortie le maximum parmi ces 3 nombres (sur le port `outmax`).

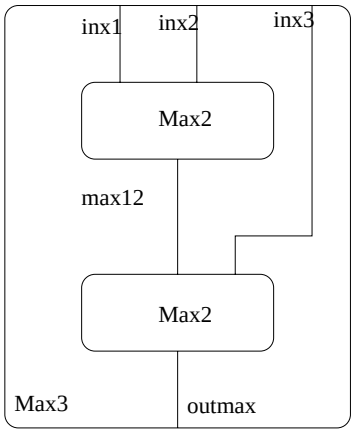

Figure 16.13: Les processus et leurs liens pour le calcul du maximum de 3 nombres

```
PROCESS Max2[inx, iny, outmax]: NOEXIT :=
  (inx ?x: nat; iny ?y: nat;
   []
   iny ?y: nat; inx ?x: nat
  );
  ( [x <= y] -> outmax !y; Max2[inx, iny, outmax]
    []
    [x >= y] -> outmax !x; Max2[inx, iny, outmax]
  )
ENDPROC
```

Figure 16.14: Processus `Max2` avec ordre arbitraire de réception (*syntaxe invalide*)

Graphiquement, les liens entre ces processus peuvent être présentés tels qu'indiqués à la figure 16.13.

Le processus `Max3` est réalisé en utilisant un processus auxiliaire qui calcule le maximum de 2 nombres: un premier processus `Max2` est créé pour calculer le maximum parmi les nombres reçus sur les deux premiers ports, alors qu'un second processus sert au calcul du maximum entre le résultat produit par le premier processus et le nombre reçu sur le troisième port (`inx3`).

On remarque, dans la description du comportement du processus `Max2`, que les 2 valeurs à comparer doivent être reçues dans un ordre fixe (`x` avant `y`). On remarque aussi l'utilisation d'alternatives avec *gardes*, c'est-à-dire d'alternatives préfixées par une expression de la forme suivante:

```
[condition] -> ...
```

Une fois les deux valeurs reçues, la valeur la plus grande parmi `x` ou `y` est alors transmise sur le port `outmax` selon que la garde appropriée est valide ou non, donc selon que `[x <= y]` ou `[x >= y]` est vraie.

La notion de choix avec gardes est semblable à celle qu'on retrouve en `Spec` dans le cas des transactions. Par exemple, une transaction `Spec` semblable au choix avec gardes du processus `Max2` aurait l'allure suivante:

```
IF WHEN x <= y -> outmax ...      -- Envoi de y
|  WHEN x >= y -> outmax ...      -- Envoi de x
FI
```

Dans cet exemple, il faut noter que, dans le cas où `x=y`, un choix non-déterministe entre les deux alternatives sera effectué, puisque les gardes seront alors toutes deux vraies; toutefois, comme les deux valeurs sont alors égales, le résultat produit sera le même.

Un autre point à souligner est que l'opérateur ";" requiert toujours que l'opérande gauche soit une action, et non pas une expression générale de comportement. Ainsi, l'expression de comportement présentée à la Fig 16.14,

```
PROCESS Max2[inx, iny, outmax]: NOEXIT :=
  inx ?x: nat;
  (
   iny ?y: nat [x <= y]; outmax !y; Max2[inx, iny, outmax]
   []
   iny ?y: nat [x >= y]; outmax !x; Max2[inx, iny, outmax]
  )
ENDPROC
```

Figure 16.15: Processus `Max2` avec ordre arbitraire de réception (syntaxe correct)

```
PROCESS Variable[lire, ecrire](val: nat): NOEXIT :=
  lire !val; Variable[lire, ecrire](val)
  []
  ecrire ?v: nat; Variable[lire, ecrire](v)
ENDPROC
```

Figure 16.16: Variable partagée avec valeur initiale

qui vise à permettre de recevoir les données `x` ou `y` dans n'importe quel ordre, *ne serait pas valide* puisque l'expression qui précède le ";" n'est pas une action simple. Pour modéliser le fait que les éléments peuvent être reçus dans n'importe quel ordre avant de pouvoir être comparés, il faudrait plutôt utiliser une expression de séquentialisation, construction que nous verrons un peu plus loin.

Une autre façon de définir `Max2` (avec ordre fixe de réception des arguments) pourrait être telle qu'indiquée à la Fig. 16.15. Dans cette version, plutôt que d'utiliser des gardes, on a utilisé des *prédicats de sélection*. Un tel prédicat permet de tester la valeur reçue et assure de ne sélectionner l'alternative contenant cette action avec déclaration de variable que si le prédicat est satisfait. Contrairement aux gardes vues précédemment, la condition du prédicat de sélection peut porter sur la valeur de l'argument; l'alternative asociée ne sera donc choisie que si la contrainte sur la valeur pouvant être reçue par la sélection de cette alternative est satisfaite. Dans l'exemple, donc, si la valeur `y` reçue sur le port `iny` est inférieur ou égale à `x` (`x <= y`), alors la première action `iny` pourra être sélectionnée; quant à la deuxième action, elle ne pourra être sélectionnée lorsque `x >= y`.

16.2.9 Modélisation d'une variable partagée

La figure 16.16 présente un autre exemple de processus `LOTOS`. Il s'agit d'un processus servant à modéliser une variable globale entière (de type `nat`) dans une mémoire partagée. Deux actions peuvent être effectuées sur une telle variable:

1. `lire`: Lire la valeur courante (`val`) de la variable.
2. `ecrire`: Écrire une nouvelle valeur (`v`) dans la variable.

Cet exemple illustre les deux sortes de paramètres qui peuvent apparaître dans une déclaration de processus ou dans un appel de processus:

- Des arguments, indiqués entre "[...]", qui dénotent les ports par lesquels le processus peut communiquer, les actions que le processus peut effectuer.
- Des arguments, indiqués entre "(...)", qui indiquent les paramètre (des valeurs, donc, et non des actions) qui peuvent être transmis lors de l'appel du processus. De telles valeurs font alors partie de *l'état interne* du processus.

Une telle distinction entre différentes sortes d'arguments formels est très semblable à celle qu'on retrouve en `Spec` entre paramètres génériques (indiqués en `Spec` par "{...}") et paramètres ordinaires (indiqués par "(...)"). La forme générale d'une spécification d'un processus `LOTOS` dont l'exécution ne se termine pas est donc la suivante:

```
PROCESS Variable[lire, ecrire]: NOEXIT :=
  HIDE generer IN
    generer ?v: nat; Var[lire, ecrire](v)
  WHERE
    PROCESS Var[lire, ecrire](val: nat): NOEXIT :=
      lire !val; Var[lire, ecrire](val)
      []
      ecrire ?v: nat; Var[lire, ecrire](v)
    ENDPROC
ENDPROC
```

Figure 16.17: Variable partagée avec valeur initiale non-déterminée

```
PROCESS NomDuProcessus[act1, ..., actk]
                      (arg1: t1, ..., argn: tn):
                      NOEXIT :=
  ... expression de comportement utilisant les actions act1, ..., actk ...
ENDPROC
```

Par contre, il est aussi possible de spécifier un processus qui se termine en retournant un certain nombre de résultats (de types `type_res1`, ..., `type_resm`):

```
PROCESS NomDuProcessus[act1, ..., actk]
                      (arg1: t1, ..., argn: tn):
                      EXIT(type_res1, ..., type_resm) :=
  ... expression de comportement utilisant les actions act1, ..., actk ...
ENDPROC
```

Pour utiliser correctement le processus `Variable` il est nécessaire, lors de la création du processus, de spécifier la valeur initiale (argument "`(val: nat)`") de son état. Cette valeur représente la valeur courante de la variable tant qu'une action d'écriture n'est pas effectuée avec l'action "`ecrire ?v: nat`". Lorsque cette dernière action est effectuée, la valeur reçue est utilisée comme nouvelle valeur courante de la variable (`Variable[lire,ecrire](v)`): le changement d'état associé à la transition (action) est donc possiblement accompagné d'une modification de l'état interne du processus.

Pour modéliser une variable dont la valeur initiale ne serait pas indiquée explicitement, on pourrait utiliser le processus présenté à la figure 16.17. L'action `generer` de l'expression de comportement du processus `Variable` modélise le choix non-déterministe d'une valeur initiale pour la variable, valeur utilisée ensuite par comme valeur initiale de l'état interne du processus `Var`.

16.2.10 Producteur–consommateur avec tampon partagé

La figure 16.18 présente finalement une description du système du producteur–consommateur avec transmission d'informations, mais cette fois avec un véritable tampon, de taille bornée. Ce tampon est défini par une paire d'arguments `n` (nombre maximum d'éléments) et `contenu` (la séquence des éléments). De plus, la génération d'un élément par le producteur est maintenant modélisée par une action interne (cachée) explicite, `generer`, la valeur résultante étant ensuite transmise au tampon par l'action `ajouter`.

Il faut souligner que les arguments décrivant l'état interne du processus `Tampon` sont définis *à la* `Spec`, donc en utilisant les types et opérations du langage `Spec`. Ceci n'est donc *pas* une spécification `LOTOS` légale. En `LOTOS`, ces variables d'état devraient plutôt être décrites en utilisant des types abstraits définis à l'aide du langage ACT ONE [dMRV92]. Nous avons choisi d'utiliser `Spec` plutôt qu'ACT ONE pour les types de façon à simplifier la présentation des concepts de base du langages `LOTOS` et ainsi éviter d'avoir à introduire en même temps une nouvelle notation pour la description des comportements (aspect contrôle) et une nouvelle notation pour les types (aspect données): une difficulté à la fois! Toutefois, pour ceux que cela intéresse, la version `LOTOS` correcte, dans la syntaxe de l'outil CADP [FGM+92], est présentée dans les figures 16.19 et 16.20.

On remarque encore, dans la spécification du processus `Tampon` de la figure 16.18, l'utilisation de *gardes*, donc de conditions devant être vérifiées pour que la transition qui suit puisse s'effectuer. L'expression décrivant le

```
SPECIFICATION Producteur_Consommateur(n: nat): NOEXIT
BEHAVIOUR
  HIDE ajouter, retirer IN
    Producteur[ajouter]
      |[ajouter]|
    Tampon[ajouter, retirer](n, [])
      |[retirer]|
    Consommateur[retirer]

WHERE
  PROCESS Producteur[ajouter]: NOEXIT :=
    HIDE generer IN
      generer ?x: nat; ajouter !x; Producteur[ajouter]
  ENDPROC

  PROCESS Consommateur[retirer]: NOEXIT :=
    retirer ?y: nat; Consommateur[retirer]
  ENDPROC

  PROCESS Tampon[ajouter, retirer](n: nat, contenu: sequence{nat}): NOEXIT :=
    [length(contenu) < n] ->
      ajouter ?x: nat;
      Tampon[ajouter, retirer](n, contenu || [x])
    []
    [length(contenu) > 0] ->
      retirer !contenu[1];
      Tampon[ajouter, retirer](n, contenu[2..length(contenu)])
  ENDPROC
ENDSPEC
```

Figure 16.18: Producteur–consommateur avec tampon partagé de taille bornée

```
SPECIFICATION Producteur_Consommateur(n: nat): NOEXIT

  LIBRARY SEQUENCENAT ENDLIB

BEHAVIOUR
  HIDE ajouter, retirer IN
    Producteur[ajouter]
      |[ajouter]|
    Tampon[ajouter, retirer](n, empty)
      |[retirer]|
    Consommateur[retirer]

WHERE
  PROCESS Producteur[ajouter]: NOEXIT :=
    HIDE generer IN
      generer ?x: nat; ajouter !x; Producteur[ajouter]
  ENDPROC

  PROCESS Consommateur[retirer]: NOEXIT :=
    retirer ?y: nat; Consommateur[retirer]
  ENDPROC

  PROCESS Tampon
          [ajouter, retirer]
          (n: nat, contenu: sequenceNat): NOEXIT :=
    [length(contenu) > 0] ->
      retirer !head(contenu);
      Tampon[ajouter, retirer](n, tail(contenu))
    []
    [length(contenu) < n] ->
      ajouter ?x: nat;
      Tampon[ajouter, retirer](n, addTail(contenu, x))
  ENDPROC
ENDSPEC
```

Figure 16.19: Version du producteur et consommateur avec tampon utilisant des types ACT ONE

```
library Natural, Boolean endlib

type sequenceNat is Natural, Boolean
  sorts sequenceNat
  opns
    empty   (*! constructor *) : -> sequenceNat
    addTail (*! constructor *) : sequenceNat, nat -> sequenceNat

    (* observateurs *)
    head:   sequenceNat -> nat
    tail:   sequenceNat -> sequenceNat
    length: sequenceNat -> nat
  eqns
   forall a: nat, s: sequenceNat
    ofsort nat
      head( addTail(empty, a) ) = a;
      head( addTail(s, a) )     = head(s);

      length( empty )           = 0;
      length( addTail(s, a) )   = length(s) + 1;

    ofsort sequenceNat
      tail( addTail(empty, a) ) = empty;
      tail( addTail(s, a) )     = addTail( tail(s), a );
endtype
```

Figure 16.20: Spécification ACT ONE des séquences utilisées dans la spécification du producteur et consommateur avec tampon

comportement de `Tampon` est donc semblable la spécification de `TRANSACTION` de la version concurrente `Spec` du problème du producteur–consommateur, qui aurait l'allure suivante:

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

Le contenu du tampon est modélisé par une séquence, laquelle est transmise en arguments "(...)" au processus `Tampon`. L'ajout d'un élément est modélisé par la concaténation de cet élément à la suite de ceux déjà présents (`contenu || [x]`), alors que le retrait est modélisé par l'envoi de l'élément en tête de la séquence (`!contenu[1]`) suivi d'un appel récursif utilisant la séquence de laquelle l'élément en tête est retiré (`contenu[2..length(contenu)]`) et modélisant le changement de l'état interne associé au retrait.

La notion d'objet mutable au sens vu dans le cadre du langage `Spec` — une entité qui possède une identité fixe avec un état interne qui évolue dans le temps — n'est donc pas une notion de base en `LOTOS`. Le langage `LOTOS` utilise plutôt une approche *fonctionnelle*, au sens des langages fonctionnels de programmation (SML, Haskell, Lisp/Scheme): en `LOTOS`, un changement d'état est modélisé par une transition et par la création, grâce à un appel récursif, d'une *nouvelle instance* de processus avec des arguments indiquant le contenu du nouvel état. Ce processus réalisera alors le comportement qui tiendra compte du changement d'état interne.

16.2.11 Le problème de paquetage/dépaquetage de M. Jackson

Un problème intéressant à résoudre dans un langage avec processus concurrents est celui présenté par M. Jackson pour illustrer la notion de *structure clash*. Le problème à résoudre est facile à exprimer. Toutefois, dans un langage séquentiel classique, il *n'est pas* facile d'exprimer simplement la solution de ce problème à cause de ce que Jackson appelle un *structure clash* — une incompatibilité de structure entre les données d'entrée et celles de sortie.

Une version simplifiée de ce problème est la suivante:

> Étant donné une suite de lignes en entrée, où chaque ligne contient un nombre variable de caractères, il faut produire en sortie la même séquence de caractères, mais avec exactement 125 caractères par ligne (sauf peut-être la dernière ligne).

La figure 16.21 présente une solution au problème indiqué alors que la figure 16.22 illustre les liens entre les processus. Le processus de niveau supérieur est `Copier`; il reçoit les lignes de caractères sur le port `entree` et produit sur le port `sortie` les lignes à 125 caractères. Pour réaliser sa tâche, `Copier` fait appel à deux processus auxiliaires:

- `Depaqueter`: Sa tâche est de recevoir sur `entree` une ligne (nombre variable de caractères) et de produire *un* (1) caractère à la fois sur le port `car`. La ligne lue est représentée par la variable d'état `ligne` et la lecture ne se fait que lorsque la ligne précédente a été complètement transmise (`length(ligne)=0`).
- `Paqueter`: Sa tâche est de recevoir un caractère à la fois sur le port `car` (`car ?c: char`) et de les accumuler dans `tampon` jusqu'à ce que `n` (= 125) caractères aient été reçus (`length(tampon) = n`); le contenu de `tampon` est alors transmis sur `sortie` (`sortie !tampon`).

Au niveau supérieur, ces deux processus sont reliés et communiquent entre eux par l'intermédiaire d'un port auxiliaire `car`: la sortie de `Depaqueter` est donc connectée à l'entrée de `Paqueter`. Et voila! Essayez d'écrire un programme Pascal ou C pour résoudre le même problème: vous verrez que la solution est loin d'être aussi simple et élégante!

16.3 Éléments additionnels du langage `LOTOS`

Dans les sections précédentes, seul un sous-ensemble du langage `LOTOS` a été examiné. Or, `LOTOS` possède de nombreuses autres constructions permettant de construire des expressions de comportement. Voyons en quelques-unes.

```
SPECIFICATION Copier[entree, sortie]: NOEXIT
BEHAVIOUR
  HIDE car IN
    Depaqueter[entree, car]([])
    |[car]|
    Paqueter[car, sortie](125, [])

WHERE
  PROCESS Depaqueter[entree, car](ligne: sequence{char}): NOEXIT :=
    [length(ligne) = 0] ->
       entree ?ligneLue: sequence{char};
       Depaqueter[entree, car](ligneLue)
    []
    [length(ligne) ~= 0] ->
       car !ligne[1];
       Depaqueter[entree, car](ligne[2..length(ligne)])
  ENDPROC

  PROCESS Paqueter[car, sortie](n: nat, tampon: sequence{char}): NOEXIT :=
    [length(tampon) = n] ->
       sortie !tampon;
       Paqueter[car, sortie](n, [])
    []
    [length(tampon) < n] ->
       car ?c: char;
       Paqueter[car, sortie](n, tampon || [c] )
  ENDPROC
ENDSPEC
```

Figure 16.21: Solution au problème de Jackson

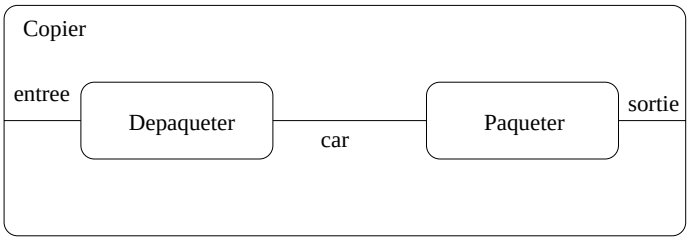

Figure 16.22: Les processus et leurs liens pour le problème de Jackson

- L'opérateur `|[...]|` permet d'indiquer de façon détaillée l'ensemble des ports par l'intermédiaire desquels des processus s'exécutant en parallèle, de façon concurrente, vont se synchroniser — donc indiquant les actions qui vont devoir s'effectuer de façon synchronisée. Deux autres opérateurs pour exécution parallèle sont aussi disponibles:

  `||` : Cet opérateur permet une exécution en parallèle avec une synchronisation complète, à chaque étape, à chaque action, entre processus. L'opérateur `||` est donc équivalent à `|[`P`]|`, où P serait l'ensemble de *tous* les ports, toutes les actions possibles.

  `|||` : Cet opérateur, appelé opérateur d'entrelacement (*pure interleaving*), peut être vu comme l'inverse du précédent en ce sens qu'il permet l'exécution en parallèle de processus mais sans qu'*aucune* synchronisation ne soit nécessaire. Il peut donc être considéré comme équivalent à `|[]|`, donc synchronisation sur un ensemble vide de ports.

- Alors que l'opérateur `STOP` indique un processus qui devient bloqué, inactif, l'opérateur `EXIT` indique un processus qui termine correctement son travail, qui termine *avec succès* son exécution. Lorsqu'un processus termine ainsi son exécution, on considère alors qu'il exécute une action spéciale nommée δ (on peut aussi dire qu'il envoie un signal de terminaison δ). Cette action peut alors servir à activer un autre processus par l'intermédiaire de l'opérateur "`>>`". Ce dernier opérateur est dit de *composition séquentielle*. Ainsi, dans l'expression "`C1 >> C2`", le processus `C2` ne commencera son exécution que lorsque `C1` terminera avec succès, avec `EXIT`, envoyant ainsi à `C2` le signal δ. Si `C1` ne termine pas avec `EXIT`, alors le processus `C2` ne sera jamais activé.

  Bien que les opérateurs ";" et "`>>`" soient deux opérateurs de composition séquentielle, il existe une différence importante entre les deux: alors que l'opérateur ";" permet de composer séquentiellement une action (gauche) et une expression de comportement (droite), l'opérateur "`>>`" peut, quant à lui, composer deux expressions générales de comportement. [5]

- Une instruction `EXIT` qui termine avec succès peut aussi produire, en terminant, un ou plusieurs résultats, comme pour une fonction dans un langage de programmation classique. Le résultat ainsi produit peut ensuite être reçu en utilisant un opérateur de composition séquentielle de la forme "`>> ACCEPT ... IN ...`". Par exemple, le résultat retourné par un processus qui termine en retournant un entier (par ex., `EXIT(0)`) pourrait être reçu par un autre processus grâce à une expression de la forme suivante:

  ```
  >> ACCEPT x: nat IN ...
  ```

  Ceci sera expliqué plus en détail à l'aide d'un exemple à la section suivante.

- L'opérateur d'interruption (*disable operator*) "`[>`" permet de modéliser l'interruption d'un processus par un autre. Par exemple, dans l'expression "`C1 [> C2`", en tout temps il est possible que `C2` interrompe l'exécution de `C1`. Plus précisément, à tout moment, un choix peut être fait entre la prochaine action de `C1` et la prochaine action de `C2`. Si l'action choisie provient de `C2`, alors `C1` est considéré comme ayant été interrompu et l'exécution se poursuit avec `C2`. Si `C1` termine son exécution sans succès, alors le processus `C2` est activé; par contre, si `C1` termine son exécution avec succès (avec `EXIT`), alors `C2` ne sera jamais exécuté et l'expression composée "`C1 [> C2`" termine elle aussi avec succès.

- L'opérateur `CHOICE`, appelé opérateur de *choix généralisé*, permet de modéliser un choix parmi un ensemble arbitraire d'éléments. Par exemple, soit l'expression suivante:

  ```
  CHOICE x: nat [] a !x; C
  ```

  Cette expression serait équivalente à la suivante, qui représente l'exécution de l'action `a` produisant une valeur entière naturelle choisie arbitrairement suivie de l'exécution de l'expression de comportement `C` (les trois points indiquent les autres possibilités: `3, 4, 5, ...`):

  ```
  a !0; C [] a !1; C [] a !2; C [] ...
  ```

[5] On peut noter qu'une différence semblable existe dans le cas du type `sequence{t}` de `Spec`, où on a les deux opérations suivantes:

- `add(x, s)`: crée une nouvelle séquence en ajoutant un élément au début d'une séquence existante.
- `s1 || s2`: crée une nouvelle séquence en concaténant deux séquences.

e1	e2	e1 [] e2	e1 || e2	e1 ||| e2	e1 |[a]| e2
a; STOP	a; STOP	{a}	{a}	{aa}	{a}
a; STOP	b; STOP	{a, b}	{}	{ab, ba}	{b}
a; b; STOP	a; b; STOP	{ab}	{ab}	{abab, aabb}	{abb}
a; b; STOP	b; a; STOP	{ab, ba}	{}	{abba, abab, baab, baba}	{bab}
a; b; STOP	a; c; STOP	{ab, ac}	{a}	{abac, aabc, aacb, acab}	{abc, acb}
a; EXIT	a; EXIT	{aδ}	{aδ}	{aaδ}	{aδ}
a; EXIT	b; EXIT	{aδ, bδ}	{}	{abδ, baδ}	{b}
a; b; EXIT	a; b; EXIT	{abδ}	{abδ}	{ababδ, aabbδ}	{abbδ}
a; b; EXIT	b; a; EXIT	{abδ, baδ}	{}	{abbaδ, ababδ, baabδ, babaδ}	{babδ}
a; b; EXIT	a; c; EXIT	{abδ, acδ}	{a}	{abacδ, aabcδ, aacbδ, acabδ}	{abcδ, acbδ}

Table 16.1: Exemples illustrant le comportement des différents opérateurs binaires

- La construction `LET ... IN ...` permet d'introduire une déclaration locale à un bloc. Une telle déclaration est donc semblable au `CHOOSE( ...SUCH THAT ... )` de `Spec`. Par exemple:

```
LET i: nat = x + 1 IN
  envoyer !i; ...
```

Comparaison des différents opérateurs binaires

La table 16.1 présente, pour différentes expressions `e1` et `e2`, l'ensemble des séquences d'actions possibles — on dit aussi l'ensemble des *traces* possibles — lorsque `e1` et `e2` sont combinées à l'aide des principaux opérateurs binaires `LOTOS` (choix et opérateurs de composition parallèle). Par exemple, la première ligne indique que si les processus `e1` et `e2` sont combinés à l'aide de l'opérateur `||`, alors la seule séquence d'actions qui pourra être observé sera `a`; par contre, si les processus sont combinés à l'aide de `|||`, alors la seule séquence possible d'actions sera `aa`. Quant à la dernière ligne, elle indique que si les processus `e1` et `e2` indiqués sont combinés avec l'opérateur `|||`, alors l'une des quatre traces indiquées pourra être observée (toutes se terminant par δ).

Quelques points intéressants à souligner sont les suivants:

- Un ensemble vide indique un processus qui ne peut effectuer aucune action, donc un processus équivalent à `STOP`. Dans la deuxième ligne de la table, avec l'opérateur `||`, ceci survient parce que `e1` attend que `e2` puisse faire l'action `a`; or, cette action ne pourra jamais être faite par `e2` qui ne peut faire que `b` (et inversement pour `b`).

  Dans la quatrième ligne de la table, toujours pour l'opérateur `||`, on obtient un ensemble vide à cause d'un *deadlock*: pour que `e1` puisse effectuer l'action `a`, `e2` devrait aussi être prêt à effectuer cette action; or `e2` doit tout d'abord faire `b` avant de faire `a`, sauf que `b` doit aussi être fait par `e1` ... mais après `a`. C'est donc un cas typique de *deadlock* où on retrouve des dépendances mutuelles cycliques.

- L'action δ dénote une forme de signal de terminaison. Un tel signal n'est produit que si tous les processus impliqués (dans le cas d'une combinaison parallèle) sont prêts à terminer: il y a donc toujours une synchronisation implicite sur cette action — en d'autres mots, `|[a1, ..., an]|` est en fait équivalent à `|[a1, ..., an,` δ`]|`. On remarque que dans certains cas ce signal n'est jamais généré. Par exemple, dans la dernière ligne, lorsque les processus sont combinées avec l'opérateur `||`, l'action `a` peut être faite en synchronie par les deux processus; par contre, une fois cette action effectuée, les deux processus deviennent bloqués parce que ni l'action `b` ni l'action `c` ne peut être effectuée par les deux processus: aucun signal de terminaison n'est donc généré, les deux processus étant bloqués.

À la section 16.5, nous présentons de façon plus formelle — à l'aide d'une forme de définition inductive — la sémantique des expressions de comportement, donc l'ensemble des transitions permises pour une expression donnée. Mais auparavant, examinons encore quelques exemples de spécification `LOTOS`.

```
PROCESS Max2[inx, iny, outmax]: NOEXIT :=
  Lire_valeurs[inx, iny]
  >>
  ACCEPT x: nat, y: nat IN
    ( [x <= y] -> outmax !y; Max2[inx, iny, outmax]
      []
      [x >= y] -> outmax !x; Max2[inx, iny, outmax]
    )
WHERE
   PROCESS Lire_valeurs[inx, iny]: EXIT( nat, nat ) :=
     inx ?x: nat; iny ?y: nat; EXIT(x, y)
     []
     iny y?: nat; inx ?x: nat; EXIT(x, y)
   ENDPROC
ENDPROC
```

Figure 16.23: Processus pour calcul du maximum de 3 nombres

16.4 Quelques exemples additionnels de spécification

16.4.1 Maximum parmi deux nombres (bis)

La figure 16.23 présente une nouvelle version d'un processus qui reçoit 2 nombres en entrée (ports `inx` et `iny`) et qui retourne ensuite le maximum sur le port `outmax`. À la différence de l'exemple de la section 16.2.8, les nombres peuvent être reçus dans n'importe quel ordre.

Le processus `Lire_valeurs`, qui est défini comme un processus auxiliaire local au processus `Max2` à l'aide de la clause `WHERE`, reçoit les deux valeurs sur les ports appropriés, dans un ordre quelconque (grace au choix). Ce processus se termine, peu importe l'alternative choisie (`x` avant `y` ou `y` avant `x`), par l'action `EXIT(x, y)`, qui retourne les deux nombres lus.

À tous les processus `LOTOS` est associé une *fonctionalité*. Cette fonctionalité, qui est une forme de typage, est indiquée entre le ":" suivant l'en-tête du processus (nom et arguments) et le ":=" donnant sa définition. Dans la plupart des cas vus jusqu'à présent, cette fonctionalité était souvent "`NOEXIT`", ce qui indiquait un processus qui ne se termine jamais. [6] Par contre, un processus peut aussi se terminer de façon correcte et normale, avec succès. Ceci est possible grâce à l'action `EXIT` (qui génère un signal de terminaison δ). Lorsqu'un processus se termine ainsi, il peut, au moment où il se termine, retourner des résultats divers. La fonctionalité du processus est alors donné par le mot-clé `EXIT` suivi du type des résultats retournés lorsque le processus termine. Dans notre exemple, la fonctionalité est donc `EXIT(nat, nat)`.

Toujours dans notre exemple du processus `Max2`, les valeurs retournées par `Lire_valeurs` sont reçues par le processus `Max2` avec la clause "`ACCEPT ... IN`". À partir de ce point, le reste de l'expression de comportement est identique à ce qu'on a vu précédemment.

16.4.2 Tri d'une suite de nombres

La figure 16.24 présente une spécification d'un processus qui permet de `Trier` (en ordre croissant) une suite de nombres naturels; le diagramme de transition correspondant est présenté à la fig. 16.25 — notons que des gardes ont été indiquées sur ce diagramme de transition. Le client peut ajouter les nombres dans la suite (avec `ajouter`); lorsque sa suite est terminée, le client l'indique par l'envoi du signal `suiteTerminee`. Le client peut ensuite commencer à retirer les éléments triés (avec `obtenir`) jusqu'à ce que la fin de la série soit indiquée au client par l'envoi du signal `suiteTerminee`. On remarque l'utilisation de la construction "`LET ... = ... IN`" qui permet de définir un identificateur local et de lui associer une valeur: ici, on définit l'index `i` — l'index de l'élément minimum de la séquence, donc l'index du prochain élément à retourner (`obtenir !elems[j]`) — et la sous-séquence `elems_o` — sous-séquence obtenue en retirant l'élément minimum et utilisée comme nouvel état

[6] Comme on l'a déjà mentionné, lorsqu'on traite des systèmes réactifs, de tels processus sont courants, puisqu'un tel système doit maintenir une interaction constante et continue avec son environnement.

```
PROCESS Trier[ajouter, obtenir, suiteTerminee]: NOEXIT :=
  TrierAjout[ajouter, obtenir, suiteTerminee]( [] )
WHERE
   PROCESS TrierAjout
           [ajouter, obtenir, suiteTerminee]
           ( elems: sequence{nat} ): NOEXIT :=
     ajouter ?x: nat;
       TrierAjout[ajouter, obtenir, suiteTerminee]( add(x, elems) )
     []
     suiteTerminee;
       TrierRetrait[ajouter, obtenir, suiteTerminee]( elems )
   ENDPROC

   PROCESS TrierRetrait
           [ajouter, obtenir, suiteTerminee]
           ( elems: sequence{nat} ): NOEXIT :=
     [elems = []] ->
       suiteTerminee;
       Trier[ajouter, obtenir, suiteTerminee]
     [elems ~= []] ->
       (LET j: nat = indexMin(elems) IN
        LET elems_o: sequence{nat} = elems[1..j-1] || elems[j+1..length(elems)] IN
          obtenir !elems[j];
          TrierRetrait[ajouter, obtenir, suiteTerminee](elems_o)
       )
   ENDPROC

   CONCEPT indexMin( elems: sequence{nat} ) VALUE( m: nat )
     WHERE
       ALL( j: nat SUCH THAT j IN domain(elems) :: elems[m] <= elems[j] )
ENDPROC
```

Figure 16.24: Tri (en ordre croissant) d'une suite de nombres

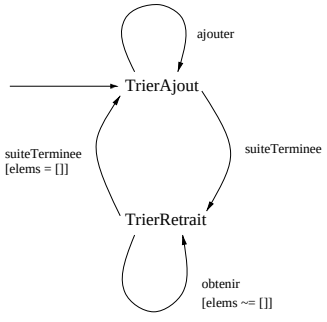

Figure 16.25: Le diagramme de transitions pour le processus de tri

```
SPECIFICATION AjouterParitePaire[recevoir, transmettre]: NOEXIT
BEHAVIOUR
  RecevoirBits[recevoir, transmettre]( [] )
WHERE
 PROCESS RecevoirBits[recevoir, transmettre]( bits: sequence{bit} ): NOEXIT :=
   [length(bits) < 7] ->
     recevoir ?b: bit;
     RecevoirBits[recevoir, transmettre]( bits || [b] )
   []
   [length(bits) = 7] ->
     (LET bitParite: bit = bitParite(bits) IN
       TransmettreBits[recevoir, transmettre]( bits || [bitParite] )
     )
 ENDPROC

 PROCESS TransmettreBits[recevoir, transmettre]( bits: sequence{bit} ): NOEXIT :=
   [length(bits) > 0] ->
     transmettre !bits[1] ;
     TransmettreBits[recevoir, transmettre]( bits[2..length(bits)] )
   []
   [length(bits) = 0] ->
     RecevoirBits[recevoir, transmettre]( [] )
 ENDPROC

 -- Concepts a la Spec (pas du LOTOS legal).
 CONCEPT bitParite( bits: sequence{bit} ) VALUE( b: bit )
   WHERE
     nombreBitsA_1(bits) MOD 2  = 0 => b = 0,
     nombreBitsA_1(bits) MOD 2 ~= 0 => b = 1,

 CONCEPT nombreBitsA_1( bits: sequence{bit} ) VALUE( n: nat )
   WHERE
     n = NUMBER( i: nat SUCH THAT i IN domain(bits) & bits[i] = 1 :: i )

 CONCEPT bit: type
   WHERE
     Subtype( bit, nat ),
     ALL( b: bit :: b = 0 | b = 1 )
ENDSPEC
```

Figure 16.26: Conversion d'une série de 7 bits en 8 bits par ajout d'un bit de parité

pour l'appel récursif. Notons, ici aussi, l'utilisation de types et concepts du langage `Spec`; de façon formelle, ces constructions ne font pas partie du langage `LOTOS`.

16.4.3 Conversion 7 bits à 8 bits par ajout d'un bit de parité

La figure 16.26 présente une spécification pour un processus qui reçoit en entrée des paquets de 7 bits et qui produit en sortie ces 7 bits auxquels a été ajouté un $8^{ième}$ bit, plus précisément, un bit de parité. Les bits de parité sont souvent utilisés pour détecter des erreurs lors de l'encodage et la transmission d'information sous forme binaire. Ici, pour un paquet de 7 bits, le bit de parité qui est ajouté à ces bits ($8^{ième}$) est choisi de façon à assurer que le nombre total de bits à 1 dans les 8 bits résultants est pair (parité paire).

Le processus de haut niveau (`AjouterParitePaire`) alterne entre les états de réception des bits (`RecevoirBits`) et de transmission des bits reçus (`TransmettreBits`). Dans l'exemple, les bits ne peuvent être transmis que lorsqu'ils ont tous été reçus et que le bit de parité a été calculé. Toutefois, les bits auraient aussi pu être transmis au fur et à mesure de leur réception, puisque les bits reçus ne sont pas modifiés et qu'on fait simplement ajouter un bit à la fin de la séquence (voir l'un des exercices à la fin du chapitre).

16.5 Sémantique opérationnelle

Dans cette section, nous allons présenter de façon formelle la sémantique d'un sous-ensemble du langage `LOTOS`. Plus précisément, nous allons définir une sémantique *opérationnelle* pour ce sous-ensemble de `LOTOS`, lequel sous-ensemble ne comprendra que les processus non-récursifs sans arguments formels et sans déclaration de variables ou valeurs et n'utilisant que les opérateurs suivants: ";", `[]`, |[...]|, |||, || et `HIDE...IN`, `STOP` et `EXIT` — on appelle parfois ce sous-ensemble *basic* `LOTOS`.

Le style de sémantique opérationnelle que nous allons présenter est appelé *sémantique opérationnelle structurée* [Hen90, Ast91]. Une telle sémantique va permettre d'indiquer, pour une expression de comportement donnée, les actions, les transitions qui peuvent s'effectuer, et ce en fonction de la structure de cette expression et en fonction du contexte dans lequel elle est utilisée. Comme on le verra, il s'agira d'une définition *inductive*: les seules actions possibles et permises pour une expression seront donc uniquement celles pouvant être déduites par un nombre fini d'applications des règles qui seront présentées. On remarquera qu'*aucune* règle ne sera spécifiée pour l'expression `STOP`: ceci signifiera donc qu'un processus réprésenté par une telle expression de comportement *ne peut rien faire*, donc représente bien un processus bloqué.

16.5.1 Règles de transition pour les expressions de comportement

Soit C une expression de comportement; nous allons dénoter par $C \xrightarrow{a} C'$ la transformation de C en C' suite à l'exécution de l'action a. En d'autres mots, $C \xrightarrow{a} C'$ indique qu'une transition associée à l'action a est effectuée et permet de passer de l'état C à l'état C' (où C et C' sont des états de processus, donc des expressions de comportement: cf. section 16.2.5).

Pour définir une sémantique opérationnelle de façon formelle, nous allons indiquer, pour une expression de comportement donnée, *toutes* les transitions possibles qui peuvent avoir lieu, selon le contexte dans lequel cette expression est utilisée. Pour ce faire, nous allons introduire des règles de transition ayant la forme suivante:

$$\frac{e_1, \ldots, e_n}{t}$$

où e_1, ..., e_n sont des transitions possibles ou des contraintes (conditions booléennes) sur les actions et où t représente une transition pouvant être effectuée si les transitions ou conditions précédentes sont valides. Un exemple concret permettra d'illustrer plus clairement cette notion de règle d'inférence.

Transitions pour une séquence d'actions Soit C une expression de comportement et a une action quelconque (y compris l'action invisible `I`). La transition suivante pour l'expression de comportement "a; C" est alors possible:

$$a;\, C \xrightarrow{a} C$$

Ceci peut être noté à l'aide de la règle suivante, où l'absence d'une pré-condition (en haut de la barre horizontale) représente une condition équivalente à `true`:

$$\frac{}{a;\, C \xrightarrow{a} C} \tag{16.1}$$

Transitions pour des expressions avec choix Soient C_1 et C_2 deux expressions de comportement et a une action quelconque (y compris `I`) telle que $C_1 \xrightarrow{a} C_1'$. Alors, les transitions suivantes sont permises pour les expressions indiquées:

$$C_1 \;[]\; C_2 \xrightarrow{a} C_1' \qquad\qquad C_2 \;[]\; C_1 \xrightarrow{a} C_1'$$

Ces deux cas peuvent être indiqués par les règles de transition suivantes:

$$\frac{C_1 \xrightarrow{a} C_1'}{C_1 \;[]\; C_2 \xrightarrow{a} C_1'} \tag{16.2}$$

$$\frac{C_1 \xrightarrow{a} C_1'}{C_2 \;[]\; C_1 \xrightarrow{a} C_1'} \tag{16.3}$$

Transitions pour des expressions en parallèle sans synchronisation Dans le cas d'un exécution parallèle sans synchronisation — donc avec l'opérateur ||| d'entrelacement — les deux processus sont libres d'évoluer indépendamment l'un de l'autre. Une exécution en parallèle signifie donc que les actions de l'un ou l'autre peuvent se faire, et ce *dans n'importe quel ordre.*

Soient C_1 et C_2 deux expressions de comportement et a une action quelconque (y compris `I`) telle que $C_1 \stackrel{a}{\rightarrow} C_1'$. Alors, les transitions suivantes sont permises:

$$C_1 \;|||\; C_2 \stackrel{a}{\rightarrow} C_1' \;|||\; C_2 \qquad\qquad C_2 \;|||\; C_1 \stackrel{a}{\rightarrow} C_2 \;|||\; C_1'$$

Ces deux cas peuvent être indiqués par les règles de transition suivantes:

$$\frac{C_1 \stackrel{a}{\rightarrow} C_1'}{C_1 \;|||\; C_2 \stackrel{a}{\rightarrow} C_1' \;|||\; C_2} \tag{16.4}$$

$$\frac{C_1 \stackrel{a}{\rightarrow} C_1'}{C_2 \;|||\; C_1 \stackrel{a}{\rightarrow} C_2 \;|||\; C_1'} \tag{16.5}$$

Transitions pour des expressions avec synchronisation Dans le cas de l'opérateur de synchronisation `|[]|`, les règles sont plus complexes car elles doivent tenir compte de ce que certains ports peuvent être utilisés pour synchronisation alors que d'autres ne le sont pas. De plus, on doit aussi prendre en compte le fait que l'action interne `I` se fait toujours de façon autonome pour chacun des processus; il n'est donc pas possible de synchroniser de telles actions, puisque par définition elles sont invisibles.

Soit $P = \{p_1, \ldots, p_n\}$ un ensemble de ports utilisés, par l'intermédiaire de l'opérateur $|P|$, pour la synchronisation entre processus (donc avec p_i différent de `I`); en d'autres mots $|P|$ dénote `|[`$p_1, \ldots, p_n$`]|`.

La première règle de transition porte sur une action qui *ne fait pas partie* de l'ensemble de synchronisation. Cette règle indique que si une action peut être faite par un processus et qu'elle ne fait pas partie de l'ensemble de synchronisation P, alors elle peut se faire de façon autonome même lorsqu'en présence de synchronisation sur P:

$$\frac{B_1 \stackrel{a}{\rightarrow} B_1',\;\; a \notin P}{B_1 \;|P|\; B_2 \stackrel{a}{\rightarrow} B_1' \;|P|\; B_2} \tag{16.6}$$

$$\frac{B_1 \stackrel{a}{\rightarrow} B_1',\;\; a \notin P}{B_2 \;|P|\; B_1 \stackrel{a}{\rightarrow} B_2 \;|P|\; B_1'} \tag{16.7}$$

Puisque l'action `I` ne peut jamais être utilisée pour la synchronisation, ce sont donc ces deux règles qui couvrent les transitions possibles pour une telle action.

Quant à l'autre règle de transition, elle porte sur des actions pour lesquelles une synchronisation doit être effectuée. La règle indique qu'une action qui doit se faire de façon synchronisée ne peut se faire que si tous les processus impliqués sont prêts à faire cette action, auquel cas les processus font cette action tous en même temps:

$$\frac{B_1 \stackrel{a}{\rightarrow} B_1',\;\; B_2 \stackrel{a}{\rightarrow} B_2',\;\; a \in P}{B_1 \;|P|\; B_2 \stackrel{a}{\rightarrow} B_1' \;|P|\; B_2'} \tag{16.8}$$

En d'autres mots, ces trois règles indiquent que des processus combinés de façon parallèle avec synchronisation peuvent exécuter toutes les actions qu'un processus peut faire indépendamment de l'autre ($a \notin P$) ainsi que toutes les actions pouvant être faites de façon synchrone ($a \in P$) lorsque tous les processus impliqués sont prêts à faire cette action.

Plus simplement, lorsque l'ensemble `P` ne contient qu'une seule action, les règles de transitions peuvent alors être décrites comme suit:

$$\frac{e_1 \stackrel{\mathtt{b}}{\rightarrow} e_1', \mathtt{b} \neq \mathtt{a}}{e_1 \;|[\mathtt{a}]|\; e_2 \stackrel{\mathtt{b}}{\rightarrow} e_1' \;|[\mathtt{a}]|\; e_2} \tag{16.9}$$

$$\frac{e_1 \stackrel{\mathtt{b}}{\rightarrow} e_1',\;\; \mathtt{b} \neq \mathtt{a}}{e_2 \;|[\mathtt{a}]|\; e_1 \stackrel{\mathtt{b}}{\rightarrow} e_2 \;|[\mathtt{a}]|\; e_1'} \tag{16.10}$$

$$\frac{e_1 \xrightarrow{\texttt{a}} e_1',\ e_2 \xrightarrow{\texttt{a}} e_2'}{e_1\ |[\texttt{a}]|\ e_2 \xrightarrow{\texttt{a}} e_1'\ |[\texttt{a}]|\ e_2'} \tag{16.11}$$

Ces règles sont complètes en ce sens que les seules actions possibles sont celles *permises* par ces règles.

Un point à souligner est que lorsqu'une action `a` est faite de façon synchrone par deux processus, l'effet visible est qu'un *unique* événement `a` vient de survenir (cf. règle 16.8). Ceci revient à exprimer ce qui avait été mentionné précédemment, à savoir qu'une action avec synchronisation représente *un seul et même événement*, mais avec plusieurs participants. Évidemment, l'interprétation de l'événement en termes d'une action effectuée par un processus dépend alors du participant impliqué (cf. l'exemple des machines distributrices).

Finalement, dans le cas des expressions obtenues avec l'opérateur || (exécution parallèle avec synchronisation sur tous les ports), les transitions possibles peuvent facilement être déduites des règles présentées plus haut et sont laissées en exercice.

Voyons un exemple simple. Soit la définition suivante de processus:

```
PROCESS P1[a, b]: NOEXIT :=
  a; b; P1[a, b]
ENDPROC
```

Soit alors le processus défini par l'expression de comportement suivante:

```
P1[tic, tac] |[tic, tac, toc]| P1[tic, toc]
```

La seule séquence d'actions qui pourra être effectuée par ce processus est la suivante:

```
  tic
```

Après avoir fait, de façon conjointe, l'action `tic`, l'expression initiale se transformera alors dans l'expression suivante:

```
tac; P1[tic, tac]  |[tic, tac, toc]| toc; P1[tic, toc]
```

Le processus à la gauche de l'opérateur `|[tic, tac, toc]` ne peut faire que `tac`, mais doit le faire en se synchronisant avec l'autre processus. Or, ce dernier ne peut pour l'instant que faire `toc` ... d'où inter-bloquage.

Transitions pour expression avec dissimulation (`HIDE`) Ici, on a deux cas possibles selon que le port associé à une action est dissimulé ou non. La première règle de transition indique qu'une action qui n'est pas dissimulée reste possible et reste visible:

$$\frac{B \xrightarrow{a} B',\ a \notin \{g_1, \ldots, g_n\}}{\texttt{HIDE}\ g_1, \ldots, g_n\ \texttt{IN}\ B \xrightarrow{a} \texttt{HIDE}\ g_1, \ldots, g_n\ \texttt{IN}\ B'} \tag{16.12}$$

L'autre règle indique qu'une action dissimulée peut quand même s'exécuter. Par contre, cette action ne sera pas visible de l'extérieur; pour un observateur extérieur, l'exécution d'une action dissimulée sera donc équivalente à l'exécution d'une action interne invisible:

$$\frac{B \xrightarrow{a} B',\ a \in \{g_1, \ldots, g_n\}}{\texttt{HIDE}\ g_1, \ldots, g_n\ \texttt{IN}\ B \xrightarrow{I} \texttt{HIDE}\ g_1, \ldots, g_n\ \texttt{IN}\ B'} \tag{16.13}$$

Transition pour l'expression `EXIT` L'expression `EXIT`, comme on l'a mentionné précédemment, dénote un processus qui termine avec succès et qui, ce faisant, produit l'action spéciale δ (qui réprésente une forme de signal de terminaison). De façon formelle, on a donc la règle d'inférence suivante:

$$\frac{}{EXIT \xrightarrow{\delta} STOP} \tag{16.14}$$

Une des particularités d'`EXIT` est la suivante: des processus composés de façon parallèle, avec ou sans synchronisation, doivent toujours terminer en même temps. En d'autres mots, une expression de comportement ne peut se terminer que si toutes les sous-expressions qui la composent sont prêtes à terminer. La formalisation de cette exigence est relativement simple puisqu'il suffit de considérer que tout opérateur de composition parallèle inclut l'action δ comme point de synchronisation. On peut donc considérer que les divers opérateurs de composition parallèle et synchronisation sont équivalents aux expressions suivantes:

`|||`	$\equiv$	`|[`δ`]|`
`||`	$\equiv$	`|[`*tous les ports possibles*`,` δ`]|`
`|[p1, ..., pn]|`	$\equiv$	`|[p1, ..., pn,` δ`]|`

16.5.2 Exemples d'application des règles de transition

Exemple avec séquences et choix Soit l'expression de comportement suivante:

```
a;b;STOP [] I;c;STOP
```

D'après la règle (16.1), les transitions possibles pour `a;b;STOP` sont les suivantes:

$$\texttt{a;b;STOP} \xrightarrow{a} \texttt{b;STOP} \xrightarrow{b} \texttt{STOP}$$

Comme aucune règle de transition n'a été spécifiée pour `STOP`, aucune transition n'est ensuite possible, donc `STOP` représente bien un processus inactif.

De façon similaire, les transitions possibles pour `I;c;STOP` sont les suivantes:

$$\texttt{I;c;STOP} \xrightarrow{I} \texttt{c;STOP} \xrightarrow{c} \texttt{STOP}$$

Si l'on combine ensuite ces deux expressions, on aura alors deux séquences possibles d'exécution selon que l'on utilise la règle (16.2) ou la règle (16.3):

Règle (16.2): $\texttt{(a;b;STOP [] I;c;STOP)} \xrightarrow{a} \texttt{b;STOP} \xrightarrow{b} \texttt{STOP}$

Règle (16.3): $\texttt{(a;b;STOP [] I;c;STOP)} \xrightarrow{I} \texttt{c;STOP} \xrightarrow{c} \texttt{STOP}$

Exemple avec synchronisation Soit l'expression de comportement suivante:

```
a;b;STOP [] I;c;STOP |[a,c]| a;STOP [] b;c;STOP
```

Tout d'abord, mentionnons que, de par la priorité des opérateurs, cette expression est équivalente à la suivante:

```
((a;b;STOP) [] (I;c;STOP)) |[a,c]| ((a;STOP) [] (b;c;STOP))
```

En `LOTOS`, la priorité des opérateurs est la suivante, en ordre décroissant de priorité:

```
;   ->   []   |[...]|   HIDE
```

Ceci permet donc d'éliminer toutes les parenthèses dans l'expression précédente sans créer d'ambiguité.

On a examiné, dans le paragraphe précédent, les transitions possibles pour la sous-expression à gauche de `|[a,c]|` lorsque cette expression est seule. Quant à l'expression de droite, les transitions possibles sont les suivantes selon la règle qui est utilisée pour effectuer la première transition (choix):

Règle (16.2): $\texttt{(a;STOP [] b;c;STOP)} \xrightarrow{a} \texttt{STOP}$

Règle (16.3): $\texttt{(a;STOP [] b;c;STOP)} \xrightarrow{b} \texttt{c;STOP} \xrightarrow{c} \texttt{STOP}$

Il nous reste maintenant à examiner le comportement de ces deux expressions lorsque mises en présence l'une de l'autre avec l'opérateur `|[a,c]|`. Encore une fois, on a diverses possibilités:

1. `a;b;STOP [] I;c;STOP |[a,c]| a;STOP [] b;c;STOP`
   $\xrightarrow{a}$ -- Règle (16.8)
   `b;STOP |[a,c]| STOP`
   $\xrightarrow{b}$ -- Règle (16.6)
   `STOP |[a,c]| STOP`

2. `a;b;STOP [] I;c;STOP |[a,c]| a;STOP [] b;c;STOP`
   $\xrightarrow{I}$ -- Règle (16.6)
   `c;STOP |[a,c]| a;STOP [] b;c;STOP`
   $\xrightarrow{b}$ -- Règle (16.7)
   `c;STOP |[a,c]| c;STOP`
   $\xrightarrow{c}$ -- Règle (16.8)
   `STOP |[a,c]| STOP`

3. `a;b;STOP [] I;c;STOP |[a,c]| a;STOP [] b;c;STOP`
   $\xrightarrow{b}$ -- Règle (16.7)
   `a;b;STOP [] I;c;STOP |[a,c]| c;STOP`
   $\xrightarrow{I}$ -- Règle (16.6)
   `c;STOP |[a,c]| c;STOP`
   $\xrightarrow{c}$ -- Règle (16.8)
   `STOP |[a,c]| STOP`

Exemple avec dissimulation Soit l'expression de comportement suivante:

```
HIDE c IN a;b;STOP [] I;c;STOP |[a,c]| a;STOP [] b;c;STOP
```

Cette expression est équivalente à la suivante:

```
(HIDE c IN ((a;b;STOP [] I;c;STOP) |[a,c]| (a;STOP [] b;c;STOP)))
```

De par l'exemple précédent, on aura alors les transitions suivantes:

1. `HIDE c IN a;b;STOP [] I;c;STOP |[a,c]| a;STOP [] b;c;STOP`
   $\xrightarrow{a}$ -- Règle (16.12)
   `HIDE c IN b;STOP |[a,c]| STOP`
   $\xrightarrow{b}$ -- Règle (16.12)
   `HIDE c IN STOP |[a,c]| STOP`

2. `HIDE c IN a;b;STOP [] I;c;STOP |[a,c]| a;STOP [] b;c;STOP`
   $\xrightarrow{I}$ -- Règle (16.12)
   `HIDE c IN c;STOP |[a,c]| a;STOP [] b;c;STOP`
   $\xrightarrow{b}$ -- Règle (16.12)
   `HIDE c IN c;STOP |[a,c]| c;STOP`
   $\xrightarrow{I}$ -- Règle (16.13)
   `STOP |[a,c]| STOP`

3. `HIDE c IN a;b;STOP [] I;c;STOP |[a,c]| a;STOP [] b;c;STOP`
   $\xrightarrow{b}$ -- Règle (16.12)
   `HIDE c IN a;b;STOP [] I;c;STOP |[a,c]| c;STOP`
   $\xrightarrow{I}$ -- Règle (16.12)
   `HIDE c IN c;STOP |[a,c]| c;STOP`
   $\xrightarrow{I}$ -- Règle (16.13)
   `HIDE c IN STOP |[a,c]| STOP`

On voit donc que les mêmes transitions peuvent être effectuées que dans l'exemple précédent, sauf que les actions `c` deviennent des actions invisibles (`I`).

16.5.3 Impact du moment du choix

Soit les deux expressions de comportement suivantes:

```
a; b; STOP [] a; c; STOP
```

```
a; (b; STOP [] c; STOP)
```

Examinons les diverses transitions possibles lorsque ces expressions sont mises en présence d'un autre processus dans le contexte suivant, où `E` représente l'une des deux expressions en question:

```
a; b; STOP |[a,b,c]| E
```

Dans le premier cas, on a les suites de transitions possibles suivantes:

1. `a; b; STOP |[a,b,c]| (a; b; STOP [] a; c; STOP)`
   $\xrightarrow{a}$ -- Règle (16.2)
   `b; STOP |[a,b,c]| b; STOP`
   $\xrightarrow{b}$ -- Règle (16.8)
   `STOP |[a,b,c]| STOP`

2. `a; b; STOP |[a,b,c]| (a; b; STOP [] a; c; STOP)`
   $\xrightarrow{a}$ -- Règle (16.3)
   `b; STOP |[a,b,c]| c; STOP`

Par contre, dans le deuxième cas, on a uniquement la suite de transitions suivante:

1. `a; b; STOP |[a,b,c]| (a; (b; STOP [] c; STOP))`
   $\xrightarrow{a}$ -- Règle (16.8)
   `b; STOP |[a,b,c]| (b; STOP [] c; STOP)`
   $\xrightarrow{b}$ -- Règle (16.8)
   `STOP |[a,b,c]| STOP`

On voit donc que les comportements ne sont pas identiques car dans le premier cas il est possible que seule la séquence d'actions contenant uniquement l'action `a` soit exécutée, ce qui n'est pas le cas pour la deuxième expression. En d'autres mots, il est possible pour la première expression de *refuser* d'exécuter l'action `b`. Cette notion de *refus* est importante et est utilisée pour définir la sémantique des expressions de comportement dans le cadre de la notation CSP [Hoa85], notation dont se sont inspirés les concepteurs de `LOTOS` (synchronisation multiple).

16.6 Équivalences entre processus

Une première façon de vérifier certaines propriétés d'un processus consiste à s'assurer que le comportement de ce processus est équivalent (non distinguable) d'un autre processus pour lequel les propriétés requises sont plus faciles à vérifier. Par exemple, soit le processus `AccepterLivrer1` présenté à la figure 16.27.

Il est relativement facile de voir que, pour un observateur externe, le comportement de ce processus consiste en une alternance (sans fin) d'actions `accepter` et `livrer`. Le fait que ce comportement est obtenu par l'intermédiaire de processus auxiliaires (`Accepteur` et `Livreur`) peut être vu comme une forme de mise en œuvre, invisible du point de vue d'un observateur externe. Soit alors le processus présenté à la figure 16.28.

Il est encore plus clair, dans ce cas, que le comportement de ce dernier processus consiste aussi en une alternance stricte (et sans fin) d'actions `accepter` et `livrer`. Ce processus peut, à cause de sa simplicité, être vu comme une spécification abstraite, de haut niveau. S'il était possible de prouver que le comportement du premier processus (fig. 16.27) est équivalent au comportement de `AccepterLivrer2`, on pourrait alors dire que le premier processus, vu comme une mise en œuvre (plein de détails), *satisfait* la spécification représentée par le processus `AccepterLivrer2`. [7]

Dans les pages qui viennent, nous allons donc étudier diverses notions d'*équivalence* entre processus. De façon informelle, deux processus sont équivalents en termes d'observation lorsque, pour un observateur externe, les comportements des deux processus ne peuvent pas être distingués l'un de l'autre. Dans les sections qui suivent, nous allons examiner différentes façons de définir une telle équivalence entre processus. Une première définition sera, on verra bientôt en quel sens, trop forte puisqu'elle considèrera comme non-équivalents certains processus qui ne peuvent pas être distingués par un observateur externe. La définition suivante correspondra mieux à la notion

[7] Notons que de telles vérifications peuvent effectivement être effectuées, et ce de façon automatique, à l'aide d'outils développés pour l'analyse de spécifications LOTOS — par exemple, CADP (*Caesar Aldebaran Development Package*) [GJM+97].

```
SPECIFICATION AccepterLivrer1[accepter, livrer]: NOEXIT
BEHAVIOUR
  HIDE envoyer, attendre IN
    Accepteur[accepter, envoyer, attendre]
    |[envoyer, attendre]|
    Livreur[envoyer, attendre, livrer]
WHERE
  PROCESS Accepteur[accepter, envoyer, attendre]: NOEXIT :=
    accepter ?x: bool;
    envoyer !x;
    attendre;
    Accepteur[accepter, envoyer, attendre]
  ENDPROC

  PROCESS Livreur[recevoir, confirmer, livrer]: NOEXIT :=
    recevoir ?x: bool;
    livrer !x;
    confirmer;
    Livreur[recevoir, confirmer, livrer]
  ENDPROC
ENDSPEC
```

Figure 16.27: Le processus `AccepterLivrer1`

```
SPECIFICATION AccepterLivrer2[accepter, livrer]: NOEXIT
BEHAVIOUR
  AL[accepter, livrer]
WHERE
  PROCESS AL[accepter, livrer]: NOEXIT :=
    accepter ?x: bool;
    livrer !x;
    AL[accepter, livrer]
  ENDPROC
ENDSPEC
```

Figure 16.28: Le processus `AccepterLivrer2`

de "comportement observable équivalent"; toutefois, elle ne possèdera pas une propriété importante, à savoir la propriété de *congruence*. Nous terminerons alors par une relation, intermédiaire entre les deux précédentes, qui incorporera les deux aspects de comportement observable équivalent et de congruence, donc qui définira véritablement une égalité entre processus.

Mentionnons, avant d'aborder plus en détails ces notions, qu'il existe de nombreuses définitions de la notion d'équivalence entre processus: ainsi [vG90], publié en 1990, compare 12 d'entre elles! Celles que nous allons présenter plus en détails dans les sections 16.6.2 à 16.6.4 correspondent à la classe des équivalences de type "*branching time*", classe qui établit le plus de distinctions entre les processus (classes d'équivalence les plus fines) et qui correspondent le mieux à l'idée de "comportement observable identique". D'autres types d'équivalences sont plutôt basées sur une notion de "*linear time*" et sont brièvement présentées à la section 16.6.1.

16.6.1 Équivalences en termes d'ensembles de traces

Les approches d'équivalence de type *temps linéaire* considèrent comme identiques des processus dont les ensembles de traces (séquences d'actions possibles) sont égaux. En d'autres mots, si deux processus ont le même ensemble de traces possibles, alors ils sont identiques. Toutefois, il faut noter qu'il existe différentes façons de définir l'ensembles de traces d'un processus. Nous en examinerons quelques-unes dans cette section.

La première façon de définir l'équivalence entre processus en termes de traces consiste à définir une trace simplement comme une séquence d'actions, *complète ou partielle*, pouvant être effectuées par un processus. De façon plus formelle, on a donc la définition suivante, où `P` $\xrightarrow{t}$ `P'` dénote la transformation de l'expression de comportement `P` en l'expression `P'` via la séquence d'actions t:

`traces(P)`
$= \{\ t$: `sequence{action} SUCH THAT SOME(` `P'`: `Processus :: P` $\xrightarrow{t}$ `P'` `) ::` $t\}$

Par exemple, soit les processus suivants:

```
P1:  a; STOP [] b; STOP
P2:  a; b; STOP
P3:  a; STOP [] a; b; STOP
```

Dans ce cas, les ensemble de traces associés à chacun de ces processus seraient les suivants, où ϵ dénote la trace vide, donc celle où aucune action n'a encore été effectuée — pour tout `P` on a donc toujours `P` $\xrightarrow{\epsilon}$ `P`:

```
traces(P1) = {ε, a, b}
traces(P2) = {ε, a, ab}
traces(P3) = {ε, a, ab}
```

Dans le cas du processus `P2`, on remarque que la trace `a` est incluse car elle correspond à une trace *partielle*. Par contre, dans le cas de `P1` cette même trace correspond à une trace complète en ce sens qu'elle ne pourra plus ensuite être étendue (parce que le processus associé est bloqué): on dit alors d'une telle trace qu'elle est *maximale*. Finalement, dans le cas du processus `P3`, cette même trace représente à la fois une trace partielle (alternative de droite) et une trace maximale (alternative de gauche).

Dans cet exemple, on remarque que les deux derniers processus ont le même ensemble de traces. De ce point de vue, ces processus devraient donc être considérés comme équivalents. Toutefois, comme on l'a vu précédemment, il est possible de distinguer ces deux processus l'un de l'autre: le processus `P3` peut parfois refuser de faire l'action `b` alors que `P2` acceptera toujours de faire cette action.

Plutôt que de travailler avec des traces partielles, une alternative consiste à utiliser des traces maximales, donc des traces qui ne peuvent plus être étendues. Dans ce cas, les processus présentés plus haut seraient alors associés aux ensembles de traces suivants:

```
traces(P1) = {a, b}
traces(P2) = {ab}
traces(P3) = {a, ab}
```

Avec cette nouvelle définition, ces trois processus seraient alors considérés comme tous différents les uns des autres. Toutefois, considérons maintenant les processus suivants:

```
P4:  a; b; STOP [] (a; (b; STOP [] c; STOP))
P5:  a; (b; STOP [] c; STOP)
```

Ici, les deux processus seraient associés au même ensemble de traces maximales: {`ab, ac`}. Par contre, il serait quand même possible de les distinguer: après avoir fait l'action `a`, `P5` est toujours disponible pour faire `b` ou `c`; par contre, dans le cas de `P4`, si l'alternative gauche est choisie, alors après avoir fait `a`, `P4` pourrait refuser de faire `c`.

Une notion possible d'équivalence consiste donc à associer à chaque processus un ensemble de traces *et* un ensemble de refus [Hoa85]. Deux processus sont alors équivalents lorsqu'ils sont associés au même ensemble de traces et peuvent refuser de faire les mêmes actions. Toutefois, même cette notion ne serait pas suffisante pour bien distinguer tous les processus. La façon la plus précise de distinguer des processus utilisant plutôt la notion de *bisimilarité*, notion que nous introduisons dans la prochaine section.

16.6.2 Bisimilarité forte

De façon informelle, deux processus sont bisimilaires lorsque les deux processus peuvent, en tout temps, faire une même action. En d'autres mots, ils peuvent se simuler l'un et l'autre, ce qui rend alors impossible de les distinguer.

Une des définitions les plus fortes de bisimilarité (qui établit le plus de distinctions entre processus) est la suivante.

- Définition: Une relation binaire $\mathcal{R}$ entre processus est une *bisimulation forte* si, lorsque $P \mathcal{R} Q$, alors pour toute action α (y compris l'action invisible `I`) on a:

  1. $P \stackrel{\alpha}{\rightarrow} P'$ implique qu'il existe Q' tel que $Q \stackrel{\alpha}{\rightarrow} Q'$ et $P' \mathcal{R} Q'$.
  2. $Q \stackrel{\alpha}{\rightarrow} Q'$ implique qu'il existe P' tel que $P \stackrel{\alpha}{\rightarrow} P'$ et $P' \mathcal{R} Q'$.

  En d'autres mots, lorsque P peut effectuer une action, Q peut alors effectuer la même action (et vice-versa) et la transition associée à cette action doit conduire les deux processus dans des états qui sont eux-mêmes bisimilaires.

- Définition: On dit que $P \sim Q$ (P et Q sont fortement équivalents) lorsqu'il existe une relation de bisimulation forte $\mathcal{R}$ telle que $P \mathcal{R} Q$.

Exemple Soient les deux expressions de comportement suivantes:

```
a; b; STOP [] a; c; STOP
a; c; STOP [] a; b; STOP
```

Ces deux expressions sont fortement bisimilaires tel qu'illustré par la relation suivante:

```
{ (a; b; STOP [] a; c; STOP, a; c; STOP [] a; b; STOP),
  (b; STOP, b; STOP),
  (c; STOP, c; STOP),
  (STOP, STOP) }
```

Exemple Soient les deux expressions de comportement suivantes:

```
a; STOP ||| b; STOP
a; b; STOP [] b; a; STOP
```

Ces deux expressions sont fortement bisimilaires tel qu'illustré par la relation suivante:

```
{ (a; STOP ||| b; STOP, a; b; STOP [] b; a; STOP),
  (STOP ||| b; STOP, b; STOP),
  (a; STOP ||| STOP, a; STOP),
  (STOP ||| STOP, STOP) }
```

Quelques équivalences découlant de la bisimilarité forte

Un intérêt de la notion de bisimilarité est qu'elle permet d'établir des équivalences générales entre processus, par exemple, associativité, commutativité.

Par exemple, soient `P1`, `P2` et `P3` des expressions de comportement. La notion de bisimilarité forte permet de conclure que les expressions suivantes sont fortement équivalentes [8]:

- `P1 [] P2` $\sim$ `P2 [] P1`
- `P1 [] (P2 [] P3)` $\sim$ `(P1 [] P2) [] P3`
- `P1 [] P1` $\sim$ `P1`
- `P1 [] STOP` $\sim$ `P1`

On a aussi les lois suivantes:

- `P1 ||| P2` $\sim$ `P2 ||| P1`
- `P1 ||| (P2 ||| P3)` $\sim$ `(P1 ||| P2) ||| P3`
- `P1 ||| STOP` $\sim$ `P1`

Par contre, la loi suivante *n'est pas* valide:

- `P1 ||| P1` $\sim$ `P1`

Par exemple, les deux processus suivants ne sont pas bisimilaires, puisque celui de gauche peut exécuter deux fois l'action `a` alors que celui de droite ne peut exécuter `a` qu'une seule fois:

- `(a; STOP ||| a; STOP)` $\not\sim$ `a; STOP`

16.6.3 Bisimilarité faible

La notion de bisimilarité introduite précédemment est trop forte en ce sens qu'elle oblige à distinguer des processus qui ne peuvent pas, du point de vue d'un observateur externe, être distingués. Ceci survient parce que la nécessité de simuler une action s'applique pour toutes les actions, *y compris les actions invisibles*, donc non-observables. En d'autres mots, elle considère deux processus comme étant équivalents si et seulement si tant leur comportement observable que leur comportement interne sont identiques.

Soit les deux expressions de comportement suivantes:

```
a; STOP
I; a; STOP
```

La notion de bisimilarité forte introduite plus haut ne nous permet pas de conclure que ces deux expressions sont fortement équivalentes, puisque la première ne pourra jamais simuler l'action `I` de la deuxième. Par contre, il est clair que, du point de vue d'un observateur externe, ces deux comportements ne peuvent pas être distingués l'un de l'autre, `I` étant invisible, donc non-observable.

Cette distinction établie par la bisimilarité forte n'est pas intéressante du point de vue conception et raffinement par abstraction. Par exemple, il est clair que les processus définissant la spécification d'un service de communication et sa réalisation par un protocole de communication n'auront pas le même comportement interne; ce qui nous importe est simplement que les comportements observables soient les mêmes, et ce peu importe le contexte dans lequel ces processus sont utilisés.

Une première tentative pour définir une relation plus faible de bisimilarité consiste à ignorer les actions invisibles et conduit à la notion de *bisimilarité faible*. Introduisons les notations suivantes:

- Notons par $P \stackrel{I^*}{\rightarrow} P'$ la transformation de P en P' via une séquence d'actions invisibles. Cette séquence peut possiblement être vide, donc on peut avoir $P \equiv P'$.
- Notons par $\widehat{\alpha}$ l'une des deux valeurs suivantes selon la valeur de α:

[8] Ces lois indiquent que les expressions de comportement avec `[]` et `STOP` forment un monoïde.

1. $\alpha = I$: Dans ce cas, $\widehat{I} = I^*$.
2. $\alpha = a \neq I$: Dans ce cas, $\widehat{a} = I^* a I^*$.

Donc, $\widehat{a}$ $(a \neq I)$ indique qu'une action a est effectuée, mais possiblement précédée ou suivie d'une séquence (possiblement vide) d'actions invisibles.

- Notons par $P \stackrel{\widehat{\alpha}}{\rightarrow} P'$ la transformation de P en P' soit par une séquence (possiblement vide) d'actions invisibles ($\alpha = I$), soit par une action α suivie ou précédée d'actions invisibles.
- Définition: Une relation binaire $\mathcal{R}$ entre processus est une *bisimulation faible* si, lorsque $P \mathcal{R} Q$, alors pour toute action α on a:

  1. $P \stackrel{\alpha}{\rightarrow} P'$ implique qu'il existe Q' tel que $Q \stackrel{\widehat{\alpha}}{\rightarrow} Q'$ et $P' \mathcal{R} Q'$.
  2. $Q \stackrel{\alpha}{\rightarrow} Q'$ implique qu'il existe P' tel que $P \stackrel{\widehat{\alpha}}{\rightarrow} P'$ et $P' \mathcal{R} Q'$.

  En d'autres mots, lorsque P peut effectuer une action, Q peut alors effectuer la même action, mais possiblement précédée ou suivie d'actions invisibles (et vice-versa); de plus, les transitions associées à ces actions doivent conduire les deux processus dans des états qui sont eux-mêmes faiblement bisimilaires.
- Définition: On dit que $P \approx Q$ (P et Q sont faiblement équivalents) lorsqu'il existe une relation de bisimulation faible $\mathcal{R}$ telle que $P \mathcal{R} Q$.

Exemple Soient les deux expressions de comportement suivantes:

```
I; a; b; STOP
a; b; STOP
```

Ces deux expressions sont faiblement bisimilaires tel qu'illustré par la relation suivante:

```
{ (I; a; b; STOP, a; b; STOP),
  (a; b; STOP, a; b; STOP),
  (b; STOP, b; STOP),
  (STOP, STOP) }
```

Exemple Soient les deux expressions de comportement suivantes:

```
I; a; b; STOP [] a; I; b; STOP
a; b; I; STOP
```

Ces deux expressions sont faiblement bisimilaires tel qu'illustré par la relation suivante:

```
{ (I; a; b; STOP [] a; I; b; STOP, a; b; I; STOP),
  (a; b; STOP, a; b; I; STOP),
  (I; b; STOP, b; I; STOP),
  (b; STOP, b; I; STOP),
  (STOP, I; STOP),
  (STOP, STOP) }
```

16.6.4 Congruence en termes d'observation

La notion de bisimilarité faible, bien qu'intéressante en termes de comportement observable puisqu'elle permet d'ignorer les actions invisibles, n'est pas idéale car elle ne définit pas une relation de *congruence*. Une relation de congruence entre processus est une relation d'équivalence qui permet, *dans n'importe quel contexte*, de remplacer un processus par un autre processus équivalent sans qu'aucune différence ne puisse être observée. Or, comme le montre l'exemple qui suit, la bisimilarité faible ne possède pas cette propriété. [9]

[9] Il faut noter que la relation de bisimilarité forte possède la propriété de congruence; toutefois, comme on l'a vu précédemment, cette forme de relation ne définit pas une relation d'équivalence en termes d'observation.

Exemple Soient les expressions de comportement suivantes, pour lesquelles on a vu qu'elles étaient faiblement équivalentes:

```
a; STOP
I; a; STOP
```

Substituons alors chacune de ces expressions dans le contexte "P `[] b; STOP`" pour obtenir les deux expressions suivantes:

```
a; STOP [] b; STOP
I; a; STOP [] b; STOP
```

Les expressions de comportement résultantes ne sont pas bisimilaires, ni fortement, ni faiblement. Il est facile de voir, de façon informelle, que ces expressions peuvent être distinguées l'une de l'autre: alors que le premier processus permet toujours, tant qu'une action `a` n'a pas été effectuée, d'effectuer une action `b`, le deuxième processus peut, si la transition interne s'effectue, rendre impossible toute action `b`; en d'autres mots, le processus pourrait refuser d'effectuer l'action `b`. Il est donc possible de distinguer ces deux processus.

Définir une bonne relation d'équivalence entre processus qui soit aussi une relation de congruence est donc loin d'être une tâche triviale, ce qui explique les nombreuses variantes qui ont été proposées dans la littérature. Pour obtenir, à partir de la notion de bisimilarité faible, une relation de congruence entre processus, il est nécessaire de s'assurer que si un processus peut, spontanément, effectuer une transition invisible, alors l'autre processus peut aussi faire une transition interne équivalente. On en arrive alors à la définition qui suit.

On dit que P et Q sont *congruents en termes d'observation*, noté $P = Q$, lorsque les conditions suivantes sont satisfaites:

1. $P \approx Q$
2. $P \stackrel{\widehat{I}}{\rightarrow} P'$ implique qu'il existe Q' tel que $Q \stackrel{\widehat{I}}{\rightarrow} Q'$ et $P' = Q'$.
3. $Q \stackrel{\widehat{I}}{\rightarrow} Q'$ implique qu'il existe P' tel que $P \stackrel{\widehat{I}}{\rightarrow} P'$ et $P' = Q'$.

Exemple Soient les deux expressions de comportement suivantes:

```
I; a; b; STOP
I; I; a; b; STOP
```

Ces deux expressions sont congruentes en termes d'observation. Par contre, comme on l'a vu précédemment, les deux expressions suivantes ne le sont pas:

```
a; b; STOP
I; a; b; STOP
```

Exemple Soient les deux expressions de comportement suivantes:

```
a; b; STOP
a; b; I; STOP
```

Ces deux expressions sont congruentes en termes d'observation.

```
MACHINE banque
  IMPORT client FROM client
  IMPORT montant FROM montant

  STATE( soldes: map{client, montant} )
  INVARIANT
    -- La balance d'un client ne peut jamais
    -- etre negative.
    ALL( c: client SUCH THAT client_actif(c) :: soldes[c] >= 0.0 )
  INITIALLY
    soldes = create(!)

  MESSAGE ajouter_client( c: client, solde_initial: montant )
    WHEN ~client_actif(c) & solde_initial > 0.0
      REPLY ajout_confirme
      TRANSITION
        soldes = bind( c, solde_initial, *soldes )
    WHEN client_actif(c)
      REPLY EXCEPTION client_deja_existant
    OTHERWISE -- ~client_actif(c) & solde_initial <= 0.0
      REPLY EXCEPTION solde_initial_invalide

  MESSAGE solde( c: client )
    WHEN client_actif(c)
      REPLY( m: montant )
        WHERE m = soldes[c]
    OTHERWISE -- ~client_actif(c)
      REPLY EXCEPTION client_invalide

  MESSAGE crediter( c: client, m: montant )
    WHEN client_actif(c) & m > 0.0
      REPLY depot_effectue
      TRANSITION
        soldes = bind( c, *soldes[c] + m, *soldes )
    WHEN ~client_actif(c)
      REPLY EXCEPTION client_invalide
    OTHERWISE -- client_actif(c) & m <= 0.0
      REPLY EXCEPTION montant_invalide

  MESSAGE debiter( c: client, m: montant )
    WHEN client_actif(c) & 0.0 < m <= solde(c)
      REPLY retrait_effectue
      TRANSITION
        soldes = bind( c, *soldes[c] - m, *soldes )
    WHEN ~client_actif(c)
      REPLY EXCEPTION client_invalide
    WHEN m <= 0.0
      REPLY EXCEPTION montant_invalide
    OTHERWISE -- client_actif(c) & m > solde(c)
      REPLY EXCEPTION montant_trop_eleve

  CONCEPT client_actif( c: client ) VALUE( b: boolean )
    WHERE b <=> c IN soldes
END
```

Figure 16.29: Spécification `Spec` d'une banque

16.7 L'exemple de la banque

L'exemple qui suit est une version `LOTOS` d'une spécification `Spec` vue précédemment (Chap. 6) qui définissait une machine pour la gestion d'une banque. La spécification `Spec` de cette banque est présentée à la figure 16.29. La spécification `LOTOS` correspondante est présentée et expliquée dans les pages qui suivent.

```
SPECIFICATION banque
[(* Constructeurs/mutateurs *) ajouter_client, debiter, crediter,
 (* Observateur *) solde,
 (* Reponses *) ajout_confirme, solde_reponse, depot_effectue, retrait_effectue,
 (* Exceptions *) client_deja_existant, solde_initial_invalide, client_invalide,
                  montant_invalide, montant_trop_eleve
]
: NOEXIT
BEHAVIOUR
  banque[ajouter_client, debiter, crediter, solde, ajout_confirme, solde_reponse,
         depot_effectue, retrait_effectue, client_deja_existant,
         solde_initial_invalide, client_invalide, montant_invalide,
         montant_trop_eleve]
         ( create(!) )
WHERE
 PROCESS banque
          [ajouter_client, debiter, crediter, solde, ajout_confirme,
           solde_reponse, depot_effectue, retrait_effectue,
           client_deja_existant, solde_initial_invalide, client_invalide,
           montant_invalide, montant_trop_eleve]
          ( soldes: map{client, montant} )
          : NOEXIT :=

  AjouterClient[ajouter_client, debiter, crediter, solde, ajout_confirme,
                solde_reponse, depot_effectue, retrait_effectue,
                client_deja_existant, solde_initial_invalide, client_invalide,
                montant_invalide, montant_trop_eleve]
               ( soldes )
  []
  Solde[ajouter_client, debiter, crediter, solde, ajout_confirme,
        solde_reponse, depot_effectue, retrait_effectue,
        client_deja_existant, solde_initial_invalide, client_invalide,
        montant_invalide, montant_trop_eleve]
       ( soldes )


  []
  Crediter[ajouter_client, debiter, crediter, solde, ajout_confirme,
           solde_reponse, depot_effectue, retrait_effectue,
           client_deja_existant, solde_initial_invalide, client_invalide,
           montant_invalide, montant_trop_eleve]
          ( soldes )
  []
  Debiter[ajouter_client, debiter, crediter, solde, ajout_confirme,
          solde_reponse, depot_effectue, retrait_effectue,
          client_deja_existant, solde_initial_invalide, client_invalide,
          montant_invalide, montant_trop_eleve]
         ( soldes )
 ENDPROC


 (* --- Les processus auxiliaires de traitement des divers messages --- *)

 PROCESS AjouterClient
         [ajouter_client, debiter, crediter, solde, ajout_confirme,
          solde_reponse, depot_effectue, retrait_effectue,
          client_deja_existant, solde_initial_invalide, client_invalide,
          montant_invalide, montant_trop_eleve]
         ( soldes: map{client, montant} )
         : NOEXIT :=
 ajouter_client ?c: client ?solde_initial: montant;
 ([~client_actif(c) & solde_initial > 0.0] ->
     ajout_confirme;
```

```
    banque[ajouter_client, debiter, crediter, solde, ajout_confirme,
           solde_reponse, depot_effectue, retrait_effectue,
           client_deja_existant, solde_initial_invalide, client_invalide,
           montant_invalide, montant_trop_eleve]
          ( bind(c, solde_initial, soldes) )
 []
 [client_actif(c)] ->
    client_deja_existant;
    banque[ajouter_client, debiter, crediter, solde, ajout_confirme,
           solde_reponse, depot_effectue, retrait_effectue,
           client_deja_existant, solde_initial_invalide, client_invalide,
           montant_invalide, montant_trop_eleve]
          ( soldes )
 []
 [solde_initial <= 0.0] ->
    solde_initial_invalide;
    banque[ajouter_client, debiter, crediter, solde, ajout_confirme,
           solde_reponse, depot_effectue, retrait_effectue,
           client_deja_existant, solde_initial_invalide, client_invalide,
           montant_invalide, montant_trop_eleve]
          ( soldes )
)
ENDPROC

(* --- *)
PROCESS Solde
        [ajouter_client, debiter, crediter, solde, ajout_confirme,
         solde_reponse, depot_effectue, retrait_effectue,
         client_deja_existant, solde_initial_invalide, client_invalide,
         montant_invalide, montant_trop_eleve]
        ( soldes: map{client, montant} )
        : NOEXIT :=
solde ?c: client;
([client_actif(c)] ->
    solde_reponse !soldes[c];
    banque[ajouter_client, debiter, crediter, solde, ajout_confirme,
           solde_reponse, depot_effectue, retrait_effectue,
           client_deja_existant, solde_initial_invalide, client_invalide,
           montant_invalide, montant_trop_eleve]
          ( soldes )
 []
 [~client_actif(c)] ->
   client_invalide;
    banque[ajouter_client, debiter, crediter, solde, ajout_confirme,
           solde_reponse, depot_effectue, retrait_effectue,
           client_deja_existant, solde_initial_invalide, client_invalide,
           montant_invalide, montant_trop_eleve]
          ( soldes )
)
ENDPROC

(* --- *)
PROCESS Crediter
        [ajouter_client, debiter, crediter, solde, ajout_confirme,
         solde_reponse, depot_effectue, retrait_effectue,
         client_deja_existant, solde_initial_invalide, client_invalide,
         montant_invalide, montant_trop_eleve]
        ( soldes: map{client, montant} )
        : NOEXIT :=
crediter ?c: client ?m: montant;
([client_actif(c) & m > 0.0] ->
    depot_effectue;
    banque[ajouter_client, debiter, crediter, solde, ajout_confirme,
           solde_reponse, depot_effectue, retrait_effectue,
           client_deja_existant, solde_initial_invalide, client_invalide,
           montant_invalide, montant_trop_eleve]
          ( bind(c, soldes[c] + m, soldes) )
 []
 [~client_actif(c)] ->
    client_invalide;
```

```
    banque[ajouter_client, debiter, crediter, solde, ajout_confirme,
           solde_reponse, depot_effectue, retrait_effectue,
           client_deja_existant, solde_initial_invalide, client_invalide,
           montant_invalide, montant_trop_eleve]
          ( soldes )
 []
 [m <= 0.0] ->
    montant_invalide;
    banque[ajouter_client, debiter, crediter, solde, ajout_confirme,
           solde_reponse, depot_effectue, retrait_effectue,
           client_deja_existant, solde_initial_invalide, client_invalide,
           montant_invalide, montant_trop_eleve]
          ( soldes )
)
ENDPROC

(* --- *)
PROCESS Debiter
        [ajouter_client, debiter, crediter, solde, ajout_confirme,
         solde_reponse, depot_effectue, retrait_effectue,
         client_deja_existant, solde_initial_invalide, client_invalide,
         montant_invalide, montant_trop_eleve]
        ( soldes: map{client, montant} )
        : NOEXIT :=
debiter ?c: client ?m: montant;
([client_actif(c) & 0.0 < m <= soldes[c]] ->
   retrait_effectue;
   banque[ajouter_client, debiter, crediter, solde, ajout_confirme,
          solde_reponse, depot_effectue, retrait_effectue,
          client_deja_existant, solde_initial_invalide, client_invalide,
          montant_invalide, montant_trop_eleve]
         ( bind(c, soldes[c] - m, soldes) )
 []
 [~client_actif(c)] ->
   client_invalide;
   banque[ajouter_client, debiter, crediter, solde, ajout_confirme,
          solde_reponse, depot_effectue, retrait_effectue,
          client_deja_existant, solde_initial_invalide, client_invalide,
          montant_invalide, montant_trop_eleve]
         ( soldes )
 []
 [m <= 0.0] ->
   montant_invalide;
   banque[ajouter_client, debiter, crediter, solde, ajout_confirme,
          solde_reponse, depot_effectue, retrait_effectue,
          client_deja_existant, solde_initial_invalide, client_invalide,
          montant_invalide, montant_trop_eleve]
         ( soldes )
 []
 [m > soldes[c]] ->
   montant_trop_eleve;
   banque[ajouter_client, debiter, crediter, solde, ajout_confirme,
          solde_reponse, depot_effectue, retrait_effectue,
          client_deja_existant, solde_initial_invalide, client_invalide,
          montant_invalide, montant_trop_eleve]
         ( soldes )
)
ENDPROC

ENDSPEC
```

Il est possible d'établir la correspondance entre la version `Spec` et la version `LOTOS` comme suit:

- Pour chaque message ou réponse associé à la version `Spec` de la banque, y compris les exceptions, la version `LOTOS` possède une action (argument générique de la spécification) correspondante du même nom.

- La spécification de haut niveau (`banque`) définit, pour chaque message de la version `Spec`, un processus auxiliaire de nom équivalent (par exemple, `AjouterClient`, `Solde`, etc.) dont le rôle est de traiter ce message. C'est à l'intérieur de chacun de ces processus que les informations appropriées (arguments) seront reçues par une action avec déclaration de variable, action dont le nom correspond lui aussi au nom de message de la version `Spec` (par exemple, `ajouter_client`, `solde`).

- La structure générale de fonctionnement de chacune des opérations est semblable dans les deux cas (`Spec` et `LOTOS`): un message avec arguments est reçu puis différentes réponses sont produites en fonction de la valeur des arguments. Alors que dans la spécification `Spec` on utilisait des clauses `WHEN` et `OTHERWISE`, on utilise plutôt, dans la spécification `LOTOS`, des gardes où les conditions sont toujours explicites (pas de clause `OTHERWISE`).

Par contre, les deux spécifications diffèrent au niveau de certains éléments qui sont explicites en `Spec` mais implicites en `LOTOS`:

- En `Spec`, chaque action ou événement est clairement identifié comme étant soit un message pouvant être reçu par le module (par exemple, `MESSAGE ajouter_client`), soit une réponse normale produite par le module (par exemple, `REPLY ajout_confirme`), soit une exception signalée par le module (par ex., `REPLY EXCEPTION client_invalide`). Par contre, en `LOTOS`, tous ces aspects sont modélisés par des noms d'action: les informations sur le *rôle* de l'action sont donc implicites et sont indiquées, dans notre exemple, sous forme de commentaires.

- En `Spec`, les changement d'état sont clairement identifiables et correspondent aux messages pour lesquels une clause `TRANSITION` est indiquée. En `LOTOS`, par contre, les changements d'état sont implicites et correspondent à des appels récursifs au processus `banque` pour lesquels les arguments formels entre "(...)" *ne sont pas les mêmes* que ceux en entrée. Par exemple, l'action `solde` effectuée par le processus auxiliaire `Solde` n'entraine pas de changement d'état (on réutilise `soldes`, reçu en entrée, comme argument dans l'appel récursif) alors que `crediter` conduit à une véritable modification de l'état interne de la banque (on utilise "`bind(c, soldes[c]+m, soldes)`" comme argument dans l'appel récursif plutôt que la variable `soldes` reçue en entrée).

  Cette différence au niveau des transitions et changements d'état entre `Spec` et `LOTOS` tient à ce que l'approche `LOTOS` est basée sur une approche purement fonctionnelle. [10]

- Un aspect un peu lourd de la spécification `LOTOS` est la nécessité de répéter, pour chaque appel de processus associé à une opération et pour chaque appel récursif, la longue liste des ports, liste qui inclut toutes les interactions possibles du système (messages, réponses, exceptions). Comme on le voit dans l'exemple, cela alourdit nettement la spécification et la rend plus longue (plus du double) que la version `Spec`.

[10] Ceci est vrai aussi dans le cas où la spécification des types de données est faite en utilisant le langage `ACT ONE`, langage basé sur la méthode algébrique de spécification types abstraits de données. Les types ainsi spécifiés définissent toujours des types abstraits *immuables*, et non des classes d'objets.

Exercices

16.1.
Définissez un processus LOTOS (son en-tête est donnée plus bas) qui va exécuter l'action `in` et qui va permettre d'exécuter l'action `oui` uniquement lorsque le nombre total d'actions `in` effectuées depuis le début est divisible par 2. (Note: 0 est divisible par 2!):

```
PROCESS Pair[in, oui]: NOEXIT
```

16.2.

16.2.1. Que fait le processus suivant:

```
PROCESS Proc[entree, oui, non]: NOEXIT :=
  oui; Proc[entree, oui, non]
  []
  entree; Proc[entree, non, oui]
ENDPROC
```

16.2.2. Dessinez le diagramme de transition de ce processus.

16.3.
Dans vos propres mots, décrivez le comportement du client défini par le processus LOTOS suivant:

```
PROCESS Client[p25, b_bisc, b_muff, bisc, muff]: NOEXIT :=
  p25; p25;
  ( p25; (b_bisc; bisc; STOP [] p25; b_muff; muff; STOP)
    []
    I; STOP
  )
ENDPROC
```

16.4.

16.4.1. En utilisant le processus `Max2`, définissez un processus qui va produire, sur le port `out`, le maximum parmi 4 nombres reçus sur les ports `in1, in2, in3, in4`.

16.4.2. Si on suppose que le traitement effectué par `Max2` prend 1 unité de temps et que le temps pour les communications est négligeable, quel sera le temps requis pour effectuer le calcul du maximum parmi 4 nombres?

16.5.
Définissez un processus LOTOS qui va recevoir un bit via le port `inb` et qui va produire immédiatement en sortie sur le port `outb` le complément du bit reçu. Le processus ne permettra donc pas de recevoir un nouveau bit tant que le complément de celui qui vient d'être reçu n'aura pas été envoyé. L'interface du processus sera la suivante:

```
PROCESS Complementer1[inb, outb]: NOEXIT
```

16.6.

16.6.1. Que fait le processus `Maxn` suivant?

```
PROCESS Maxn[nb, entree, sortie]: NOEXIT :=
  nb ?n: nat; Max[nb, entree, sortie](n, [])
ENDPROC
  WHERE
    PROCESS Max[nb, entree, sortie](n: nat, s: sequence{nat}): NOEXIT :=
      [length(s) < n] ->
```

```
      entree ?x: nat;
        Max[nb, entree, sortie](n, add(x, s))
    []
    [length(s) = n] ->
      sortie !MAXIMUM( i: nat SUCH THAT i IN domain(s) :: s[i] );
        Maxn[nb, entree, sortie]
ENDPROC
```

16.6.2. Que peut-on observer des expressions de comportement suivantes:

1. ```
   Maxn[nb, entree, sortie]
   |[nb, entree, sortie]|
   (nb !3; entree !10; entree !20; entree !30; sortie ?y: nat; STOP)
   ```

2. ```
   HIDE nb, entree IN
     Maxn[nb, entree, sortie]
     |[nb, entree, sortie]|
     (nb !3; entree !10; entree !20; entree !30; sortie ?y: nat; STOP)
   ```

3. ```
   HIDE entree IN
     Maxn[nb, entree, sortie]
     |[nb, entree, sortie]|
     (nb !4; entree !10; entree !20; entree !30; sortie ?y: nat; STOP)
   ```

16.7.
Définissez un processus LOTOS qui va recevoir des bits via le port `inb` et qui va produire en sortie, sur le port `outb`, le complément des bits reçus. Contrairement à l'exercice précédent, il ne sera pas nécessaire de retirer le complément aussitôt qu'il sera reçu; les bits pourront plutôt être conservés dans un tampon et retirés au fur et à mesure qu'ils seront requis.

16.8.
Supposons que l'on ait un processus, dont l'interface est donnée plus bas, qui remplace les *paires* de caractères de la forme "`**`" (reçus *un à la fois* par l'intermédiaire du port `entree`) par le caractère '`^`" transmis sur le port `sortie`. Les autres caractères sont transmis inchangés sur `sortie`. Le nom de ce processus est `Changer_exposant`, puisque "`**`" et "`^`" sont deux façons de représenter l'opérateur d'exponentiation (à la FORTRAN vs. à la Pascal ou Spec).

```
PROCESS Changer_exposant[entree, sortie]: NOEXIT
```

16.8.1. Donnez une spécification `LOTOS` de ce processus.

16.8.2. Combiner le processus `Changer_exposant` avec les processus `Paqueter` et `Depaqueter`, vus dans les notes de cours, de façon à produire une expression de comportement qui va lire en entrée (sur le port `entree`) des lignes de caractères, va remplacer les exposants du style "`**`" par ceux de style "`^`", va ensuite produire en sortie (sur le port `sortie`) des lignes de 125 caractères.

16.9.
Soit l'expression de comportement suivante:

```
(a; STOP [] b; c; STOP) || (a; STOP [] I; b; c; STOP [] d; STOP)
```

Trouvez une expression de comportement plus simple qui serait équivalente.

16.10.
Soient les deux expressions de comportement suivantes:

```
e1 = a; b; STOP [] a; c; STOP

e2 = a; (b; STOP [] c; STOP)
```
```

Donnez toutes les transitions possibles pour chacune des expressions suivantes:

1. `a; b; STOP |[b]| e1`
2. `a; b; STOP |[b]| e2`

16.11.
Donnez les règles de transition de la sémantique opérationnelle pour les opérateurs `||` et `|||`.

16.12.
Soit la fonction `Spec` suivante:

```
FUNCTION somme
  MESSAGE( x y: nat )
    REPLY( z: nat ) WHERE z = x + y
END
```

Définissez, en `LOTOS`, un `PROCESS Somme` qui réalise cette même fonction, mais sans lecture sur un port (il ne s'agit donc pas véritablement d'un processus, mais plutôt d'une fonction au sens classique). Puis, définissez un véritable processus `Somme2` qui utilise cette fonction comme suit, et ce de façon cyclique:

```
Lire deux valeurs sur les ports inx et iny.
Ecrire, sur le port outsomme, la somme des deux nombres lus.
```

16.13.
Soit l'exemple réalisant la conversion d'une série de 7 bits en 8 bits par ajout d'un bit de parité. Serait-il possible de réaliser la même tâche mais en transmettant les bits au fur et à mesure où ils sont reçus? Si oui, sans entrer dans les détails, comment procéderait-on? (En d'autres mots, quels sont les principaux changements que l'on devrait apporter aux processus de l'exemple?)

16.14.

16.14.1. Les expressions suivantes sont-elles fortement bisimilaires?

```
a; b; STOP
```

```
a; b; STOP || a; b; STOP
```

16.14.2. Pour une expression arbitraire `P`, les deux expressions suivantes sont-elles toujours bisimilaires?

```
P
```

```
P || P
```

16.15.
La figure 16.30 présente la spécification (partielle) d'une machine `Spec` qui permet de recevoir une série de valeurs entières (avec `ajouter_valeur`) et qui permet ensuite d'obtenir le maximum parmi les valeurs transmises (avec `obtenir_max`).

Définissez un processus `LOTOS` pouvant effectuer les mêmes actions, mais qui assure toutefois qu'`obtenir_max` ne peut être effectuée que si au moins une valeur a été transmise. De plus, il faut aussi permettre, en tout temps, d'annuler la série d'ajouts en cours par un appel à `reinitialiser`. Finalement, une fois que `nb_max_elems` éléments ont été ajoutés dans l'ensemble, alors aucun nouvel élément ne peut être ajouté avec `ajouter_valeur` (il faut donc s'assurer de faire respecter l'invariant).

Note: Vous pouvez utiliser la notation `Spec` pour définir les types de données ainsi que les concepts auxiliaires dont vous avez besoin.

```
MACHINE trouver_max{ nb_max_elems: nat SUCH THAT nb_max_elems > 0 }
  STATE( vals: set{integer} )
  INVARIANT size(vals) <= nb_max_elems
  INITIALLY vals = {}

  MESSAGE ajouter_valeur( v: integer )
    TRANSITION
      vals = *vals U {v}

  MESSAGE obtenir_max
    REPLY( m: integer )
      WHERE m = MAXIMUM( x:integer SUCH THAT x IN vals :: x )
    TRANSITION
      vals = {}

  MESSAGE reinitialiser
    TRANSITION
      vals = {}
END
```

Figure 16.30: Machine pour trouver le maximum parmi une série de nombres entiers

16.16.
Cet exercice porte sur le système de contrôle des feux de circulation présenté à la section 17.2.
Supposons qu'on ne puisse pas savoir, lorsqu'un feu (primaire ou secondaire) est initialement mis en marche, quelles lumières seront allumées ou éteintes.

16.16.1. Quelles modifications faudrait-il effectuer à la spécification des divers processus pour assurer le fonctionnement sécure du système?

16.16.2. Dans ce cas, serait-il acceptable d'éliminer la synchronisation `secondairePret` (le port `initTermine` dans le processus `SecondaireInit`)? Justifiez.

16.17. Soient les processus suivants:

```
PROCESS Proc1[a, b]: NOEXIT :=
  a; a; b; STOP
ENDPROC

PROCESS Proc2[a, b]: EXIT :=
  a; a; b; EXIT [] b; a; b; EXIT
ENDPROC
```

Donnez l'ensemble des traces complètes (maximales) possibles pour les expressions suivantes:

1. `Proc1[x, y] [] Proc2[x, y]`
2. `I; Proc1[x, y] || Proc1[x, y]`
3. `Proc1[x, y] || Proc2[x, y]`
4. `Proc1[x, y] |[x, y]| Proc2[x, y]`
5. `Proc1[x, y] |[x]| Proc1[x, y]`
6. `Proc1[x, y] |[x]| Proc2[x, y]`
7. `Proc1[x, y] ||| Proc1[x, y]`
8. `Proc2[x, y] |[x]| Proc2[x, y]`

16.18. Soient les processus suivants:

```
PROCESS Proc0[a, b](j: nat): EXIT :=
  a !j; b; EXIT
ENDPROC

PROCESS Procs[a, b](n: nat): EXIT :=
  [n = 0]  ->
    EXIT
  []
  [n ~= 0] ->
    Proc0[a, b](n)
    |||
    Procs[a, b](n-1)
ENDPROC
```

Donnez l'ensemble des traces complètes (maximales) possibles pour les expressions suivantes:

1. `Procs[a, b](3) |[a]| (a !1; EXIT [] a !2; EXIT)`
2. `Procs[a, b](3) |[a]| (a !1; EXIT ||| a !2; EXIT)`
3. `Procs[a, b](3) |[a]| (a !1; EXIT |[a]| a !2; EXIT)`

Note: Un effet équivalent pourrait être obtenu en définissant `Proc0` comme suit:

```
PROCESS Proc0[a, b](j: nat): EXIT :=
  a ?x: nat [x = j]; b; EXIT
ENDPROC
```

Chapitre 17

Spécifications de propriétés à l'aide de la logique temporelle

17.1 Propriétés d'un système réactif: sécurité et vivacité

La notation `LOTOS`, comme nous l'avons vu jusqu'à présent, permet de décrire le comportement d'un processus ou système en décrivant les suites possibles d'actions pouvant être effectuées par ce processus ou système. Cette forme de description est semblable à la façon de procéder dans un langage de programmation impératif classique: "le système fait cela, ensuite il fait cela ou bien cela, etc." On dit donc qu'il s'agit d'une description *opérationnelle*. Pour certains, ce qui est peut-être nouveau en `LOTOS` est le fait que le comportement ainsi décrit peut être concurrent, car on peut utiliser des opérateurs parallèles et de synchronisation.

Bien qu'une méthode opérationnelle soit utile pour décrire de façon détaillée les diverses étapes du comportement d'un système, elle possède malheureusement certaines faiblesses. Par exemple, alors qu'en `Spec` on pouvait utiliser une clause `INVARIANT` pour décrire une propriété devant toujours être respectée par le système, ceci, pour l'instant, n'est pas possible en `LOTOS`.

Un système réactif et concurrent est généralement composé d'un ensemble de processus qui collaborent entre eux pour réaliser le comportement global du système. Dans certains cas, bien qu'il puisse être relativement facile de voir que le comportement de chacun des processus satisfait une propriété donnée, rien n'assure que cette propriété sera ensuite satisfaite par l'ensemble des processus travaillant de façon concurrente. L'aspect concurrence introduit donc des difficultés supplémentaires.

Pour les systèmes réactifs et concurrents, une classe de méthodes souvent utilisée pour décrire des propriétés du comportement de tels système est la "*logique temporelle*". Contrairement à la logique classique qui ne traite pas explicitement de l'écoulement du temps, la logique temporelle introduit divers opérateurs, appelés *modalités*, qui permettent de traiter cet aspect. Par exemple, on peut indiquer qu'une proposition *sera toujours vraie*, ou encore, qu'elle *deviendra vraie* dans le futur.

Dans ce qui suit, nous introduisons et illustrons quelques-unes des propriétés généralement requises d'un système réactif et concurrent, et ce à l'aide d'exemples utilisant une logique temporelle combinant la logique HML [Mil89, Sti96, Bru97] et diverses modalités de la notation LTAC présentée dans [QS83]. Cette forme de logique est essentiellement celle utilisée dans l'outil `CADP` [GJM+97]. Toutes les propriétés présentées dans les exemples qui suivent ont donc été vérifiées à l'aide de cet outil. Notons toutefois que, pour des raisons de lisibilité, la syntaxe concrète que nous avons utilisée dans nos exemples diffère légèrement de celle traitée par l'outil `CADP`.

17.1.1 Les deux grandes classes de propriété

Avant d'aborder la présentation de la logique temporelle, il est bon de situer les deux principales classes de propriété généralement requises d'un système réactif et concurrent [Pnu86, MP92]:

- Propriétés de *sécurité*: Une propriété de sécurité (*safety*) vise à indiquer qu'un comportement non désiré est certain de ne pas survenir. En d'autres mots, une propriété de sécurité assure que rien de mauvais ne va arriver (*nothing bad will ever happen*).

- Propriétés de *vivacité*: Une propriété de vivacité (*liveness*) vise à indiquer qu'un comportement désiré est certain de survenir. En d'autres mots, une propriété de vivacité assure que quelque chose de bon va arriver (*something good will eventually happen*).

Les propriétés de sécurité peuvent, règle générale, être exprimées à l'aide d'invariants sur l'état du système. Par exemple, le fait que la condition `p` est certaine de ne jamais survenir peut être décrite par un invariant de la forme suivante:

```
INVARIANT ~p
```

Par contre, les propriétés de vivacité impliquent généralement non pas une quantification universelle sur tous les instants et états possibles du système mais plutôt une description de contraintes sur des événements futurs: *éventuellement*, une certaine propriété deviendra ou pourrait devenir vraie. On verra à la prochaine section comment modéliser ces deux formes de propriété.

17.1.2 Logique modale et temporelle

Dans la section qui suit, nous allons présenter certains éléments de base d'une logique modale et temporelle. Dans un premier temps, nous allons voir comment exprimer des propriétés sur les actions pouvant survenir, pour une expression de comportement donnée, à partir de l'instant présent. Nous verrons ensuite comment formuler des propriétés portant sur le comportement futur, propriétés inévitables, potentielles ou toujours vraies.

Modalités pour les actions

Formules valides (syntaxe)

Définissons l'ensemble Φ des formules modales valides de façon inductive comme suit:[1]

- `true`, `false` $\in \Phi$;
- si $\phi \in \Phi$ alors `not` $\phi \in \Phi$;
- si $\phi_1, \phi_2 \in \Phi$ alors ϕ_1 `and` $\phi_2 \in \Phi$;
- si $\phi_1, \phi_2 \in \Phi$ alors ϕ_1 `or` $\phi_2 \in \Phi$;
- si $\phi \in \Phi$ et A est un ensemble d'actions, alors `<`A`>`$\phi \in \Phi$;
- si $\phi \in \Phi$ et A est un ensemble d'actions, alors `[`A`]`$\phi \in \Phi$;

Interprétation (sémantique)

Une formule modale est interprétée relativement à une expression de comportement. En d'autres mots, la signification d'une formule modale est définie en fonction d'une expression donnée. Soit `e` une telle expression de comportement. La sémantique de diverses expressions modales peut alors être définie de façon inductive comme suit, où `e` $\models \phi$ indique que l'expression `e` *satisfait* la formule modale ϕ:

- `e` $\models$ `true`
- `e` $\not\models$ `false`
- `e` $\models$ `not` ϕ si et seulement si `e` $\not\models \phi$
- `e` $\models \phi_1$ `and` ϕ_2 si et seulement si `e` $\models \phi_1$ et `e` $\models \phi_2$
- `e` $\models \phi_1$ `or` ϕ_2 si et seulement si `e` $\models \phi_1$ ou `e` $\models \phi_2$

[1] Notons que la définition de cet ensemble de formules pourrait être simplifiée en utilisant les équivalences suivantes:

- `false` $\equiv$ `not true`;
- ϕ_1`or` $\phi_2 \equiv$ `not (not` ϕ_1`and not` ϕ_2`)`;
- `<`A`>`$\phi \equiv$ `not [`A`](not` ϕ`)`;

Il suffirait alors de présenter les règles introduisant `true`, `not`, `and` et `[]`, tant dans la description de l'ensembles des formules valides que dans la description de la sémantique présentée plus bas.

- `e` $\models$ `<`A`>`ϕ si et seulement si il existe `e'` et `a` $\in A$ tel que `e` $\xrightarrow{a}$ `e'` et `e'` $\models \phi$
- `e` $\models$ `[`A`]`ϕ ssi pour tout `e'` et `a` $\in A$ tel que `e` $\xrightarrow{a}$ `e'`, on a que `e'` $\models \phi$

Exemple Voyons quelques exemples simples de formules modales:

- `e` $\models$ `<`a`>true`: l'action `a` peut être effectuée par le processus `e`.
- `e` $\models$ `[`a`]false`: l'action `a` ne peut pas être effectuée par le processus `e`.
- `e` $\models$ `[`a`][`b`]false`: après avoir effectué une action `a`, il n'est pas possible pour `e` d'effectuer une action `b`.
- `e` $\models$ `[`a, b`]<`c, d`>true`: après avoir effectué une des actions `a` ou `b`, `e` peut aussi effectuer une des actions `c` ou `d`.
- `e` $\models$ `<`$*$`>true`: `e` peut effectuer une action (n'importe laquelle). Ici, le symbole "`*`" dénote l'ensemble de toutes les actions pouvant être effectuées par l'expression de comportement concernée.
- `e` $\models$ `[`$*$`]false`: `e` ne peut effectuer aucune action.

Modalités temporelles

Les modalités introduites plus haut permettent d'indiquer que certaines actions peuvent ou non être faites de façon immédiate par une expression de comportement `e`. Bien que ceci soit utile, ce n'est pas suffisant car cela ne permet pas d'exprimer que certaines propriétés pourront, *dans le futur*, devenir vraies.

Pour comprendre les modalités qui vont être introduites, il faut réaliser que, à cause de la présence possible de choix non-déterministes (opérateur `[]`), le futur d'une expression de comportement *n'est pas nécessairement unique*. En d'autres mots, il existe *plusieurs futurs possibles*. Les modalités qui suivent permettent donc de distinguer entre la *possibilité* qu'une propriété devienne vraie, *l'assurance* qu'elle le *deviendra* ou encore l'assurance qu'elle sera *toujours* vraie.

- `e` $\models$ `POT`(ϕ): Il existe un futur possible de l'expression de comportement `e` tel que la propriété ϕ *deviendra* vraie — ϕ est `POT`entiellement vraie. En d'autres mots, parmi les diverses alternatives possibles pour le comportement de `e`, l'une d'entre elles peut satisfaire la propriété ϕ.
- `e` $\models$ `INEV`(ϕ): Dans tous les futurs possibles de l'expression de comportement `e`, la propriété ϕ *deviendra* vraie — ϕ est `INEV`itable. En d'autres mots, pour toutes les alternatives possibles du comportement de `e`, chacune d'entre elles va éventuellement, à un moment futur, satisfaire ϕ.
- `e` $\models$ `ALL`(ϕ): Dès maintenant et dans tous les futurs possibles, la propriété ϕ est et sera vraie. En d'autres mots, pour toutes les alternatives possible du comportement de `e`, chacune d'entre elles va, *à tout instant*, satisfaire ϕ.

De façon un peu plus formelle [2], on pourrait définir l'interprétation de ces formules comme suit:

- `e` $\models$ `INEV`(ϕ) si et seulement si `e` satisfait la propriété X définie récursivement comme suit:

  $X = \phi$ `or (<*>true and [*]`X`)`

  De façon intuitive, on peut donc dire qu'une propriété ϕ est inévitablement (éventuellement) vraie si est elle déjà vraie ou si on peut faire une action et qu'après avoir fait cette action la propriété ϕ est à nouveau éventuellement vraie.

- `e` $\models$ `ALL`(ϕ) si et seulement si `e` satisfait la propriété X définie récursivement comme suit:

  $X = \phi$ `and [*]`X

  De façon intuitive, une propriété ϕ est toujours vraie si elle est présentement vraie et qu'après avoir effectué n'importe quelle action ϕ est à nouveau vraie (et pour toujours).

[2] En fait, pour être vraiment formel, il nous faudrait introduire les notions de *points fixes* d'équations récursives: plus petit point fixe pour `POT`, plus grand point fixe pour `ALL`. Nous nous restreindrons plutôt aux définitions semi-formelles qui suivent.

- e $\models$ POT(ϕ) si et seulement si on a la propriété suivante:[3]

  e $\models$ not (ALL(not ϕ))

  En d'autres mots, une propriété ϕ est potentiellement vraie s'il n'est pas possible qu'elle soit toujours fausse.

De façon générale, on aura donc qu'une propriété de sécurité aura généralement l'une ou l'autre des formes suivantes:

- not POT(ϕ): il n'est pas possible qu'une condition ϕ devienne vraie.
- ALL(ϕ): une condition ϕ est toujours vraie (invariant).

De façon similaire, une propriété de vivacité aura généralement la forme suivante:

- INEV(ϕ): éventuellement, la condition ϕ deviendra vraie.

17.1.3 Quelques exemples

Examinons maintenant quelques exemples de propriétés spécifiées à l'aide de la logique temporelle introduite dans les paragraphes précédents.

Exemple Soit le processus décrivant la machine distributrice de biscuits et muffins présentée à la fig. 16.4. Les propriétés suivantes peuvent être formalisées en logique modale et sont satisfaites par le comportement de la machine:

1. Dans on état initial, la machine accepte tant une pièce de 0.25$ que de 1.00$.

   ```
   <p25> true and <p100> true
   ```

2. Après avoir mis une pièce d'un dollar, il n'est plus possible de mettre une autre pièce.

   ```
   [p100]([p100] false and [p25] false)
   ```

3. Après avoir mis trois pièces de 0.25$, on peut soit mettre une autre pièce de 0.25$, soit sélectionner un biscuit; il n'est toutefois pas possible de sélectionner un muffin.

   ```
   [p25][p25][p25](<p25, b_bisc> true and [b_muff] false)
   ```

4. Après avoir sélectionné un item, l'item doit nécessairement être livré avant que d'autres pièces puissent être insérées.

   ```
   POT(
      <b_bisc, b_muff> true
      and
      [b_bisc, b_muff]<bisc, muff><p25, p100> true
   )
   ```

5. En tout temps, la machine peut continuer à fonctionner, sans jamais arriver à un blocage (*deadlock*).

   ```
   ALL( <*> true )
   ```

6. La machine effectue un cycle sans fin de livraisons d'items. En d'autres mots, de façon continue, la machine pourra potentiellement livrer des biscuits ou muffins.

   ```
   ALL( POT( <bisc, muff> true )  )
   ```

[3] Notons qu'il existe aussi une modalité SOME définie de façon analogue relativement à INEV, modalité que nous n'utiliserons pas dans les exemples qui suivent:

SOME(ϕ) = not (INEV(not ϕ)).

Exemple Soit le processus décrivant la machine distributrice de café et bouillon présentée à la fig. 16.7. Les propriétés suivantes peuvent être formalisées et sont satisfaites par le comportement de la machine:

1. Il n'est pas possible d'insérer 4 pièces de suite.

   ```
   [p25][p25][p25][p25] false
   ```

2. Après avoir inséré une pièce, et uniquement à partir du moment où une première pièce a été insérée, on peut presser sur le bouton d'annulation.

   ```
   [b_annul] false
   and
   [p25]<b_annul> true
   and
   [p25][p25]<b_annul> true
   and
   [p25][p25][p25]<b_annul> true
   ```

Exemple Soit le processus suivant:

```
PROCESS Horloge[tic]: NOEXIT :=
  tic; Horloge[tic]
ENDPROC
```

Le fait que ce processus, en tout temps, va pouvoir émettre le signal `tic` pourrait être formalisé par l'expression suivante:

```
ALL ( <tic> true )
```

Cette propriété pourrait être interprétée comme la propriété satisfaite par l'équation récursive suivante, qui indique que `tic` peut être effectuée et qu'après avoir effectué n'importe quelle action, on peut à nouveau effectuer `tic`:

```
X = <tic> true and [*] X
```

On pourrait aussi formaliser qu'aucune autre action ne peut être effectuée comme suit, où "`*-tic`" dénote l'ensemble de toutes les actions possibles sauf l'action `tic`:

```
ALL ( [*-tic] false )
```

Soit maintenant l'horloge suivante:

```
PROCESS Horloge[tic, tac]: NOEXIT :=
  tic; tac; Horloge[tic, tac]
ENDPROC
```

La propriété suivante indique qu'après avoir fait `tic` ou peut toujours fait `tac`, et vice-versa; de plus, on peut toujours faire `tic` ou `tac` mais rien d'autre:

```
ALL (
  [tic]<tac> true
  and
  [tac]<tic> true
  and
  ( <tic> true or <tac> true )
  and
  [*-tic, tac] false
)
```

```
SPECIFICATION Semaphore[a1, a2, b1, b2] : NOEXIT
  BEHAVIOUR
   HIDE p, v IN
    ( Proc[a1, a2, p, v] ||| Proc[b1, b2, p, v] )
    |[p, v]|
    Semaphore[p, v]
   WHERE

      PROCESS Proc[ac1, ac2, p, v] : NOEXIT :=
        p; ac1; ac2; v; Proc[ac1, ac2, p, v]
      ENDPROC

      PROCESS Semaphore[p, v] : NOEXIT :=
         p; v; Semaphore[p, v]
      ENDPROC
ENDSPEC
```

Figure 17.1: Deux processus concurrents avec un sémaphore

Exemple Soit la spécification présentée à la fig. 17.1, qui décrit deux processus utilisant un sémaphore de façon à exécuter leurs actions respectives (`a1; a2` ou `b1; b2`) de façon mutuellement exclusive (sans entrelacement)

Voici deux propriétés de ce système, exprimées à l'aide de la logique modale:

1. Sans fin, une action `a1` sera immédiatement suivie d'une action `a2`; il en sera de même pour les actions `b1` et `b2`.

   ```
   ALL(
      POT( <a1><a2> true )
      and
      [a1]<a2> true
      and
      POT( <b1><b2> true )
      and
      [b1]<b2> true
    )
   ```

2. On n'aura jamais une série d'actions où les `a` et `b` sont entrelacées, par exemple, de la forme `a1, b1, a2, b2` ou `b1, a1, a2, b2`, etc.

   ```
   not
     POT (
      <a1><b1> true or <a1><b2> true
      or
      <b1><a1> true or <b1><a2> true
     )
   ```

17.2 Système de contrôle de feux de circulation

17.2.1 Description du problème

On veut spécifier un système de contrôle des feux de circulation pour une intersection entre une route principale avec trafic dense et régulier, et une route secondaire où la circulation est beaucoup moins fréquente. [4] La figure 17.2 illustre l'allure de cette intersection.

[4] Cet exemple est inspiré d'un exemple similaire présenté dans [Fen96] et utilisant la notation CCS.

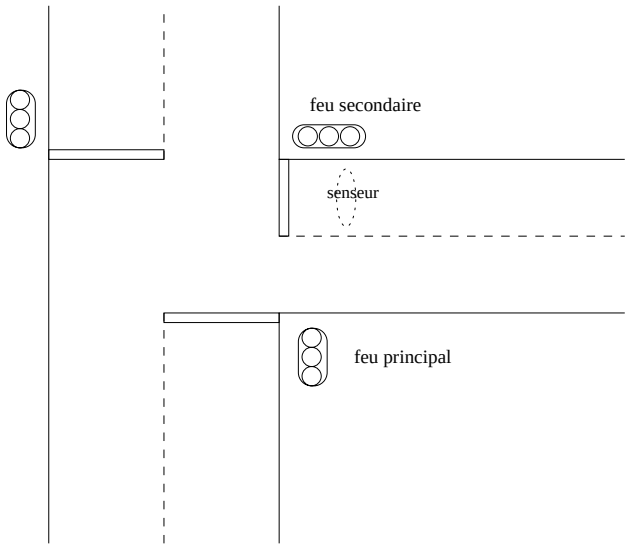

Figure 17.2: L'intersection entre la route principale et la route secondaire

Le feu de circulation qui contrôle le passage des véhicules sur la route secondaire sera toujours laissé au rouge, et ceux de la route principale laissés au vert, sauf lorsque la présence d'un véhicule sera détectée au feu de la route secondaire. Dans ce cas, les feux de la route principale seront alors mis au rouge et ensuite, lorsque ce changement d'état des feux de circulation sera complété, le feu de la route secondaire sera mis au vert. Après un certain délai, les feux retourneront à leur position initiale. Durant la période où le passage des véhicules de la route secondaire sera permis, le détecteur de véhicule devra être rendu inactif et ne sera réactivé que lorsque le feu de la route secondaire sera remis au rouge.

17.2.2 Description détaillée des signaux et ports d'interaction

Les signaux et messages échangés entre l'environnement et le système de contrôle des feux seront les suivants:

- `activerSysteme`: signal pour activer le fonctionnement du système de contrôle des feux (mise en marche).
- `vehiculePresent`: signal qui indique qu'un véhicule est présent à l'intersection sur la route secondaire. Ce signal sera interprété par un `Senseur` qui enverra alors un signal approprié au processus principal de contrôle des feux.
- `feuPrincipal`: signal permettant d'allumer les diverses lumières des feux de circulation de la route principale. Avec ce signal est transmis des informations indiquant la couleur d'une des lumières (`rouge`, `vert` ou `orange`) et une valeur indiquant si cette lumière doit être allumée ou éteinte (`allumer` vs. `eteindre`). On suppose, pour simplifier, que le même signal contrôle l'allumage des lumières des feux dans chacune des directions.
- `feuSecondaire`: signal similaire à `feuPrincipal`, mais pour le feu de la route secondaire.

La figure 17.3 illustre l'organisation générale du sysème de contrôle des feux de circulation, lequel est composé de trois (3) processus communiquant et collaborant entre eux. Une spécification détaillée en `LOTOS` de ce système est présentée à la section suivante.

17.2.3 Spécification `LOTOS`

```
SPECIFICATION ControleurIntersection
              [activerSysteme, vehiculePresent, feuPrincipal, feuSecondaire]
              : NOEXIT
```

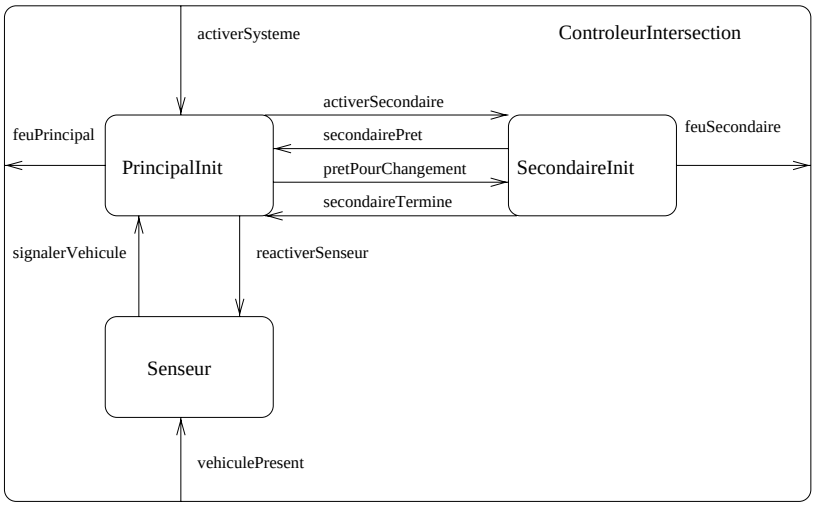

Figure 17.3: L'organisation du sysème de contrôle des feux de circulation

```
BEHAVIOUR
  HIDE activerSecondaire, secondairePret, reactiverSenseur,
       signalerVehicule, pretPourChangement, secondaireTermine IN

   Senseur[vehiculePresent, signalerVehicule, reactiverSenseur]

   |[signalerVehicule, reactiverSenseur]|

   PrincipalInit[feuPrincipal, activerSysteme,
                 activerSecondaire, secondairePret,
                 pretPourChangement, secondaireTermine,
                 reactiverSenseur, signalerVehicule]

   |[activerSecondaire, secondairePret, pretPourChangement, secondaireTermine]|

   SecondaireInit[feuSecondaire, activerSecondaire, secondairePret,
                  pretPourChangement, secondaireTermine]
WHERE

PROCESS PrincipalInit
        [feu, activer, activerSecondaire, secondairePret, pretPourChangement,
         secondaireTermine, reactiverSenseur, signalerVehicule]
        : NOEXIT :=
  activer;
  feu !rouge !allumer;
  activerSecondaire;
  secondairePret;
  feu !rouge !eteindre;
  feu !vert !allumer;
  Principal[signalerVehicule, feu, pretPourChangement, secondaireTermine,
            reactiverSenseur]
WHERE
  PROCESS Principal
          [signalVehicule, feu, pretPourChangement,
           secondaireTermine, reactiverSenseur]
```

```
          : NOEXIT :=
    signalVehicule;
    feu !vert !eteindre;
    feu !orange !allumer;
    feu !orange !eteindre;
    feu !rouge !allumer;
    pretPourChangement;
    secondaireTermine;
    reactiverSenseur;
    feu !rouge !eteindre;
    feu !vert !allumer;
    Principal[signalVehicule, feu, pretPourChangement,
              secondaireTermine, reactiverSenseur]
  ENDPROC
ENDPROC

PROCESS SecondaireInit
        [feu, activer, initTermine, pretChangement, termine]
        : NOEXIT :=
  activer;
  feu !rouge !allumer;
  initTermine;
  Secondaire[pretChangement, feu, termine]
WHERE
  PROCESS Secondaire[pretChangement, feu, termine]: NOEXIT :=
    pretChangement;
    feu !rouge !eteindre;
    feu !vert !allumer;
    feu !vert !eteindre;
    feu !orange !allumer;
    feu !orange !eteindre;
    feu !rouge !allumer;
    termine;
    Secondaire[pretChangement, feu, termine]
  ENDPROC
ENDPROC

PROCESS Senseur
        [vehiculePresent, signalerVehicule, reactiverSenseur]
        : NOEXIT :=
  vehiculePresent;
  signalerVehicule;
  reactiverSenseur;
  Senseur[vehiculePresent, signalerVehicule, reactiverSenseur]
ENDPROC

ENDSPEC
```

Quelques points intéressants à souligner dans cette spécification `LOTOS` sont les suivants:

- La façon la plus simple de voir comment connecter les divers processus avec l'opérateur de synchronisation parallèle est la suivante: les synchronisations devraient refléter les liens qui existent, dans la figure 17.3, entre ces processus. Par exemple, on voit sur cette figure que le processus `Senseur` communique avec `PrinicipalInit` par l'intermédiaire de `signalerVehicule` et `reactiverSenseur`. Ce sont donc ces deux ports qui devraient être utilisés comme point de synchronisation dans l'opérateur `|[]|`:

  ```
  PrincipalInit[...] |[signalerVehicule, reactiverSenseur]| Senseur[...]
  ```

- Initialement (`PrincipalInit`), pour assurer un fonctionnement sécure, le feu de la route principale est laissé au rouge jusqu'à ce que l'on soit assuré que le feu de la route secondaire est bien au rouge (`secondairePret`); ce n'est qu'ensuite que le feu est mis au vert. Évidemment, ceci suppose que les feux, dans leur état initial, sont tous éteints.

- Le mécanisme général de fonctionnement est le suivant. Lorsque le senseur détecte un véhicule présent, il envoie alors le signal `signalerVehicule` au processus `Principal`. Ce dernier fait alors tourner au rouge les feux de la route principale puis fait devenir vert le feu de la route secondaire en envoyant le signal `pretPourChangement`. Le feu secondaire indique ensuite la fin de son cycle en envoyant le signal `secondaireTermine`, signal qui assure que le feu de la route secondaire est retourné au rouge. Le senseur est alors réactivé, le feu de la route principale est remis au vert et le cycle recommence.

- Un point important à noter, point caractéristique des spécifications `LOTOS`, est qu'il n'y a pas de notion de temps avec durée. Par exemple, dans le processus `Secondaire`, le feu rouge est tout d'abord éteint, puis le vert, "immédiatement" après, est allumé, puis ensuite éteint.

  En `LOTOS` les transitions et actions sont considérées comme étant instantanées, donc de durée nulle. Par contre, le temps requis entre le moment où une transition est prête à s'exécuter et le moment où elle est effectivement exécutée est variable, non-contrôlable, non-déterministe. L'aspect durée temporelle, dans une spécification `LOTOS`, n'est donc pas spécifié ou contrôlable, à moins d'introduire explicitement un signal d'horloge et de mesurer l'écoulement du temps à partir de ce signal.

  Cette façon de traiter le temps en `LOTOS` — de façon qualitative plutôt que quantitative — reflète bien l'origine du langage. À l'origine, `LOTOS` a été conçu pour la spécification de protocoles de communication et non pour les systèmes en temps réel (système de contrôle où des contraintes *dures* de temps doivent être respectées). L'élément central lié à un protocole de communication est la présence d'un réseau de télécommunication. Une caractéristique importante d'un tel réseau est que le temps requis pour une communication est imprévisible: il n'y donc généralement aucune façon de prédire le temps requis pour effectuer une communication parce que trop d'éléments entrent en jeu (chemins d'accès variables, congestion, perte d'information et nécessité de retransmission, etc.). De plus, pour assurer le bon fonctionnement d'un protocole, on utilise aussi des alarmes: par exemple, lorsqu'une confirmation de réception d'un message n'a pas été reçue après un certain délai, le message est retransmis; règle générale, la durée exacte de l'alarme n'a pas et ne doit pas avoir d'influence sur le comportement du protocole: le protocole doit fonctionner même si l'alarme arrive trop tôt (la confirmation arrive juste après la retransmision), donc indépendamment de la durée exacte du délai d'alarme.

17.2.4 Propriétés requises (sécurité et vivacité)

Pour un développement complet et sûr de ce système, il est intéressant de spécifier de façon formelle certaines des propriétés devant être satisfaites par le comportement de ce système et de s'assurer que ces propriétés sont bien respectées par la spécification `LOTOS` présentée plus haut. Voici donc quelques propriétés satisfaites par cette spécification:

1. Sécurité: Le feu de la route principal et le feu de la route secondaire ne sont jamais verts en même temps.

```
ALL (
   not POT (
     <feuPrincipal !vert !allumer> true
     and
     <feuSecondaire !vert !allumer> true
   )
)
```

2. Vivacité: Si un véhicule arrive à l'intersection de la route secondaire, alors éventuellement le feu de la route secondaire va devenir vert pour laisser passer le véhicule, et ce en tout temps.

```
ALL (
  INEV (
    <vehiculepresent> true
    and
    [vehiculepresent] (POT (<feusecondaire !vert !allumer> true))
  )
)
```

3. Vivacité: Éventuellement, le feu de la route principale va redevenir vert.

```
ALL (
  INEV (
    <feuprincipal !vert !allumer> true
  )
)
```

Références

[Ast91] E. Astesiano. Inductive and operational semantics. In E.J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, pages 51–136. Springer-Verlag, 1991.

[BB87] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[BL90] V. Berzins and Luqi. An introduction to the specification language `Spec`. *IEEE Software*, 7(2):74–84, March 1990.

[BL91] V. Berzins and Luqi. *Software Engineering with Abstractions.* Addison-Wesley Publishing Co., 1991. [En réserve: QA76.76D47B47].

[Bru97] G. Bruns. *Distributed Systems Analysis with CCS.* International Series in Computer Science. Prentice-Hall, 1997.

[Dam94] M. Dam. Temporal logics, automata, and classical theories — an introduction. Lecture Notes for the 6th European Summer School on Logic, Language and Information, 1994.

[dMRV92] Jan de Meer, Rudolf Roth, and Son Vuong. Introduction to algebraic specifications based on the language ACT ONE. *Computer Networks and ISDN Systems*, 23(5):363–392, 1992.

[Fen96] C. Fencott. *Formal methods for concurrency.* International Thomson Computer Press, 1996.

[FGK+92] J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP: A protocol validation and verification toolbox. In *Computer Aided Verification*, pages 437–440. Springer-Verlag, LNCS-1102, 1992.

[FGM+92] J.-C. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodriguez, and J. Sifakis. A toolbox for the verification of LOTOS programs. In *ICSE '92*, pages 246–259, 1992.

[Gar90] H. Garavel. Introduction au langage LOTOS. *Revue de l'Association des Anciens Élèves de l'ENSIMAG*, 1990. ftp://ftp.imag.fr/imag/SPECTRE/LOTOS/ PAPERS/Garavel-90-b.ps.Z.

[GH93] J.V. Guttag and J.J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993. [En réserve: QA76.6G88].

[GJM+97] H. Garavel, M. Jorgensen, R. Mateescu, C. Pecheur, M. Sighireanu, and B. Vivien. CADP '97 — status, applications, and perspectives. In *Proceedings of the 2nd COST 247 International Workshop on Applied Formal Methods in System Design*, Zagreg, Croatia, June 1997.

[GLO91] S. Gallouzi, L. Logrippo, and A. Obaid. Le LOTOS: Théorie, applications, outils. In *CFIP '91 — Ingénierie des Protocoles*, pages 385–404, 1991.

[Hen90] M. Hennessy. *The Semantics of Programming Languages — An elementary introduction using structural operational semantics.* John Wiley and Sons Ltd, 1990.

[Hoa85] C.A.R. Hoare. *Communicating sequential processes.* Prentice Hall international series in computer science, 1985. [QA76.6H63].

[Hol92] G.J. Holzmann. Protocol design: Redefining the state of the art. *IEEE Software*, 9(1):17–22, Jan. 1992.

[LFHH92] L. Logrippo, M. Faci, and M. Haj-Hussein. An introduction to LOTOS: Learning by examples. *Computer Networks and ISDN Systems*, 23:325–342, 1992.

[Mil80] R. Milner. *A Calculus of Communicating Systems.* Springer-Verlag, LNCS-92, 1980.

[Mil89] R. Milner. *Communication and concurrency.* Prentice-Hall, 1989.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems — Specification.* Springer-Verlag, 1992. [QA76.6M3564(V1].

[Pnu86] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In *Current Trends in Concurrency*, pages 510–584. Springer-Verlag, LNCS-224, 1986.

[QS83] J.P. Queille and J. Sifakis. Fairness and related properties in transition systems — a temporal logic to deal with fairness. *Acta Informatica*, 19:195–220, 1983.

[Sti96] C. Stirling. Model and temporal logics for processes. In *Logics for Concurrency — Structure versus Automata*, pages 149–237. Springer-Verlag, LNCS-1043, 1996.

[Tur93] K.J. Turner. *Using formal description techniques: an introduction to Estelle, LOTOS, and SDL.* J. Wiley & Sons, 1993. [En réserve: QA76.6U85].

[vG90] R.J. van Glabbeek. The linear time – branching time spectrum. In *CONCUR '90*, pages 278–297. Springer-Verlag, LNCS-458, 1990. [QA75.5L28V458].