Technique de Description Formelle (FDT)

Estelle

Un Langage de Spécification pour les Protocoles

Université François Rabelais de Tours
Faculté des Sciences et Techniques - Antenne Universitaire de Blois
Master Informatique
Option Systèmes d'Information et Analyse Décisionnelle
1ère Année

Mohamed Taghelit
taghelit@univ-tours.fr

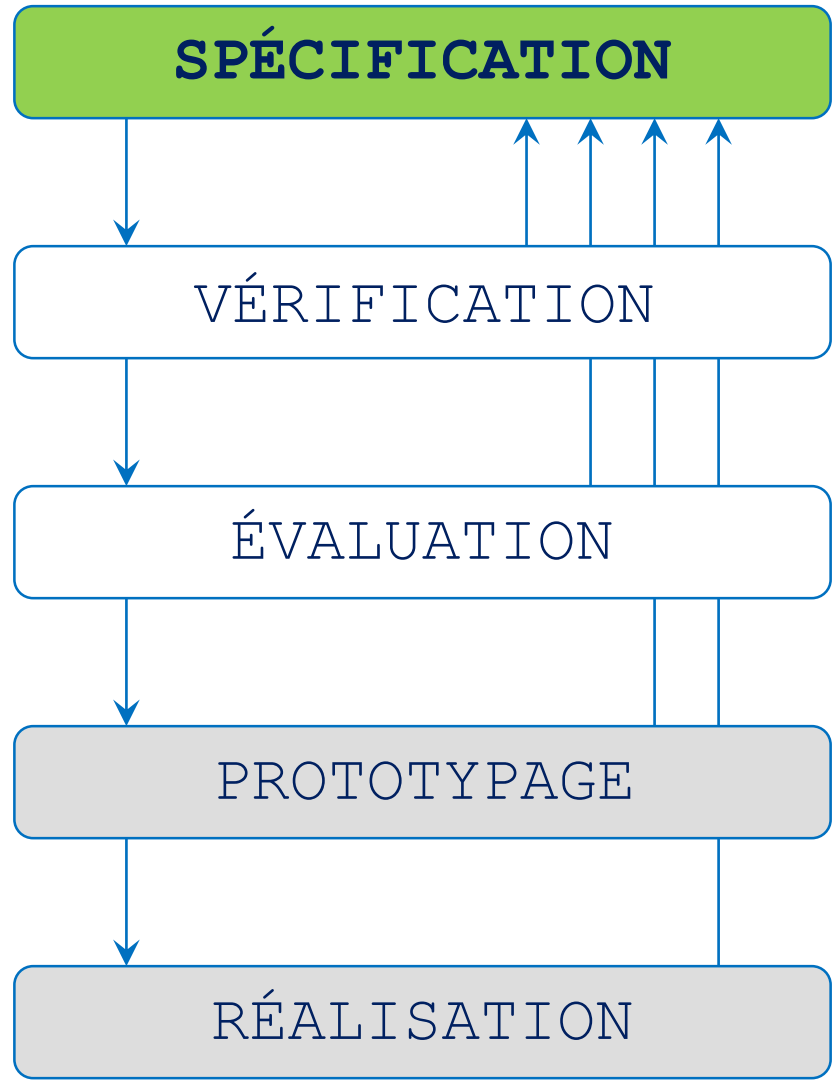
Contenu

- Introduction et Contexte
- Notions Principales
- Aspects Syntaxiques et Sémantiques
- Exemple

Positionnement – Cycle de Vie

Conception d'applications réparties ou coopératives

- Approche qui repose sur l'expérimentation une fois le système réalisé (génie logiciel)
- Approche qui consiste à développer l'application à partir d'un cycle à plusieurs étapes pour élimer le plus d'erreurs avant réalisation



Informations Générales

ISO International Standard : ISO 9074

- Début des travaux en 1981

  FDT/TC97/SC21/FDT subgroup B chaired by Richard TENNEY.

- Fin des travaux en 1989

Informations Générales

- Technique de Description Formelle (FDT) pour la spécification de systèmes concurrents distribués, et particulièrement les protocoles de communication et les services.

- Repose sur un modèle étendu de transition d'états (automates non-déterministes étendus par PASCAL).

  Modélise un système comme une structure hiérarchique d'automates communicants qui :
  - peuvent s'exécuter en parallèle, et
  - peuvent communiquer par échange de messages et partage, d'une manière restrictive, de certaines variables.

- Reflète les acquis des techniques antérieures
  - Langages PDIL, NBS, FDT-B, …

- Reflète la coopération avec le CCITT
  - Certaines notions sont communes avec le langage SDL

Informations Générales

Définition Formelle

- Syntaxe
- Sémantique

Support d'outils

- Conception incrémentale
- Validation
- implémentation

Informations Générales

Dans le même environnement, il est possible de :

- Supporter
  - Conception de haut niveau
  - Conception de bas niveau
- Valider la conception à chaque niveau
- Générer automatiquement du code
- Exécuter ce code dans un environnement donné
  - Prototypage facilité, ou
  - Implémentation finale

Notions Principales - Structure Hiérarchique

- Le système spécifié
  → un arbre de tâches

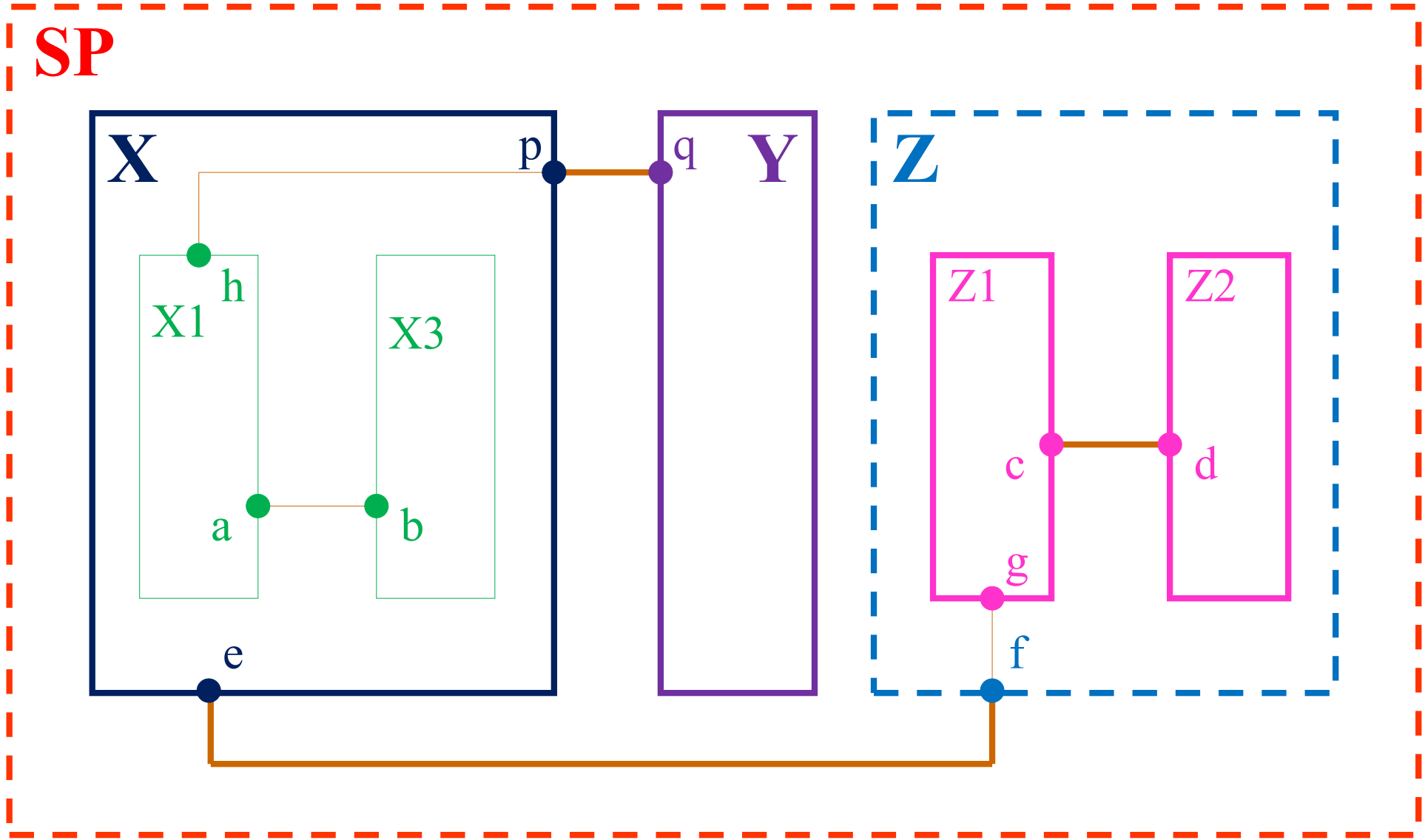
- Chaque tâche a un nombre fixe de points d'accès (entrée/sortie)
  → points d'interaction

- Des liens de communication bidirectionnels peuvent exister entre les tâches (entre leurs points d'interaction)

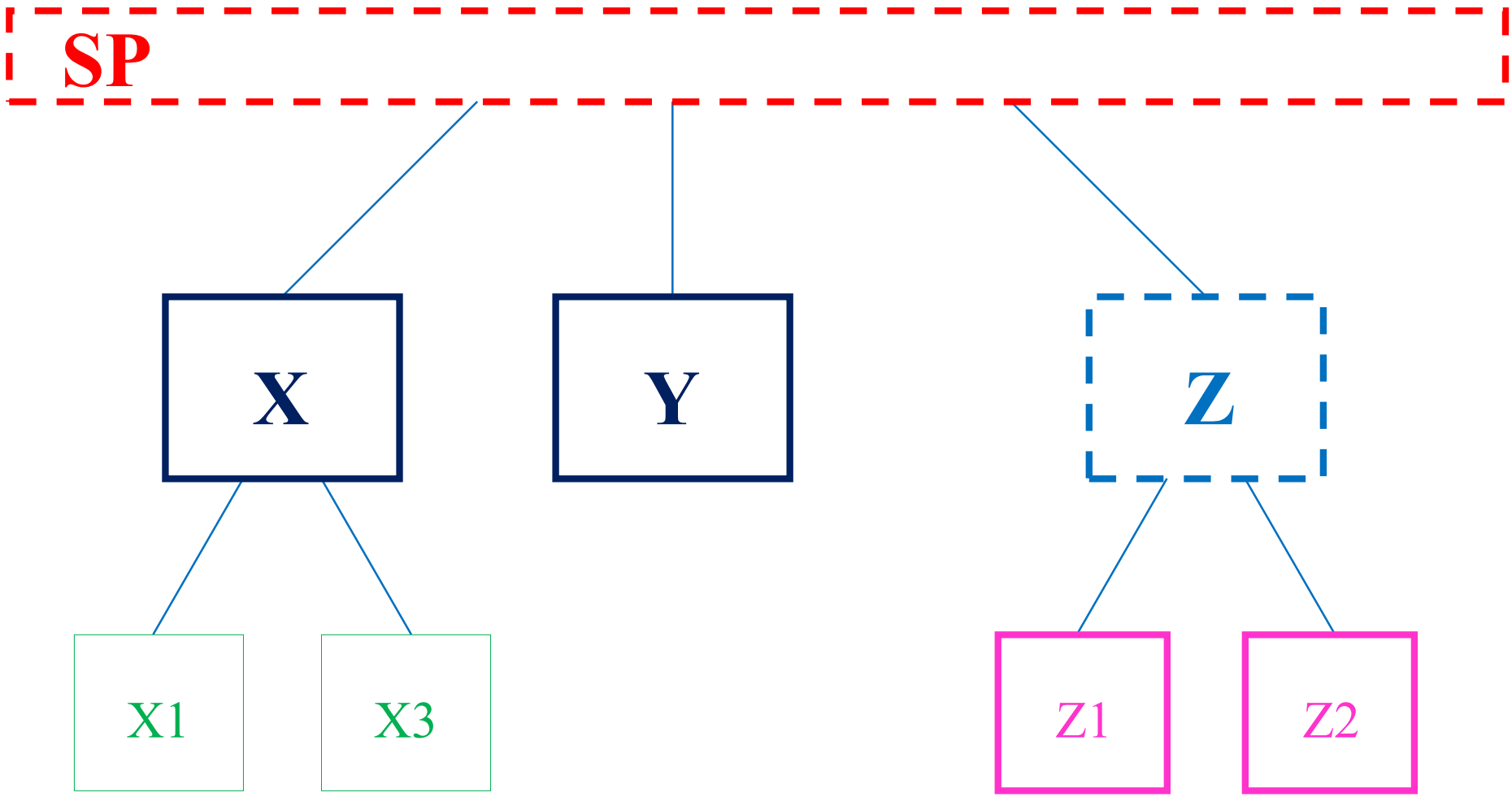
- Dans un système spécifié donné, il existe une structure fixe de sous-systèmes (sous-arbres de tâches) et de liens de communication (entre sous-systèmes)

- Dans un sous-système, les deux structures (des tâches et des liens de communication) peuvent changer dynamiquement

Notions Principales - Structure Hiérarchique



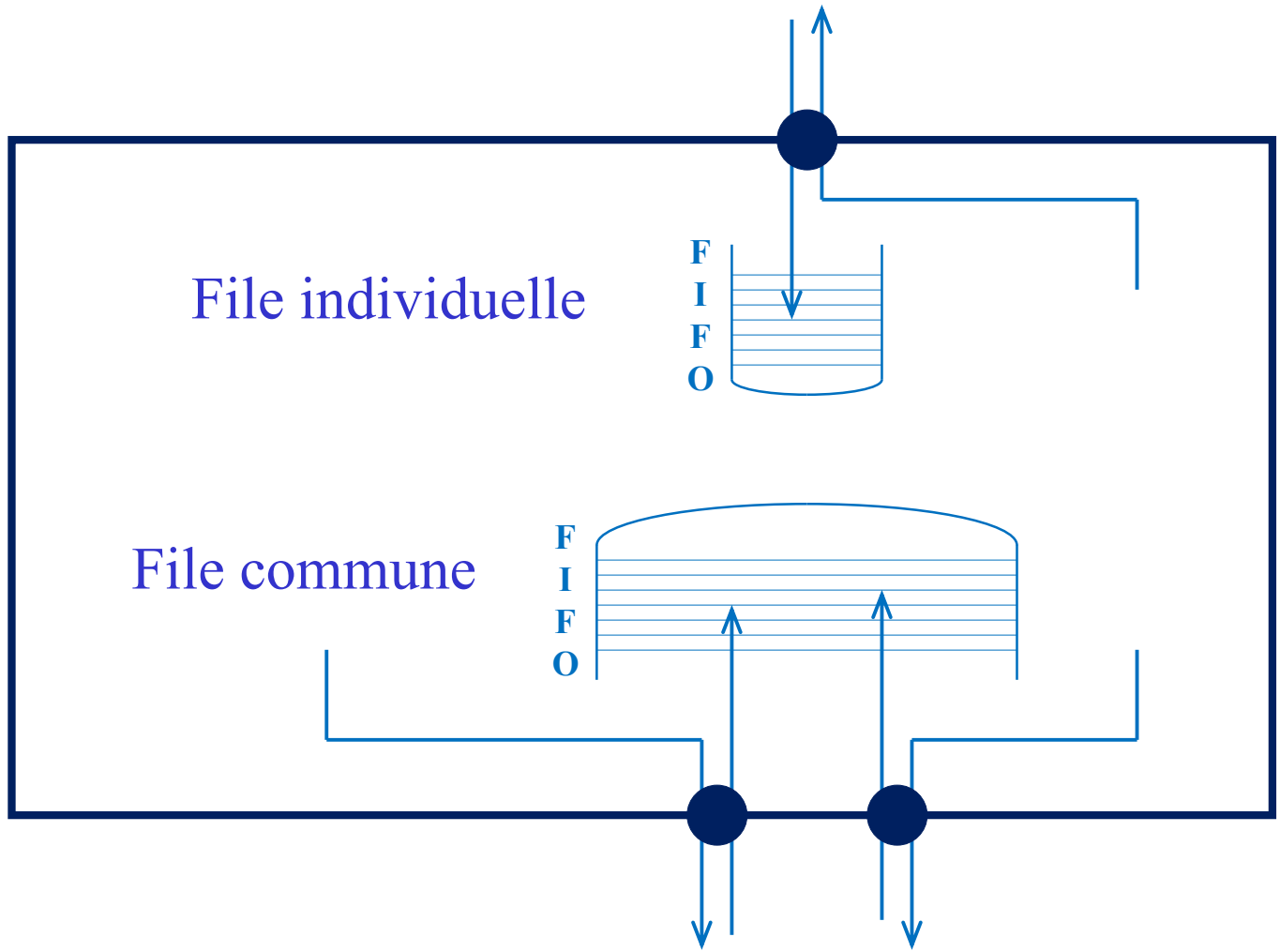
Notions Principales - Structure hiérarchique



Notions Principales - Communication

- Les tâches échangent des interactions
  - Une tâche peut émettre une interaction, à travers son point d'interaction, à une tâche au quelle elle est liée
  - Une tâche peut toujours émettre une interaction
  - Une interaction reçue par une tâche, sur son point d'interaction, est stockée dans la file associée à ce point d'interaction
  - Une file (FIFO) peut être associée à :
    - Un point d'interaction → file individuelle
    - plusieurs points d'interaction → file commune
- une tâche peut exporter des variables vers son parent qui peut les accéder (lecture et écriture)

Notions Principales - Communication



Notions Principales - Parallélisme

- Deux types de parallélisme peuvent être définis
  - Parallélisme asynchrone – uniquement entre (actions de) tâches de différents sous-systèmes
  - Parallélisme synchrone – uniquement entre (actions de) différentes tâches d'un même sous-système

    Parallélisme synchrone entre actions

    → toutes les actions doivent compléter leur exécution parallèle avant que d'autres actions puissent être exécutées en parallèle

Notions Principales - Temps

- Le temps d'exécution des actions est supposé inconnu
  → dépendant de l'implémentation

- Certaines actions peuvent être spécifiées de façon que leur exécution soit différée
  - Deux valeurs de durée (MIN et MAX) peuvent être spécifiées
  - Les valeurs de ces durées sont supposées être modifiées par un processus indépendant
  - La sémantique ESTELLE impose uniquement des contraintes faibles sur la progression du temps

!!! Le temps progresse quand l'exécution progresse !!!

Notions Principales - Typage

Tous les objets manipulés sont fortement typés

- Variables PASCAL
- Objets ESTELLE
  - Variables états (de contrôle) → type énuméré (énumération des identificateurs d'états)
  - Variables modules → type MODULE
  - Points d'interaction → type CHANNEL

Canaux de Communication

Syntaxe

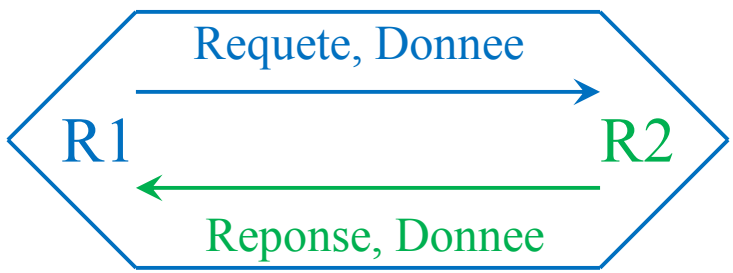
```
CHANNEL nom_canal ( role1, role2 )
        BY role1 :
                nom_interaction_1 (parametres types);
                ..........;
                ..........;
        BY role2 :
                nom_interaction_1 (parametres types);
                ..........;
                ..........;
        BY role1, role2 :
                nom_interaction_1 (parametres types);
                ..........;
                ..........;
```

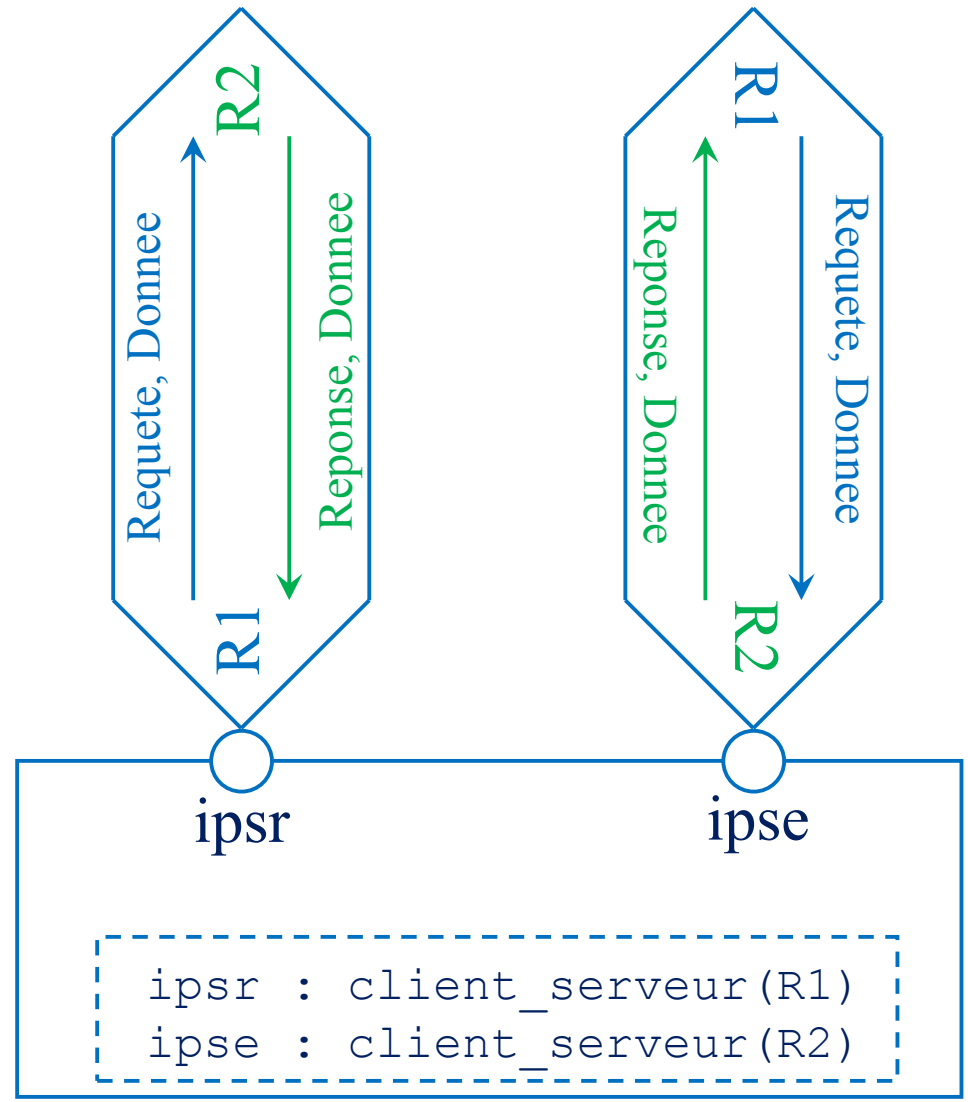
Canaux de Communication

- La définition CHANNEL détermine deux types CHANNEL
  - le type nom_canal(role1), et
  - le type nom_canal(role2)
- Un type CHANNEL définit deux ensembles d'interactions
  - celles qui peuvent être émises à travers un point d'interaction de ce type
  - celles qui peuvent être reçues à travers un point d'interaction de ce type

Canaux de Communication

```
CHANNEL client_serveur (R1, R2)
    BY R1 : Requete;
    BY R2 : Reponse (a : boolean);
    BY R1, R2 : Donnee;
```





Messages de Communication

Un rôle de canal est explicité par une liste d'identifiants de variables de messages susceptibles d'être véhiculés par le canal.

Les identifiants peuvent être :

- des messages simples
  - ⇨ m1, m2, ack, nack, jeton, ...
- des classes de messages
  - ⇨ msg (x : `INTEGER`, y : `BOOLEAN`)

Exemple de Canaux et de Messages

```
CHANNEL Canal_ABC ( role1, role2 )
        BY role1 : m1, m2, …, mi;
        BY role2 : n1, n2, …, nj;
```

Module A

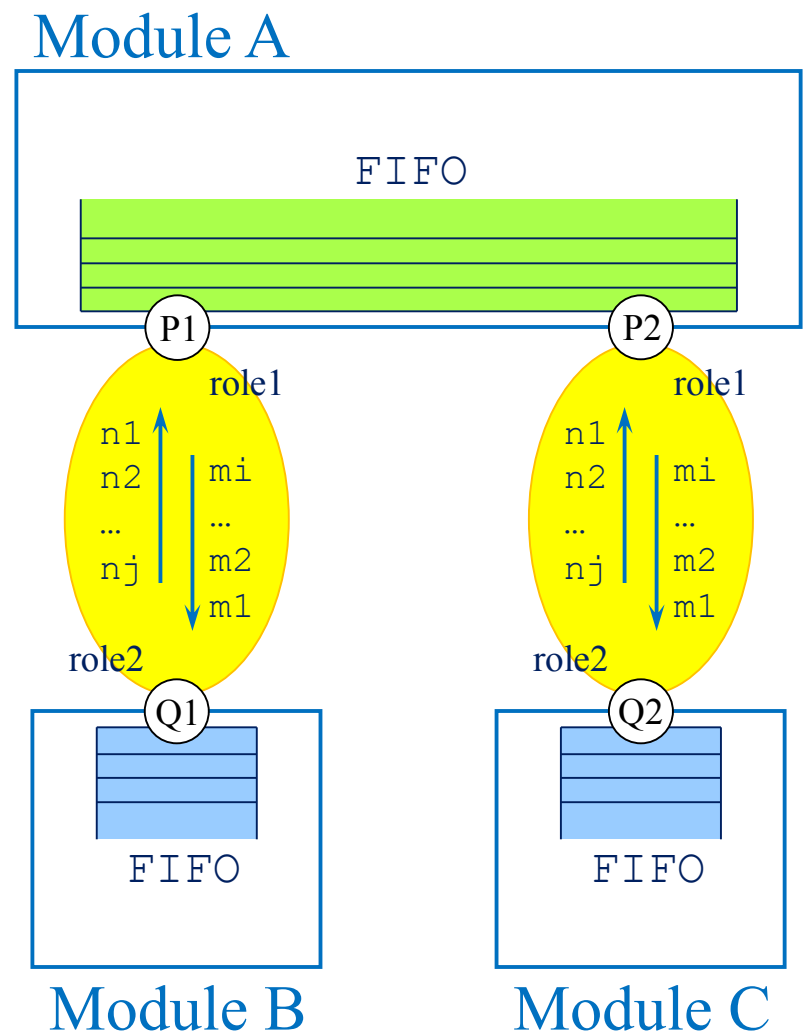
```
IP P1, P2 : Canal_ABC ( role1 ) COMMON QUEUE
```

Module B

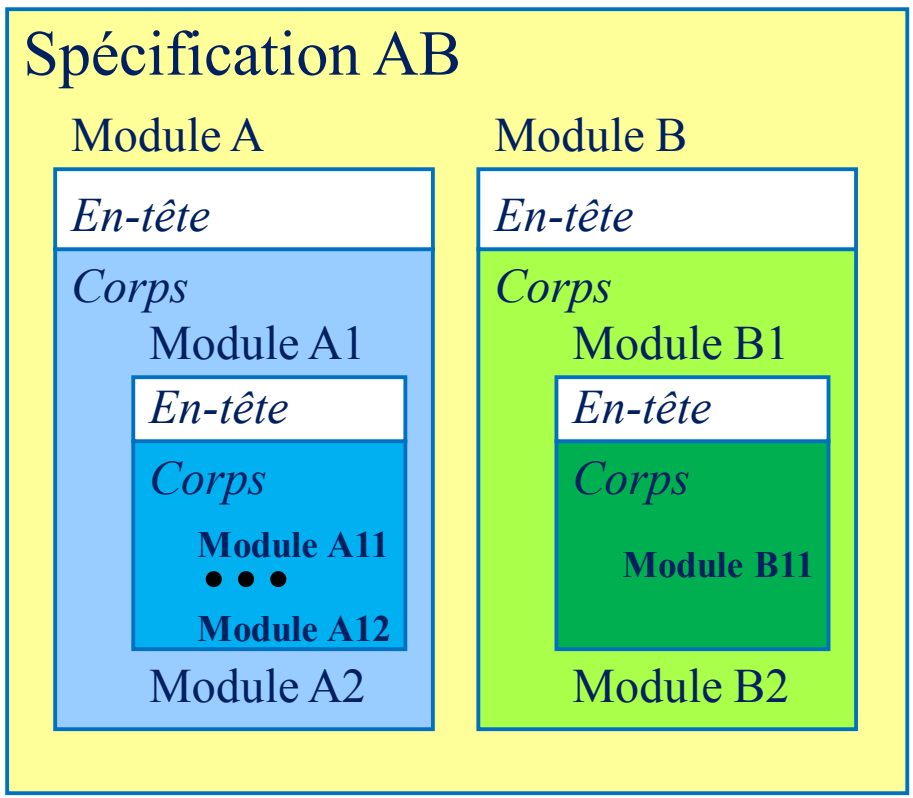
```
IP Q1 : Canal_ABC ( role2 ) INDIVIDUAL QUEUE
```

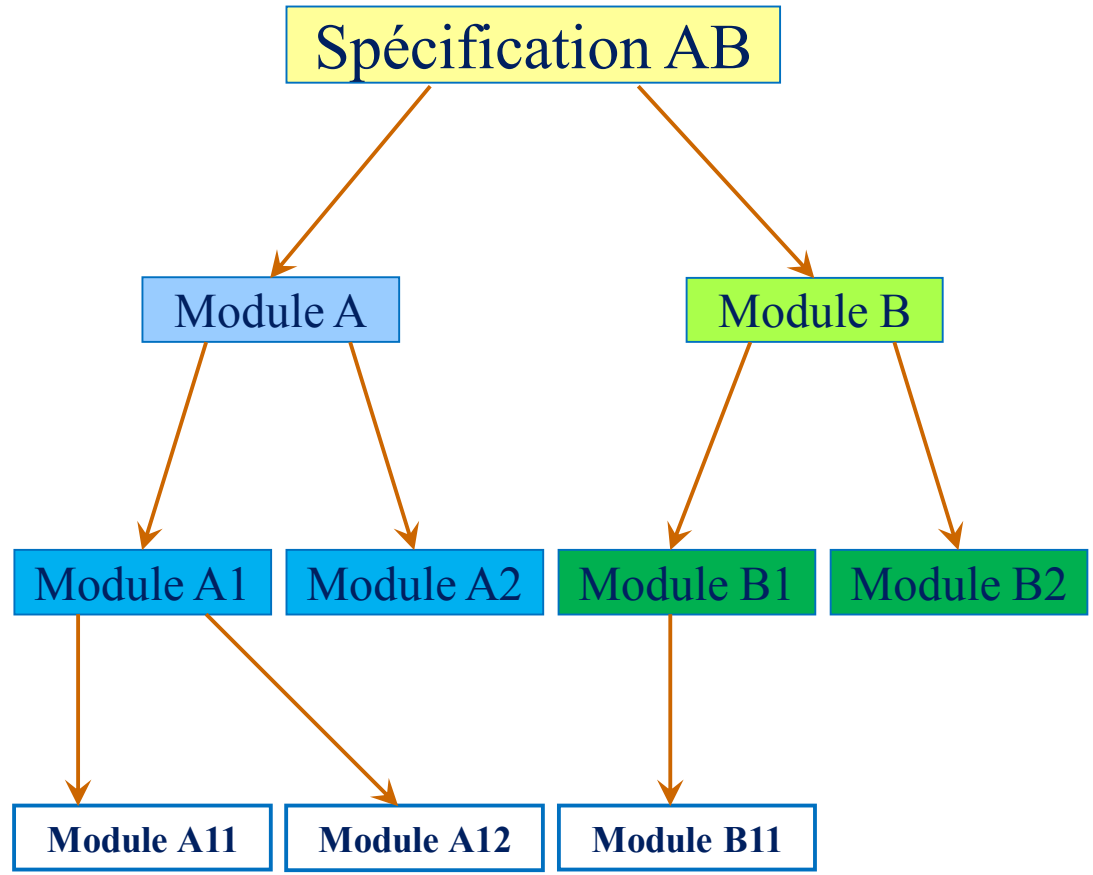
Module C

```
IP Q2 : Canal_ABC ( role2 ) INDIVIDUAL QUEUE
```

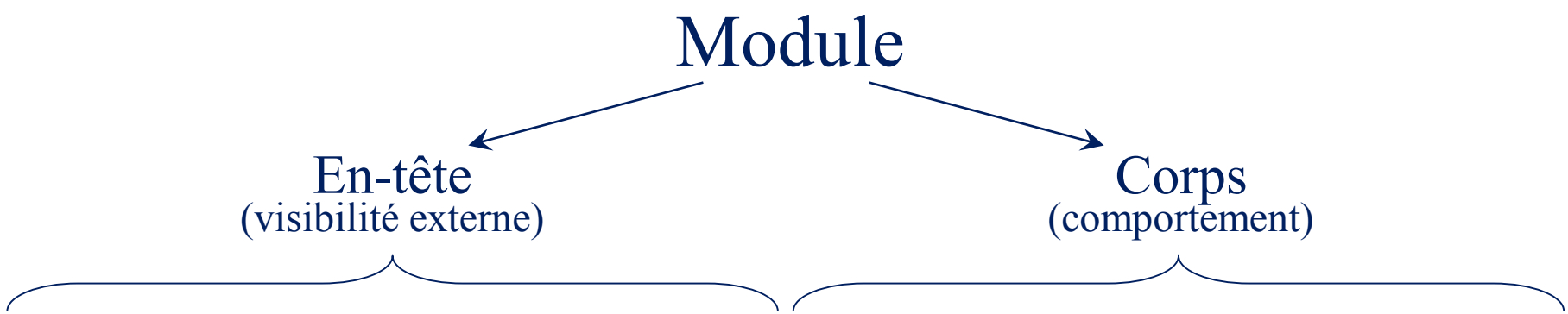


Spécification Estelle





Structuration d'un Module



- En-tête de module
  - Définition des variables exportées
  - Définition des points d'interaction externes

Syntaxe

```
MODULE nom_en-tete_module
        /* Définition des variables et
                        des Ips externes  */
END;
```

- Corps de module
  - Définition des variables et des Ips internes
  - Initialisation des variables du module
  - Description du comportement interne

Syntaxe

```
BODY nom_corps_module FOR nom_en-tete_module
        /* Définition du corps du module */
END;
```

Définition d'un En-tête de Module

- La définition d'un en-tête de module spécifie un type MODULE
- Au moins une définition de corps doit être déclarée pour chaque définition d'en-tête
- Une instance de module correspond à une tâche
  (Définition en-tête de module & UNE définition de corps de module)
- L'en-tête de module définit la visibilité externe du module
  → seuls les éléments définis dans l'en-tête seront connus et donc utilisables par les (instances de) modules frères et le module père

Syntaxe

```
MODULE nom_en-tete_module [ attribut_classe ]
    /*  Déclaration des variables  externes  */
    ▪  Les points d'interaction externes
    ▪  Les paramètres formels
    ▪  Les variables exportées
END;
```

Définition des Points d'Interaction

Un point d'interaction externe d'un module n'est visible et utilisable que par les modules frères et le module père.

Il permet l'échange de messages entre le module et son père, et entre le module et ses frères.

Syntaxe

`IP` nom_ip : **nom_canal** (**role-associe**) **strategie_stockage ;**

- **nom_canal** (**role_associe**)
  Définit les messages pouvant être émis via le point d'interaction.

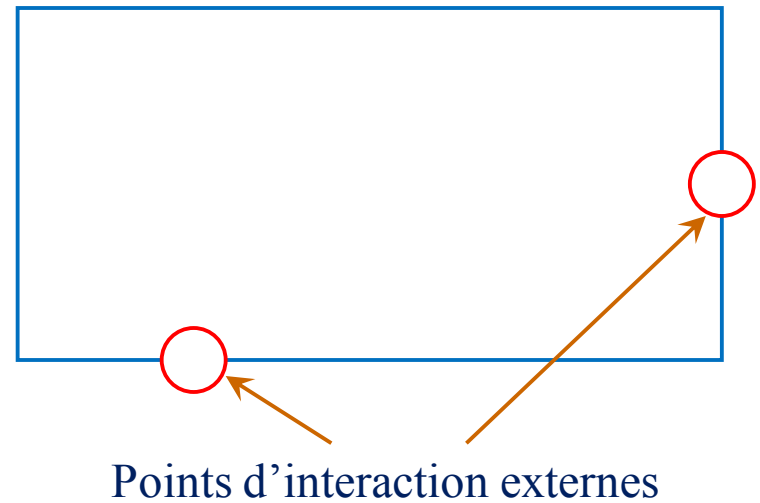
- **strategie_stockage**
  Définit la stratégie de stockage des messages dans la file associée au point d'interaction.

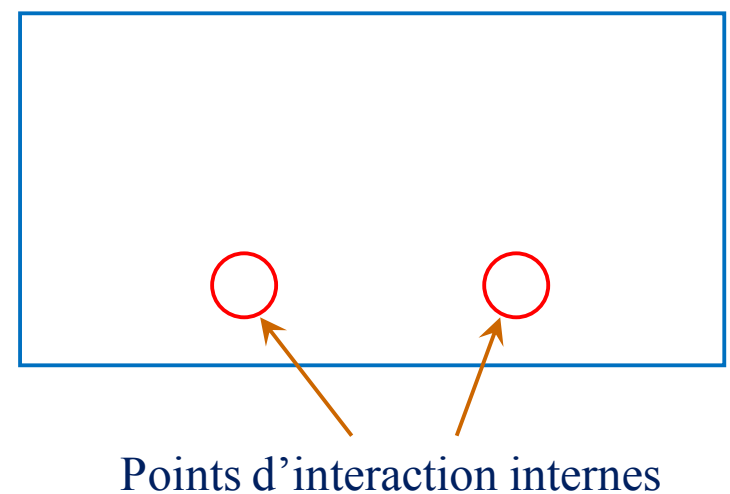
Schématisation des Points d'Interaction

Les points d'interaction peuvent être déclarés à deux endroits d'une spécification :

- En-tête du module
  ⇨ Points d'interaction externes

- Corps du module
  ⇨ Points d'interaction internes



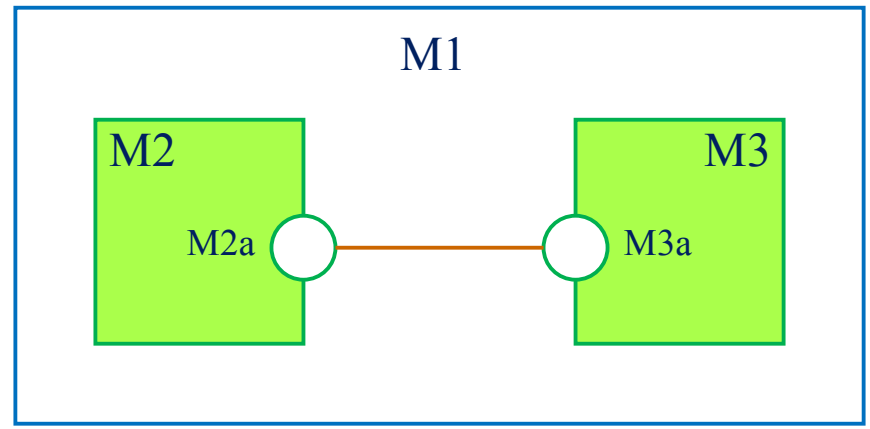


Liens entre Points d'Interaction

- Points d'interaction *connectés*
  - Deux points d'interaction externes de deux instances de modules frères

    Syntaxe

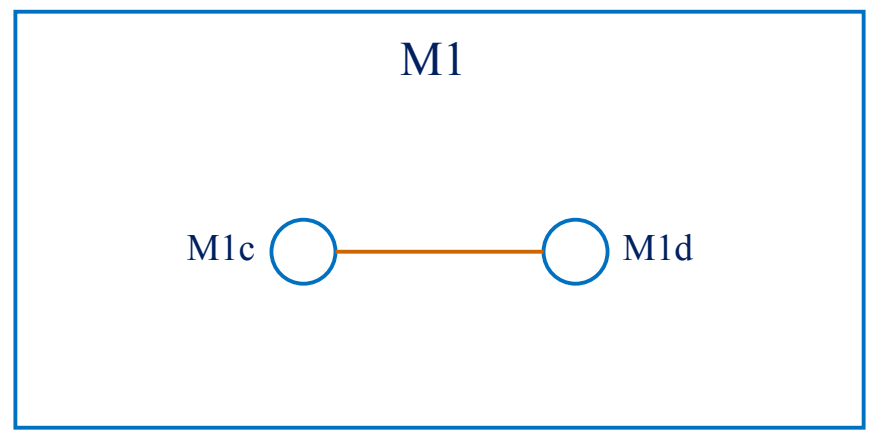
    `CONNECT` M2.M2a `TO` M3.M3a



  - Deux points d'interaction internes d 'une même instance de module

    Syntaxe

    `CONNECT` M1c `TO` M1d



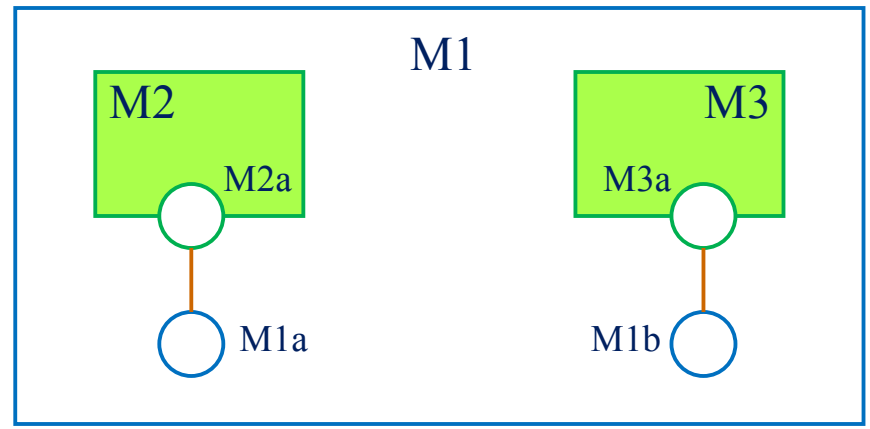
Liens entre Points d'Interaction

- Un point d'interaction interne d'une instance de module père et un point d'interaction externe d'une instance de module fils

Syntaxe

`CONNECT` M1a `TO` M2.M2a

`CONNECT` M1b `TO` M3.M3a



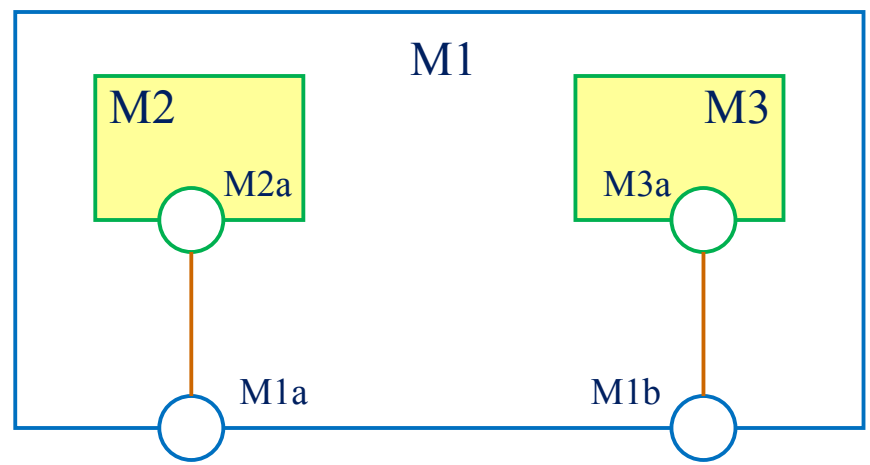
❑ Points d'interaction *attachés*

- Un point d'interaction externe d'une instance de module père et un point d'interaction externe d'une instance de module fils

Syntaxe

`ATTACH` M1a `TO` M2.M2a

`ATTACH` M1b `TO` M3.M3a



Points d'Interaction *Connectés*

Deux points d'interaction *connectés* doivent être associés à un même canal et à des rôles opposés.

⇨ échange de messages possible

Exemple

```
CHANNEL Canal_A ( rôle1, rôle2 )
...
CHANNEL Canal_B ( rôle1, rôle2 )
...
Module M1
    IP P1 : Canal_A ( rôle1 )
       P2 : Canal_B ( rôle2 )

Module M2
    IP P1 : Canal_A ( rôle1 )
       P2 : Canal_A ( rôle2 )
```

Connections possibles

~~CONNECT M1.P1 TO M2.P1~~

CONNECT M1.P1 TO M2.P2

~~CONNECT M1.P2 TO M2.P1~~

~~CONNECT M1.P2 TO M2.P2~~

Points d'Interaction *Attachés*

Deux points d'interaction *attachés* doivent être associés à un même canal et à un même rôle.

⇨ chaque point d'interaction remplace l'autre

Exemple

```
CHANNEL Canal_A ( rôle1, rôle2 )
...
CHANNEL Canal_B ( rôle1, rôle2 )
...
Module M1
    IP P1 : Canal_A ( rôle1 )
       P2 : Canal_B ( rôle2 )

Module M2
    IP P1 : Canal_A ( rôle1 )
       P2 : Canal_A ( rôle2 )
```

Attachements possibles

ATTACH M1.P1 TO M2.P1

~~**ATTACH M1.P1 TO M2.P2**~~

~~**ATTACH M1.P2 TO M2.P1**~~

~~**ATTACH M1.P2 TO M2.P2**~~

Variables *Exportées*

Variables partagées entre un module père et une instance de module fils

⇨ Déclaration dans l'en-tête du module fils
visibilité externe

⇨ Communication entre père et fils
lues et écrites par le module père et le module fils
échange d 'information rapide et privilégié

⇨ Inexistence de conflits d'accès
principe de priorité d 'exécution père/fils

Syntaxe

```
/* Déclaration */
    EXPORT ID_Var_Exportee : Type_Var;
/*  Référence */
    ID_Var_Exportee
    ID_Var_Module.ID_Var_Exportee
```

Exemple

```
MODULE id_en-tete;
    EXPORT X, Y : INTEGER;
END;

BODY id_corps FOR id_en-tete;
    ...
    IF ( X = Y ) THEN ...
    ...
END;
```

Paramètres Formels

On peut définir une liste de paramètres associée à l'en-tête d'un module.

⇨ Ces paramètres seront transmis aux instances du module lors de leur création

Syntaxe

MODULE id_en-tete [att_classe] (par_1, …, par_N) ;

Exemple

```
MODULE id_en-tete ( ID : INTEGER;
    Tab : ARRAY [ 1..10 ] OF INETEGER );
    …
END;


/* Corps du module père */
…
FOR i := 1..10 DO vecteur[i] := 0;
i := 3;
INIT X WITH id_corps (i, vecteur) ;
…
```

Corps d'un Module

Ossature d'un corps de module

```
BODY nom_corps FOR nom_en-tete
        /* Partie déclaration */
INITIALIZE
    BEGIN
        /* Partie initialisation du module */
    END;
        /* Partie définition du comportement interne */
END;
```

Partie Déclaration

Corps d'un module

La partie déclaration d'un corps de module comprend :

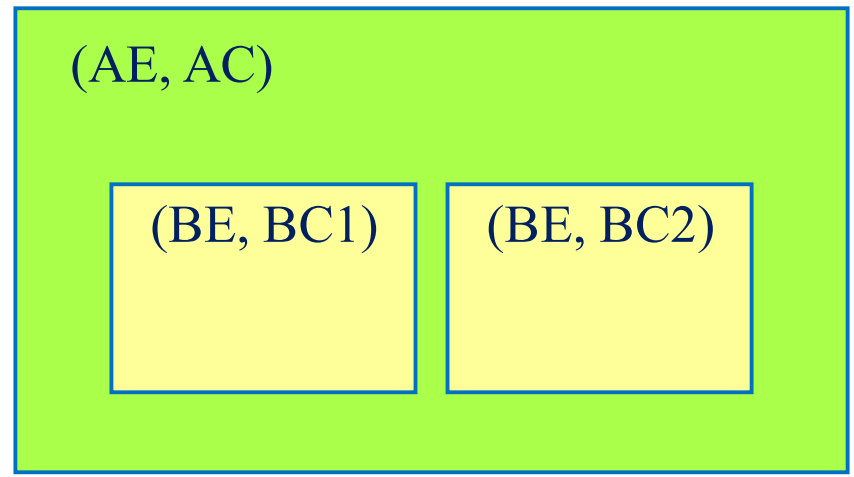
- déclaration écrite en langage PASCAL,
- déclaration d'objets spécifiques à Estelle
  - les canaux et les messages de communication
  - les modules fils
  - les variables de module
  - les états du module
  - les points d'interaction internes

Les modules fils

Parti déclaration - Corps d'un module

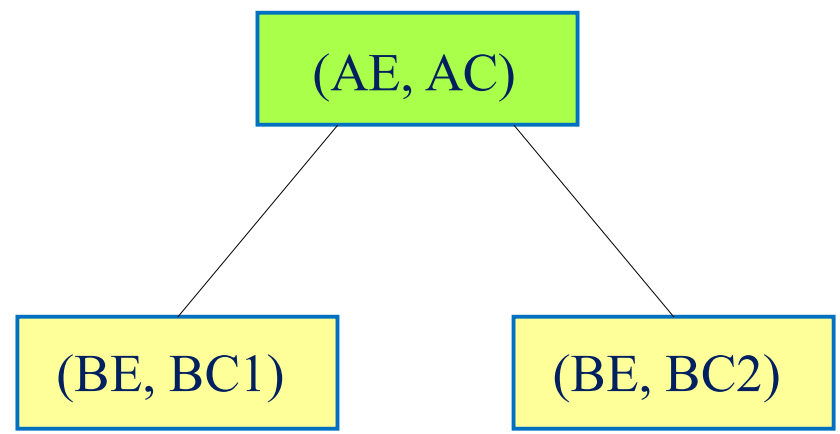
La définition d'un corps de module peut contenir la spécification d'autres modules
⇨ Structuration hiérarchique des modules

La définition d'un module X à l'intérieur d'un module A explicite la relation père/fils entre X et A (X est fils de A).



Exemple

```
MODULE AE … END;

BODY AC FOR AE;
        MODULE BE … END;

        BODY BC1 FOR BE; … END;
        BODY BC2 FOR BE; … END;
END;
```



Les variables de Modules

Parti déclaration - Corps d'un module

Variable de module ⇨ variable d'en-tête de module

⇨ instance d'en-tête de module

Dans la partie *Initialisation*, chaque variable de module, définie dans la partie *Déclaration*, sera initialisée avec un corps déjà spécifié.

⇨ création d'instance de module

Syntaxe

```
MODVAR nom_instance_module : nom_en-tete;
```

Exemple

```
MODULE A ... END;

...
MODVAR X, Y, Z  : A;
```

Dans la partie *Initialisation*, des corps seront également attribués à ces variables de modules : munis d'un en-tête et d'un corps, X, Y et Z constitueront des instances de modules.

La déclaration de variables de modules n'induit pas la création (d'instances) de modules.

Les États et Ensembles d'États

Parti déclaration - Corps d'un module

Le comportement interne d'un module (ou instance de module) est défini en terme d'automate dont les états sont déclarés ici par l'énumération de leur identifiant.

Syntaxe

`STATE` $etat_0$`,` $etat_1$`, ...,` $etat_N$`;`

On peut également référencer des groupes de variables qui sont alors déclarés comme ensemble d'états.

Syntaxe

`STATESET` nom_ens_etats = `(`$etat_0$`,` $etat_1$`, ...,` $etat_i$`)`

Les Déclarations PASCAL

Parti déclaration - Corps d'un module

Déclaration des objets du langage PASCAL qui seront utilisés par la suite :

- constantes, types, variables, procédures et fonctions.

Exemples

```
CONST   TempsMin = 1;
        TempsMax = 10;

TYPE    msg_type = RECORD
        id : INTEGER;
        type : (data, ack);
        END;

VAR     X : INTEGER;
        Y : (1, 2);
        Z : msg_type;
```

```
PROCEDURE nom_proc ( message : msg_type, VAR n : INTEGER )
        BEGIN … END;

FUNCTION nom_func ( n : INTEGER ) : BOOLEAN
        BEGIN … END;
```

Partie Initialisation

Corps d'un module (1/4)

La partie *initialisation* d'un corps de module est introduite par le mot clé `INITIALIZE`. Cette partie spécifie les valeurs des variables du module avec lesquelles chaque nouvelle instance du module commencera son exécution. La partie *initialisation* définit une procédure devant être exécutée qu'une seule fois, lorsqu'une instance de module est créée.

- Les variables locales

  Initialisation des variables PASCAL → utilisation des instructions PASCAL

  Exemple `X := 10;`

- Les variables exportées

  Les variables exportées des instances de modules fils sont initialisées à ce niveau.

  Exemple

  Soient X une instance de module fils et var_exp une de ses variables exportées. L'initialisation de la variable var_exp est réalisée de la façon suivante :

  `X.var_exp := 0;`

Partie Initialisation

Corps d'un module (2/4)

- Les variables états

L'initialisation de ces variables permet d'indiquer l'état initial du module : un seul état doit être état initial. Le mot réservé `TO` doit précéder l'identifiant de cet état.

Syntaxe

`TO` id_etat_initial

Le processus associé au module commencera son exécution à partir de son état *id_etat_initial*.

- Les variables instances de modules

L'initialisation des instances de modules consiste à leur attribuer un corps.

Syntaxe

`INIT` nom_var_mod `WITH` nom_corps`;`

Si une liste de paramètres est attachée à l'en-tête du module, on aura :

`INIT` nom_var_mod `WITH` nom_corps `(`par_1`, …,` par_N`);`

Le passage des paramètres se fait par valeur. Chaque instance reçoit sa propre copie des paramètres de l'en-tête avec les valeurs des paramètres évalués au moment de son initialisation.

Partie Initialisation

Corps d'un module (3/4)

Exemple

```
MODULE AE ( id : INTEGER ) … END;
…
BODY AC FOR AE … IF ( id = x ) … END;
…
MODVAR TAB = ARRAY [ 1..10 ] OF AE;
…
VAR i : INTEGER;
…
i := 1;
INIT TAB [i] WITH AC (i);
```

Par cette initialisation, la variable de module TAB[1] devient instance du module (AE, AC) et reçoit pour id la valeur 1.

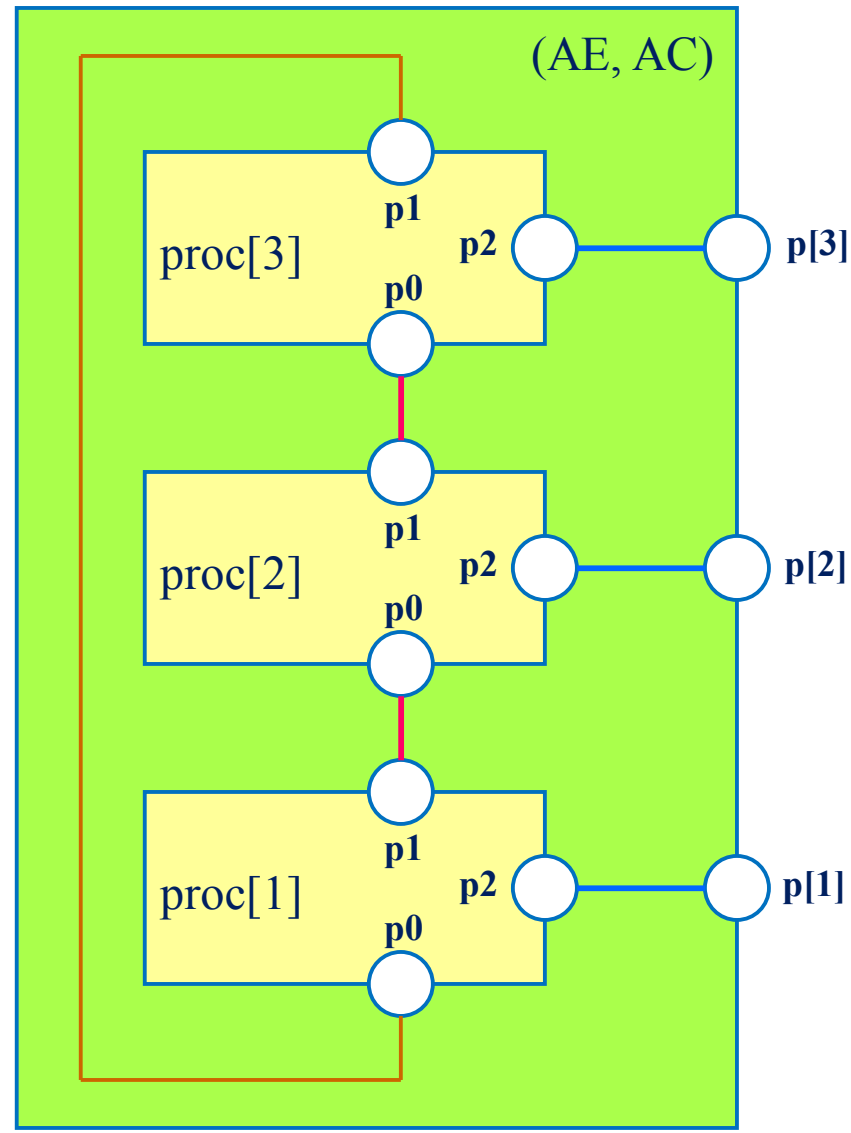
Les liens entre les points d'interaction des instances créées doivent également être initialisés (CONNECT, ATTACH).

Partie Initialisation

Corps d'un module (4/4)

Exemple

```
MODULE AE
    IP p : ARRAY[1..3] OF Can (R2) … END;
BODY AC FOR AE;
    MODULE BE
        IP p0 : Can (R1) INDIVIDUAL QUEUE
           p1, p2 : Can (R2) INDIVIDUAL QUEUE
    END;
    BODY BC FOR BE; … END;
    …
    MODVAR proc : ARRAY[1..3] OF BE;
    INITIALIZE
        BEGIN
            …
            FOR i := 1 TO 3 DO INIT proc[i] WITH BC;
            CONNECT proc[1].p0 TO proc[3].p1;
            CONNECT proc[3].p0 TO proc[2].p1;
            CONNECT proc[2].p0 TO proc[1].p1;
            FOR i := 1 TO 3 DO ATTACH p[i] TO proc[i].p2;
        END;
…
END;
```



Partie Comportement

Corps d'un module (1/2)

Le comportement interne d'un module est toujours séquentiel et se définit par un ensemble de transitions permettant au module de passer d'un état à un autre.

Une transition comporte :

- une partie condition
  - clauses simples : **`FROM`**, **`TO`**, **`ANY`**, **`WHEN`**, **`PROVIDED`**
  - clauses complexes : **`DELAY`**, **`PRIORITY`**
- une partie action
  - instructions PASCAL
  - instructions Estelle

Partie Comportement

Corps d'un module (2/2)

Syntaxe

```
TRANS   /* condition */

        /* déclarations PASCAL locales à la transition */

        BEGIN

                /* action */

        END;
```

Si la partie *condition* (formée d'un ensemble de clauses) est vérifiée, alors les instructions de la partie *action* sont susceptibles de s'exécuter.

Le franchissement d'une transition est une opération indivisible qui ne peut être interrompue.

⇨ Le tir d'une transition est atomique

Clause FROM

Partie condition - Partie comportement - Corps d'un module (1/2)

- **FROM**

La clause FROM fait partie de la condition de franchissement d'une transition. La clause FROM spécifie les états de contrôle à partir desquels une transition peut être exécutée.

Syntaxe

FROM etat

FROM $etat_0$, $etat_1$, ..., $etat_N$

FROM ETATS /* où ETATS = ($etat_i$, $etat_j$, ..., $etat_k$) */

La clause "FROM ETATS" est satisfaite si le processus se trouve dans l'un des états spécifiés par ETATS.

Clause FROM

Partie condition - Partie comportement - Corps d'un module (2/2)

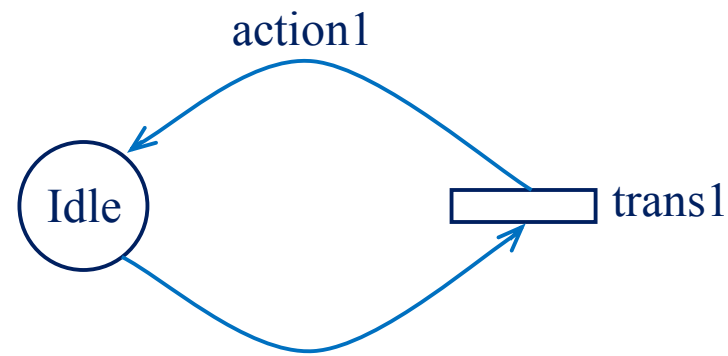
Exemple

Si le processus se trouve dans l'état `Idle`, les trois clauses suivantes sont vérifiées :

```
FROM Idle
FROM Idle, Wait, Suspend
FROM Suspend,  IDWA
          où   IDWA = (Idle, Wait)
```

Après exécution des instructions de la partie *action*, le processus reste dans le même état qui a satisfait la clause.

```
TRANS
    FROM Idle NAME trans1 :
        BEGIN
            action1
        END;
```



FROM Idle /* condition */

Clause TO

Partie condition - Partie comportement - Corps d'un module (1/2)

- **TO**

La clause TO spécifie le prochain état de contrôle lorsque la transition est exécutée.

Syntaxe

TO etat_arrivee

TO SAME ≡ FROM $etat_i$ TO $etat_i$

"*etat_arrivee*" sera le prochain état de contrôle lorsque la transition sera exécutée.
Si "*etat_arrivee*" = SAME, alors l'état de contrôle ne change pas.
Si TO est omis, alors l'état de contrôle ne change pas.

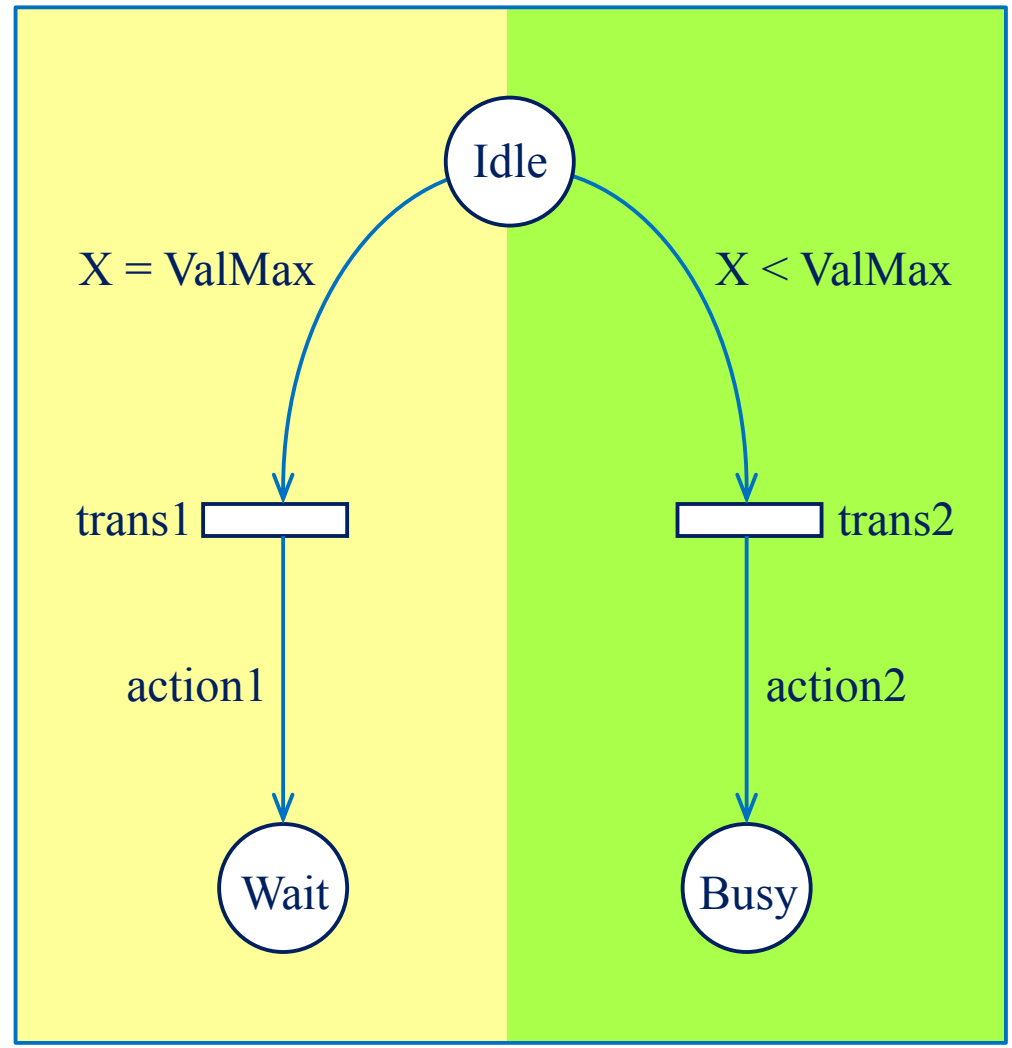
Clause TO

Partie condition - Partie comportement - Corps d'un module (2/2)

Exemple

```
TRANS
    FROM Idle TO Wait
    PROVIDED (X = ValMax) NAME trans1 :
            BEGIN
                    action1
            END;
```

```
    FROM Idle TO Busy
    PROVIDED (X < ValMax) NAME trans2:
            BEGIN
                    action2
            END;
```



Clause WHEN

Partie condition - Partie comportement - Corps d'un module (1/2)

- **WHEN**

La clause WHEN fait partie de la condition de franchissement d'une transition. Elle est satisfaite si l'*interaction* spécifiée est en tête de la file associée au point d'interaction. L'interaction est supprimée de la file uniquement après exécution de la transition.

Syntaxe

WHEN nom_ip . id_message [(par_1 , … , par_N)]

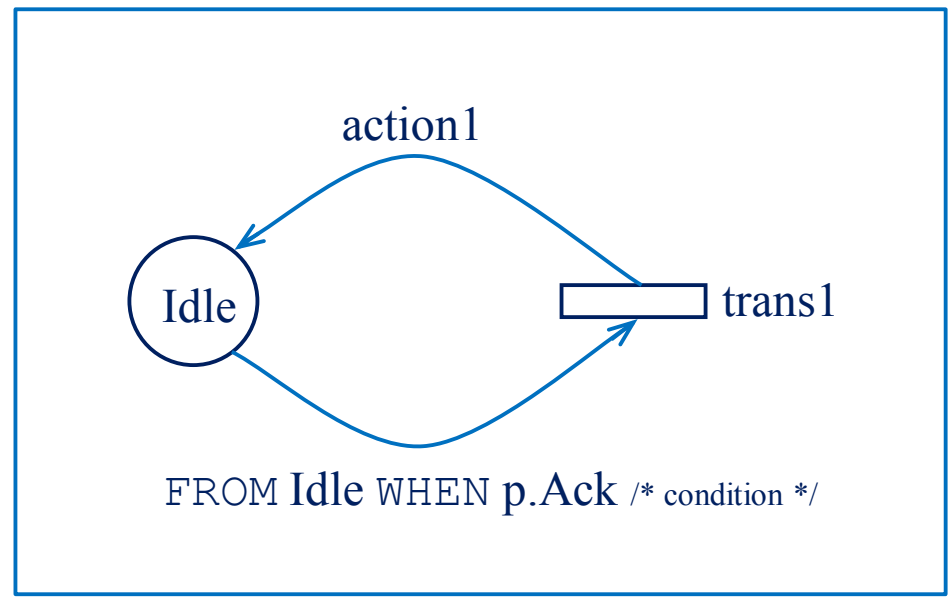
La clause "WHEN nom_ip . id_message" est satisfaite dans un état du processus si le message "*id_message*" est en tête de la file associée au point d'interaction "*nom_ip*".

Clause WHEN

Partie condition - Partie comportement - Corps d'un module (2/2)

Exemple

```
TRANS
    FROM Idle
    WHEN p.Ack NAME trans1 :
        BEGIN
            action1
        END;
```



Si le processus est dans l'état *Idle* et si le premier message dans la file associée au point d'interaction p est le message *Ack*, alors :

- le message *Ack* est consommé (retiré de la file),
- les instructions de la partie *action1* sont exécutées,
- le processus reste dans le même état *Idle*.

Clause ANY

Partie condition - Partie comportement - Corps d'un module (1/2)

- **ANY**

  Une transition dans laquelle la clause `ANY` apparaît constitue une écriture compactée d'un ensemble de transitions.

  Syntaxe

  `ANY` domaine_liste `DO`

  "*domaine_liste*" doit spécifier des listes de type ordinal qui doivent être finies et dont les bornes sont statiques et connues.

Clause ANY

Partie condition - Partie comportement - Corps d'un module (2/2)

Exemple

```
IP ports : ARRAY[1..10] OF nom_canal (sens1);

...

TRANS
    FROM Idle TO Receive
    WHEN ports[1].msg NAME trans1:
        BEGIN action END;

    ...
    ...

    FROM Idle TO Receive
    WHEN ports[10].msg NAME trans10:
        BEGIN action END;
```

```
FROM Idle TO Receive
ANY i:1..10 DO
WHEN ports[i].msg NAME trans1a10:
    BEGIN action END;
```

Clause PROVIDED

Partie condition - Partie comportement - Corps d'un module (1/2)

- **PROVIDED**

  La clause PROVIDED fait partie de la condition de franchissement d'une transition.

  Elle est satisfaite si l'expression booléenne spécifiée est vraie.

  Syntaxe

  ```
  PROVIDED (expression_booleenne | OTHERWISE)
  ```

  La clause "PROVIDED expression_booleenne" est satisfaite dans un état du processus si l'expression booléenne "expression_booleenne" est vraie.

  Si la clause PROVIDED est omise, ceci est équivalent à spécifier "PROVIDED true".

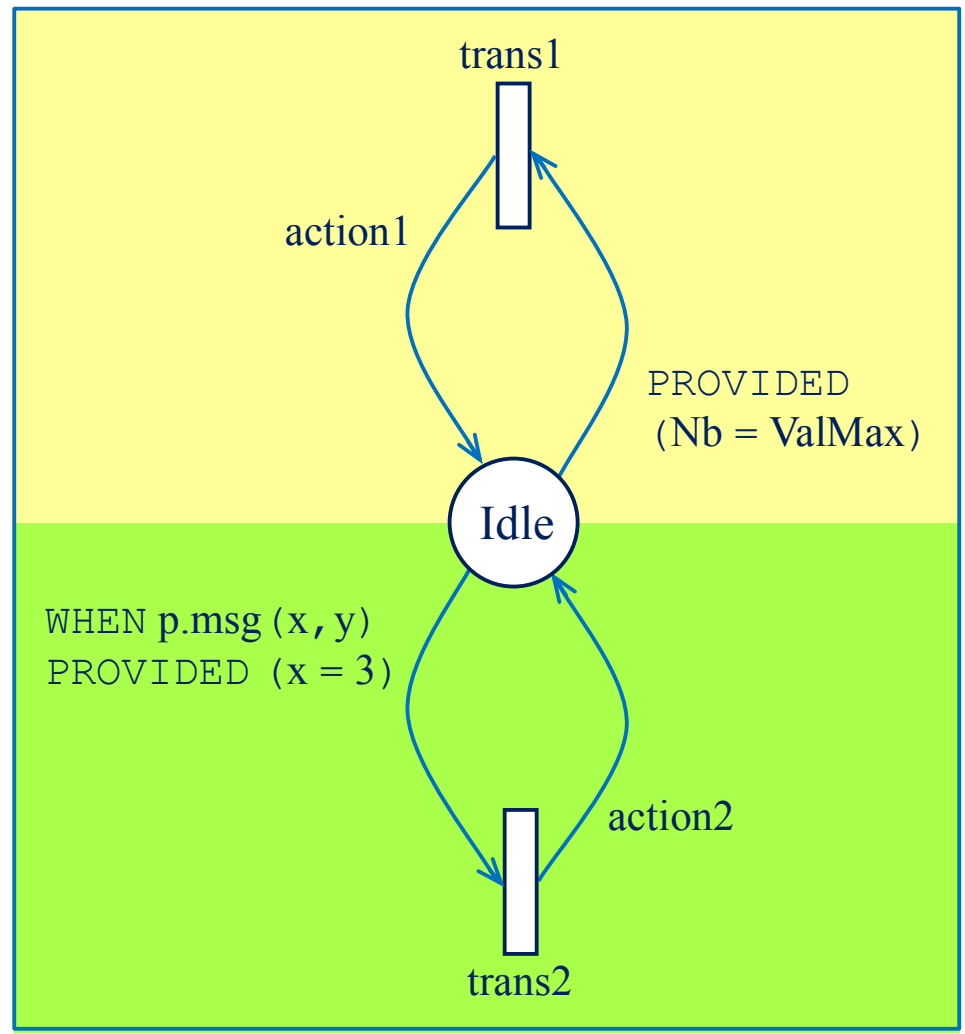
Clause PROVIDED

Partie condition - Partie comportement - Corps d'un module (2/2)

Exemple

```
TRANS
    FROM Idle
    PROVIDED (Nb = ValMax) NAME trans1 :
        BEGIN
            action1
        END;
```

```
    FROM Idle
    WHEN p.msg (x, y)
    PROVIDED (x = 3) NAME trans2 :
        BEGIN
            action2
        END;
```



Clause DELAY

Partie condition - Partie comportement - Corps d'un module (1/2)

- **DELAY**

La clause DELAY fait partie de la condition de franchissement d'une transition. Elle spécifie un intervalle de temps dans lequel l'exécution de l'action doit être effectuée.

Syntaxe

```
DELAY ( Min, Max
      | Min, *
      | Min
    )
```

Si les clauses simples de la transition sont restées vérifiées durant *Min* unités de temps, alors la transition est susceptible d'être exécutée entre le temps *Min* et le temps *Max* de façon indéterministe, ou après *Max* de façon déterministe.

Clause DELAY

Partie condition - Partie comportement - Corps d'un module (2/2)

Exemple

```
TRANS
    FROM Idle
    DELAY (3 , 10) NAME trans1 :
        BEGIN
            action1
        END;
```

≡

```
TRANS
    FROM Idle
    DELAY (Exp1, Exp2) NAME trans1 :
        BEGIN
            action1
        END;
    /* Où Exp1 = 3 et Exp2 = 10 */
```

Notes

Les valeurs spécifiées par *Min* et *Max* dans la clause `DELAY` doivent être de type `INTEGER`.

La clause `DELAY` ne peut être utilisée que pour des transitions spontanées, c.a.d. des transitions qui ne contiennent pas de clause `WHEN`.

Clause PRIORITY

Partie condition - Partie comportement - Corps d'un module (1/2)

- **PRIORITY**

  La clause PRIORITY fait partie de la condition de franchissement d'une transition. Elle permet d'ordonner les transitions en fonction de leur priorité. L'ordonnancement concerne uniquement les transitions d'**une même instance** de module.

  Syntaxe

  PRIORITY val_priorite

  La priorité est un des éléments pris en compte dans la sélection des transitions *franchissables* dans un état donné parmi celles qui sont *sensibilisées*.

  "*val_priorite*" doit être un entier positif. La valeur entière la plus petite conférant la priorité la plus grande.

Clause PRIORITY

Partie condition - Partie comportement - Corps d'un module (2/2)

Exemple

```
TRANS
    FROM Idle NAME trans1 :
        BEGIN
            action1
        END;
```

```
TRANS
    FROM Idle NAME trans2 :
        BEGIN
            action2
        END;
```

Si *trans1* et *trans2* sont sensibilisées, *trans1* ou *trans2* sera franchie de façon indéterministe.

```
TRANS
    FROM Idle
    PRIORITY Haute NAME trans1 :
        BEGIN
            action1
        END;
```

```
TRANS
    FROM Idle
    PRIORITY Basse NAME trans2 :
        BEGIN
            action2
        END;
```

Si *trans1* et *trans2* sont sensibilisées, *trans1* sera franchie en premier (si valeur(*Haute*) < valeur(*Basse*)).

Note : UNE TRANSITION D'UN MODULE PARENT EST TOUJOURS PLUS PRIORITAIRE QU'UNE TRANSITION D'UN MODULE FILS !

Création Dynamique d'Instances de Modules

Partie action - Partie comportement - Corps d'un module (1/2)

- **`INIT`**

  L'instruction `INIT` peut apparaître dans les parties *initialisation* et *transition* ⇨ création dynamique d'instances de modules

  Son exécution sélectionne un corps particulier pour une instance de module et initialise cette dernière.

  Elle assigne une valeur à la variable d'instance de module référencée et assigne les valeurs aux paramètres formels.

  Syntaxe

  /* *Identique à celle utilisée dans la partie initialisation.* */

  `INIT` id_var_mod `WITH` id_corps [(par_1, …, par_N)];

Création Dynamique d'Instances de Modules

Partie action - Partie comportement - Corps d'un module (2/2)

Exemple

```
TRANS
    FROM Idle TO SAME
    WHEN p.msg(x, b)
    PROVIDED (b) NAME trans1:
        BEGIN
            INIT proc[x] WHITH BC;
        END;
```

L'exécution de l'instruction `INIT` inclut l'exécution de la partie *initialisation* du module fils pour l'instance créée.

Si `INIT` référence une variable de module qui identifie déjà une instance de module, une nouvelle instance est créée et la variable référencée reçoit une nouvelle valeur. La désignation de l'instance initiale est perdue, à moins que la valeur de la variable référencée ait été sauvegardée dans une autre variable de module (de même type) par une instruction d'affectation.

Toutefois, la désignation de l'instance initiale peut être recouvrée par l'utilisation des instructions `ALL` et `FORONE`.

Suppression et Terminaison d'Instances de Modules

Partie action - Partie comportement - Corps d'un module (1/2)

- **`RELEASE`, `TERMINATE`**

  Conséquences des instructions `RELEASE` et `TERMINATE`

  – tous les points d'interaction externes, de l'instance, attachés (resp. connectés) sont détachés (resp. déconnectés),

  – l'instance ainsi que toutes ses instances descendantes sont supprimées.

  Les valeurs de toutes les variables de modules qui identifiaient les instances supprimées deviennent indéfinies.

  Syntaxe

```
RELEASE id_var_mod;
TERMINATE id_var_mod;
```

```
DETACH X;
TERMINATE X;
```

} `RELEASE X;`

Suppression et Terminaison d'Instances de Modules

Partie action - Partie comportement - Corps d'un module (2/2)

Exemple

```
TRANS
    FROM Idle TO SAME
    WHEN p.msg(x, b)
    PROVIDED (b) NAME trans1:
        BEGIN
            RELEASE proc[x];          /* l'instance proc[x] est définitivement supprimée */
        END;
```

OU

```
TRANS
    FROM Idle TO SAME
    WHEN p.msg(x, b)
    PROVIDED (b) NAME trans1:
        BEGIN
            TERMINATE proc[x];        /* l'instance proc[x] est définitivement supprimée */
        END;
```

Création et Suppression Dynamiques de Liens

Partie action - Partie comportement - Corps d'un module (1/2)

- **CONNECT, ATTACH**

  Syntaxe

```
/* Identique à celle utilisée dans la partie initialisation. */
CONNECT nom_var_mod.nom_ip TO nom_var_mod.nom_ip
ATTACH nom_ip TO nom_var_mod.nom_ip
```

- **DISCONNECT, DETACH**

  Syntaxe

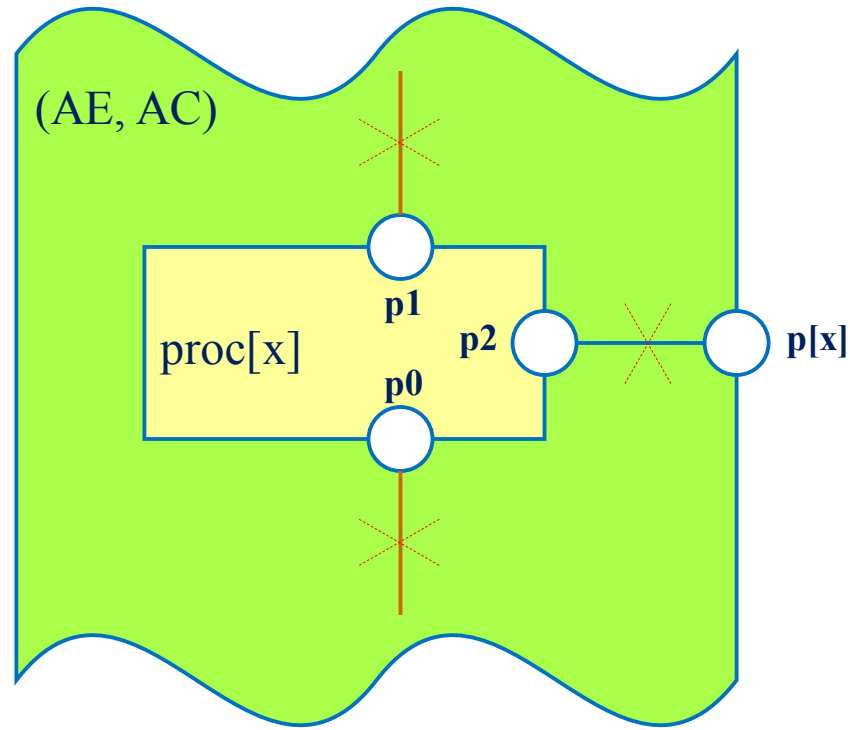
```
DISCONNECT nom_ip                    /* ip interne */
DISCONNECT nom_var_mod.nom_ip        /* ip externe */
DISCONNECT nom_var_mod               /* ips externes */
DETACH nom_ip                        /* ip externe */
DETACH nom_var_mod.nom_ip            /* ip externe */
```

Création et Suppression Dynamiques de Liens

Partie action - Partie comportement - Corps d'un module (2/2)

Exemple

```
TRANS
        FROM Idle TO SAME
        WHEN p.msg(x, b)
        PROVIDED (b) NAME trans1:
            BEGIN
                DISCONNECT proc[x].p0;
                DISCONNECT proc[x].p1;
                DETACH proc[x].p2;
            END;
```



INIT, **RELEASE**, **TERMINATE**, **CONNECT**, **DISCONNECT**, **ATTACH** et **DETACH**

⇨ modification dynamique de la configuration.

Émission de Messages

Partie action - Partie comportement - Corps d'un module (1/2)

- **OUTPUT**

Le résultat de l'instruction `OUTPUT` est que l'interaction spécifiée (ainsi que ses éventuels paramètres) sera ajoutée à la file assignée au point d'interaction de l'autre extrémité.

Syntaxe

`OUTPUT` id_ip`.`id_var_message [`(`par_1`,` … `,` par_N`)`]

L'instruction `OUTPUT` permet d'envoyer le message "id_var_message" via le point d'interaction "id_ip".

Un point d'interaction étant lié à au plus un autre point d'interaction

⇨ le processus récepteur du message est connu et unique.

De même, un message reçu ne peut provenir que d'un seul émetteur.

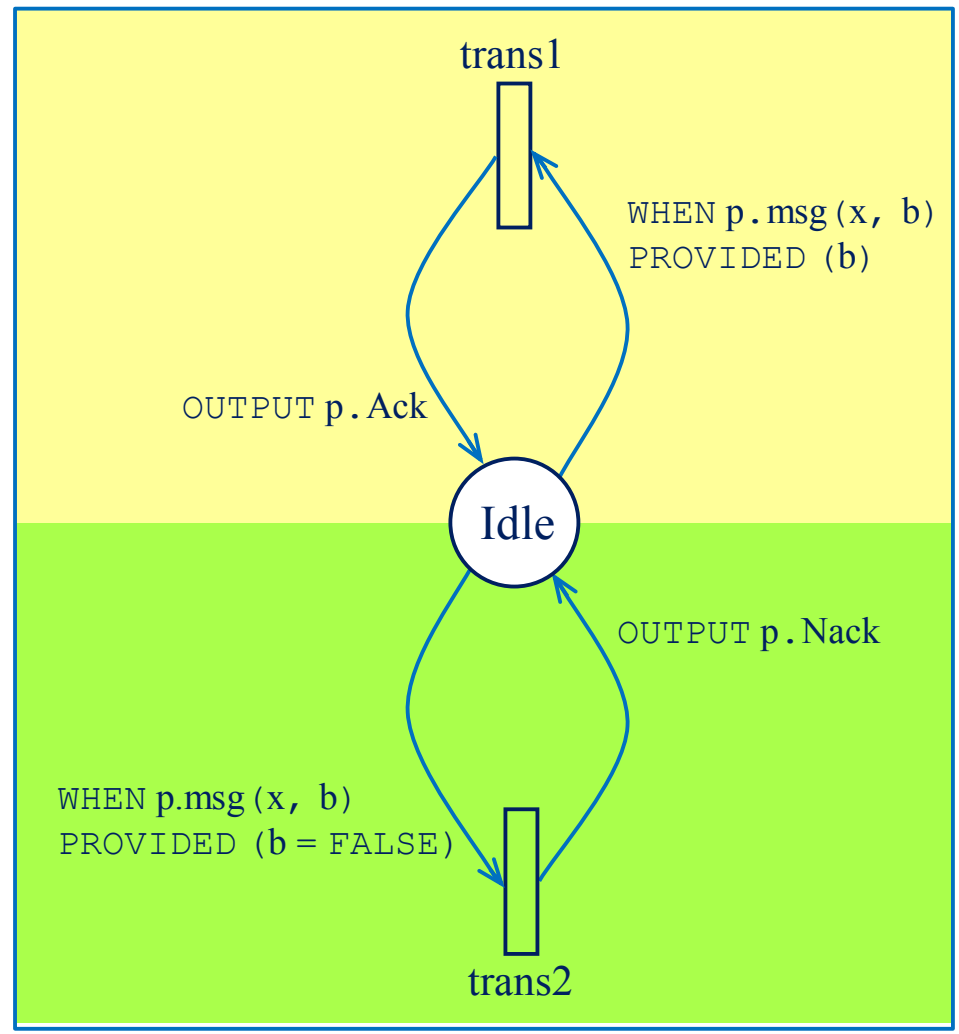
Émission de Messages

Partie action - Partie comportement - Corps d'un module (2/2)

Exemple

```
TRANS
   FROM Idle TO SAME
   WHEN p.msg(x, b)
   PROVIDED (b) NAME trans1:
      BEGIN
         OUTPUT p.Ack;
      END;
```

```
TRANS
   FROM Idle TO SAME
   WHEN p.msg(x, b)
   PROVIDED (b = FALSE) NAME trans2:
      BEGIN
         OUTPUT p.Nack;
      END;
```



Instruction ALL

Partie action - Partie comportement - Corps d'un module (1/2)

- **ALL ... DO**

L'instruction ALL est une instruction de répétition qui permet l'itération sur un type ordinal ou un ensemble d'instances de module.

Syntaxe

```
ALL id : domaine_liste DO instructions;
ALL id : domaine_module DO instructions;
```

L'occurrence d'un identifiant dans "domaine_liste" ou "domaine_module" d'une instruction ALL constitue son point de définition comme identifiant de variable pour la région définie par l'instruction ALL.

Le résultat d'une instruction ALL est l'exécution des instructions qui suivent le mot clé DO pour :

- toutes les valeurs de type ordinal données par "domaine_liste", ou
- toutes les instances fils dont les définitions d'entêtes sont identifiées par l'identifiant d'entête du "domaine_module".

Instruction ALL

Partie action - Partie comportement - Corps d'un module (2/2)

Exemple

```
MODVAR proc : ARRAY[1..N] OF id_en-tete
...
INITIALIZE TO Idle
    BEGIN
        FOR i := 1 TO N DO
            INIT proc[i] WITH id_corps;
    END;

TRANS
    FROM Idle TO SAME NAME trans1 :
        BEGIN
            ALL i : 1..N DO RELEASE proc[i];
        END;
```

```
MODVAR T : id_en-tete
...
INITIALIZE TO Idle
    BEGIN
        FOR i := 1 TO N DO
            INIT T WITH id_corps;
        END;

TRANS
    FROM Idle TO SAME NAME trans2 :
        BEGIN
            ALL t : id_en-tete DO RELEASE t;
        END;
```

Instruction FORONE

Partie action - Partie comportement - Corps d'un module

- **FORONE ... SUCHTHAT ... DO**

Permet l'exécution d'un ensemble d'instructions si une expression booléenne donnée est vérifiée par au moins un élément.

Syntaxe

```
FORONE ( domaine_liste | domaine_module )
SUCHTHAT expression_booleenne;
DO instructions
[OTHERWISE instructions];
```

Les instructions spécifiées après le mot clé `DO` sont exécutées si l'expression booléenne introduite par `SUCHTHAT` est vérifiée par au moins un élément de l'ensemble :

- des valeurs de type ordinal données par "domaine_liste", ou
- des instances fils dont les définitions d'entêtes sont identifiées par l'identifiant d'en-tête du "domaine_module".

Si l'expression booléenne est évaluée à faux pour tous les éléments de tels ensembles, alors les instructions contenue dans la clause `OTHERWISE`, si présente, sont exécutées.

Expression EXIST

Partie action - Partie comportement - Corps d'un module

- **EXIST … SUCHTHAT**

  Permet d'évaluer s'il existe un élément vérifiant une expression booléenne donnée.

  Syntaxe

  ```
  EXIST ( domaine_liste | domaine_module )
  SUCHTHAT expression_booleenne;
  ```

  Une expression EXIST est une expression booléenne qui est satisfaite s'il y a au moins un élément de l'ensemble :

  - des valeurs de type ordinal données par "domaine_liste", ou
  - des instances fils dont les définitions d'entêtes sont identifiées par l'identifiant d'en-tête du "domaine_module".

  qui satisfait "expression_booleenne".

Extensions/Restrictions au Langage Pascal

Partie action - Partie comportement - Corps d'un module

- **Fonctions et Procédures**
  - Les fonctions peuvent retourner des données de types structurés.
  - Les fonctions et les procédures ne doivent pas référencer des objets non Pascal (i.e., variables de modules, points d'interactions, interactions ou états).
  - Les instructions Estelle `ALL`, `FORONE` et `EXIST` qui apparaissent dans des fonctions ou des procédures doivent être utilisées avec "domaine_liste" limité au type ordinal.
  - Les instructions Estelle `INIT`, `RELEASE`, `CONNECT`, `DISCONNECT`, `ATTACH`, `DETACH` et `OUTPUT` ne doivent pas être utilisées dans les fonctions et procédures.

Module Spécification

Toutes les définitions de modules, telles que décrites précédemment, sont contenues dans un module principal (racine) appelé **module spécification**

⇨ *module de la spécification complète du système de l'utilisateur*.

Syntaxe

```
SPECIFICATION nom_specification [ attribut_classe ]
[ DEFAULT ( INDIVIDUAL | COMMON ) QUEUE ];
[ TIMESCALE ( second | millisecond | … ) ];
    /* Définition du corps */
END.
```

Il est supposé qu'il n'existe qu'une seule instance du module spécification.

Attributs de Classe et Règles d'Attribution

- Un module peut avoir l'un des attributs de classe suivants ou aucun
  - `SYSTEMPROCESS, SYSTEMACTIVITY, PROCESS, ACTIVITY`
- Toutes les instances d'un module ont le même attribut
- Les modules ayant l'attribut `SYSTEMPROCESS` ou `SYSTEMACTIVITY` sont appelés *modules système*.
- Les règles d'attribution suivantes doivent être observées dans une hiérarchie de modules :
  - Chaque module actif doit être attribué.
  - Les modules système ne doivent pas apparaître à l'intérieur d'un module attribué.
  - Les modules attribués `PROCESS` ou `ACTIVITY` doivent être des descendants d'un module système.
  - Un module attribué `SYSTEMPROCESS` (ou `PROCESS`) peut contenir des modules attribués `PROCESS` ou `ACTIVITY`.
  - Un module `SYSTEMACTIVITY` (ou `ACTIVITY`) peut contenir des modules attribués `ACTIVITY`.

Conditions de Franchissement des Transitions

- Une transition est sensibilisée dans un état donné d'un module si les clauses `WHEN`, `FROM` et `PROVIDED` (si présentes) sont toutes satisfaites dans cet état.

- Une transition est dite franchissable (ou prête à être tirée) dans un état donné d'un module et à un instant donné si :

  a) elle est sensibilisée dans cet état, et si elle contient une clause `DELAY(Min, Max)` alors la transition est restée au moins Min unités de temps sensibilisée, et

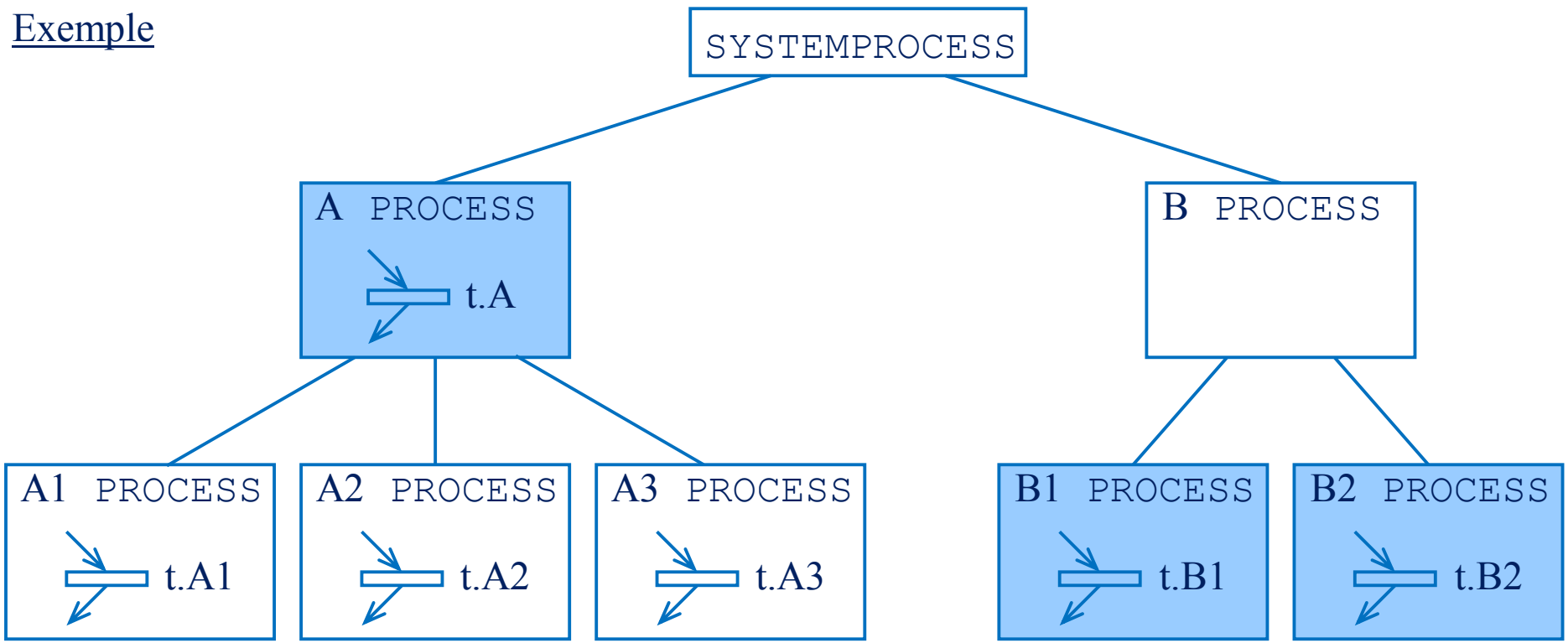
  b) elle a la plus haute priorité parmi celles satisfaisant **a**)

Influence des Attributs sur le Fonctionnement (1/3)

- Attributs **SYSTEMPROCESS**, **PROCESS**

  Dans ce cas, toutes les transitions franchissables qui ne sont pas en conflit hiérarchique sont sélectionnées et exécutées.

Exemple



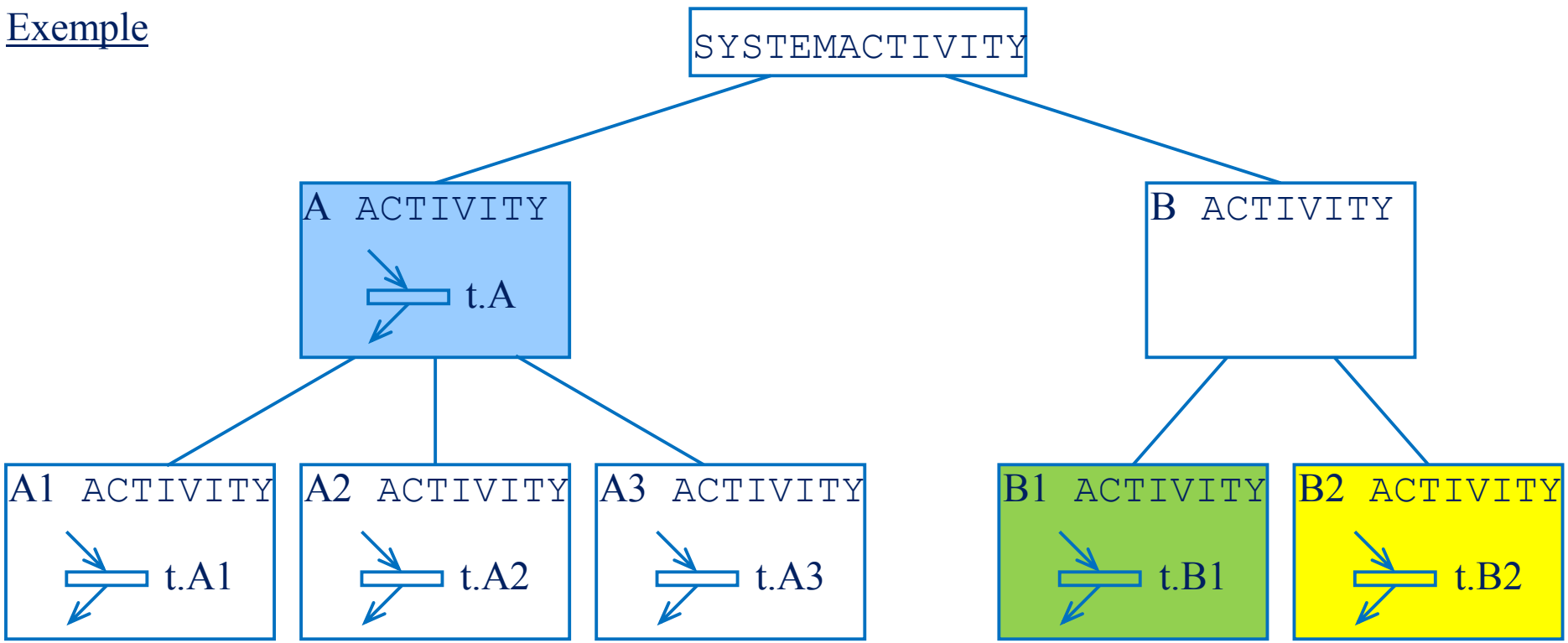
Les transitions t.A, t.B1 et t.B2 sont exécutées en parallèle.

Influence des Attributs sur le Fonctionnement (2/3)

- Attributs **SYSTEMACTIVITY**, **ACTIVITY**

Dans ce cas, une seule des transitions franchissables est sélectionnée et exécutée. On a un comportement non-déterministe : la transition à exécuter est choisie de façon non-déterministe.

Exemple



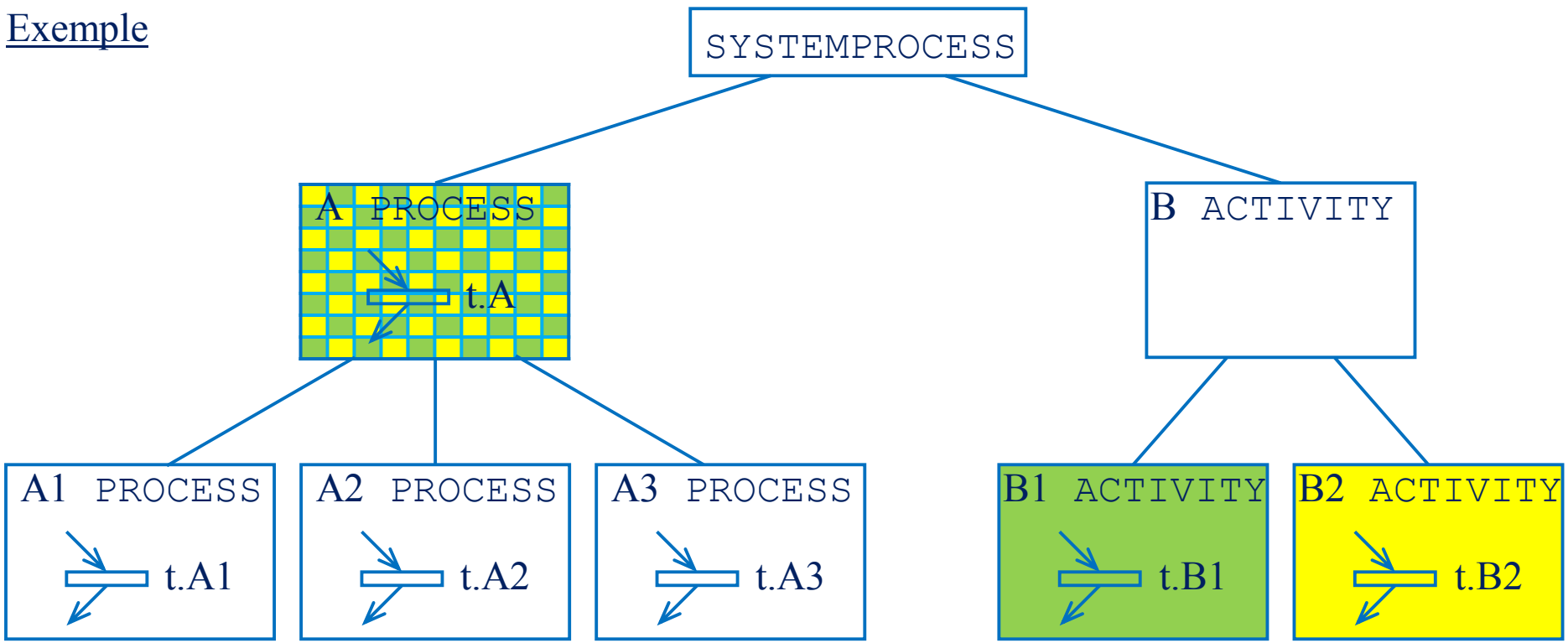
La transition t.A ou t.B1 ou t.B2 sera exécutée.

Influence des Attributs sur le Fonctionnement (2/3)

- Attributs mixtes

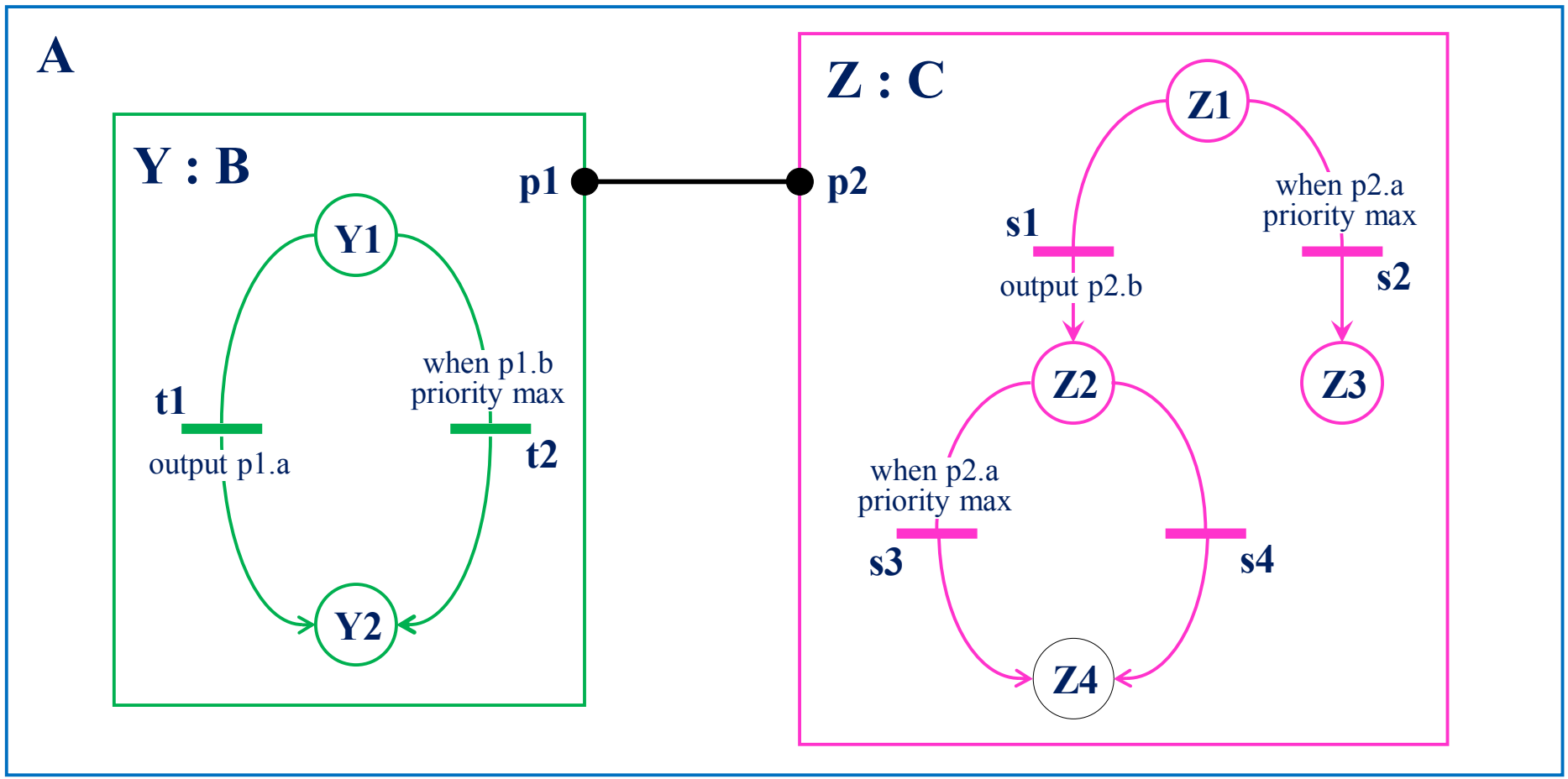
Dans ce cas, les deux principes, donnés précédemment, s'appliquent à chacune des branches du sous-système, et on retrouve également la notion de non-déterminisme.

Exemple

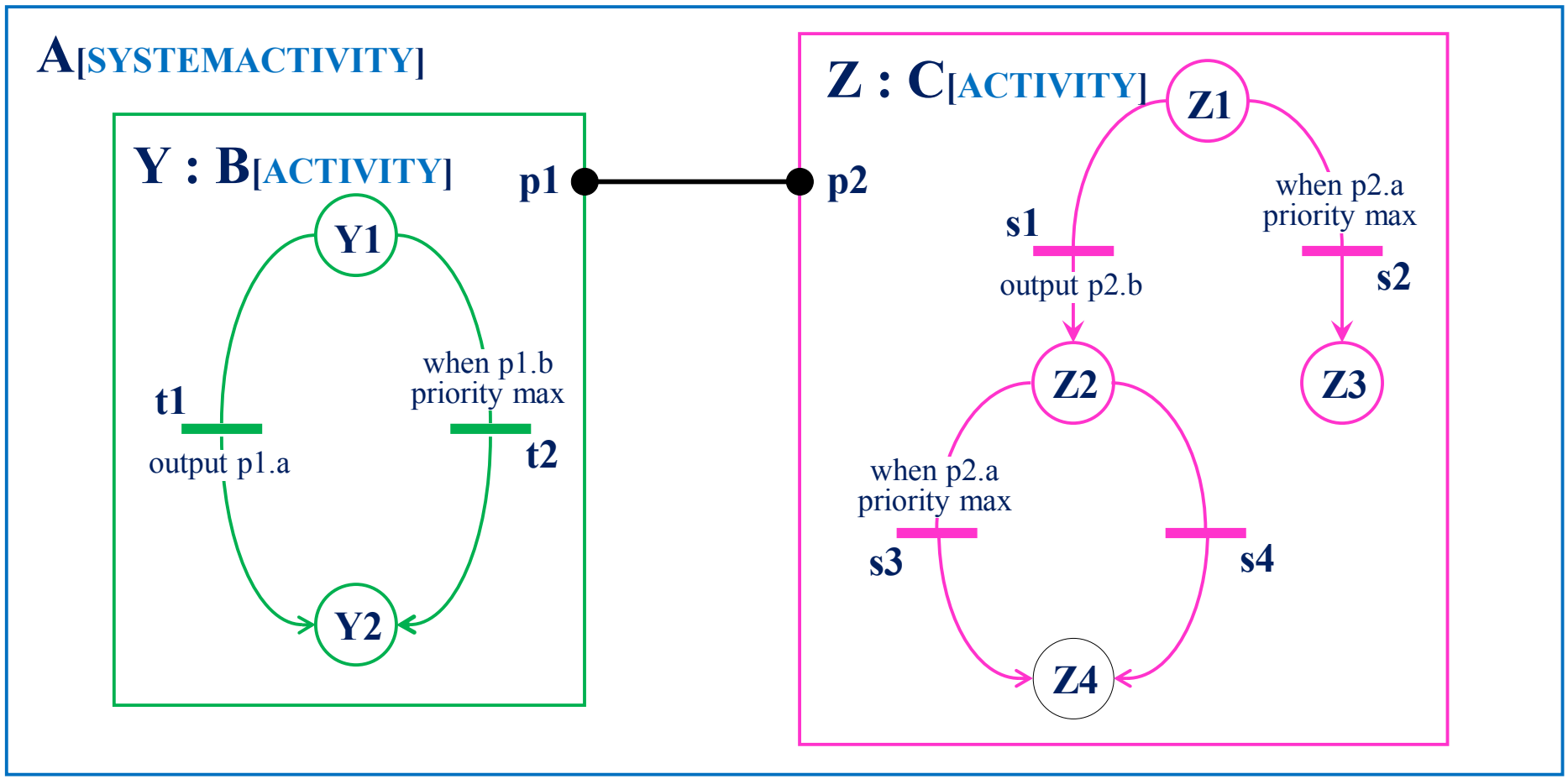


Les transitions t.A et t.B1 ou bien les transitions t.A et t.B2 seront exécutées.

Exemples de Comportements



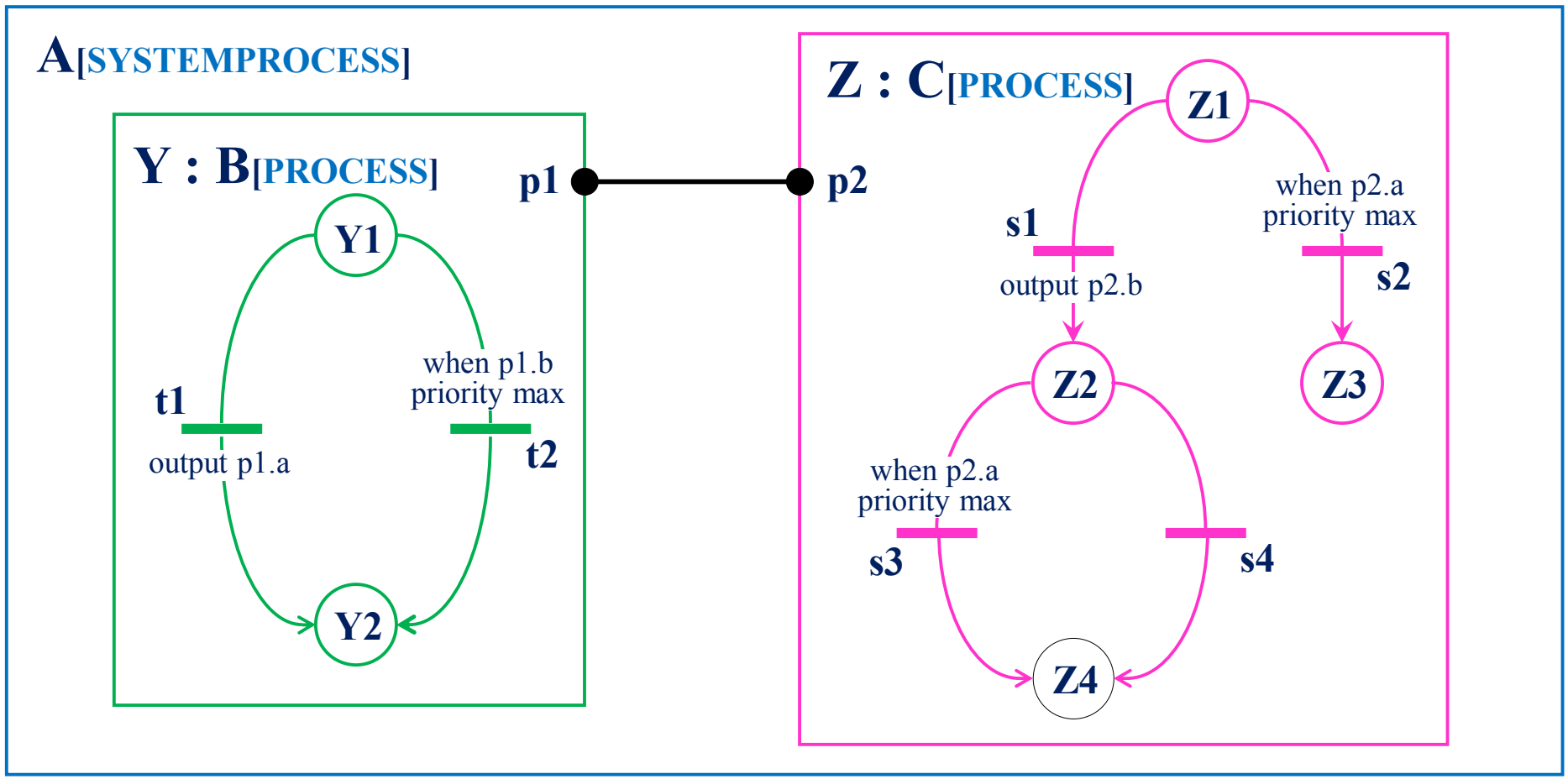
Comportement Non-Déterministe



Séquences d'exécutions admissibles :

<t1, s2>, <s1, t2, s4> et <s1, s4, t2>

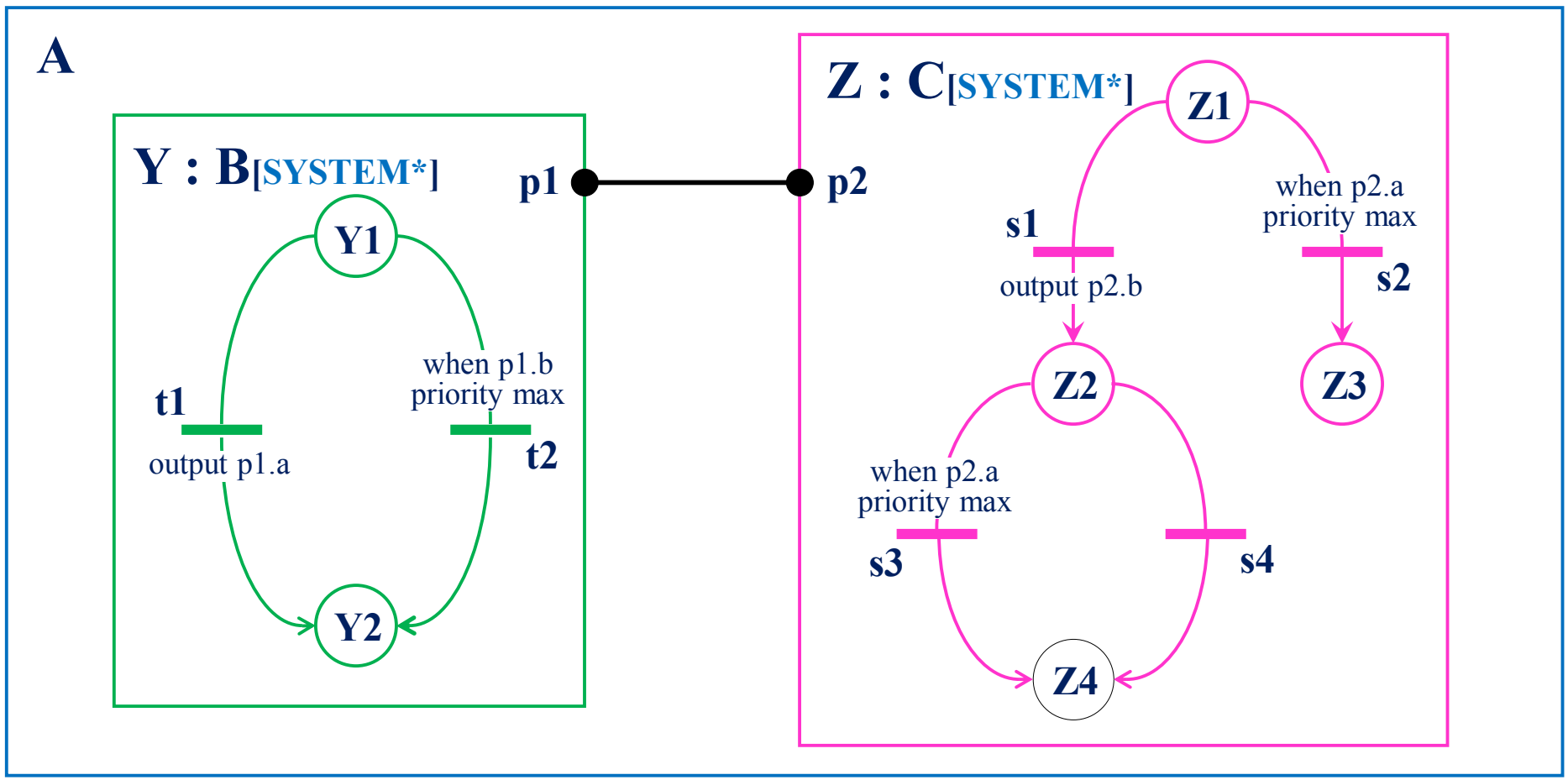
Comportement Synchrone



Séquences d'exécutions admissibles :

<t1, s1, s3> et <s1, t1, s3>

Comportement Asynchrone entre Systèmes



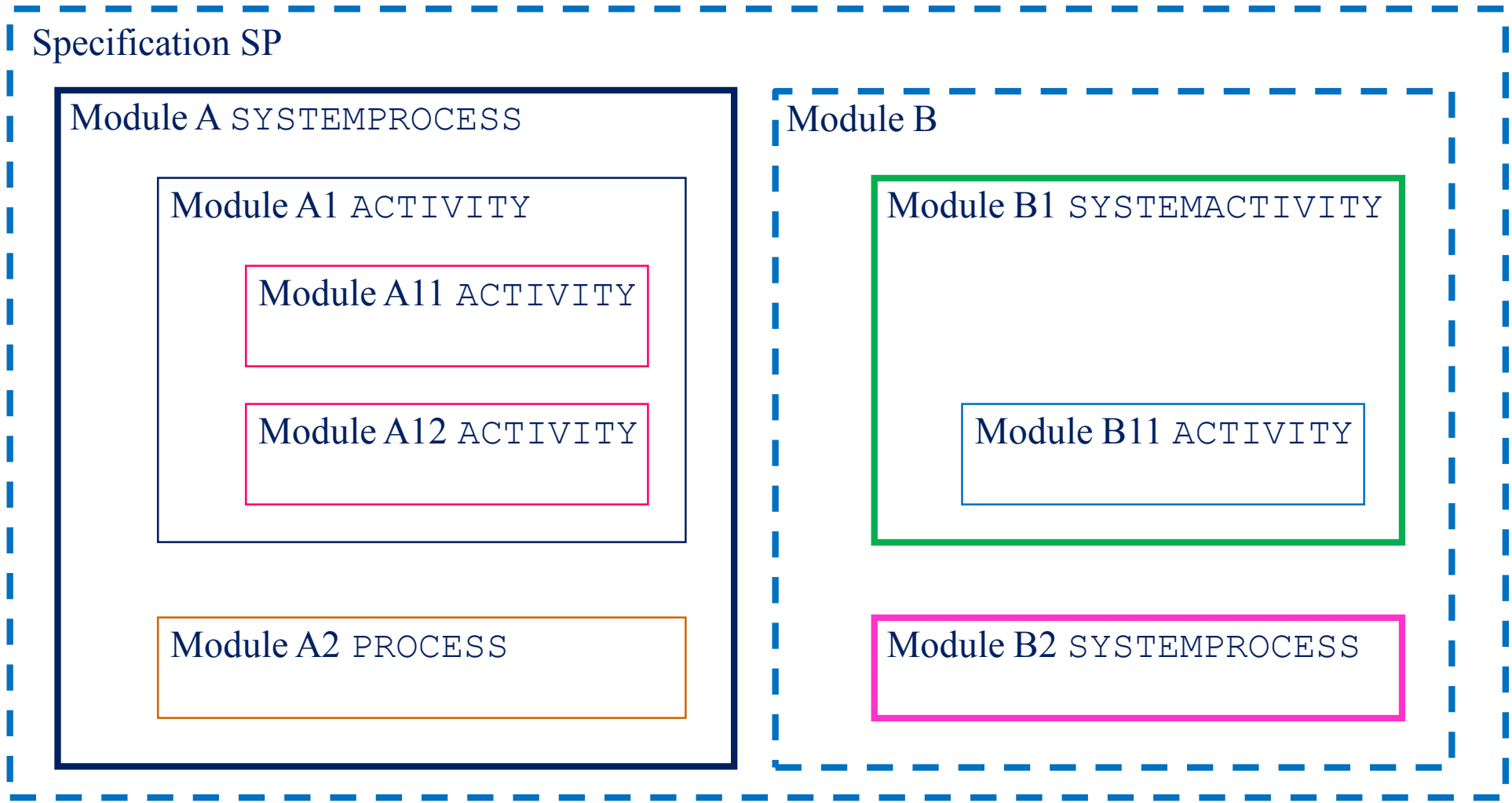
Séquences d'exécutions admissibles :

<t1, s2>, { Y commence et termine t1 }

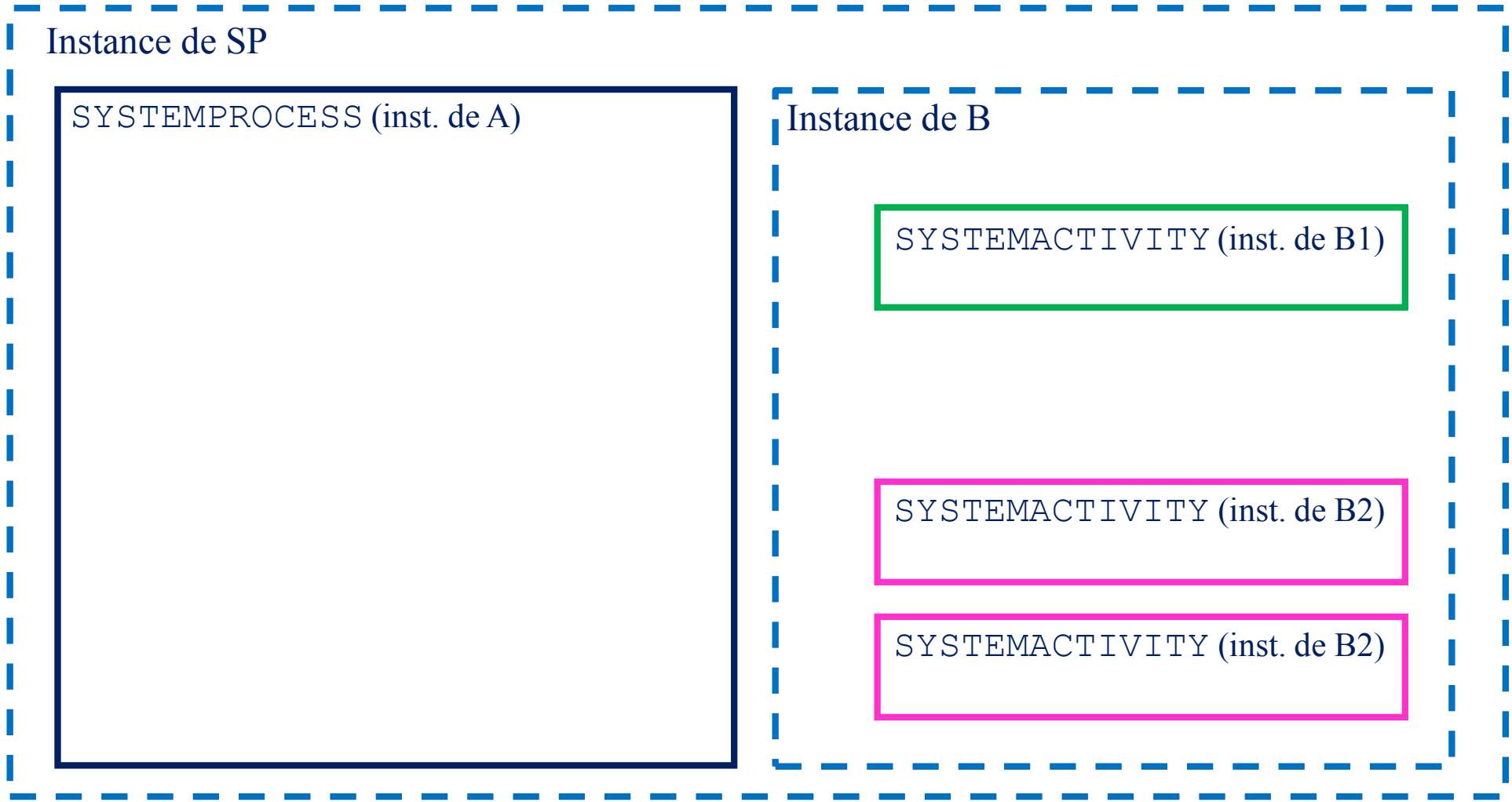
<s1, t2, s4>, <s1, s4, t2>, {Z commence et termine s1 }

<t1, s1, s3>, <s1, s4, t1>, <s1, t1, s4> et <s1, t1, s3>

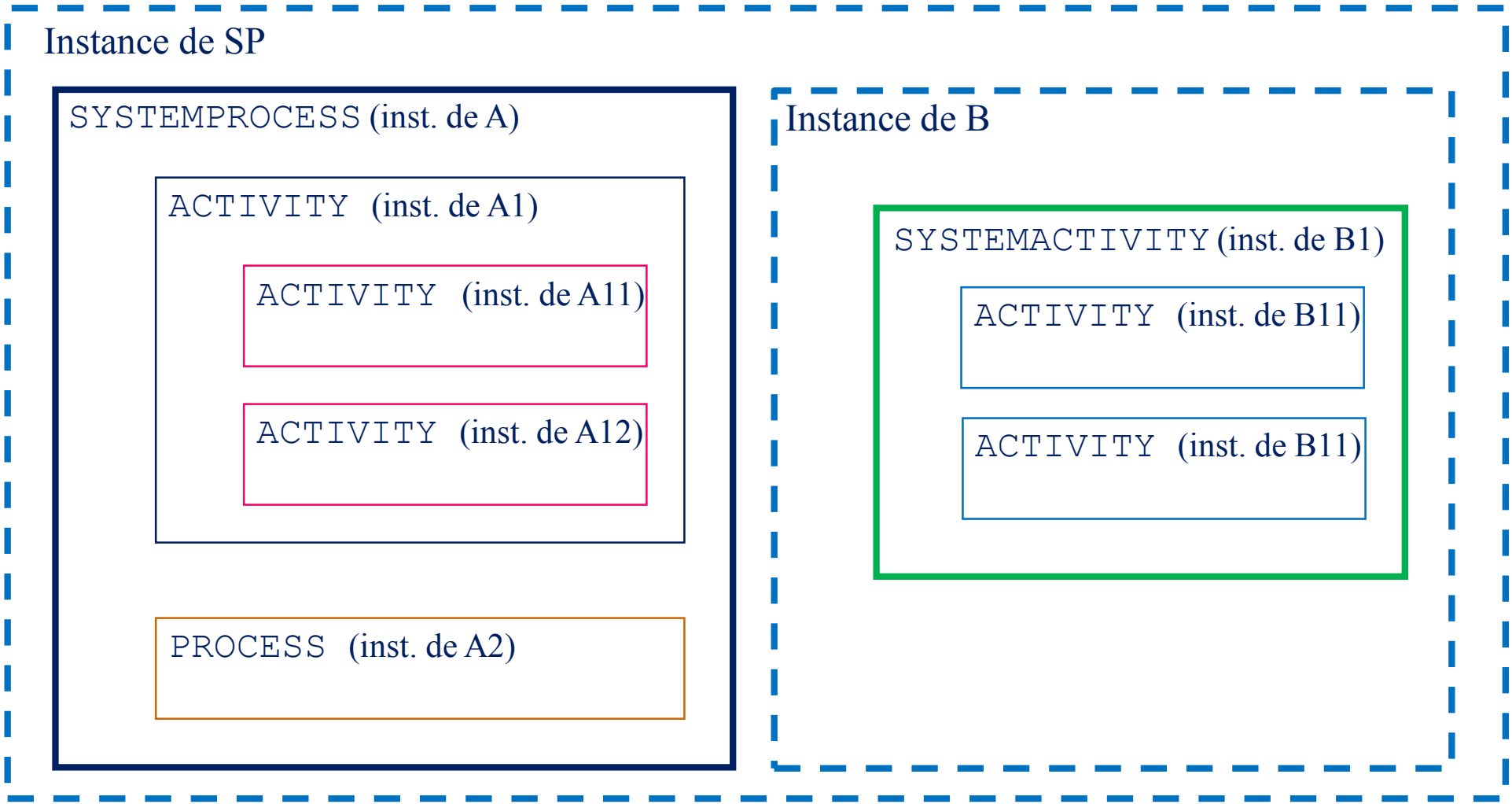
Structure Hiérarchique d'une Spécification



Initialisation Possible de la Structure



Autre Initialisation Possible de la Structure



Exemple

Description du système

Soit un système client/serveur composé d'un serveur et de trois clients.

Le serveur a les tâches suivantes :

- sur réception du message "vrai", émis par un client, il renvoie le message "ack",
- sur réception du message "faux", émis par un client, il renvoie le message "nack".

Le serveur doit également comptabiliser :

- les nombres de messages "vrai" et "faux" reçus (variables "NbVraiRecus" et "NbFauxRecus"),
- les nombres de messages "ack" et "nack" émis (variables "NbAckEmis" et "NbNackEmis").

Le serveur utilisera **une file commune** pour l'ensemble des points d'interaction qu'il utilise pour communiquer avec les trois clients.

Chaque client a les tâches suivantes :

- il peut, de manière spontanée, émettre vers le serveur soit un message "vrai" soit un message "faux",
- après émission du message ("vrai" ou "faux), il se met en attente du message retourné par le serveur,
- il ne peut émettre un nouveau message avant d'avoir reçu le message "ack" ou "nack") du serveur.

De plus, chaque client doit comptabiliser :

- les nombres de messages "vrai" et "faux" émis (variables "NbVraiEmis" et "NbFauxEmis"),
- les nombres de messages "ack" et "nack" reçus (variables "NbAckRecus" et "NbNackRecus").

Exemple

Spécification du système (1/4)

```
SPECIFICATION ServerClient SYSTEMPROCESS;
TIMESCALE second;

TYPE message = (ack, nack, vrai, faux);

CHANNEL Serv_Cli (sens1, sens2);
        BY sens1, sens2 : msg(m : message);
```

```
MODULE m_Server PROCESS;
        IP sip : ARRAY[1..3] OF Serv_Cli(sens1) COMMON QUEUE;
END;
```

```
MODULE m_Client PROCESS;
        IP cip : Serv_Cli(sens2) INDIVIDUAL QUEUE;
END;
```

Exemple

Spécification du système (2/4)

```
BODY c_Server FOR m_Server;
        VAR NBVraiRecus, NbFauxRecus, NbAckEmis, NbNackEmis : INTEGER;

STATE Attente;
INITIALIZE  TO Attente;
        BEGIN NbVraiRecus := NbFauxRecus := NbAckEmis := NbNackEmis := 0;  END;

TRANS
FROM Attente TO Attente
        ANY i : 1..3 DO WHEN sip[i].msg(m) PROVIDED (m = vrai) NAME Ts1:
        BEGIN       NbVraiRecus := NbVraiRecus + 1; NbAckEmis := NbAckEmis + 1;
                    OUTPUT sip[i].msg(ack); END;
FROM Attente TO Attente
        ANY i : 1..3 DO WHEN sip[i].msg(m) PROVIDED (m = faux) NAME Ts2:
        BEGIN       NbFauxRecus := NbFauxRecus + 1; NbNackEmis := NbNackEmis + 1;
                    OUTPUT sip[i].msg(nack); END;
END;  { end body c_Server }
```

Exemple

Spécification du système (3/4)

```
BODY c_Client FOR m_Client;
      VAR NBVraiEmis, NbFauxEmis, NbAckRecus, NbNackRecus : INTEGER;

STATE Repos, Attente;
INITIALIZE TO Repos;
      BEGIN NBVraiEmis := NbFauxEmis := NbAckRecus := NbNackRecus := 0; END;

TRANS
FROM Repos TO Attente NAME Tc1:
      BEGIN      NbVraiEmis := NbVraiEmis + 1; OUTPUT cip.msg(vrai); END;
FROM Repos TO Attente NAME Tc2:
      BEGIN      NbFauxEmis := NbFauxEmis + 1; OUTPUT cip.msg(faux); END;
FROM Attente TO Repos WHEN cip.msg(m) PROVIDED ( m = ack ) NAME Tc3:
      BEGIN      NbAckRecus := NbAckRecus + 1; END;
FROM Attente TO Repos WHEN cip.msg(m) PROVIDED ( m = nack ) NAME Tc4:
      BEGIN      NbNackRecus := NbNackRecus + 1; END;
END;  { end body c_Client }
```

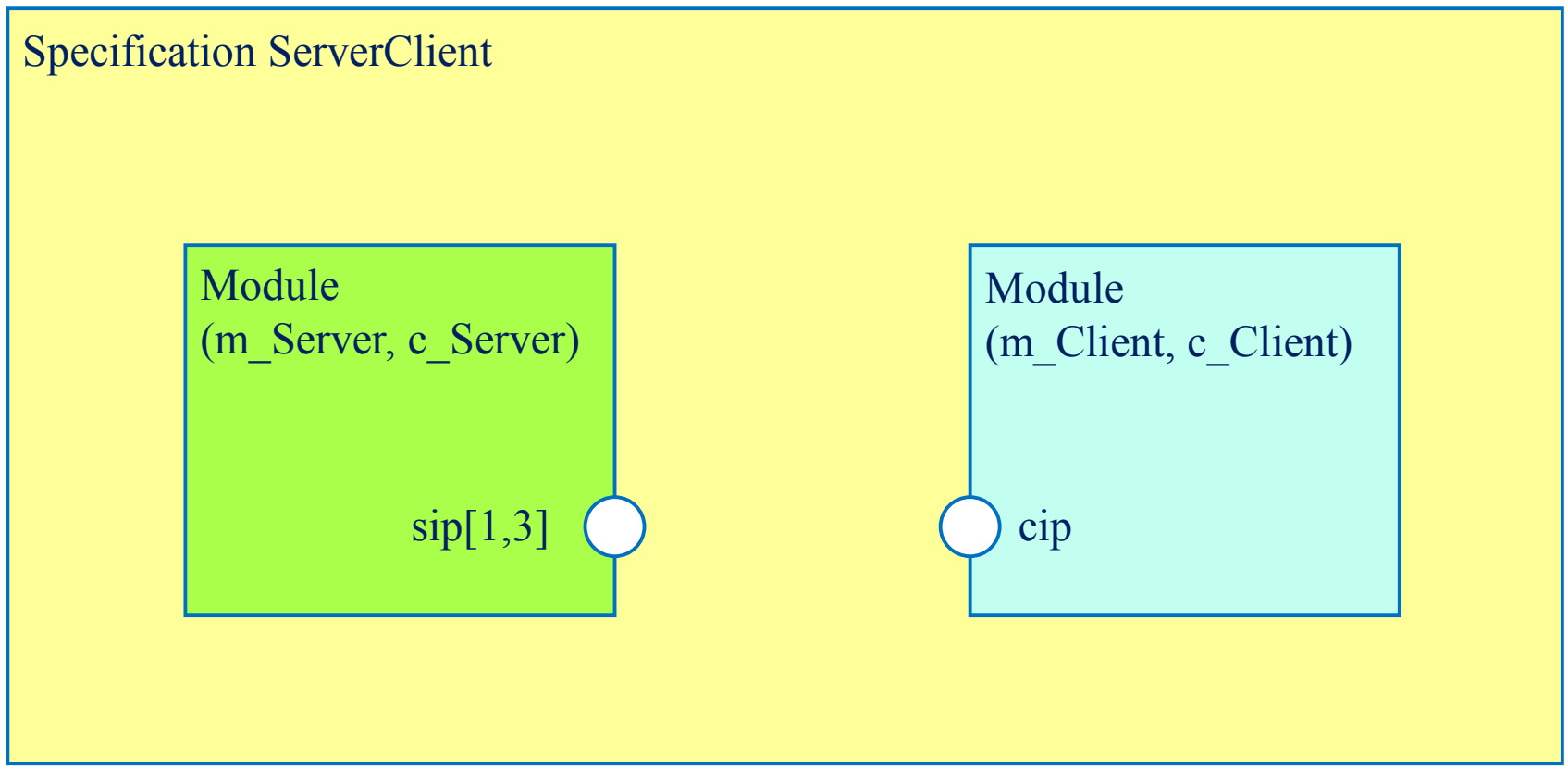
Exemple

Spécification du système (4/4)

```
MODVAR server : m_Server; clients : ARRAY[1..3] OF m_Client;

INITIALIZE
     VAR i : INTEGER;
     BEGIN
          INIT server WITH c_Server;
          FOR i := 1 TO 3 DO INIT clients[i] WITH c_Client;

          FOR i := 1 TO 3 DO CONNECT server.sip[i] TO clients[i].cip;
     END;

END.  { end specification ServerClient }
```

Exemple

Structure hiérarchique spécifiée



Exemple

Configuration initiale