Introduction aux algèbres de processus : définition textuelle et structurée de systèmes

à l'aide de l'outil CADP

Nicole Levy

CNAM

Vérification de modèles ou model checking

Problèmes :
Comment garantir la correction de systèmes (informatiques, électroniques ou ...) ?
Comment vérifier la satisfaction de propriétés ?

Solution :
Faire un modèle du système et
vérifier sur ce modèle les propriétés désirées

Modèle choisi :
Système de transition d'états avec transitions étiquetées
$\longrightarrow$ **Automate avec un nombre fini d'états**
(finite-state automaton ou finite state machine ou FSM)

Vérification de modèles ou model checking

Partir d'un **modèle** et d'une **propriété**
pour **vérifier si la propriété est vérifiée** et,
le cas échéant, avoir un contre-exemple d'une exécution du système qui falsifie la propriété

Partant d'un modèle de système, vérifier **exhaustivement et automatiquement** les propriétés d'exactitude des systèmes à états finis.

Exigences de sécurité : absence de blocages et d'états critiques

Vérification de modèles ou model checking

Modèle : système de transition d'états ou graphe orienté et étiqueté ou structure de Kripke

- Un **sommet** représente un état du système
- Chaque **arc** représente une transition (évolution possible du système d'un état donné vers un autre état)
- Chaque **état** du graphe orienté est étiqueté par l'ensemble des propositions atomiques vraies à ce point d'exécution

Un peu d'histoire : quelques noms

- **Amir Pnueli**, Ses travaux ont notamment porté sur les notions de vivacité et d'équité dans les systèmes concurrents et leur intégration dans les techniques de model checking.
  Il a reçu le prix Turing en 1996 pour un « travail fondateur introduisant la logique temporelle en informatique et pour des contributions exceptionnelles à la vérification des programmes et systèmes ».
- **Edmund M. Clarke, E. Allen Emerson et Joseph Sifakis** au tout début des années 1980. Ils se sont vu attribuer le prix Turing 2007 pour leurs travaux sur le model checking.
- **Leslie Lamport**, prix Turing 2013.

La vérification de modèles (Model-Checking)

Un lien intéressant :
Le cours au collège de France de Gérard Berry du 25/03/2015 :
http://www.college-de-france.fr/site/gerard-berry/course-2015-03-25-16h00.htm

Les algèbres de processus

Issue des travaux fondateurs de **C.A.R. Hoare** *et* **R. Milner** *— tous deux lauréats du prix Türing — l'école des algèbres de processus a produit des résultats théoriques profonds, à commencer par une vision unifiée, élégante et féconde du parallélisme asynchrone.*

Ces idées ont conduit à la définition de multiples langages, trop nombreux pour qu'on les cite tous ici : **CSP** *[HOA 78],* **CCS** *[MIL 80, MIL 89],* **TCSP** *[BRO 84, HOA 85, ROS 97, SCH 99],* **ACP** *[BER 84, BER 85],* **FP2** *[JOR 84],* **MEIJE** *[de 85],* **CIRCAL** *[MIL 85],* **LCS** *[BER 94, BER 95, BER 96],* **LOTOS** *[ISO 89],* **PSF** *[MAU 90, MAU 93],* μ**CRL** *[GRO 95, GRO 97],* **FSP** *[MAG 99b],* **E-LOTOS** *[ISO 01],* **LOTOS NT** *[SIG 00]. . .*

Hubert Garavel
INRIA Rhône-Alpes

Les algèbres de processus

... dont ces 3 langages pour la définition, validation et vérification de systèmes

- **CSP**: Communicating Sequential Processes
- **CCS**: Calculus of Communicating Systems
- **ISO FDT group**: Formal Description Techniques for specification of protocols and services : **Estelle** et **LOTOS**
  (...et sa nouvelle version **LNT**)

CSP: Communicating Sequential Processes

C.A.R. Hoare *(1934,-),*
University of Oxford

Bibliographie

- C.A.R. Hoare: Communicating Sequential Processes, Prentice Hall, 1985
- A.W. Roscoe: The theory and Practice of Concurrency, Prentice Hall, 1998
- S. Schneider: Concurrent and Real-time Systems: the CSP Approach , Wiley, 1999
- http://usingcsp.com/cspbook.pdf/

Outils de Validation de spécifications CSP

- FDR: Model-Checker
- PVS: Theorem prover
- CTJ : CSP to JAVA, permet l'exécution de programmes CSP, en les traduisant (compilant) en Java.

CCS: Calculus of Communicating Systems

Robin Milner *(1934-2010),*
University of Edinburg (1973-99)
Computer Laboratory in Cambridge
(1995-... 2010)

Bibliographie

- *R. Milner: A Calculus of Communicating Systems, Springer, 1980*

Syntaxe de CCS

$$P ::= \emptyset \mid a.P_1 \mid A \mid P_1 + P_2 \mid P_1 \mid P_2 \mid P_1[b/a] \mid P_1 \backslash a$$

ISO FDT group: Formal Description Techniques for specification of protocols and services

Objectifs : abstraction, indépendance vis à vis de l'implémentation, sémantique formelle, et support pour la vérification

2 orientations:

- Machines d'états finis étendues: Estelle
- Techniques d'ordonnancement temporel: LOTOS

LOTOS

"Language Of Temporal Ordering Specification", ISO 8807 (1989)

- Défini pour spécifier les protocoles de télécommunication
- Utilisable pour les systèmes distribués en général
- Standardisé par l'ISO SC21/WG1 pour soutenir la normalisation de l'OSI (Open Systems Interconnection)
- Permet de décrire
  "des ensembles de processus qui interagissent et échangent des données entre eux et avec leur environnement"

LOTOS: grands principes

- Deux formalismes complémentaires pour
  les données : ACT-ONE et
  le contrôle: un langage basé sur CCS et CSP

  La définition des types de données était un des plus grands problèmes de LOTOS et d'une des raisons de l'évolution vers LNT
- Sémantique par entrelacement: les évènements sont atomiques et concurrents
- Définition formelle:
  syntaxe formelle,
  sémantique statique définie par une grammaire attribuée et
  sémantique opérationnelle définie par des règles d'inférence

LOTOS: grands principes

- Preuves de propriétés : Algèbre de processus
- Efficacité des outils de vérification automatique
- Prototypage rapide : exécutabilité des comportements
- Modularité et réutilisation (par paramétrage des modules)

$\Rightarrow$ *Définition de* ***processus*** *qui s'exécutent en parallèle et se synchronisent par rendez-vous*

Utilisation de la boîte à outils **CADP**

*Site de l'****outil CADP****: cadp.inria.fr*

La boîte à outils **CADP**

Beaucoup d'outils très puissants et variés allant de la simulation pas à pas à la vérification de modèles massivement parallèles.
Plusieurs langages peuvent être utilisés en entrée : LOTOS (ISO 8807), LNT, FSP, pi-calcul, etc.
La boîte à outils CADP fournit des compilateurs (CAESAR, CAESAR.ADT, LNT2LOTOS, FSP2LOTOS, etc.) qui traduisent les descriptions de haut niveau en code C à utiliser pour la simulation, la vérification et les tests.

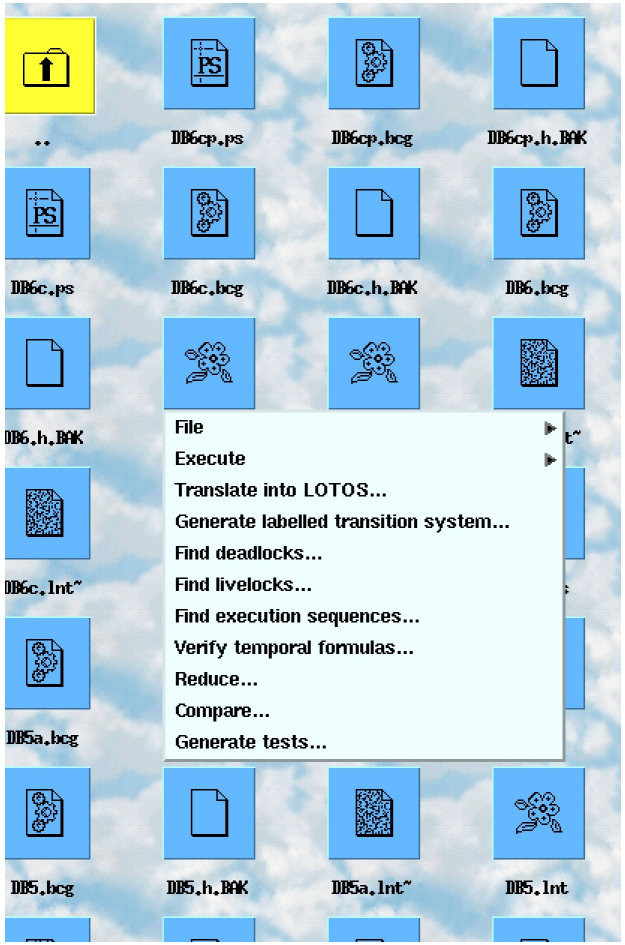
La boîte à outils **CADP**

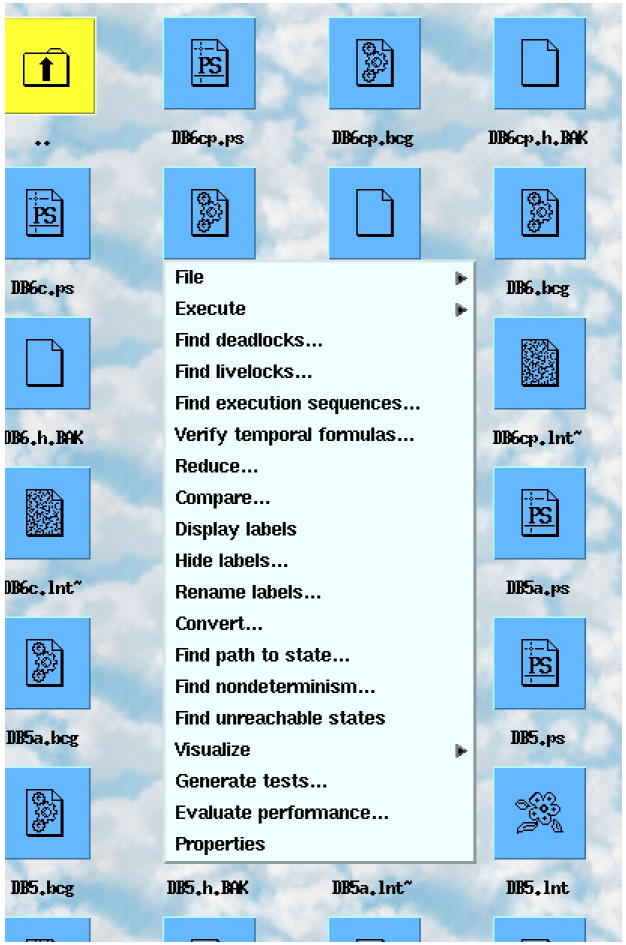
Plusieurs outils sont proposés tels que :

- outils de vérification d'équivalence (réduction et comparaisons de relations de bissimulations), tels que BCG_MIN et BISIMULATOR.
- outils de simulation
- outils de visualisation graphique
- outils de vérification de modèles pour diverses logiques temporelles et mu-calculs, tels que EVALUATOR et XTL
- .. plus d'autres outils avec des fonctionnalités avancées telles que la vérification visuelle, l'évaluation des performances, etc.

La boîte à outils **CADP**

http://cadp.inria.fr/





LOTOS → LNT

Les algèbres de processus plus agréables à utiliser

LNT ("LOTOS New Technology") :

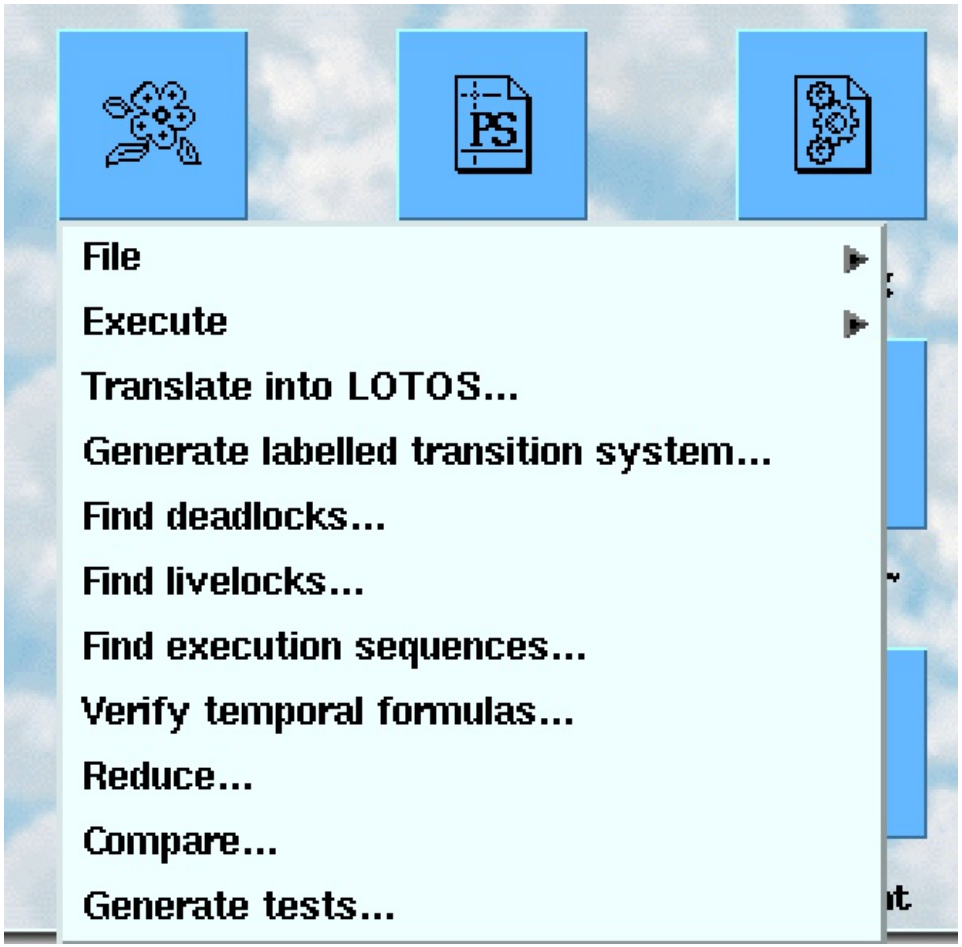
- conçu et implémenté dans CADP depuis 2005 pour être :
  - concis,
  - expressif,
  - facilement lisible et convivial
- combinaison du calcul de processus, de langages fonctionnels et de langages impératifs
- intensivement utilisé pour spécifier et vérifier les systèmes concurrents utilisant CADP
- LNT remplace progressivement LOTOS comme langage de modélisation

LNT : 2 parties

- La partie processus : description des échanges et interactions entre composants
- La partie données : définir des structures de données et accéder aux données d'un système
  - mémorisées ou
  - échangées entre composants

LNT → LOTOS

- (À l'heure actuelle,) LNT est mis en œuvre par traduction vers LOTOS : **LNT2LOTOS**
- LNT est intégré dans CADP



Algèbre de processus ...
Mais qu'est-ce qu'on entend par processus ?

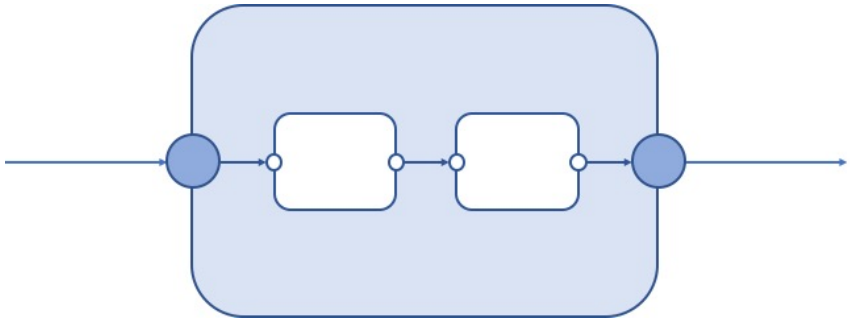
Processus = Composant

- Un processus est considéré comme une boîte noire à un certain niveau d'abstraction: seul son comportement externe est étudié

- Un processus possède des points d'interaction appelés **portes**
- Une porte est activée par un **évènement** ou une **action**
- Une **porte** est la partie visible d'un composant qui peut accepter un **évènement** qui va provoquer une **action**. Cette action peut être visible ou interne.

Comportement des processus

Système = ensemble de **composants** qui interagissent avec leur environnement via leurs portes
Vue compositionnelle: Deux composants interagissant forment un système plus large les englobant



Comportement d'un processus

- Un processus est spécifié en précisant son
  comportement (behaviour)
- La définition d'un processus décrit le comportement visible en terme d'une **suite d'événements** auxquels il participe
- Le comportement est décrit par un terme obtenu par combinaison des opérateurs de comportement que nous allons étudier

Comportement d'un processus : un jeu de "lego"

Constructions par opérateurs primitifs

Un petit nombre d'opérateurs primitifs pouvant être composés arbitrairement (orthogonalité), chacun exprimant un concept

- pas de comportement : **stop**
- communication par rendez-vous sur une porte
- composition séquentielle : **;**
- comportement itératif: **loop**
- choix déterministe: **if..then..else**
- choix non déterministe : **select**
- composition parallèle : **par**
- masquage d'actions : **hide**
- appel de processus
- ...

Comportement d'un processus

- L'expression du comportement d'un processus définit ses **états** et les **actions** (ou **événements**) auxquels il peut prendre part
- A chaque instant, un processus peut:
  - exécuter un comportement interne
  - ne plus rien faire: comportement inactif *stop* (deadlock)
  - terminer en succès
  - se synchroniser sur une porte avec un autre processus ou avec l'environnement

Événement

- Un **événement** représente une synchronisation entre processus (ou avec son environnement)
- Un événement demande la participation d'au moins 2 processus (des valeurs peuvent être échangées)
- **Les événements sont atomiques** ; la synchronisation des processus et les informations échangées sont réalisées en une unité de temps
- **Alphabet** d'un processus: l'ensemble des événements auxquels il peut participer
- Si aucune synchronisation n'est possible, il y a **interblocage (deadlock)**
- Si le processus ne propose plus de synchronisation, il y a **perte de vivacité (livelock)**

Spécification en LNT

Une spécification en LNT est un ensemble de définitions de modules

Un **module**
peut importer d'autres modules
et comprendre des définitions de :

- Types
- Fonctions
- Canaux de communication
- Processus

Spécification en LNT

Une spécification en LNT est un ensemble de définitions de modules

Un **module** *M*

peut importer d'autres modules : via ses paramètres *M*0, ...*Mn*

et comprendre des définitions de :

- Types : `type` id `is ... end type`
- Fonctions : `function` id `is ... end function`
- Canaux de communication : `channel` id `is ... end channel`
- Processus : `process` id `is ... end process`

Définition d'un module

Module == **module** $M\ (M0, ..Mn)$ **is**
Types
Fonctions
Canaux
Processus
end module

Définition d'un processus

Processus == **process** *P* [*Porte**,](*Parametre**,) **is**
Comportement
end process

Porte == *P*0, ...*Pn* : *Canal*
Parametre == *v*0, ...*vn* : *Type*
Comportement == (...cf ci-dessous !)

Un exemple pour commencer : un passage à niveau

Un exemple simple :
Modélisation d'un **passage à niveau** ou **croisement** avec une seule voie de train et une barrière qui se lève et descend
Le train arrive près du croisement, il entre dans le croisement puis en sort
La barrière doit se baisser avant son entrée, puis de lever après sa sortie
Ce qui nous intéresse est de savoir s'il y a un chemin d'événements où le train pourrait entrer dans le croisement avant la descente de la barrière

Voir la présentation de l'exemple du TRAIN

Comportement d'arrêt: STOP

Le comportement «**stop**» met fin à l'exécution du processus
Cette fin est alors considérée comme infructueuse,
car tout comportement est impossible après un «stop»

$\Rightarrow$ «stop» représente un blocage (deadlock)

Comportements action et séquence ;

Un comportement (d'un processus ou dans un processus) peut juste être une **action** ou un **rendez-vous sur une porte** ou un **évènement**
L'action :

a

dénote un rendez-vous sur la porte **a**

L'opérateur ; dénote la séquence des comportements, par exemple des actions

a ; **b**

où **a** et **b** sont des actions (ou événements ou portes)
Le comportement décrit la séquence de l'action **a** suivie de l'action **b**

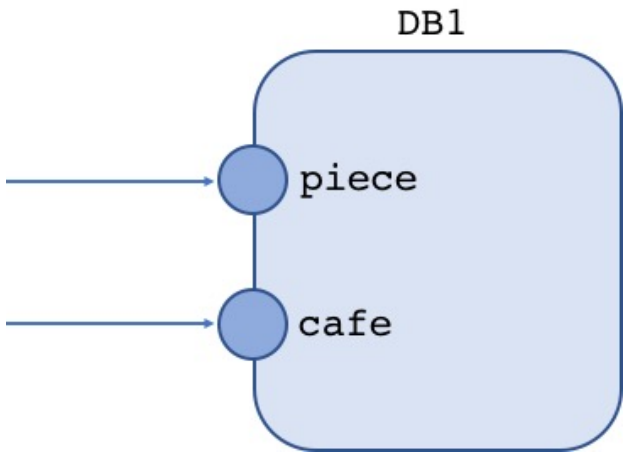
Exemple: un distributeur de boissons

Actions :

- pièces (le distributeur reçoit des pièces)
- café (le distributeur distribue du café)

```
process DB1 [piece, cafe : none] is
      piece ; cafe
end process
```

Le processus réalise les actions `piece` puis `cafe`, puis s'arrête



Sémantique

Arbres de transitions: toutes les transitions possibles du système à partir de l'état initial

Dans le graphe des états, une transition est une flèche étiquetée de la forme :

$$e_1 \rightarrow_a e_2$$

e_1, e_2:	état	
a:	action qui peut être	un événement externe un événement terminal, une action interne (i)

Sémantique

Exemple du distributeur de boissons:

Traces ou chemins possibles :

(0)

$(0) \rightarrow_{piece} (1)$

$(0) \rightarrow_{piece} (1) \rightarrow_{cafe} (2)$

$(0) \rightarrow_{piece} (1) \rightarrow_{cafe} (2) \rightarrow_{exit} (3)$

Itération infinie

Il est utile de pouvoir définir des processus qui *tournent* sans arrêt, qui vivent indéfiniment... : ils bouclent (Loop)

```
process Proc [porte : none] is
    loop
       porte
    end loop
end process
```

On choisit `porte` à l'infini

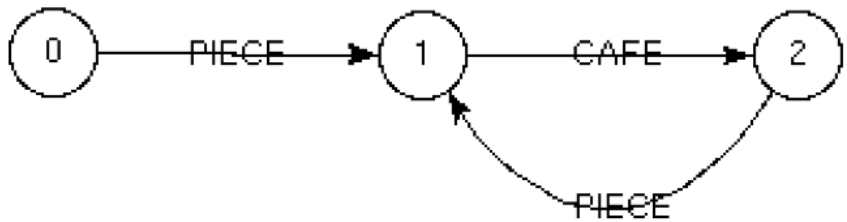
Itération infinie

Exemple du distributeur de boissons:

```
process DB2 [piece, cafe : none] is
     loop
         piece ;  cafe
     end loop
end process
```

On choisit alternativement `piece` puis `cafe` à l'infini

Une **trace** parmi l'infinité de traces :
$<$ piece , cafe , piece , cafe , piece $>$



Choix indéterministe

B1, **B2**: comportements

select B1 [] B2 end select

*se comporte soit comme **B1** soit comme **B2** de manière indéterministe*

- L'opérateur de choix [] est associatif et commutatif
- Dans un choix, les alternatives qui mènent à un deadlock ne sont pas prises en compte:
  (select B [] stop end select) = B

Choix indéterministe gardé

a, b: événements
B1, B2: comportements

select **a** ; **B1** [] **b** ; **B2** **end select**

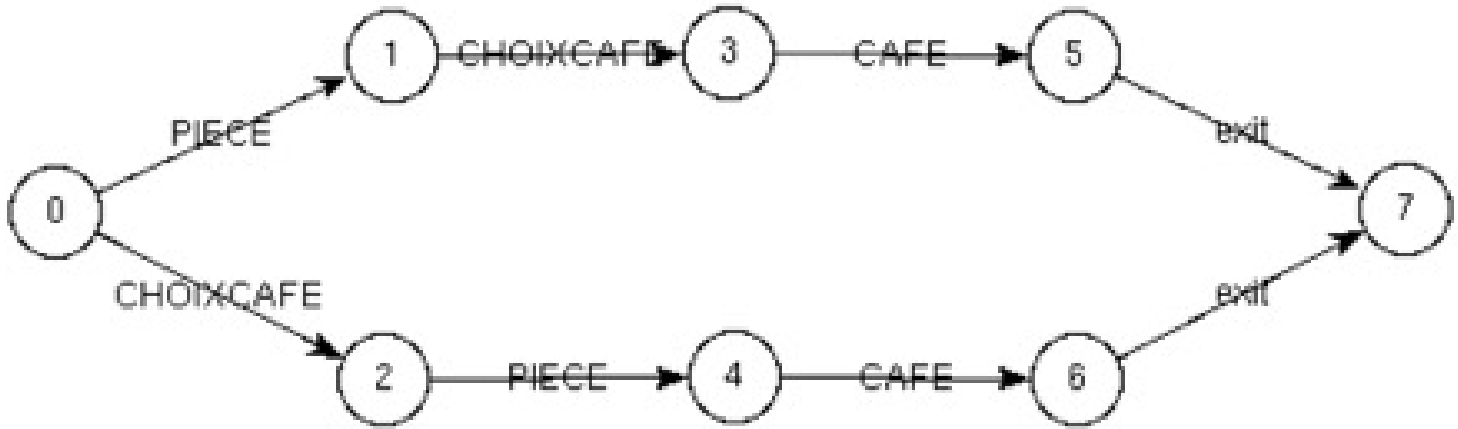
Processus qui peut faire :

a *puis se comporter comme* **B1**
ou
b *puis se comporter comme* **B2**

Le choix de la branche est fait par l'environnement du processus :
si l'évènement **a** survient alors le processus fera **B1**
si l'évènement **b** survient alors le processus fera **B2**

Exemple du distributeur de boissons

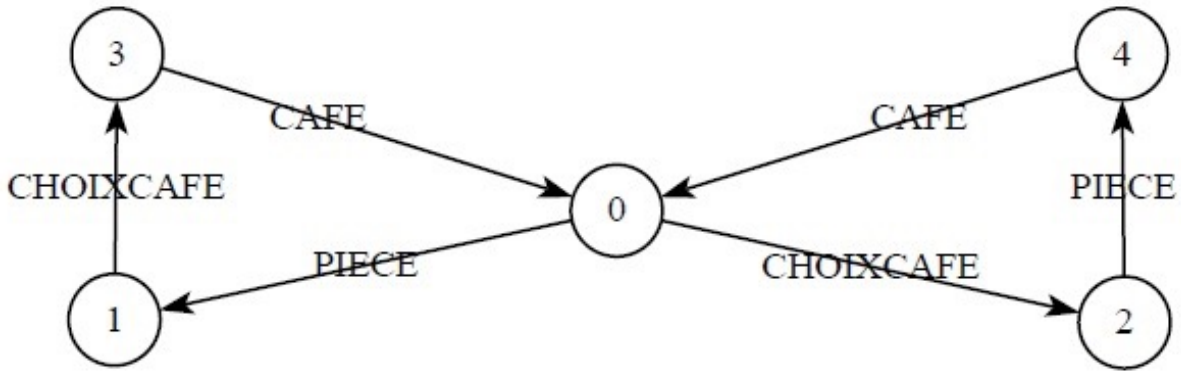
```
process DB3 [piece, choixcafe, cafe : none] is
    select
       piece ; choixcafe ; cafe
    []
       choixcafe ; piece ; cafe
    end select
end process
```



Version itéractive du distributeur de boissons

Avec une boucle (loop) sans fin

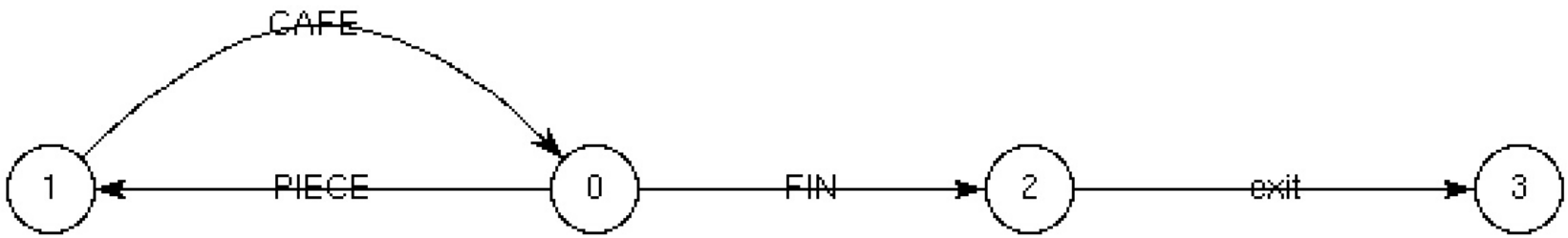
```
process DB4 [piece, choixcafe, cafe : none] is
    loop
        select
                piece ; choixcafe ; cafe
        []
                choixcafe ; piece ; cafe
        end select
    end loop
end process
```



Opérateur d'itération

Utilisation de l'opérateur de choix gardé pour décrire un processus itératif que se termine après plusieurs itérations :

```
process DB2a [piece, cafe, fin : none] is
loop arret in
        select
            piece ; cafe
        []
            fin ; break arret
        end select
    end loop
end process
```



Synchronisation parallèle

Les processus se synchronisent sur leur portes communes ... ou pas : il faut lister les portes de synchronisation

a, b: événements
B1, B2: comportements acceptant **a** et **b**

B1 et **B2** s'exécutent en parallèle :
Ils se synchronisent par rendez-vous sur les événements **a** et **b** :

par a, b in B1 || B2 end par

Ils se synchronisent par rendez-vous sur l'événement **a** :

par a in B1 || B2 end par

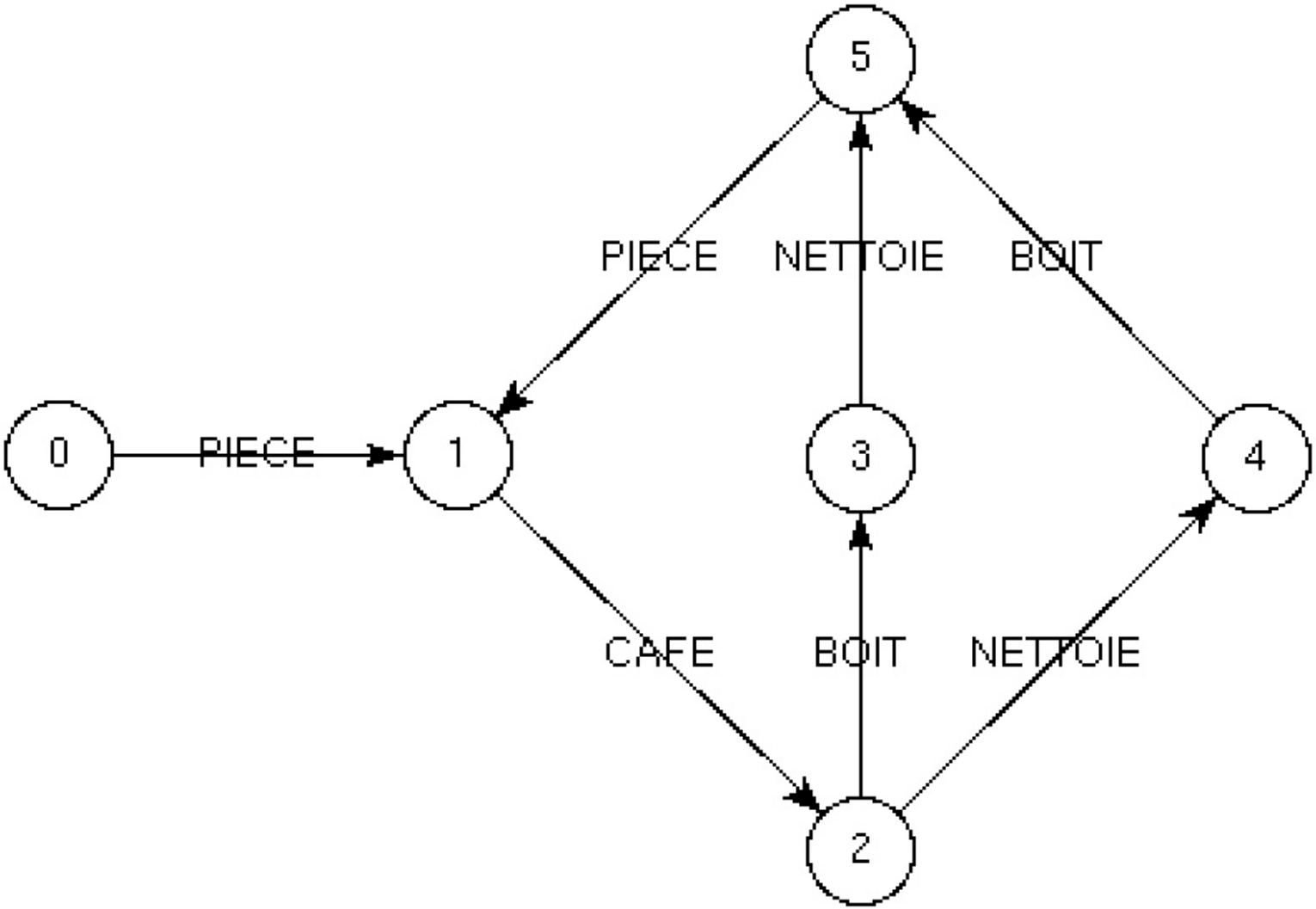
Ils ne se synchronisent pas : ils sont indépendants. Les actions de **B1** et celles de **B2** s'entrelacent de toutes les façons possibles

par B1 || B2 end par

Distributeur de boissons avec synchronisation parallèle

```
module DB5 is
process MAIN [piece, cafe, boit, nettoie : none] is
    par piece, cafe in
        DB5[piece,cafe,nettoie]
    ||
        CLIENT[piece,cafe,boit]
    end par
end process
process DB5[piece,cafe,nettoie : none] is
    loop
         piece; cafe; nettoie
    end loop
end process
process CLIENT[piece,cafe,boit : none] is
    loop
        piece; cafe; boit
    end loop
end process
```

Distributeur de boissons avec synchronisation parallèle



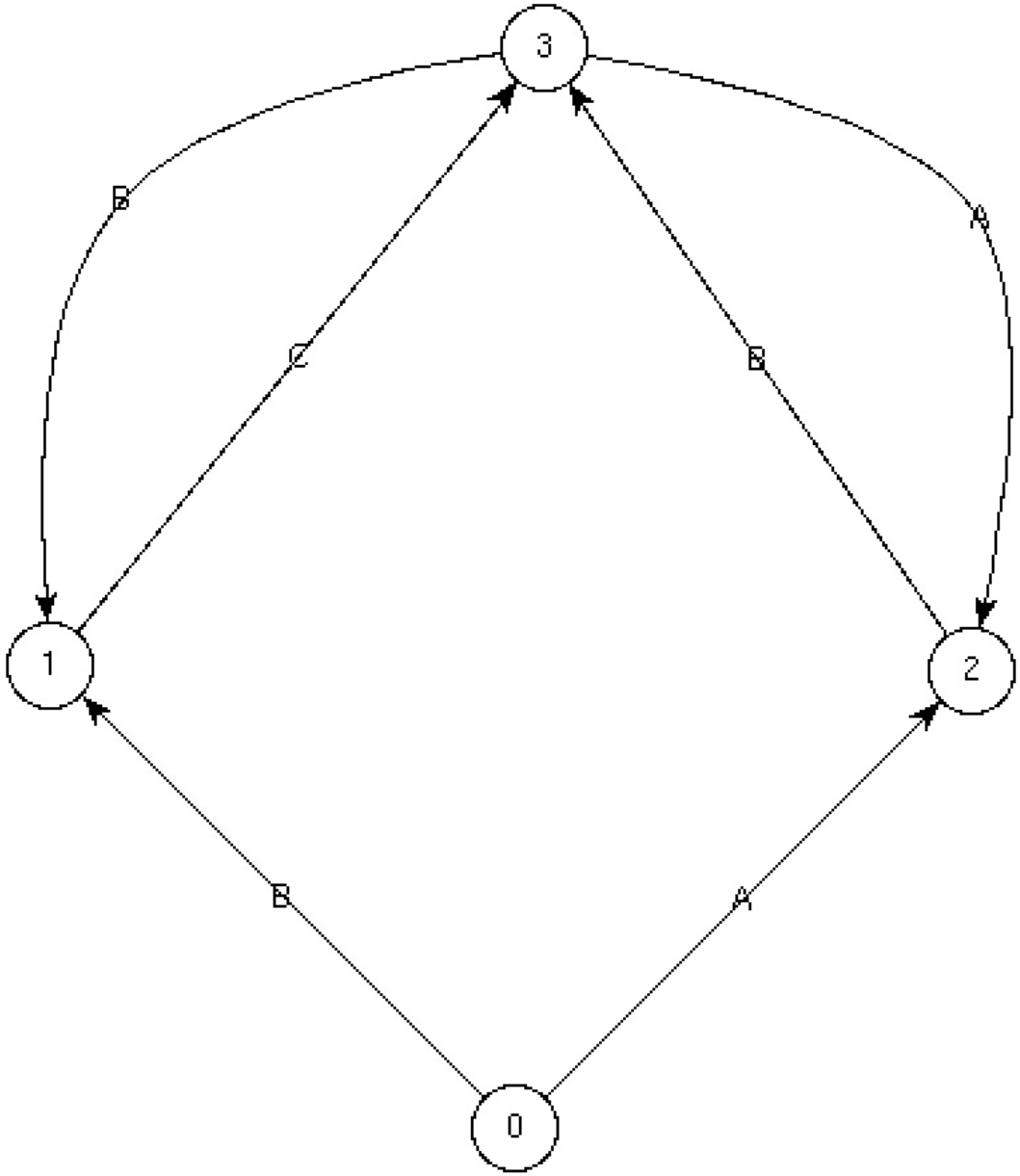
La synchronisation

- Synchronisation n-aire (multiway) qui vient de CSP
- Synchronisation symétrique : pas de direction, pas de notion d'initiateur de la synchronisation, tous les processus participent de façon équitable
- Vers l'environnement : synchronisation anonyme
- Synchronisation non-déterministe
- La terminaison par exit est toujours synchrone; le processus qui se termine en premier attend les autres

La synchronisation n-aire: un exemple

```
module NAIRE is
process MAIN [a, b, c : none] is
    par
        a, b    -> A1[a,b]
    ||  a, b, c -> A2[a,b,c]
    ||  b, c    -> A3[b,c]
    end par
end process
process A1 [a,b : none] is loop
        select a [] b end select
end loop end process
process A2 [a,b,c : none] is loop
        select a ; b [] b ; c end select
end loop end process
process A3 [b,c : none] is loop
        select b [] c end select
end loop end process
end module
```

La synchronisation n-aire: un exemple



Distributeur de boissons avec ou sans synchronisation

Avec

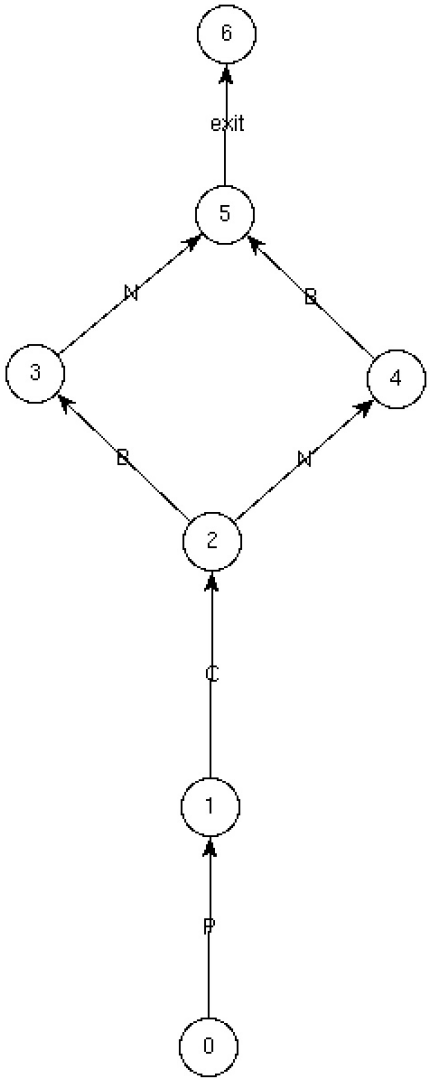
```
module DB6cp is
process MAIN[P,C,N,B:none] is
     par P,C in
        DB6[P,C,N]
     ||
        CLIENT[P,C,B]
     end par
end process
process DB6[P,C,N:none] is
     P; C; N
end process
process CLIENT[P,C,B:none] is
     P; C; B
end process
end module
```

Sans

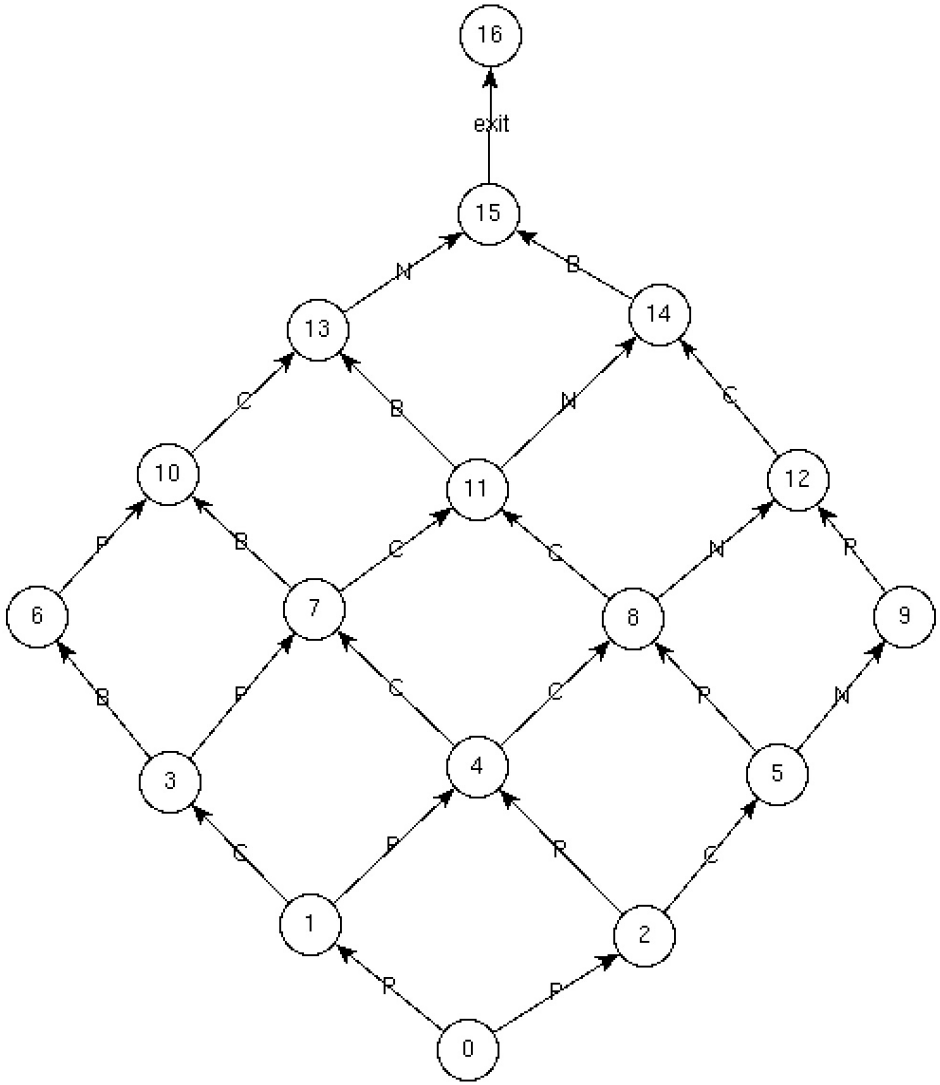
```
module DB6c is
process MAIN[P,C,N,B:none] is
     par
        DB6[P,C,N]
     ||
        CLIENT[P,C,B]
     end par
end process
process DB6[P,C,N:none] is
     P; C; N
end process
process CLIENT[P,C,B:none] is
     P; C; B
end process
end module
```

Distributeur de boissons avec ou sans synchronisation

Avec



Sans



Masquage d'actions **hide**

a, b, c: événements
P : Comportement avec actions **a, b**

hide `a : A , b : B` **in** `P` **end hide**

masque les actions typées `a, b` *dans le comportement* `P`

Les actions masquées ne sont pas observables : elles sont transformées en actions internes **i** *(invisibles)*

Exemple :

```
hide b : none in a ; b ; c end hide
```

équivaut à

```
a ; i ; c
```

Autres structures de contrôles

- Déclaration locale de variables **var**
- Opération vide (ne fait rien) : **null**
- Composition séquentielle **;**
- Affectation **:=**
- Choix déterministe **if ... then ... else**, **case**
- Itération **loop**, **while**, **for** avec ou sans sortie **break**

Ces structures de contrôle utilisent des valeurs.

Différents types d'itérations

B un comportement, *P* une propriété, *I* une instruction, L un nom d'interruption :

- Itération infinie :
  loop *B* end loop
  B est itéré à l'infini
- Itération tant qu'une propriété ***P*** est vérifiée :
  while *P* loop *B* end loop
- Itération pour :
  for I_0 while *P* by I_i loop *B* end loop
- Itération avec sortie sur interruption (**break**) :
  loop *L* in *B incluant un* break *L* end loop

Prise en compte de valeurs et de variables valuées

Les valeurs peuvent être :

- affectées à des variables locales à un processus
- paramètres d'un processus : elles représentent son état
- **échangées lors de communications**

Typage des variables

Les variables sont typées
Un type peut être :

- Un type de base prédéfini : **bool, char, nat, int, real, string**
- Un type structuré :
  - Tableau : **array [n..m] of T**
  - Liste : **list of T**
  - Ensemble : **set of T**
  - ...
- Un type défini dans une clause **type ... end type** du module, comme par exemple :
  - Un type énuméré :
    type T is v1, v2,... end type
  - Un Produit cartésien (record) :
    type T is tuple (n1 : T1, n2 : T2,...) end type
  - ...

Échange de valeurs lors d'une communication

Un processus communique par synchronisation avec son environnement via ses portes de communication

Lors d'une synchronisation, il peut échanger des valeurs

La communication est contrainte par le type de la valeur attendue :

- **any** : pas de contrainte sur le type de la valeur échangée
- **none** : pas de valeur échangée
- **C**
  où **C** est le nom d'un canal de communication défini

Rappel : définition d'un module intégrant la déclaration des canaux

Les canaux de communication sont déclarés dans un module

Module == **module** *M* $(M0, ..Mn)$ **is**
Types
Fonctions
Canaux
Processus
end module

Déclaration de canaux de communication

channel C is (T1, T2, ...) end channel
liste de (tuples de) types de valeurs possibles échangées

Example :

channel C is (nat), (bool, bool), () end channel

Signifie que sur le canal de communication **C** *peuvent être échangées des valeurs d'un des types suivants : nombre naturel ou couple de booléens ou pas de valeur*

Il existe 2 types de canaux de communication prédéfinis :

- **any** : pas de contrainte sur le type de la valeur échangée
- **none** : pas de valeur échangée
  est équivalent à : **channel none is () end channel**

Rappel : Définition d'un processus

Processus == **process** *P* [*Porte**,](*Parametre**,) **is**
Comportement
end process

Porte == *P*0, ...*Pn* : *Canal*
Parametre == *v*0, ...*vn* : *Type*
Comportement == (...cf ci-dessus !)

Chaque porte est caractérisée par son **canal de communication**
Les canaux doivent être définis avant les processus
Les canaux peuvent mentionner des types qui doivent être aussi définis avant
(Jusqu'à présent on a utilisé **none** car aucune valeur n'était échangée)

Échange de données lors d'une synchronisation

Une action de synchronisation est composée :

- d'une porte,
- d'une liste d'événements où un évènement peut
  - offrir une valeur : **!**
  - accepter une valeur : **?**
- optionnellement d'un prédicat **where pred**

Exemple :

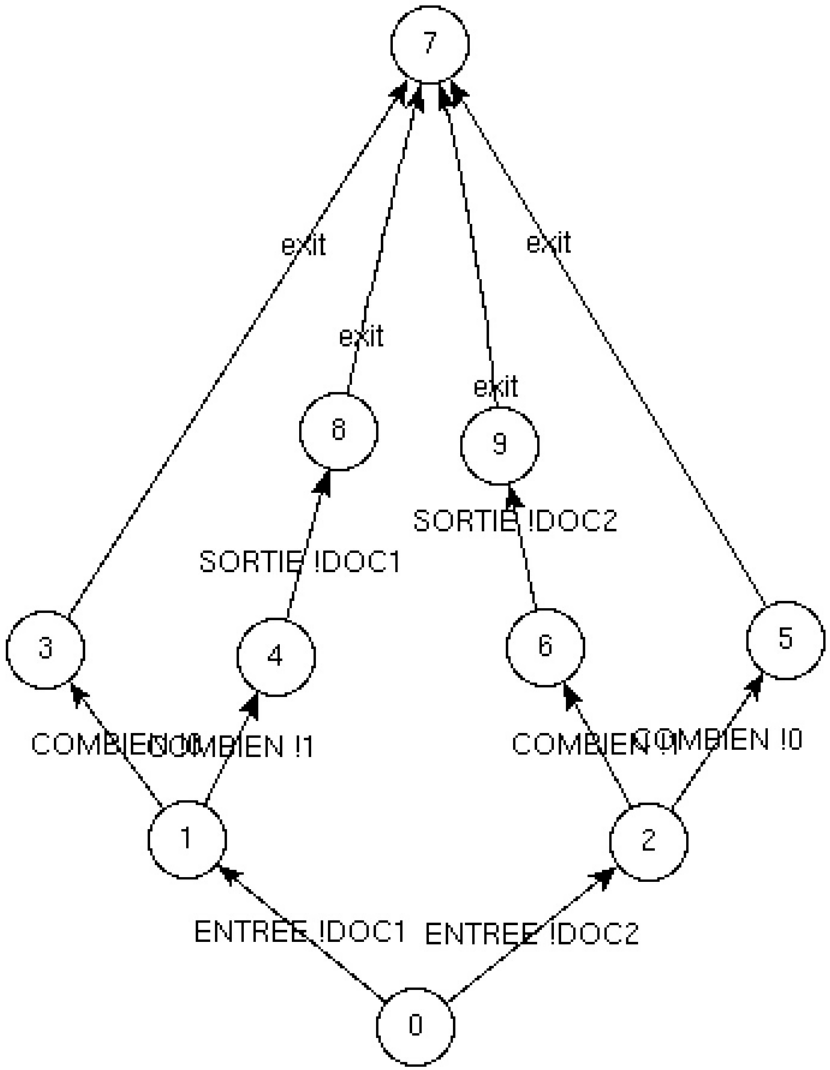
```
var x : Nat in ...
       P (? x, ! 1)  where x > 0 ; ...
end var
```

attend sur la porte `P` *un naturel strictement positif qui sera affecté à* `x` *et émet la valeur* `1`

Entrées - Sorties: exemple Copie

```
module COPIE is
type DOCUMENT is
      doc1, doc2 with "==", "!="
end type
-----------------------
channel D is (DOCUMENT) end channel
channel N is (NAT) end channel
-----------------------
process COPIE [entree, sortie : D, combien : N] is
        var original : DOCUMENT, i, n : NAT in
            entree(?original) ;
            combien(?n) where n<3 ;
            for i := 1 while (i <= n) by i := i +1 loop
                sortie (!original)
            end loop
        end var
end process
end module
```

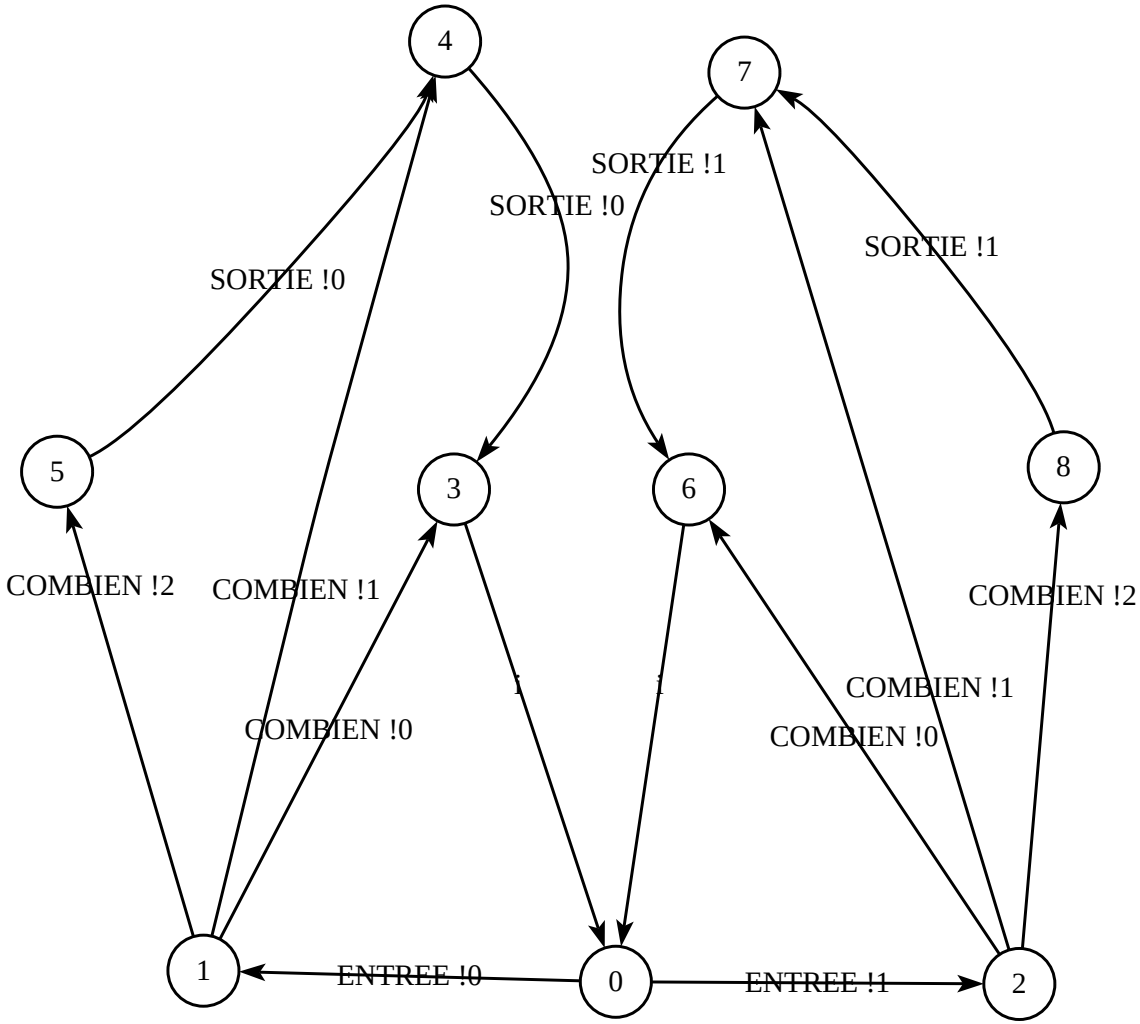
Entrées - Sorties: exemple Copie 2 fois



Entrées/Sorties: exemple Copie avec itération

```
module COPIEi is
type Document is
     doc1, doc2    with "==", "!="
end type
channel D is (DOCUMENT) end channel
channel N is (NAT) end channel
process COPIEi [entree, sortie : D, combien : N] is
    var original : DOCUMENT, i, n : NAT in
        loop
            entree(?original) ;
            combien(?n) where n<3 ;
            for i := 1 while (i <= n) by i := i +1 loop
                sortie (!original)
            end loop
        end loop
    end var
end process
end module
```

Entrées/Sorties: exemple Copie 3 fois avec itération



Choix gardé:

Utiliser des valeurs pour sélectionner un comportement

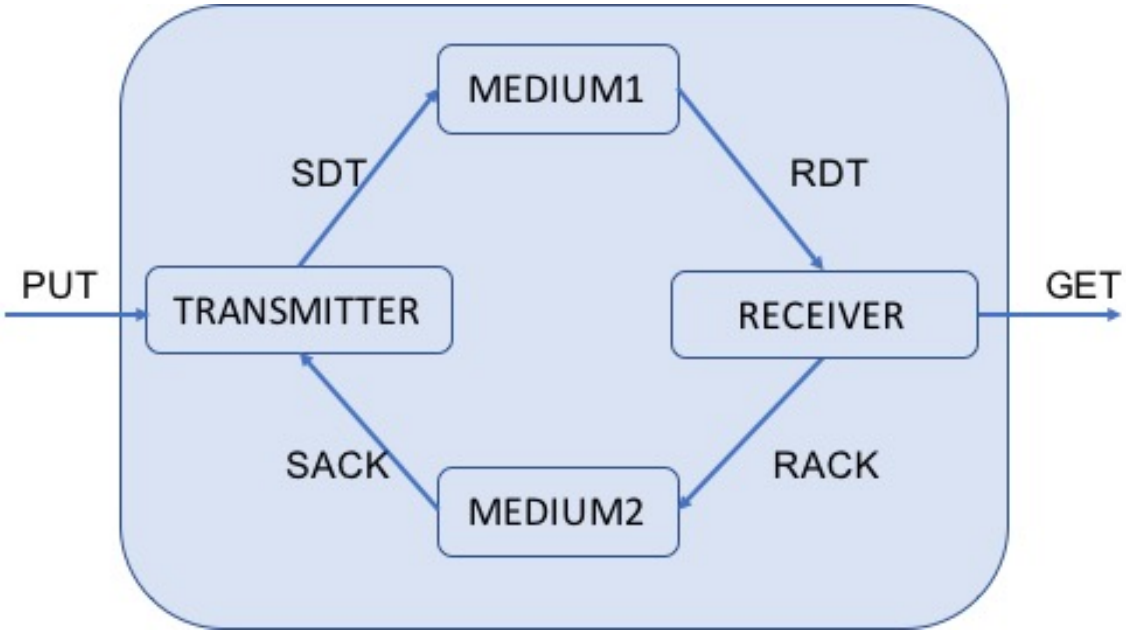
P: porte
a, b: événements : offre **!v** ou réception **?v** d'une valeur **v**

select
P a where cond1
[]
P b where cond2
end select

si **cond1** *est vérifié, alors sur la porte* **P** *faire l'événement* **a**
ou
si **cond2** *est vérifié, alors sur la porte* **P** *faire l'événement* **b**

Exemple du protocole du bit alterné

1er protocole spécifié par un automate de transitions d'états [Bartlett et Scantlebury, Comm ACM, May 1969]



Les canaux de communication MEDIUM1 et MEDIUM2 peuvent perdre les messages.
Alors tout message est associé à un bit (false ou true) qui alterne et le récepteur renvoie le bit en acquittement

Fonctionnement du protocole : cas nominal

envoi du message $m1$

$\rightarrow PUT\ m1$

TRANSMITTER envoie le message étiqueté 0

$\rightarrow SDT<m1,0>$

$\rightarrow RDT<m1,0>$

RECEIVER attend un message étiqueté 0

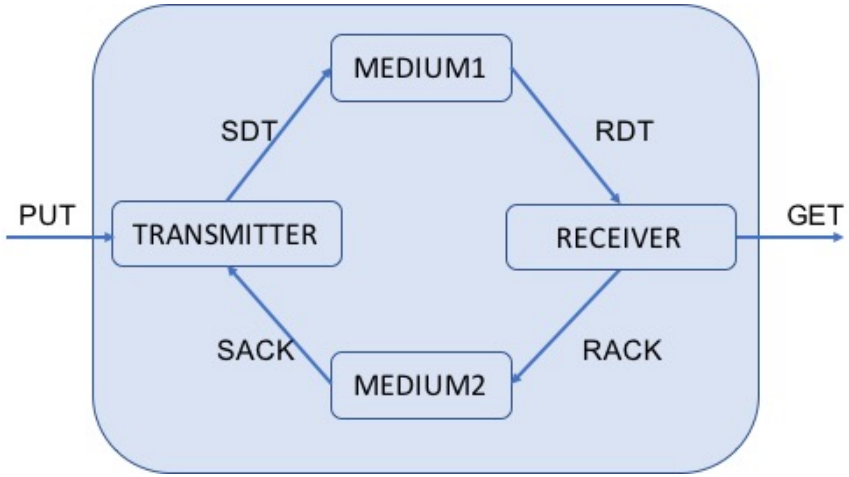
RECEIVER reçoit m1 et renvoie l'acquittement 0

$\rightarrow GET\ m1$

$\rightarrow RACK<0>$

$\rightarrow SACK<0>$

TRANSMITTER reçoit l'acquittement 0



$\rightarrow PUT\ m2$ envoi du message $m2$

$\rightarrow SDT<m2,1>$ TRANSMITTER envoie le message étiqueté 1

$\rightarrow RDT<m2,1>$ RECEIVER attend un message étiqueté 1

$\rightarrow GET\ m2$ RECEIVER reçoit m2

$\rightarrow RACK<1>$ RECEIVER renvoie l'acquittement 1

$\rightarrow SACK<1>$ TRANSMITTER reçoit l'acquittement 1 ...

Fonctionnement du protocole : perte d'un message (1)

envoi du message $m1$

$\rightarrow PUT\ m1$

TRANSMITTER envoie le message étiqueté 0

$\rightarrow SDT<m1,0>$

MEDIUM1 perd le message étiqueté 0

RECEIVER attend et ne reçoit rien

ne voyant pas venir l'acquittement

TRANSMITTER renvoie le message

$\rightarrow SDT<m1,0>$

cette fois MEDIUM1 ne perd pas le message

$\rightarrow RDT<m1,0>$

RECEIVER attend un message étiqueté 0

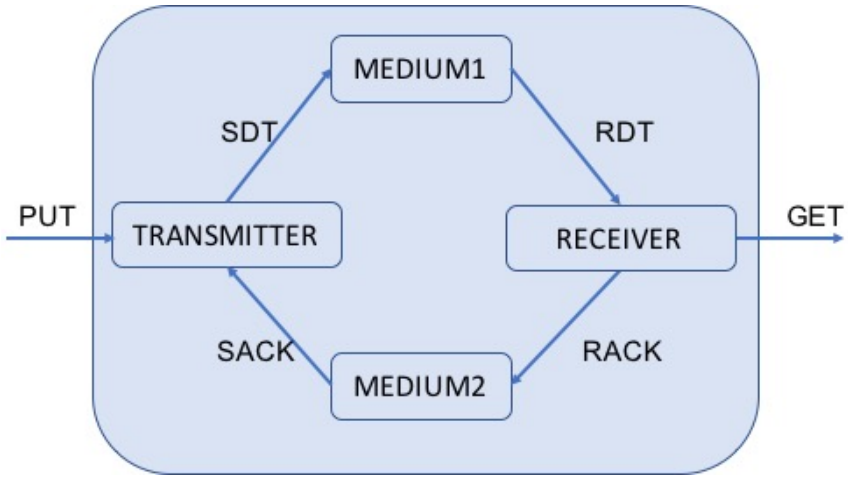
RECEIVER reçoit m1 et renvoie l'acquittement 0

$\rightarrow GET\ m1$

$\rightarrow RACK<0>$

$\rightarrow SACK<0>$

TRANSMITTER reçoit l'acquittement 0



Fonctionnement du protocole : perte d'un message (2)

envoi du message $m1$

$\rightarrow PUT\ m1$

TRANSMITTER envoie le message étiqueté 0

$\rightarrow SDT<m1,0>$

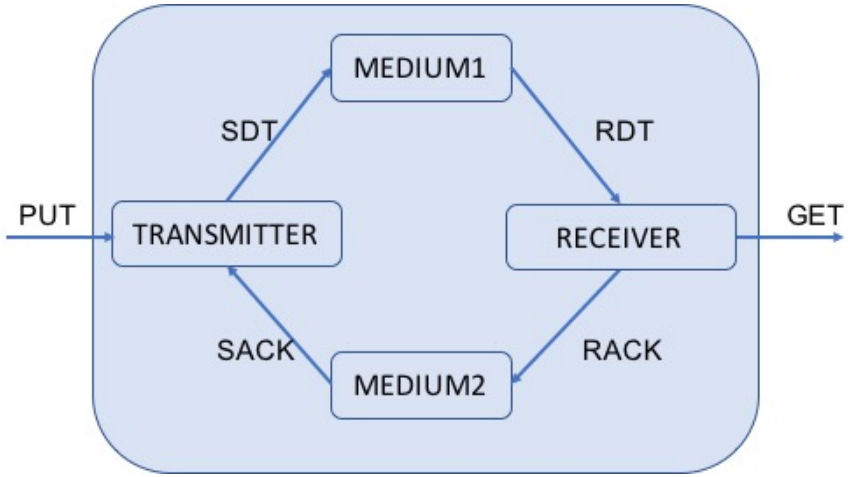
$\rightarrow RDT<m1,0>$

RECEIVER attend un message étiqueté 0

RECEIVER reçoit m1 et renvoie l'acquittement 0

$\rightarrow GET\ m1$

$\rightarrow RACK<0>$



MEDIUM2 perd l'acquittement 0, ne voyant pas venir l'acquittement, TRANSMITTER renvoie le message

$\rightarrow SDT<m1,0>$

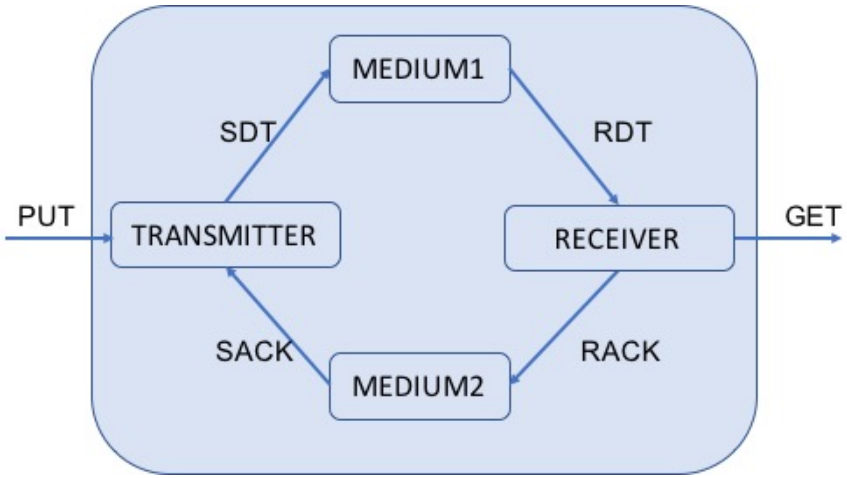
$\rightarrow RDT<m1,0>$

RECEIVER reçoit le message étiqueté 0 : il en déduit que son acquittement a été perdu et renvoie l'acquittement 0

$\rightarrow RACK<0>$ MEDIUM2 ne perd pas l'acquittement 0

$\rightarrow SACK<0>$ TRANSMITTER reçoit l'acquittement 0

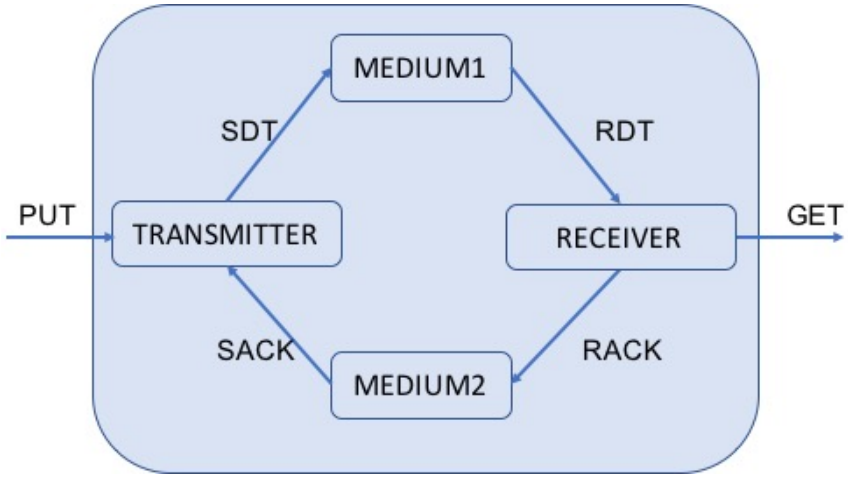
Protocole du bit alterné en LNT: Module BITALT




```
module BITALT is
type Msg     ... end type
channel      ... end channel
process MAIN                    ...  end process
process TRANSMITTER             ...  end process
process RECEIVER                ...  end process
process MEDIUM1                 ...  end process
process MEDIUM2                 ...  end process
end module
```

Protocole du bit alterné : Portes

Remarque : pour le processus `MAIN`, le protocole et ses processus internes sont cachés : de l'extérieur on ne connaît que les portes `PUT` et `GET`




```
process MAIN  [GET, PUT: C]
process TRANSMITTER[PUT: C, SDT: M, SACK: A]
                   (in var b: Bool)
process RECEIVER [GET: C, RDT: M, RACK: A]
                   (in var b: Bool)
process MEDIUM1 [SDT, RDT: M]
process MEDIUM2 [RACK, SACK: A]

channel C is      (Msg)          end channel
channel M is      (Msg, Bool)    end channel
channel A is      (Bool)         end channel
```

Protocole du bit alterné : Type Msg et canaux de communication

```
type Msg is
     msg1, msg2
end type

channel C is      (Msg)          end channel
channel M is      (Msg, Bool)    end channel
channel A is      (Bool)         end channel
```

Protocole du bit alterné : Processus MAIN

```
process MAIN [GET, PUT: C] is
hide SDT, RDT: M, RACK, SACK: A  in
    par SDT, RDT, RACK, SACK in
        par
            TRANSMITTER [PUT, SDT, SACK]
                        (false of Bool)
        ||
            RECEIVER [GET, RDT, RACK]
                        (false of Bool)
        end par
    ||
        par
            MEDIUM1 [SDT, RDT]
        ||
            MEDIUM2 [RACK, SACK]
        end par
    end par
end hide
end process
```

Protocole du bit alterné : TRANSMITTER

```
process TRANSMITTER[PUT:C, SDT:M, SACK:A](in var b:Bool)is
var m: Msg in
    loop
        PUT (?m);
        loop L in
            SDT (!m, !b);
            select
                SACK (!b);
                b := not(b);
                break L
            []
                SACK (!not(b))
            []
                i
            end select
        end loop
    end loop
end var
end process
```

Protocole du bit alterné : RECEIVER

```
process RECEIVER[GET:C, RDT:M, RACK:A](in var b: Bool)is
var m: Msg in
    loop
        select
            RDT (?m, !b);
            GET (!m);
            RACK (!b);
            b := not(b)
        []
            RDT (?any Msg, !not(b));
            RACK (!not(b))
        []
            i;
            RACK (!not(b))
        end select
    end loop
end var
end process
```

Protocole du bit alterné : MEDIUM1

```
process MEDIUM1 [SDT, RDT: M] is
    var m: Msg, b: Bool in
        loop
            SDT (?m, ?b);
            select
                RDT (!m, !b)
            []
                i
            end select
        end loop
    end var
end process
```

Protocole du bit alterné : MEDIUM2

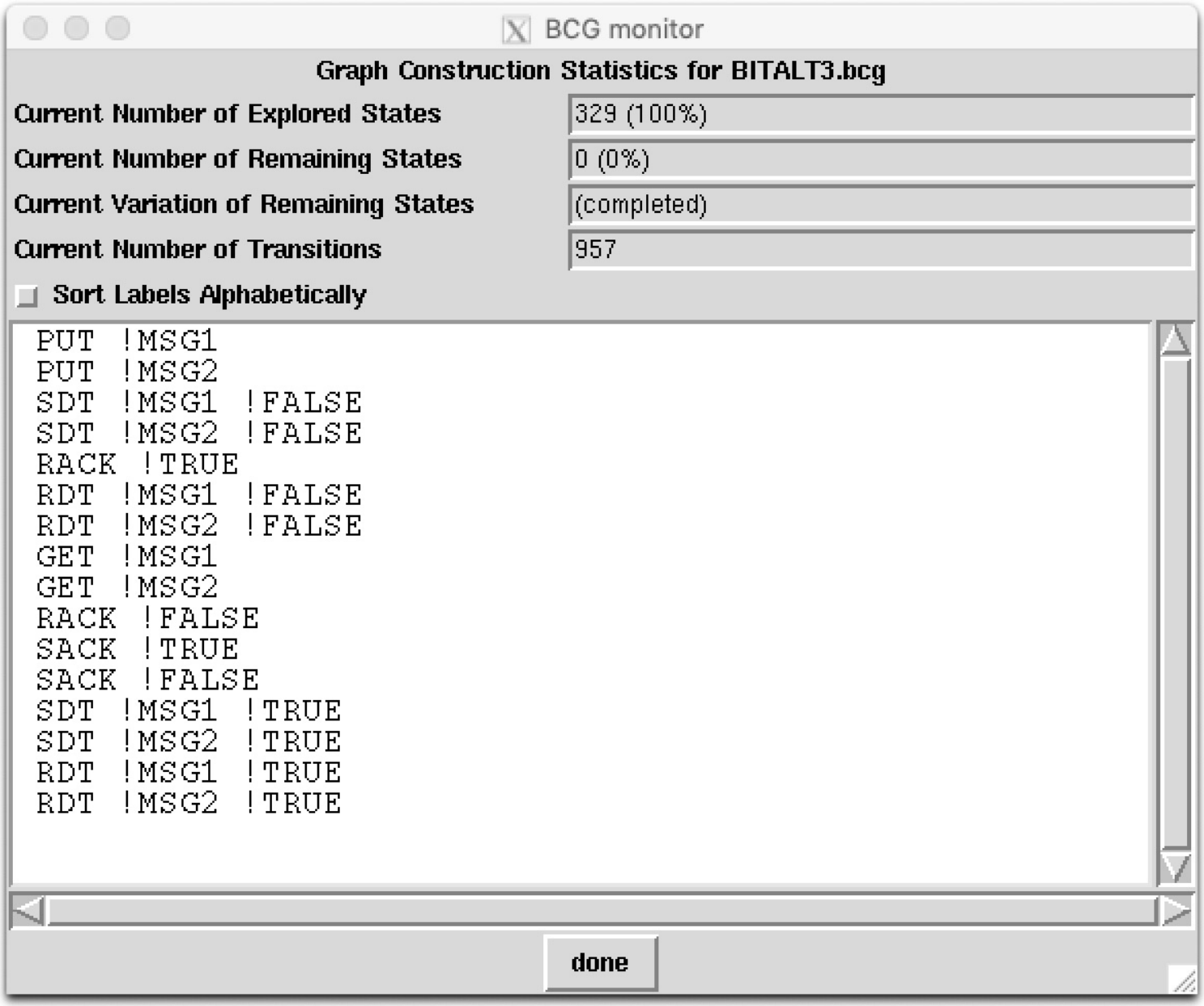
```
process MEDIUM2 [RACK, SACK: A] is
    var b: Bool in
        loop
            RACK (?b) ;
            select
                SACK (!b)
            []
                i
            end select
        end loop
    end var
end process
```

Pour tester le protocole du bit alterné

On "ouvre" le processus `MAIN` :

```
process MAIN [GET, PUT: C, SDT, RDT: M, RACK, SACK: A] is
-- process MAIN [GET, PUT: C] is
    --  hide SDT, RDT: M, RACK, SACK: A  in
        ...
    --  end hide
end process
```

Test du protocole du bit alterné



CADP: Construction et Analyse de Processus Distribués

CADP :

- Boîte à outils pour la compilation, la vérification et la validation de programmes Lotos
- Développée à l'Inria Rhônes–Alpes
- Elle intègre tout un ensemble d'outils, nous en verrons quelqu'uns
- Deux catégories de composants =
  - les compilateurs
  - les vérificateurs

Utilisation de CADP: l'outil eucalyptus **xeuca**

Il faut être sous Linux
Appeler l'outil : **xeuca &**
Choisir le répertoire courant où se trouve la spécification LOTOS NT (fichier.lnt)
Se positionner sur l'icône du fichier à étudier et clic droit pour avoir le menu
Rq : le menu est différent selon le type de fichier

Utilisation de CADP: l'outil eucalyptus **xeuca**

Générer le système de transition étiqueté (labelled transition system)

- en utilisant BCG pour pouvoir visualiser le graphe, l'éxecuter, etc... Génère fichier.bcg, fichier.h et fichier.net
- en utilisant aldebaran pour utiliser les outils de simulation ou de vérification Génère fichier.aut, fichier.h et fichier.net

Puis utiliser les outils proposés :

- Visualize -> draw pour voir le graphe
- Execute -> 3 choix pour faire une simulation ... à explorer !
- Find deadlocks pour trouver les points d'interblocage
- Find livelocks pour trouver les points de non vivacité
- Compare pour comparer des spécification
- Reduce pour réduire la taille du graphe généré
- ...

Vérification formelle: quelles propriétés veut-on étudier ?

- **Absence de blocage** (deadlock freeness): le système ne se trouve jamais dans une situation où il ne peut plus progresser, dans laquelle il ne peut plus faire ce qu'il doit faire
- **Propriété de vivacité** (liveness) :
  "quelque chose de bien finira par avoir lieu"
  Propriété de vivacité conditionnelle:
  "sous certaines conditions, certaines actions sont possibles"

Vérification formelle: quelles propriétés veut-on étudier ? (suite)

- **Propriété de sûreté** (safety):
  “quelque chose de mauvais ne se produit jamais”
  Propriété de sûreté conditionnelle:
  “sous certaines conditions, certaines actions ne sont pas offertes”
  Propriété invariante :
  “une condition donnée est satisfaite pour tout état atteignable”
- **Propriété d’atteignabilité** (reachability):
  “une situation donnée peut-être atteinte”

Mais attention: nous travaillons sous l’**hypothèse d’équité** :
“Lorsque 2 actions sont offertes infiniment souvent, un système ne peut pas choisir éternellement une action aux dépens de l’autre”.

Sémantique

- **Sémantique opérationnelle**: en terme de diagrammes de transition avec des actions (visibles ou invisibles) permettant de passer d'un état à un autre.
- **Sémantique dénotationnelle**: définit le comportement d'une spécification LOTOS en terme de traces.
- **Sémantique algébrique**: un ensemble de règles algébriques sont définies à partir desquelles des propriétés, telles que l'équivalence entre processus, peuvent être dérivées.

Nous allons étudier la sémantique opérationnelle utilisée par CADP.

Avant d'aller plus loin : retour sur les portes prédéfinies : porte **i**

Il y a 3 portes prédéfinies

- La porte interne (ou invisible) : **i**
  Action invisible souvent notée ρ
  Porte normale (pas une exception), mais elle ne peut pas être utilisée pour la communication ou la synchronisation.
  Le canal de cette porte est "none".

Avant d'aller plus loin : retour sur les portes prédéfinies : portes δ et ξ

- La porte de continuation (ou de terminaison réussie) est notée δ
  Cette porte n'apparaît pas explicitement dans la syntaxe de Lotos et Lnt, mais apparaît dans la sémantique dynamique des processus Lotos et Lnt.
  Cette porte apparaît chaque fois qu'un comportement se termine, cédant le contrôle à un autre comportement qui sera exécuté en séquence. Par exemple, le comportement "null" de Lnt génère une action sur la porte δ . Le canal de cette porte est "any".
- L'exception anonyme est notée "unexpected" dans la syntaxe concrète (identifiant de porte prédéfini) et ξ dans la sémantique. Cette porte est déclarée implicitement au niveau supérieur et ne devrait donc jamais apparaître dans les déclarations de portes.Le canal de cette porte est "none".

Sémantique opérationnelle: relation de transition

La sémantique opérationnelle spécifie les règles d'évolution des programmes.
Elle est définie par la relation de transition : $B_1 \longrightarrow^L B_2$
où
B_1 et B_2 sont des expressions de comportement
L représente une action $GV_1, \ldots V_n$ où G peut être l'action interne i ou l'action d'arrêt δ

$B_1 \longrightarrow^L B_2$
signifie : on peut passer du comportement B_1 au comportement B_2 en effectuant l'action L
Elle est définie par récurrence sur la structure syntaxique des comportements.

Sémantique opérationnelle: Règles d'inférence

La règle d'inférence $\frac{Premisse_1 ... Premisse_n}{Conclusion}$ où

$Premisse_1 \wedge ... \wedge Premisse_n$ et $Conclusion$ sont des Prédicats signifie :

$Premisse_1 ... Premisse_n$ implique $Conclusion$

On peut écrire $\frac{\frac{P}{Q}}{R}$

qui signifie : si P est vrai alors Q aussi et donc R est aussi vrai

Équivalence de Processus $\frac{P \longrightarrow^a P'}{N \longrightarrow^a P'} \ [N \equiv P]$

Signifie : lorsque les processus N et P sont équivalents,
si P devient P' par l'action a,
alors N devient aussi P' par l'action a

Sémantique opérationnelle: règles d'inférence

Pas de règle pour STOP (pas de transition)

Les règles d'inférence portent sur des couples $< B, \sigma >$ *où B est un comportement et* σ *un état mémoire.*

L'état initial d'un module B_0 *est* $< B_0, [] >$

Une transition est notée : $< B, \sigma > \longrightarrow^{a} < B', \sigma' >$

Sémantique opérationnelle: exemples de règles d'inférence

Composition séquentielle :

$$\frac{<B_1,\sigma> \longrightarrow^a <B'_1,\sigma'>}{<B_1;\ B_2,\sigma> \longrightarrow^a <B'_1;\ B_2,\sigma'>}$$

Choix non déterministe :

$$\frac{<B_i,\sigma> \longrightarrow^a <B'_i,\sigma'>}{<\textit{select } B_1 [] ... [] B_n \textit{ end select},\sigma> \longrightarrow^a <B'_i,\sigma'>} \quad i \in \{1..n\}$$

... ainsi sont définies les sémantiques de LNT et de LOTOS

Sémantique opérationnelle: utilisation des règles d'inférence

- Les règles d'inférence peuvent être utilisées pour exécuter une spécification LOTOS ou LNT en la transformant
- Le graphe étiqueté du système est obtenu à partir de l'état initial par application des règles d'inférence possibles
- De ce fait, le graphe comporte tous les états atteignables
- L'interblocage ou deadlock est le cas où aucune règle d'inférence ne peut être appliquée
- Interblocage et stop sont exactement la même chose en LOTOS et en LNT : Il n'y a pas de règles d'inférence pour stop

Sémantique opérationnelle : définition du graphe états/transitions

Graphe états/transitions ou LTS (Labelled Transition System) est défini par : (Σ, σ_0, A, T)

- Σ : l'ensemble des états
- $\sigma_0 \in \Sigma$: l'état initial
- A : l'union des actions (ou portes) des processus du système.
- $T \in \Sigma * A * \Sigma$: la relation de transitions entre états de Σ, notées $\rightarrow^a$ étiquetées par les actions $a \in A$ entre les états de Σ

Tous les états sont supposés être atteignables à partir de l'état initial.
i est l'action invisible : $i \in A$
On note T^* la fermeture réflexive et transitive de T

Deadlock et Livelock

Définition: Un system concurrent est dit être en **deadlock** si aucun des composants (processus) ne peut faire d'action, parce qu'il est en train d'attendre pour communiquer avec d'autres.

Définition: Un system concurrent est dit être en **livelock** si un réseau communique indéfiniment de manière interne sans que ses composants communiquent avec l'extérieur.

Exemples d'interblocage

Voir l'exemple du protocole du bit alterné :

- si TRANSMITTER et RECEIVER ne sont pas initialisés de la même manière
- si TRANSMITTER ne renvoie pas le message au bout d'un laps de temps

Le parcours exhaustif du graphe permet de détecter les interblocages

Relations d'équivalence : relations de bissimulation

- Bissimulation forte
- Bissimulation Observationnelle
- Bissimulation *Tau*

Relation de Forte Bissimilation

Soient deux spécifications $S1$ et $S2$,
$S1 = (\Sigma_1, \sigma_{01}, A_1, T_1)$ et $S2 = (\Sigma_2, \sigma_{02}, A_2, T_2)$

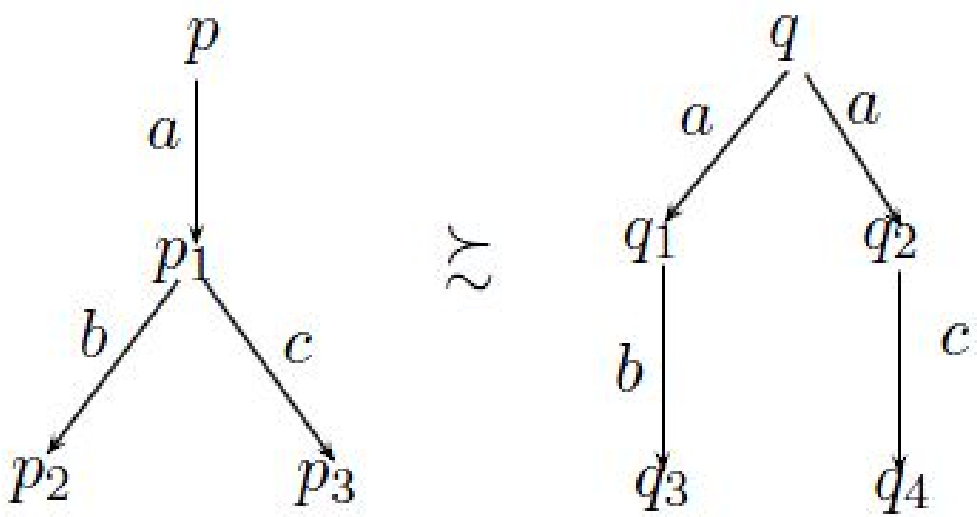
S_1 et S_2 sont **fortement bissimulables** ssi
il existe une relation de bissimulation forte $\equiv$ entre les états de S_1 et S_2 telle que

- $\sigma_{01} \equiv \sigma_{02}$
- $\forall\, \sigma_1 \in \Sigma_1,\ \sigma_2 \in \Sigma_2$.

$$\sigma_1 \equiv \sigma_2 \Rightarrow \begin{cases} \forall (\sigma_1 \rightarrow^{a1} \sigma_1') \in T_1 \ \exists (\sigma_2 \rightarrow^{a2} \sigma_2') \in T_2 \,.\, \sigma_1' \equiv \sigma_2' \\ \forall (\sigma_2 \rightarrow^{a2} \sigma_2') \in T_2 \ \exists (\sigma_1 \rightarrow^{a1} \sigma_1') \in T_1 \,.\, \sigma_1' \equiv \sigma_2' \end{cases}$$

Remarque: $a1$ et $a2$ peuvent être invisibles.

Relation de Forte Bissimilation



Une relation S est une simulation forte si p S q entraîne pour tout p' tel que p $\rightarrow^a$ p', il existe q' tel que q $\rightarrow^a$ q' et p' S q'

Relation de Bissimulation Observationnelle

Soient deux spécifications $S1$ et $S2$,
$S1 = (\Sigma_1, \sigma_{01}, A_1, T_1)$ et $S2 = (\Sigma_2, \sigma_{02}, A_2, T_2)$

S_1 et S_2 sont **observationnellement bissimulables** ssi
il existe une relation $\equiv_{obs}$ de bissimulation observationnelle entre les états de S_1 et S_2 telle que

- $\sigma_{01} \equiv_{obs} \sigma_{02}$
- $\forall \sigma_1 \in \Sigma_1,\ \sigma_2 \in \Sigma_2 \ .\ \sigma_1 \equiv_{obs} \sigma_2 \Rightarrow$

$$\begin{cases} \forall (\sigma_1 \rightarrow^{i} \sigma_1') \in T_1 \ \exists (\sigma_2 \rightarrow^{i^*} \sigma_2') \in T_2^* \ .\ \sigma_1' \equiv_{obs} \sigma_2' \\ \forall (\sigma_1 \rightarrow^{a} \sigma_1') \in T_1[a \neq i] \ \exists (\sigma_2 \rightarrow^{i^*.a.i^*} \sigma_2') \in T_2^* \ .\ \sigma_1' \equiv_{obs} \sigma_2' \\ \forall (\sigma_2 \rightarrow^{i} \sigma_2') \in T_2 \ \exists (\sigma_1 \rightarrow^{i^*} \sigma_1') \in T_1^* \ .\ \sigma_1' \equiv_{obs} \sigma_2' \ \bigstar \\ \forall (\sigma_2 \rightarrow^{a} \sigma_2') \in T_2[a \neq i] \ \exists (\sigma_1 \rightarrow^{i^*.a.i^*} \sigma_1') \in T_1^* \ .\ \sigma_1' \equiv_{obs} \sigma_2' \ \bigstar \end{cases}$$

Sans les 2 dernières propriétés (notées $\bigstar$) on a : $S_1 \geq_{obs} S_2$

Relation de Bissimulation *Tau*

Soient deux spécifications $S1$ et $S2$,
$S1 = (\Sigma_1, \sigma_{01}, A_1, T_1)$ et $S2 = (\Sigma_2, \sigma_{02}, A_2, T_2)$

S_1 et S_2 sont *Tau* **bissimulables** ssi
il existe une relation $\equiv_{tau}$ de bissimulation *Tau* entre les états de S_1 et S_2 telle que

- $\sigma_{01} \equiv_{tau} \sigma_{02}$
- $\forall\, \sigma_1 \in \Sigma_1,\ \sigma_2 \in \Sigma_2 \ .\ \sigma_1 \equiv_{tau} \sigma_2 \Rightarrow$
  $\begin{cases} \forall (\sigma_1 \rightarrow^{i^* a} \sigma_1') \in T_1[a \neq i]\ \exists (\sigma_2 \rightarrow^{i^* . a} \sigma_2') \in T_2^* \ .\ \sigma_1' \equiv_{tau} \sigma_2' \\ \forall (\sigma_2 \rightarrow^{i^* a} \sigma_2') \in T_2[a \neq i]\ \exists (\sigma_1 \rightarrow^{i^* . a} \sigma_1') \in T_1^* \ .\ \sigma_1' \equiv_{tau} \sigma_2' \ \bigstar \end{cases}$

Sans la dernière propriété (notée $\bigstar$) on a : $S_1 \geq_{Tau} S_2$

Relations d'équivalence : relations de bissimulation

- **Bissimulation forte** :
  les arcs étiquetés par *i* sont traités comme les autres
- **Bissimulation Observationnelle**:
  au lieu de $\rightarrow^a$ on a $\rightarrow^{i^*.a.i^*}$ et au lieu de $\rightarrow^i$ on a $\rightarrow^{i^*}$
- **Bissimulation** *Tau*:
  au lieu de $\rightarrow^a$ on a $\rightarrow^{i^*.a}$

Théorème :
La Bissimulation forte implique la Bissimulation *Tau* qui implique la Bissimulation Observationnelle

Bissimulation forte $\Rightarrow$ Bissimulation *Tau* $\Rightarrow$ Bissimulation Observationnelle

Traces

- **Traces** : séquence finie d'actions observables à partir de l'état initial
- **Sémantique de Traces** : $Tr(p)$ est l'ensemble des traces à partir de l'état initial de p
- **Equivalence de Traces** : 2 processus p et q sont en équivalence de traces si $Tr(p) = Tr(q)$
- La bissimulation forte implique l'équivalence de traces, mais pas l'inverse.

Exemple du protocole de bit alterné

On introduit la spécification simplifiée qui correspond au comportement observable souhaité :

```
module COMMUNICATION is
type Msg is msg1, msg2 with "eq", "ne" end type
channel C is (Msg) end channel

process MAIN [GET, PUT: C] is
    var m: Msg in
        loop
            PUT (?m);    -- receive a message
            i;
            GET (!m)     -- delivery of message
        end loop
    end var
end process

end module
```

Exemple du protocole de bit alterné

Comparaison des spécifications COMMUNICATION et BITALT :

- COMMUNICATION $\equiv_{Obs}$ BITALT
- COMMUNICATION $\equiv_{Tau}$ BITALT

mais ce n'est pas le cas pour l'équivalence forte

Raffinement

Une spécification est une description de tous les comportements acceptables.
Une description en LNT d'un processus définit un ensemble de traces: c'est une spécification.
Un autre processus P_1 satisfera la spécification décrite par un processus P_0 si toutes ses traces sont acceptables par P_0. On dit que P_1 raffine P_0:
$P_0 \sqsubseteq P_1 = traces(P_1) \subseteq traces(P_0)$

Raffiner c'est faire des choix

Exemple:
$P_0 = a \rightarrow P_0 \mid b \rightarrow P_0$
$P_1 = a \rightarrow b \rightarrow P_1$

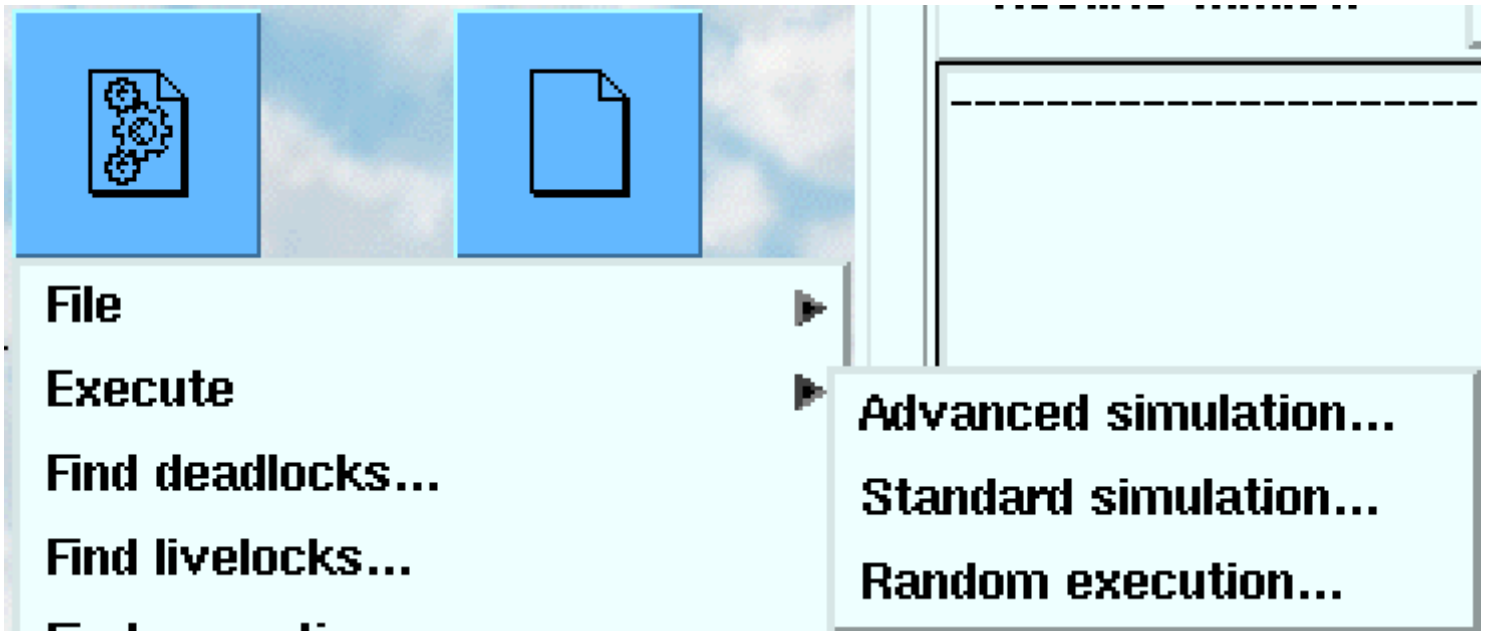
Tester un système

Le test d'un système peut se faire de plusieurs façons :

- Par simulation
- Par définition de traces de test et recherche de comportements redoutés
- Par recherche de deadlock

Tester un système : simulation

CADP 3 manières de faire des simulations



Test par exécution de traces et recherche de comportements redoutés

Il faut définir les traces à rechercher : find execution sequence

- Préparer la recherche dans un fichier .seq
- Exprimer la recherche dans le format SEQ
  - Suite d'actions : `"action !1 !FALSE"`
  - Utilisation possible de `<until>`, `<while>`, `<deadlock>`, `<any>`
  - Utilisation possible de `*, +, |, ...` comme en UNIX ou LINUX
- Remarques:
  - `<while> label` est équivalent à `label*`
  - `<until> label` est équivalent à :
    `(~ label)*`
    `label`

Recherche de deadlock et de livelock