LotosNT : modélisation et vérification

Construction Formelle de Logiciels

Christian Attiogbé, Meriem Ouederni

Janvier 2012

Introduction

- *Algèbres de processus* =
  une catégorie de formalismes qui permettent de *décrire* et d'*analyser* les comportements de *systèmes (processus) concurrents*.
- Historique
  - Théorie des automates,
  - Théorie des réseaux de Petri,
  - Algèbres de processus :
    CSP (Communicating Sequential Processes) ; C.A.R. Hoare, 1978, `occam + Transputer`
    CCS (Calculus of Communicating Systems) ; R. Milner, 1985
    ...

Caractéristiques principales des algèbres de processus

- Spécification et étude des systèmes concurrents (communication, synchronisation)
- Abstraction sur les comportements,
- Mode de synchronisation (synchrone, RdV, actions complémentaires), etc
- Mode de composition (parallèle, entrelacement, etc),
- Modèles sémantiques (opérationnelle, traces, bissimulation, *failure-divergence*).

Concepts fondamentaux

Alphabets
L'évolution d'un processus est décrite à l'aide des noms des actions qu'il entreprend : *alphabet*.
Expressions régulières
Combinaison de noms d'action (*alphabet*) à l'aide d'opérateurs prédéfinis.
Automate à états.
Processus élémentaire
L'évolution séquentielle, exprimée à l'aide de combinaison d'actions et d'opérateurs (**séquence, choix, arrêt**) : on parle de *comportement* (***behaviour***).

Concepts fondamentaux (suite)

Processus prédéfinis

- Un procesus qui ne termine pas (***deadlock***) :
  il ne peut plus effectuer aucune action de son alphabet.
  En CSP c'est **STOP**, en CCS c'est **0**.
- Un processus qui termine mais qui ne peut effectuer aucune action. En CSP c'est `skip`.

Principaux opérateurs (illustration avec CSP)

Séquence (notée ;)

Une action suivie d'un comportement (le reste des actions).
Soit un alphabet $A = \{lire, ecrire, traiter, ouvrir\}$

```
lire;traiter;ecrire;STOP
```

Principaux opérateurs (illustration avec CSP)

Choix de comportements (noté `[]`)
Le processus s'engage dans un comportement ou un autre :

```
lire;STOP [] ecrire;STOP

lire;STOP [] ecrire;(traiter;STOP [] ouvrir;STOP)
```


Interruption

Interruption (notée [>)
Alphabet pour la suite : $\mathcal{A} = \{a, b, c, d, e, f, g\}$
Soit P un processus avec le *comportement principal* suivant nommé P1 :

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

On veut exprimer que P peut être interrompu à tout moment et adopter le comportement `e;f;g;STOP`
On écrit :

```
P = P1 [> e;f;g;STOP
```

Exercice Donner une représentation graphique de P

Communication

Communication : seul moyen de **synchroniser** et d'**échanger des valeurs** entre les processus.
La communication est effectuée via des **actions** ou des **canaux**.
Utilisation de **canaux** de communication (CSP) ou
de **ports** de communication (CCS).
En CSP

Opérateur d'émission sur un canal : **!**

Opérateur de réception sur un canal : **?**

Exemples

```
ca?va:int ; cb?vb:int ; cc!(va + vb) ; STOP

g1?xx:Typx !yy[xx>0] plus d'expression avec LOTOS
```


Plan de ce cours

Communication par rendez-vous (à la CSP)

Rendez-Vous = moyen de **synchronisation** et de **communication**

- **symétrique** : bloquant pour l'émission **et** la réception
- **binaire** ou **n-aire**

Il existe d'autres modes de communication : asynchrone, diffusion

structure *if then else*

Exemple

```
ca?vx:int;
if vx >= 0
then cb!vx;STOP
else cb!-vx;STOP
endif
```

Exercice

Donnez une représentation graphique du processus ainsi décrit.

Comportements gardés (avec `[garde] -> ...`)

Exemple

```
ca?vx:int;
(
 [ vx > 0 ] -> cb!vx;STOP
 []
 [ vx = 0 ] -> cb!0;STOP
 []
 [ vx < 0 ] -> cb!vx;STOP
)
```

Exercice

Donner une représentation graphique du processus ainsi décrit.

Opérateurs de composition parallèle

Notation de CSP : `|[L]|`
`P1 |[L]| P2` signifie :

- P1 et P2 s'exécutent en parallèle
- L est une liste d'actions ou de canaux
- P1 et P2 doivent se *synchroniser* sur les actions de L
- P1 et P2 ne doivent pas se synchroniser sur des actions n'appartenant pas à L.

Exemple de comportements en parallèle

Sous forme de réseau de Pétri

```
P1 := a;b;c;STOP
P2 := a;b;d;STOP
P1 |[a,b]| P2
```

Modèles sémantiques (interpréter les comportements)

Sémantique opérationnelle
Relation de transition entre les comportements.

$$Terme \rightarrow_{action} Terme$$

Sémantique axiomatique
On donne les propriétés algébriques des différents opérateurs
Failure-divergence
On donne l'ensemble des comportements non autorisés

Conclusion

- De nombreuses autres algèbres processus : ACP, μCRL,...
- Extensions avec gestion de contraintes de temps: TCSP, ...
- Peu utilisées dans les milieux industriels
- LOTOS normalisé, utilisé en industrie
- Des recherches toujours en cours...

Algèbres de processus (AdP) : Conclusion

AdP : formalisme théorique sur lequel s'appuient divers outils de spécification et de vérification.

Exemples d'AdP pour les systèmes asynchrones :

- CCS (Calculus of Communicating Systems) [Milner-89]
- CSP (Communicating Sequential Processes) [Hoare-85]
- ACP (Algebra of Communicating Processes) [Bergstra-Klop-84]
- Langage normalisé : LOTOS [ISO-88]
- Langage plus moderne : LOTOS NT (abrév. LNT)

Dans ce cours on présentera partiellement LNT et sa sémantique formelle.

Plan de ce cours

Références

- H. Garavel, *Notes de Cours : systèmes temps réels* , ENSIMAG, Grenoble
- Hubert Garavel, *Introduction au Langage LOTOS*, Revue de l'Association des Anciens Elèves de l'ENSIMAG, 1990
- H. Garavel, `http://www.inrialpes.fr/vasy/cadp/`
- L. Logrippo, M. Faci, M. Haj-Hussein, *An introduction to LOTOS: Learning by Examples*, Computer Networks and ISDN Systems 23(5), 1992

Références (suites)

- A. Strohmeier et D. Buchs, *Génie logiciel: principes, méthodes et techniques*, Presses polytechniques et universitaires romandes, 1996
- Ken Turner, `http://www.cs.stir.ac.uk/ kjt/research/well/index.html`
- Ken Turner, *Using Formal Description Techniques - An Introduction to Estelle, LOTOS and SDL*, John Wiley and Sons Ltd., 1993

Le langage LOTOS - Généralités

Language of Temporal Ordering Specification

> Normalisé ISO 8807, en 1987 après des Drafts en 1985 et 1986
> Résultat de travaux de l'OSI dans le cadre de ISO
> (années fin 70 - début 80)

Contexte : Techniques de description formelle des protocoles et services

2 groupes de travail OSI/ISO:

- Extension des machines à états ⇒ ESTELLE
- Techniques d'ordonnancement temporel ⇒ LOTOS

LOTOS - Généralités

LOTOS

utilise CSP et CCS pour décrire le comportement des processus

utilise ACT ONE pour décrire les données

possède de nombreux outils (vérification des propriétés)
par exemple CADP (CAESAR/ALDEBARAN Dev. Pack.) à Grenoble

LOTOS NT (LNT) - Historique

On trouve **trois principales versions** :

- LNT
- Full LOTOS
- Enhancement of LOTOS (E-LOTOS)
  nouvelle norme ISO/IEC 15437, 2001
- LNT

Complémentarité entre les aspects **données** et **comportements**,
une expression de comportement représente un état.

Le langage LNT

Héritier de LOTOS (norme internationale [ISO 8807] pour la spécification formelle des protocoles de télécommunication et des systèmes distribués)

Variante de la norme E-LOTOS [ISO 15437] Combinaison des concepts AdP avec un langage algorithmique classique (impératif / fonctionnel)

- partie données (pour le calcul sur les structures de données) : types, fonctions, instructions
- partie contrôle = sur-ensemble de la partie données : comportements

Une partie données est nécessaire pour décrire des systèmes réalistes. Il existe des sous-ensembles des AdP sans données (“pure CCS”, “LNT”), mais qui sont inutilisables en pratique.

LNT - Généralités

- LNT supporté par la boîte à outils CADP (INRIA / VASY) : simulation, vérification formelle (model checking), ...
- Traduction vers LOTOS (lnt2lotos, lnt.open)

Formalisme de présentation du langage

- Grammaire EBNF (Extended Bachus Naur Form)
  - Mots-clés et symboles terminaux en rouge
  - Meta-symboles en bleu
  - Symboles non-terminaux dans les autres couleurs
- Règles de la forme Θ ::= $E_0|...|E_n$ où :
  - Θ : symbole non-terminal
  - $E_0, \ldots, E_n$: séquences de symboles (terminaux et non-terminaux)
  - [E] : production optionnelle
  - $E_0 \ldots E_n$: rèpètition non-vide ($E+$)
  - $E_1 \ldots E_n$: rèpètition possiblement vide (E*)

Définition de types de données

- Identificateurs de types $T, T_1, T_2, \ldots$
  - Prédéfinis : bool, char, nat, int, real, string
  - Définis par l'utilisateur : type T is E end type
- Expressions de types $E, E_1, E_2, \ldots$

$$E ::= D_0, \ldots, D_n$$
$$\mid \textit{set of } T \mid \textit{list of } T \mid$$
$$\mid \textit{array } [\, n \,..\, m] \textit{ of } T \mid \ldots$$
$$D ::= C\ [(X_1{:}T_1, \ldots, X_n{:}T_n)]$$

D_i: Définition de constructeur
X_i: variables
C: identificateur de constructeur

Types à constructeurs

- Issus des langages fonctionnels (ML, Haskell)
- Exemples :
  - **Enuméré** :
    type color is red, green, blue end type
  - Record : type pers is pair (name : string, age : nat) end type
  - Union : type IntBool is anInt (n : int), aBool (b : bool) end type
  - Liste : type int_list is nil, cons (hd : int, tl : int_list) end type
    équivalent à type int_list is list of int end type
  - Arbre binaire : type tree is leaf, node (fg : tree, fd : tree) end type

Définitions de fonctions

- **function** $F(X_1 : T_1, \ldots, X_n : T_n) : T$ **is**
  I
  end function

- Ici : passage de paramètres par valeur (autres modes de passage permis, voir plus loin)

- Effet défini par une instruction : (affectations, séquence, conditionnelles, boucles, return, ...)

Exemple

```
function fact (n : nat) : nat is
        var result : nat in
           result := 1;
           while n > 1 loop
                result := result * n; n := n - 1
          end loop;
          return result
       end var end function
```

Exemples d'appel :

fact (4 of nat)
fact (X)

Plan de ce cours

Passage de paramètres par référence (paramètres out et inout)

- Permis aussi par LNT mais non-présenté dans la grammaire ci-dessus ;
- Exemple (incrémentation compteur 16 bits avec vérification d'overflow) :

```
function incr16 (inout n : nat, in delta : nat,
                        out overflow : bool) is
        if n ≥ 65535 - delta then overflow := true
        else overflow := false; n := n + delta end if
end function
```

Expressions

- Notées V, V1, V2, ...
- Expression = terme qui a une valeur unique dans un contexte donné

V **::=** X
| C [(V1, ... , Vn)]
| F [(V1, ... , Vn)]
| V[X]
| V.X | V.{X0 ⇒ V0, ... , Xn ⇒ Vn}
| (V0) | V0 of T | ...

Surcharge de symboles

- Possibilité d'avoir des fonctions ou constructeurs de même nom mais de types différents
  function odd (n : nat) : bool **is** ... **end function**
  function odd (n : int) : bool **is** ... **end function**
- Idem pour les fonctions prédéfinies
  function _ + _ (b1 : Bool, b2 : Bool) : Bool **is**
  ...
  end function
- **Attention** : pas de règles de priorité entre opérateurs ⇒ toujours parenthéser!
  a and b or c **doit être** écrit (a and b) or c

Motifs (patterns)

- Notés P, P_1, P_2, ...
- Motif = terme qui représente un ensemble de valeurs possibles dans un contexte donné

  P ::= X
  | any T
  | C [(P_0, ... , P_n)]
  | P_0 where V_0 | ...

- Intérêt : filtrage de motifs (patternmatching) hérité des langages fonctionnels (ML, Haskell, ...)

Filtrage de motif

- Instruction case
  case V in [var X_1 : T_1, ..., X_m : T_m] in
  P_0 -> I_0 | ... | P_n -> I_n
  end case
- Si V appartient à l'ensemble de valeurs représentées par P_i ($i \in$ 1..n) alors exécuter I_i (avec priorité de gauche à droite)
- Les variables de P_i prennent la valeur des sous-termes correspondant dans V

Partie contrôle

Permet la définition des processus (comportements B, B_1, B_2, ...)
Particularité de LNT : symétrie entre données et contrôle

- Les comportements englobent toutes les instructions de la partie donnée
- Exception : return (réservé à la partie données)

Sont ajoutés des comportements décrivant :

- le non-déterminisme
- le parallélisme asynchrone
- la communication
- l'abstraction de comportement

Comportements existant dans la partie données (1/2)

B ::= null
| B_1; B_2
| X := V
| X[V0] := V_1
| var X_1:T_1, ..., X_n:T_n in B_0 end var
| case V in [var X_1:T_1, ..., X_m:T_m] in
P_0 -> B_0 | ... | P_n -> B_n
end case
| ...

Comportements existant dans la partie données (2/2)

B ::= ...
| if V_0 then B_0
[elsif V_1 then B_1 ... elsif V_n then B_n]
[else B_{n+1}] end if
| loop [L in] B_0 end loop
| break L
| while V loop B_0 end loop
| for B_0 while V by B_1 loop B_2 end loop
| ...

Sémantique de “null”

“null” décrit l’inaction.

“null” se termine immédiatement sans changer le contexte.

Comportements spécifiques à la partie contrôle (1/2)

B ::= ...
- | **stop**
- | Π **[[** G_0**, ...,** G_m **]] [(**V_1**, ...,** V_n**)]**
- | **X := any T [where V]**
- | **select** B_0 **[] ... []** B_n **end select**
- | **par [** G_0**, ...,** G_n **in]** B_0 **|| ... ||** B_m **end par**
- | **hide** G_0 **:** Γ_0**, ...,** G_n **:** Γ_n **in** B_0 **end hide**
- | **disrupt** B_1 **by** B_2 **end disrupt**
- | **...**

Plan de ce cours

Comportements spécifiques à la partie contrôle (2/2)

Communication

B ::= ...
- | **G [(**O_1**, ...,** O_n**)]**

Offres de communication O, O_1, O_2, **...**

O ::= [!] V
- | **? P**

Sémantique de “stop”

Pour l’opérateur stop (inaction) : aucune règle sémantique associée.

On ne peut dériver aucune action à partir de stop.

L’opérateur stop n’existe pas dans la partie données.

Opérateur “disrupt”

- **LNT permet d’exprimer l’interruption d’un comportement par un autre (*eq.* exception) :**

  disrupt B1 by B2 end disrupt

  signifie que l’exécution de B1 peut être à tout instant interrompue et continuée avec B2

- **Exemple : disrupt a ; b by d ; stop end disrupt**
- **N’existe pas dans la partie données**

Channels

- **Possibilité, au moment de déclarer une porte, de contraindre le type des données échangées sur cette porte**
- **Identificateurs de channels Γ, Γ_1, Γ_2, ...**
  - **Prédéfini : any (aucune contrainte)**
  - **Définis par l'utilisateur**

  channel Γ is K_0, ..., K_n end channel
  avec K::=$(T_1, ..., T_n)$ permet le transit de tuples dont les valeurs ont les types respectifs T_1, ..., T_n

Exemples

Variables: N : nat, B : bool

- **channel ch1 is (nat) end channel**
  G (4), G (?N), etc.
- **channel ch2 is (nat), (bool) end channel**
  G (4), G (?N), G (true), G (?B), etc.
- **channel ch3 is (), (nat, bool) end channel**
  G , G (4, ?B), G (?N, ?B), G (4, true), etc.

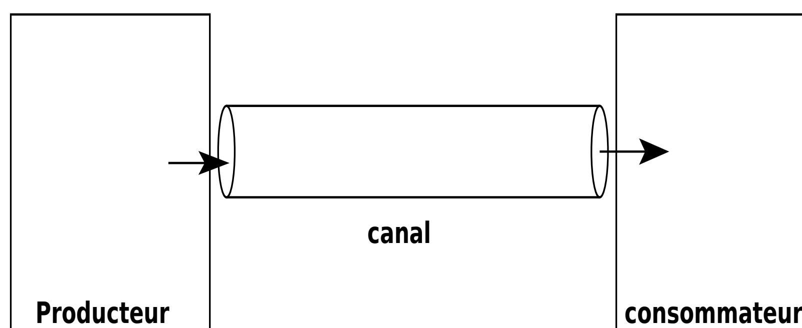
Définitions de processus

process Π **[[** G_0**:**Γ_0**, ...,** G_n**:**Γ_n **]] [(X1:T1, ..., Xn:Tn)] is** B
end process
où :

- Π **= nom du processus**
- G_0**, ...,** G_n **= paramètres formels portes de** Π
- Γ_0**, ...,** Γ_n **= channels de** G_0**, ...,** G_n
- X_1**, ...,** X_n **= paramètres formels données de** Π
- T_1**, ...,** T_n **= types de** X_1**, ...,** X_n
- B **= comportement de** Π

Un exemple pour l'étude : producteur-consommateur de ressources

Exemple classique, simple, mais intéressant sur plusieurs points :



Etude (1/28)

- Quel est le comportement du (processus) producteur ?
- Quel est le comportement du consommateur ?
- Qu'est-ce que le canal ? quel modèle ? quel comportement ? Informellement structure FIFO à 1 place, 2 places, ...

Quelles stratégies appliquées pour la gestion des ressources ?

- Produire tout, puis consommer tout ?
- produire et consommer alternativement ?

Plan de ce cours

LNT / étude (2/28)

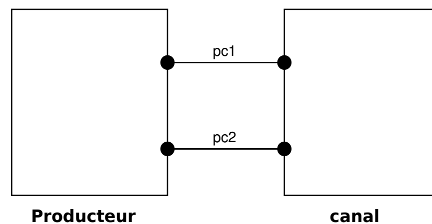
Hypothèses de travail :

- Le canal représente une mémoire tampon (buffer) de 2 places
- On modélise le canal comme un processus
- Le producteur est un processus qui **doit** produire 2 ressources puis s'arrêter
- Le consommateur est un processus qui doit consommer 2 ressources (ce qui est produit) et s'arrêter.

Soient `pc1`, `pc2` les deux actions du processus producteur : `pc1` représente la production de la ressource 1 vers l'environnement (ici le canal).

LNT / étude (2'/28)

Une action identifiant une porte, on peut percevoir le système producteur-canal comme suit :



pc1, pc2 sont des portes.

Plan de ce cours

LNT / étude (2"/28)

Le **comportement du processus producteur** s'écrit alors

```
process Producteur [pc1:any, pc2:any] is
     pc1 ; pc2 ; stop
end process
```

pc1, pc2 sont ici des (portes) paramètres formels.
stop indique qu'aucune action n'est plus possible.

LNT / étude (3/28)

Le processus consommateur est spécifié de la même façon par :

```
process Consommateur [cc1:any, cc2:any] is
     cc1 ; cc2 ; stop
end process
```

cc1, cc2 sont ici les actions de consommation des ressources.
cc1 représente l'action de consommer la première ressource dans le canal.

LNT / étude (4/28)

La **spécification du canal** est comme suit :

```
process Canal [pc1:any, pc2:any, cc1:any, cc2:any] is
      pc1; pc2; cc1; cc2; stop
end process
```

Maintenant il suffit de composer ces trois processus en parallèle
(après avoir étudier les différents opérateurs de composition parallèle)

LNT / étude (5/28)

Nouvelles hypothèses de travail :

- considérons que le canal **peut** délivrer au consommateur, la première ressource produite par le producteur, avant que la deuxième ne soit produite.

Plusieurs possibilités :

- soit le canal se comporte comme dans le cas précédent
- soit après la première production, il se synchronise avec le consommateur sur la porte `cc1`.

Plan de ce cours

LNT / étude (6/28)

On a un choix dans les comportements du canal.
La **spécification du canal** devient comme suit :

```
process Canal [pc1:any, pc2:any, cc1:any, cc2:any] is
     pc1;
        select
           pc2 ; cc1 ; cc2 ; stop
        []
          cc1 ; pc2 ; cc2 ; stop
        end select
end process
```

On a utilisé la structure de choix (select A `[]` B end select).

LNT / étude (6'/28)

L'opérateur de choix `[]` est :

- *commutatif* et
- *associatif*.

On parle alors de *comportements équivalents*,

avec la notion d'**équivalence observationnelle**.

Il y a d'autres types d'équivalence [Milner,Van Glabbek,...].

LNT / étude (8/28) Comportement séquentiel

Dans le comportement précédent du canal, le sous-comportement `cc2;stop` est effectué à la fin des deux possibilités.

On peut en faire un processus composé séquentiellement avec le comportement du début du canal.

LNT permet de coder les comportements séquentiels au moyen d'opérateurs de choix (select), séquencement (stop et ;), boucles (loop) et/ou processus récursif terminal.

LNT / étude (9/28)

La **spécification du canal** devient comme suit :

```
process Canal [pc1:any, pc2:any, cc1:any, cc2:any] is
    pc1;
      select
        pc2 ; cc1
      []
        cc1 ; pc2
      end select ;
      cc2 ; stop
end process
```


LNT / étude (10/28)

Nouvelles hypothèses de travail :

- On considère que le canal n'est pas fiable.
  Il peut perdre des messages.
- La spécification du système doit prendre en compte cette propriété.
- Les ressources consommées sont celles qui ne sont pas perdues sur le canal.

Du point de vue de la spécification, la perte d'un message sur le canal peut se traduire par une *action nulle* du canal pour ce message.

LNT / étude (11/28)

Un message produit (pc1 ou pc2) peut ne pas être consommé ; donc absence de cc1 ou cc2.

L'action interne **i** peut simuler les pertes comme action nulle.

Différentes possibilités pour le comportement du canal :

- récupérer pc1
  - récupérer pc2 ; délivrer pc1
  - récupérer pc2 ; perte de pc1
  - délivrer pc1 ; récupérer pc2
  - perte de pc1 ; récupérer pc2
- puis soit délivrer pc2 soit perdre pc2

LNT / étude (12/28)

Une **spécification du canal** est comme suit :

```
process Canal [pc1:any, pc2:any, cc1:any, cc2:any] is
     pc1;
        select
           pc2 ; cc1
        []
           select
              cc1 ; pc2
           []
           i ; pc2 ; stop
           end select
        end select ;
        select cc2 ; stop [] i ; stop end select
end process
```

LNT / étude (13/28)

Modification de la spécification du consommateur.
Il peut :

- consommer les deux ressources si aucune perte,
- consommer une des deux ressources non perdues,
- ne rien consommer si les deux ressources sont perdues.

```
process Consommateur[cc1:any, cc2:any] is
      select
         cc1 ; select cc2 ; stop [] stop end select
         (* pas de perte *)
      []
         cc2 ; stop (* cc1 perdue *)
      []
      stop (* les deux perdues *)
      end select
end process
```

Plan de ce cours

LNT: opérateurs de composition

A présent, étude des opérateurs de composition parallèle.

LNT / étude comp. parallèle (14/28)

LNT offre **trois opérateurs** pour la composition parallèle des processus.
Nous les étudions afin d'utiliser ceux qui sont adéquats pour l'étude du producteur-consommateur.

- opérateur de composition avec *entrelacement* (noté `... ||| ...`)
- opérateur de composition *sélective* (noté `par ... in ... || ...`)
- opérateur de composition avec *synchronisation complète* (noté `... || ...`)

Plan de ce cours

LNT / étude comp. parallèle (15/28)

Opérateur parallèle entrelacement `|||` :
utilisé lorsque les processus parallèles ne se synchronisent pas.
Exemple :

```
b ; c ; stop ||| c ; stop
```

est équivalent à

```
b ;( c ; c ; stop [] c ; c ; stop) [] c ; b ; c ; stop)
```

ou

```
(b ; c ; c ; stop [] c ; b ; c ; stop)
```

L'opérateur ||| est **commutatif** et **associatif**

LNT / étude comp. parallèle (16/28)

Opérateur parallèle sélectif par L in ... || ... :
on utilise une liste d'actions ou portes de synchronisation L.
La synchronisation **doit se faire sur les actions de L**.
Exercice : écrire le comportement équivalent à

```
par a in a ; b ; c ; stop || d ; a ; c ; stop
```

L'opérateur || est **commutatif**.
L' **associativité** dépend de L.

Plan de ce cours

LNT / étude comp. parallèle (17/28)

Opérateur parallèle de synchronisation complète || :
Ici l'alphabet d'actions en entier correspond à la liste de synchronisation.

C'est comme si || = par **alphabet** in ... || ...

Les processus doivent donc se synchroniser sur toutes les actions.

`a ; b ; c ; stop || c ; a ; b ; stop` est équivalent à `stop` (deadlock).

`a ; b ; c ; stop || (a ; b ; stop [] a ; c ; stop)` est équivalent à `a ; stop [] a ; b ; stop`

LNT / étude comp. parallèle (18/28)

Opérateur parallèle || (suite)

```
a ; b ; stop || (select a ; b ; stop [] i ; b ; stop end
select)
```

est équivalent à

`a ; b ; stop` ou à

`i ; stop`

selon le choix issu de `[]`.

L'opérateur || est **commutatif** et **associatif**

LNT / étude (19/28)
Module et point d'entrée LNT

Spécification globale du producteur-consommateur :

- Un programme LNT est défini dans un module de même nom que le fichier qui le contient
- Possibilité d'importer des modules
- Un processus nommé MAIN (sans paramètres valeur) définit le point d'entrée du programme
- Exemple :

```
module mon_module (module_1, module_2) is
    ...
    process MAIN [...] is ... end process
end module
```

LNT / étude (19/28): suite

```
module producteur_consommateur is
     process MAIN[pc1:any,pc2:any,cc1:any,cc2:any] is
        (Producteur [pc1, pc2]
           |||
        Consommateur [cc1, cc2])
        ||
        Canal [pc1, pc2, cc1, cc2]
        (* du fait de || le comportement de Canal prime *)
     ...
```


LNT / étude (20/28)

Spécification globale du producteur-consommateur (suite)

```
... (* proc. définis auparavent mais avec
paramètres formels *)
process Producteur [o1:any, o2:any] is ...
process Consommateur [i1:any, i2:any] is ...
process Canal [rc1:any, rc2:any, cr1:any, cr2:any] is ...
end process
end module
```

Paramètres formels et paramètres effectifs : **instanciation**.

LNT / étude (21/28)

Instanciation des paramètres formels
La liste de portes formelles d'un processus est renommé avec les paramètres effectifs.
Le renommage est fait **dynamiquement** et non statiquement
Le renommage est fait avant que les actions renommées ne soient offertes à l'environnement du processus renommé.

LNT / étude (22/28)

Soit un processus P

```
process P [a, b, c] is
      par a in a ; b ; stop || a ; c ; stop
endprocess
```

`P[c, c, a]` équivaut à `c;(c;a;stop [] a;c;stop)`
car transformé en `a;(b;c;stop[]c;b;stop)` avant renommage

LNT: Exercice

Dessiner le STE des processus suivants:

```
par G2 in
     G1; G2; G3
     ||
     G4; G2; G5
end par;
G6; stop

var X, Y, Z : bool in
     par G in
        G(?X)
     ||
        G(?Y); Z := true
     end par;
     H(Z); stop
```

LNT / étude (23/28)

Spécification globale du producteur-consommateur

```
module producteur_consommateur is
     process MAIN[pc1:any,pc2:any,cc1:any,cc2] is
     par pc1, pc2 in
        Producteur [pc1, pc2]
     ||
         par cc1, cc2 in
           Canal [pc1, pc2, cc1, cc2]
           ||
           Consommateur [cc1, cc2]
        end par
     end par
     ...
```

LNT / étude (24/28)

Spécification d'un canal non contraint
(qui retire et délivre dans n'importe quel ordre)

```
process Canal [pc1:any, pc2:any, cc1:any, cc2:any] is
pc1 ; cc1 ; stop ||| pc2 ; cc2 ; stop
end process
```


LNT / étude (25/28)

Spécification d'un canal non fiable

```
process Canal [pc1:any, pc2:any, cc1:any, cc2:any] is
     pc1 ; (select cc1 ; stop [] i ; stop end select)
     |||
     pc2 ; (cc2 ; stop [] i ; stop)
end process
```

LNT / étude (26/28)

L'opérateur d'échappement

LNT propose l'opérateur **disrupt ... by ... end disrupt** qui modélise l'interruption **à tout moment** d'un processus principal par un autre auxilliaire.
L'expression `disrupt Cp by Ca end disrupt` signifie :
à tout moment pendant l'exécution du comportement principale `Cp`, on a le choix entre une action de `Cp` et une action du comportement auxilliaire `Ca`.
Si `Cp` est terminé avant son interruption, `Ca` n'est plus faisable.
Si `Ca` est commencé, on ne peut plus exécuter aucune action de `Cp`.

Plan de ce cours

LNT / étude (27/28)

Spécification d'un canal non fiable

Supposons que le canal peut devenir défectueux à tout moment.
Il pourrait devenir silencieux après n'importe quelle action.

```
process Canal [pc1, pc2, cc1, cc2] is
     disrupt
      ... (* comme précédemment *)
     by
      i ; stop (* à tout moment le canal peut 'tomber' *)
end process
```

Dans un tel cas le comportement du consommateur doit changer.
Exercice : écrire la spécification du consommateur dans le cas où le canal peut est défectueux.

LNT / étude (28/28)

L'opérateur de masquage

Lors de la composition de processus, on peut masquer certaines actions.

LNT propose l'opérateur `hide L in ...` qui permet de masquer une liste d'actions L dans un comportement donné.

Les actions masquées ne sont pas observables, elles sont transformées en action interne : **i**.

Exemple :

`(hide b in a ; b ; c ; stop) || a ; c ; stop`

équivaut à `a;i;c;stop`

Exercice : hide `b` **in** `(a ; b ; c ; exit || a ; c ; exit)` = ??

Références

- H. Garavel, *Notes de Cours : systèmes temps réels* , ENSIMAG, Grenoble
- Hubert Garavel, *Introduction au Langage LOTOS*, Revue de l'Association des Anciens Elèves de l'ENSIMAG, 1990
- H. Garavel, `http://www.inrialpes.fr/vasy/cadp/`
- L. Logrippo, M. Faci, M. Haj-Hussein, *An introduction to LOTOS: Learning by Examples*, Computer Networks and ISDN Systems 23(5), 1992

Références (suite)

- A. Strohmeier et D. Buchs, *Génie logiciel: principes, méthodes et techniques*, Presses polytechniques et universitaires romandes, 1996
- Ken Turner, `http://www.cs.stir.ac.uk/ kjt/research/well/index.html`
- Ken Turner, *Using Formal Description Techniques - An Introduction to Estelle, LOTOS and SDL*,John Wiley and Sons Ltd., 1993