

SDL : Specification and Description Language

Marc BOYER

ENSEEIHT - 3SN

Septembre 2020

1 – Introduction

- SDL = Specification and Description Language
- norme de l'ITU-T (recommandation Z.100–Z.109)
- principales caractéristiques (issu du site de [Soc])
 - *the ability to be used as a wide spectrum language from requirements to implementation;*
 - *suitability for real-time, stimulus-response systems;*
 - *presentation in a graphical form;*
 - *a model based on communicating processes (extended finite state machines)*
 - *object oriented description of SDL components.*

Références

- [Dol01] Laurent Doldi. *SDL Illustrated – Visually design executable models*. Number ISBN 2-9516600-0-6. Doldi, 2001.
- [SDL99] Specification and description language (SDL). Recommendation Z.100, International Telecommunication Union, 1999.
- [Soc] SDL Forum Society. SDL forum society home page.
[http ://www.sdl-forum.org/](http://www.sdl-forum.org/).
- [UML03] *UML 2 illustrated : developping real-time and communication systems*. Number ISBN 2-9516600-1-4. TMSO, 2003.

1.1 Historique

1968 premières études

1976 *Orange Book SDL* – langage graphique élémentaire

1980 *Yellow Book SDL* – définition de la sémantique des processus

1984 *Red Book SDL* – types de données, premiers outils

1988 *Blue Book SDL* – concurrence, compositions, outils effectifs

⇒ SDL-88 : langage simple mais sémantique formelle

1992 *White Book* – approche objet ⇒ SDL-92 : langage complexe

1995 Ajout de ASN.1 à SDL (Z.105)

1996 corrections et ajouts mineurs

⇒ SDL-96

2000 intégration avec UML

⇒ SDL-2000

1.2 Les caractéristiques principales

- représentation graphique
- sémantique non ambiguë
- adapté aux systèmes parallèles communicants (donc distribués)
- plusieurs niveaux de granularité possibles
- environnements de développement, test, validation...

1.3 Le problème de la conception

Arriver à découper un système en sous-systèmes sans présumer de leur implantation matérielle.

Un travail d'ingénieur.

Analogie avec le bâtiment : concevoir, c'est faire les plans, coder, c'est monter les murs.

1.4 Phases d'utilisation dans le cycle en V

SDL peut être utilisé :

- en spécification
- en conception (architecturale et détaillée)
- en implantation (génération automatique de code)

Dans le langage de la norme :

Spécification : description du comportement souhaité

Description : description du comportement réel

Intérêt dans une spécification ou une conception

- Système ouvert : on peut simuler/exécuter un système partiellement décrit (l'outil demande à l'utilisateur)
- Syntaxe et compilateur \Rightarrow vérifie la cohérence de l'architecture
- Sémantique formelle \Rightarrow aucune ambiguïté avec client ou développeur
- Sémantique comportementale vs. diagrammes de séquence : les diagrammes de séquence décrivent des interactions possibles, mais pas les règles, donc pas ce qui est interdit.
- Le cycle en V a des soubresauts : spécifier et faire une conception de taille conséquence nécessitent des tests (syntaxiques et comportementaux) des spécifications et de la conception.
- Interfacé avec UML

Intérêt dans l'implantation

- Indépendance envers l'API de la cible (très variable pour prog. parallèle et communications)
- Codage d'automates fastidieux, source d'erreurs
- Suppose confiance dans le générateur...

Intérêt dans toutes les phases descendantes

On peut garder un même langage dans toute les phases descendantes du projet ^a !

On peut raffiner petit à petit le même modèle.

a. Insuffisant à lui tout seul, mais c'est déjà ça...

Utilisation dans les tests

Les outils offrent souvent :

- génération de jeux de tests (dont MSC)
- test de cohérence entre modèle SDL et diagrammes de séquence

1.5 Quel SDL dans ce cours ?

Norme actuelle SDL : SDL-2000

- peu outillée
- lourde, mal adaptée aux objectifs de ce cours

Un outils : RTDS 5.0

- sous-ensemble de SDL-92
 - adapté à nos besoins
- ⇒ certains aspects seront notés comme “non supportés” (†)

2 – Un langage graphique *et* textuel

SDL possède deux formats :

- SDL/PR (*Phrase Representation*) textuel
- SDL/GR (*Graphical Representation*) graphique

Il existe un format d'échanges entre outils : CIF (*Common Interchange Format* – Z.106), une représentation textuelle agrémentée de commentaires `/* CIF . . . */` qui spécifient le placement des symboles SDL.

3 – SDL et UML

Conception orientée composants *vs* Conception orientée objet

Les deux font de l'abstraction, en publiant des interfaces :

- COO : un objet offre une liste de méthodes
- COC : un composant
 - accepte / requière des messages
 - émet / offre des messages
- ADL (Architecture Description Language) : Composant / Connecteur


Les deux font une distinction entre processus/démon et fonction/ressource.

- UML : Objets actifs *vs* passifs
- SDL : Processus *vs* procédure

A - Un SDL minimal

Architecture statique – Types de base –
Comportements basiques

4 – Règles lexicales

- SDL-2000 est sensible à la casse, pas les versions antérieures (RTDS 5.0 l'est dans certains contextes)
- les identificateurs peuvent contenir des lettres, des chiffres, le souligné `_` mais aussi `#`, `@`, `\`
- il existe plusieurs formes de commentaires
 - à la C, entre `/* ... */` : `/* Un commentaire */`
 - une chaîne entre `'` qui suit le mot clef `comment` : `COMMENT 'un autre commentaire'`
 - symbole graphique du `COMMENT` est : 

5 – Architecture d'un modèle

5.1 La décomposition

Un modèle SDL est composé d'un niveau *système*, composé d'un ou plusieurs *blocs*, eux même composés de blocs ou de processus ^a.

La visibilité des types et constantes est classique (on voit les déclaration de ses parents).

Les entités communiquent par files de messages, nommés *route de signaux* (*signal route*) si une extrémité est un processus, *canal* (*channel*) sinon.

Ces files sont typées.

Il n'y a pas de variable partagée !

a. le "ou" est ici exclusif !

5.2 Déclaration de signaux (sans paramètre)

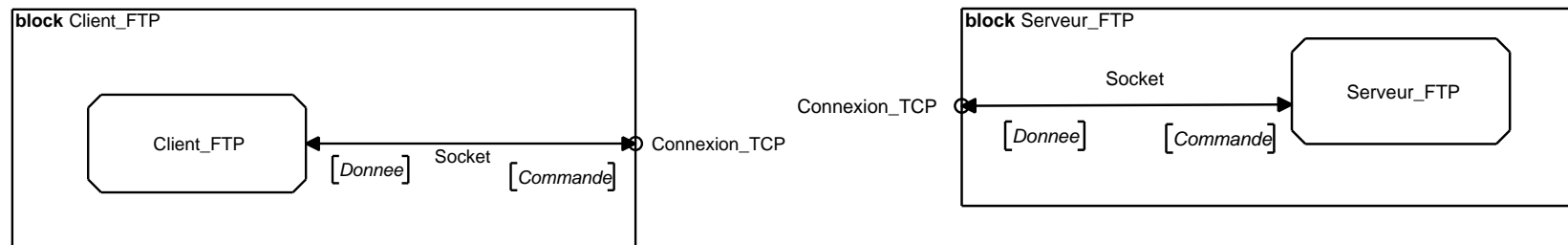
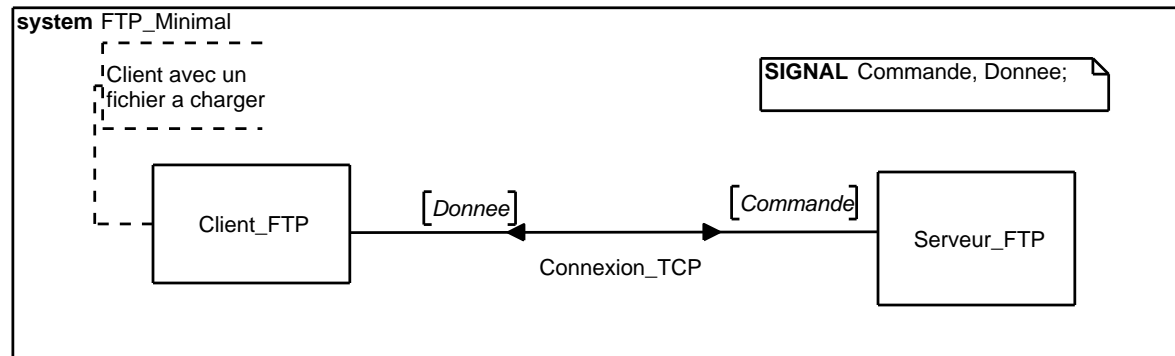
Dans un bloc de texte : `SIGNAL <ident> (, <ident>)*`;



SIGNAL Command, Data;

Dans ce cours, on distinguera un *signal* (qui est un type), et une *occurrence de signal* (qui est un “message” typé).

5.3 Un premier exemple







Sur ce premier exemple, on voit :

- architecture du système
- vérification de la cohérence des échanges

Erreur du débutant : oublier de nommer les connecteurs.

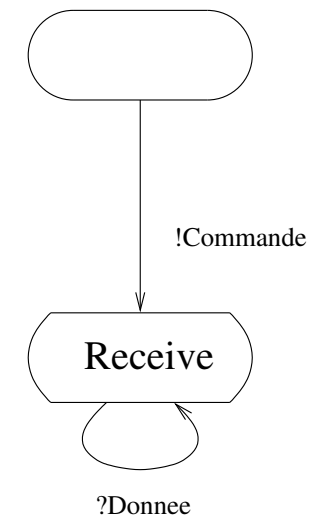
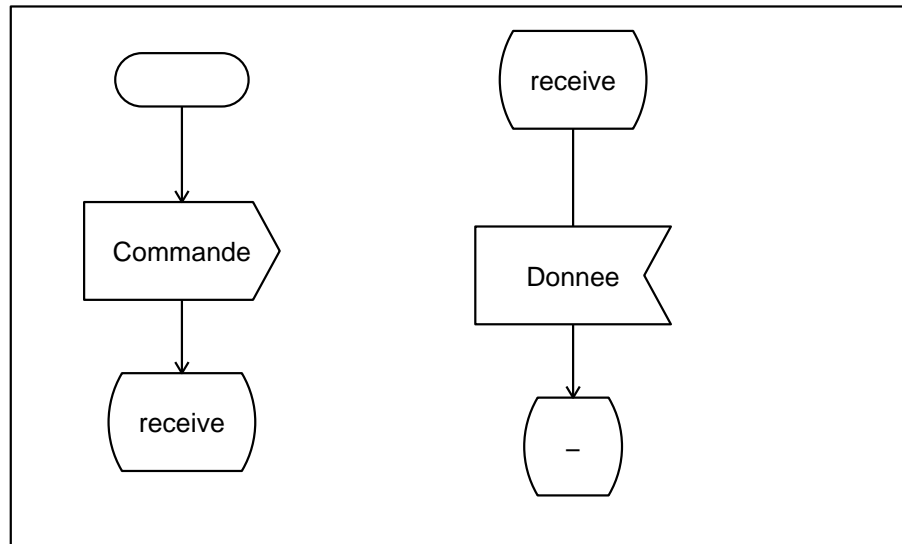
6 – Ajoutons des comportements : processus

Un processus est une machine à état (automate) où, en première approche, les transitions sont conditionnées par la réception d'une occurrence signal, et peuvent engendrer des occurrences de signaux.

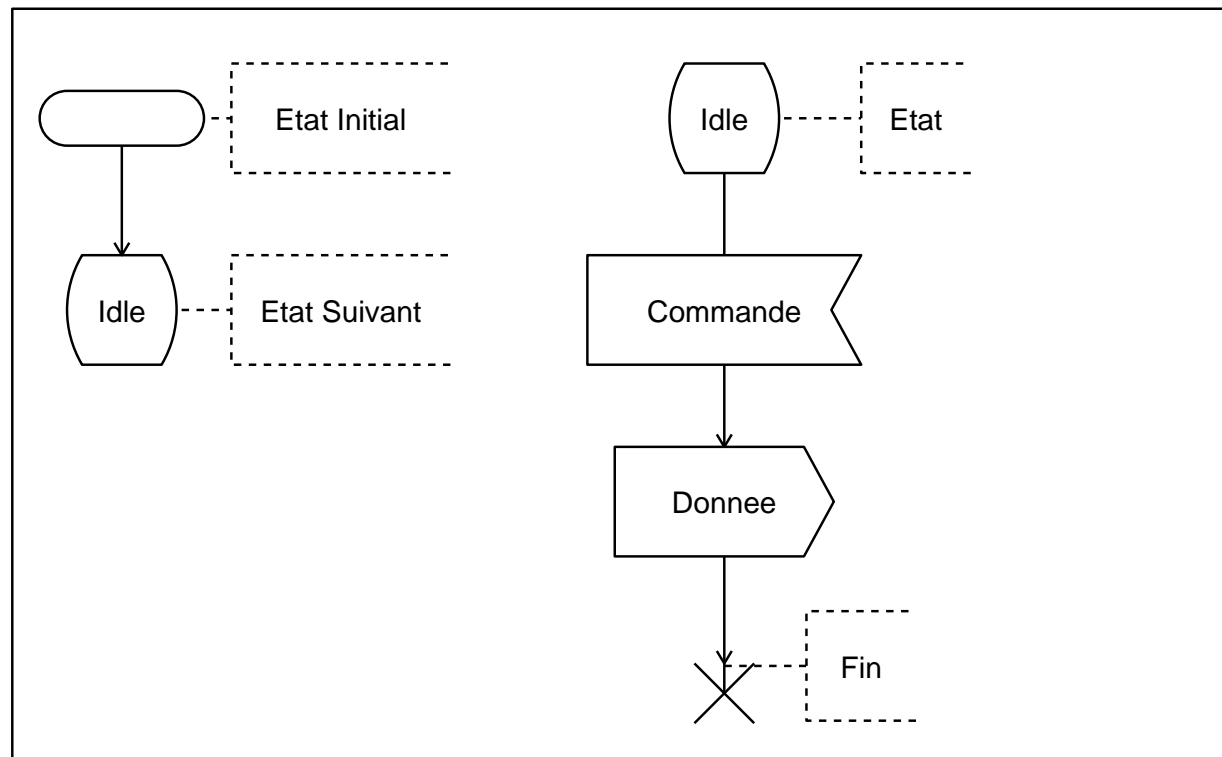
- Symbole d'état :  (et état initial 
- Symbole d'envois : 
- Symbole de réception : 
- le *type* de l'occurrence de signal est noté dans le symbole

On représente l'automate comme un ensemble de transitions et non comme un graphe connexe.

6.1 Exemple : processus du client



6.2 Exemple : processus du serveur



Attention à ne pas confondre :


- état initial (unique, anonyme)
- état (nommé, sert de définition)
- état suivant (référence à un état)

6.3 Gestion des signaux dans les files

Les occurrences de signaux qui arrivent à un processus^a sont stockés dans une FIFO.

Dans un état donné, seul un certain nombre de signaux peuvent être lus^b.

Par défaut, le système détruit tous les messages en tête de file non attendu !

⇒ introduction d'une possibilité de sauvegarde d'un ensemble de signaux :
symbole  .

a. Par une *signal route*

b. ceux pour lesquels l'état courant possède une réception

Lecture prioritaire

On peut briser l'aspect FIFO en décidant que certaines lectures sont prioritaires sur d'autres.

On utilise le symbole : .

Les occurrences de signaux dans la FIFO avant l'occurrence lue sont conservées.

7 – Ajoutons quelques données

7.1 Types de base

Boolean True, False

Character 'A', '1', '-', classique, sans accents

deux opérateurs : Num et Chr permettent de convertir entier en caractère et inversement

Integer : entier relatif (bornes non définies)

Natural : entier naturel (bornes non définies)

Real : flottant (précision et bornes non définies)

Charstring : chaîne de caractères 'exemple de chaîne'

Pid : identificateur de processus, avec seulement deux opérateurs = (égalité), /= (différence)

Time, Duration : utilisé avec les timers (échelle de temps non spécifiée)

7.2 Déclaration de variables

Dans un bloc texte,  :

DCL <ident> (, <ident>)* <type> (, <ident> (, <ident>)* <type>)* ;

process declarations_type_base

DCL a Character;
DCL b Character, i Integer;
DCL c, d Character, n, m Natural;

7.3 Déclaration de constantes

Dans un bloc texte :

SYNONYM <ident> <type> = <constante littérale> ;

8 – Les comportements

Ajoutons sur nos transitions de quoi faire des conditions, des boucles, etc.

8.1 Atomicité

Le tir d'une transition est dit *atomique* dans le sens où aucune transition ne peut commencer à être tirée si une autre ailleurs dans le système est encore en cours d'exécution.

Remarque : cette hypothèse est difficile à assurer dans l'implémentation...

8.2 Signaux transportant des valeurs

Les occurrences de signaux peuvent transporter des valeurs.

Il faut pour cela définir les types des paramètres lors de la définition des signaux :

SIGNAL Data(**Integer**, **Boolean**);

SIGNAL Ack(**Boolean**);

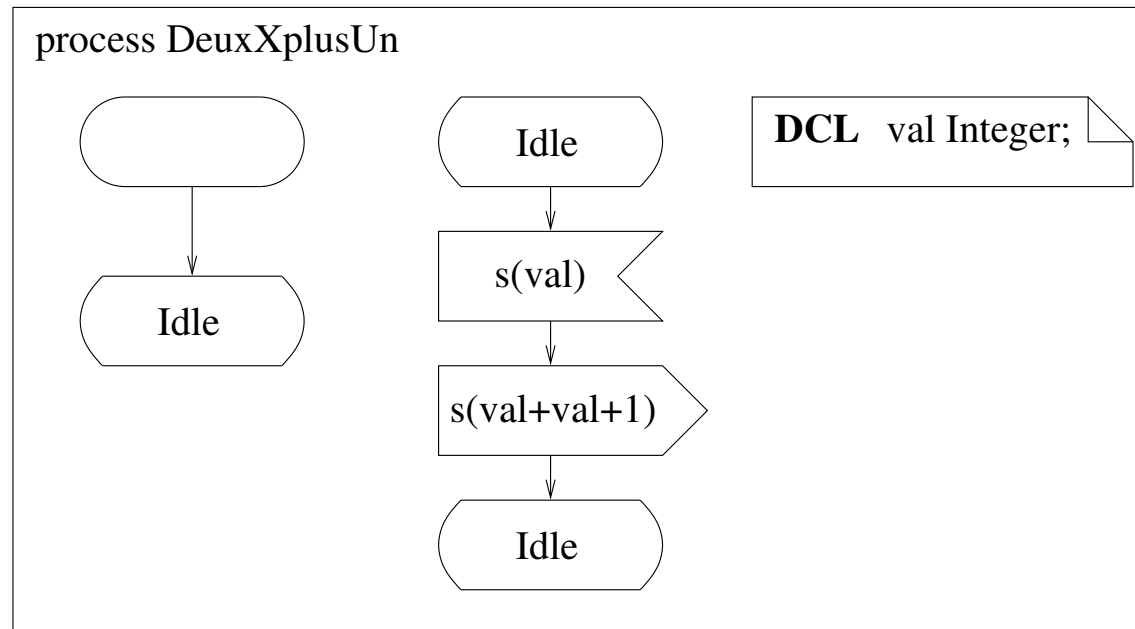
Quand on émet une occurrence de signal avec des paramètres, il faut préciser leur valeur.

Quand on lit une occurrence de signal, il faut passer les variables dans lesquelles les valeurs transportées seront stockées.

Exemple

Avec dans le système la déclaration :

SIGNAL s(**Integer**) ;



8.3 Signal Continu

On peut quitter un état sur un test sur une valeur : c'est un *signal continu* (*continuous signal*).

$$\langle x < 8 \text{ PRIORITY } 0 \rangle$$
$$\langle x > 8 \text{ PRIORITY } 2 \rangle$$

Mais, les signaux continus sont toujours moins prioritaires que les signaux.

L'idée est que dans un système réactif, les requêtes extérieures sont plus prioritaires que les évolutions internes.

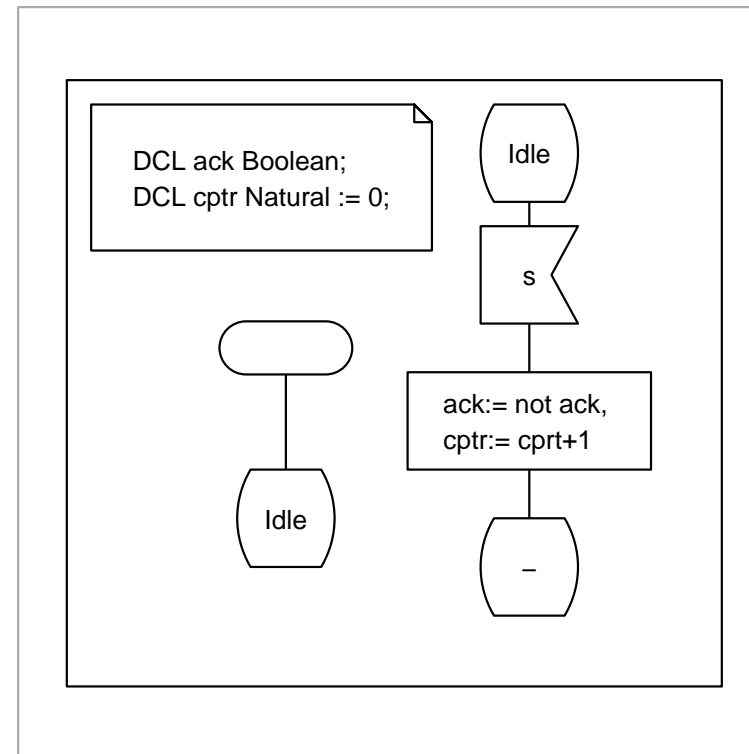
Bug RTDS 5.0 : la priorité est obligatoire.

8.4 Tâche (action)

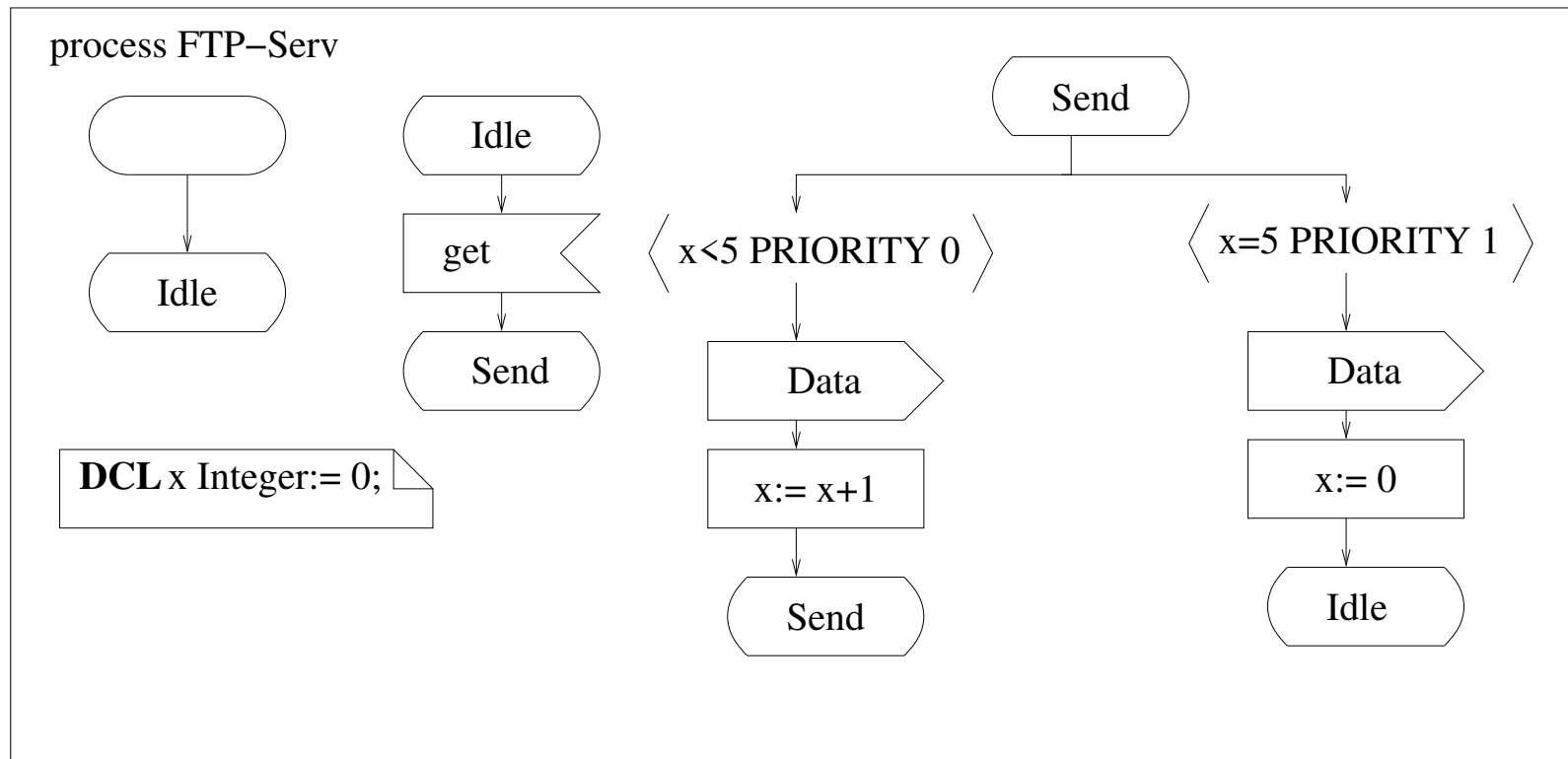
On peut mettre, après l'entrée d'une transition, une tâche (*task*) : Tache

L'action la plus courante est l'affectation, on utilise `:=`. Pour mettre plusieurs affectations dans un bloc tâche, on utilise la virgule `,` comme séparateur^a !

^a. et non le point-virgule ;



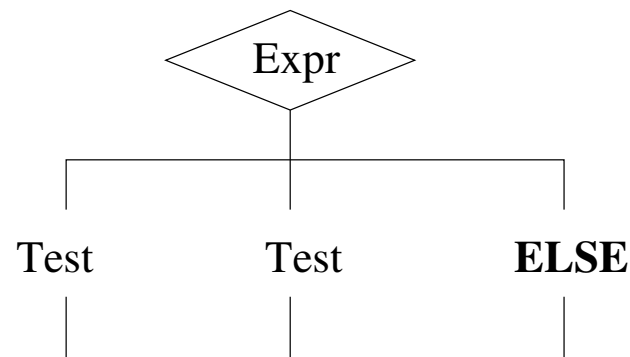
Avec ces deux éléments (signal continu + tâche), nous pouvons faire un peu mieux pour notre serveur FTP.



8.5 Décision

Après l'entrée d'une transition (réception signal ou signal continu), on peut introduire une *décision* qui permet de décider la suite de la transition.

Principe



Exemples



8.6 Expression conditionnelle

Il existe un **IF THEN ELSE FI** utilisable dans les expressions (un peu comme le `?:` du C).

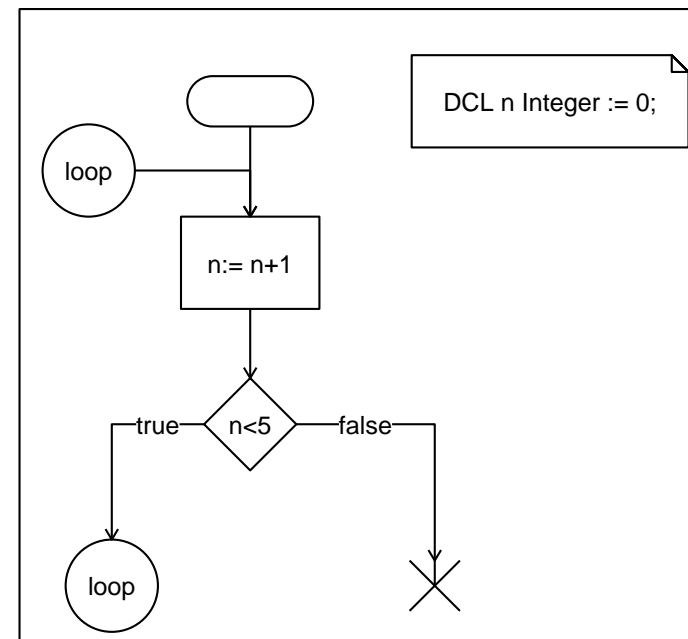
Exemple (dans un bloc tâche) :

```
n := IF (n < 16) THEN n+1 ELSE 0 FI
```

8.7 Branchements


On peut introduire à tout moment après l'entrée d'une transition, un
une *étiquette* (*label*), $\text{label} \rightarrow$, et en
fin de transition, un *branchement*

(*join*) $\downarrow \text{label}$.

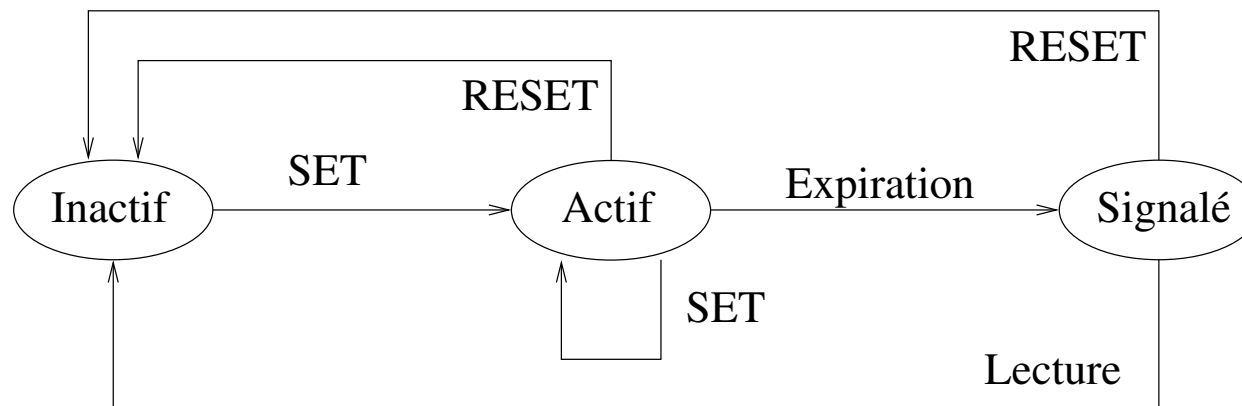


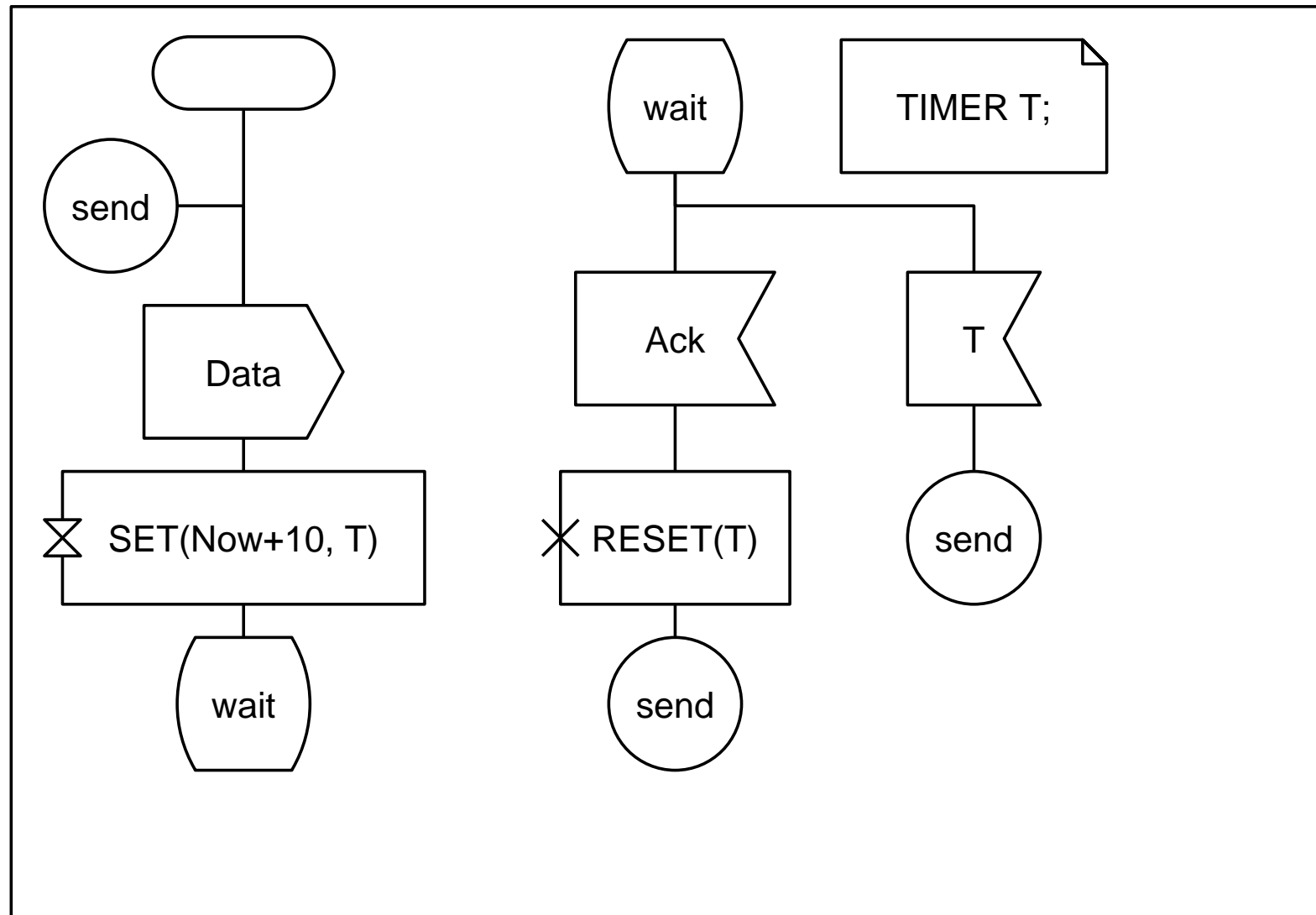
8.8 Timers

On peut définir une variable de type **TIMER**. Une telle variable a trois états : inactif, actif, signalé.

Quand on déclare un timer, il est *inactif*. On *active* un timer avec **SET**(<echeance>,<timer>) en lui donnant une date d'échéance. Quand la date d'échéance arrive, le timer passe en mode *signalé*. Le processus peut lire cette information avec le symbole de réception :  Timer .

Le cycle de vie complet est :





Remarques :

- on peut obtenir l'instant courant avec **NOW**
- on peut tester si un timer est actif : **ACTIVE**(<timer>)
- on peut faire des timers avec paramètres \dagger (^a)

a. Nous verrons comment on peut faire sans...

Timer avec paramètre(s) †

Quand on déclare une variable **TIMER**, dans la norme, on peut lui associer des paramètres, comme à un signal.

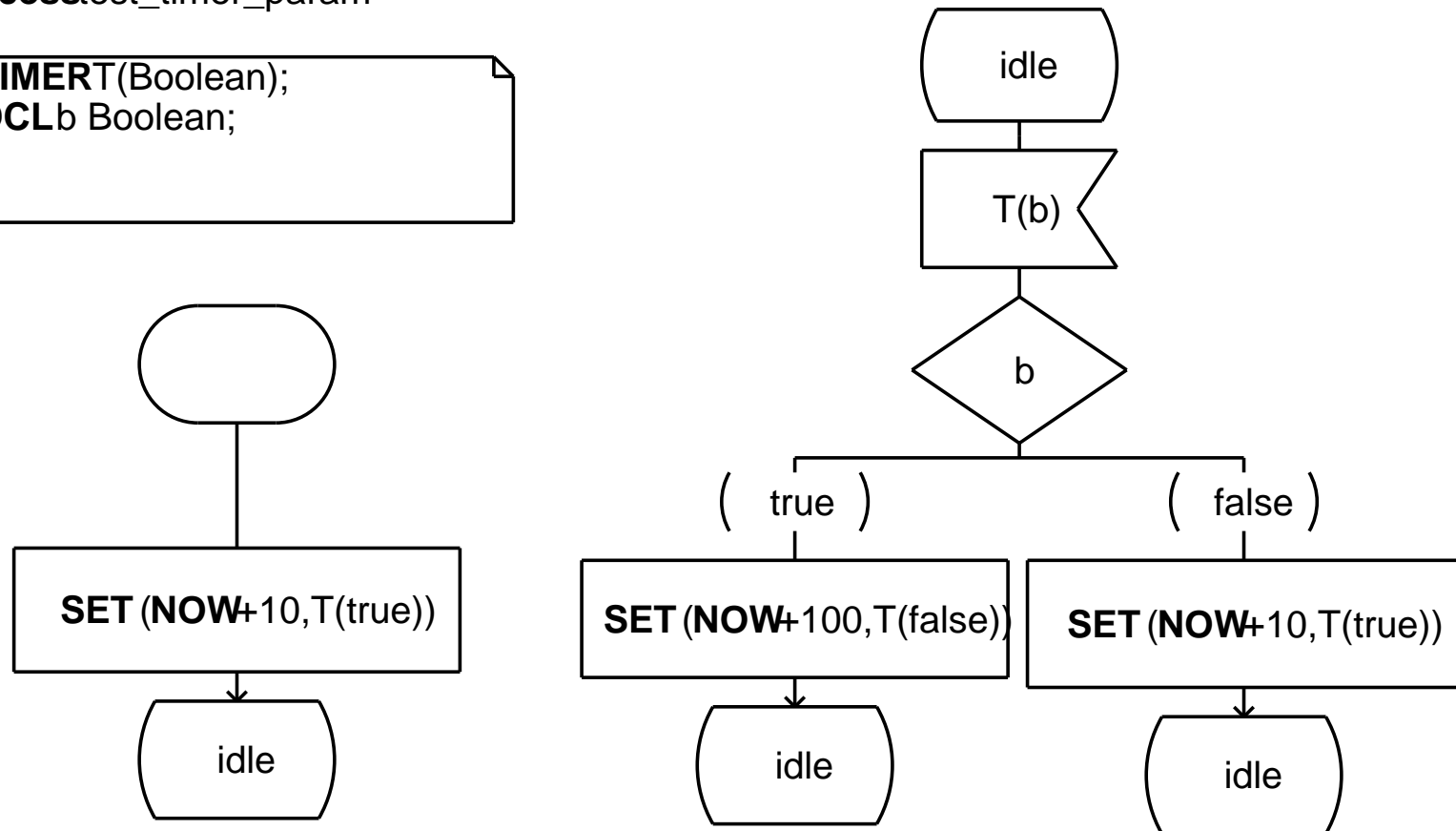
Avec RTDS 5.0 on ne peut pas déclarer de type **TIMER** avec paramètres.

Quand on ne peut pas avoir de timer avec paramètre, on peut :

- gérer un unique timer auquel est associé une liste de couples :
échéance - valeur
- créer dynamiquement des processus, chaque processus ayant un timer et une variable paramètre

processtest_timer_param

TIMERT(Boolean);
DCLb Boolean;

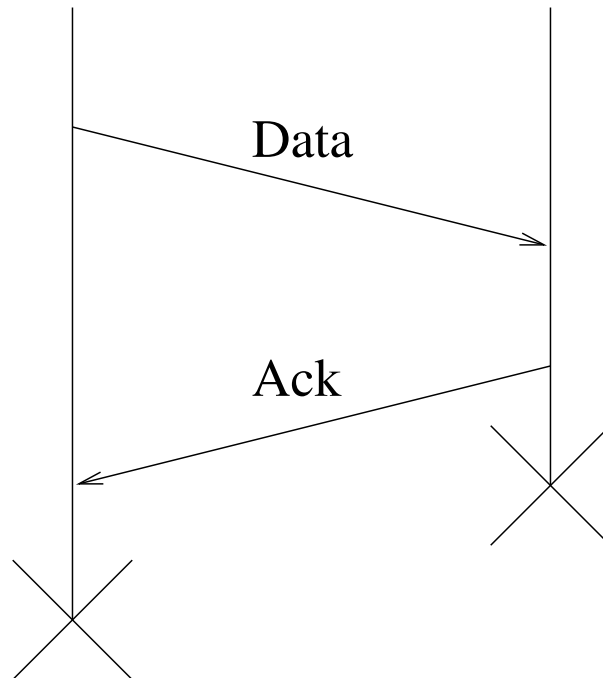


9 – Exercice

Écrire le système SDL qui permet cet échange simpliste.

Emetteur

Récepteur



B - SDL avancé

Types de données utilisateur – Appel de procédures –
Processus dynamiques – Chemins de donnée

10 – Types de donnés utilisateur

10.1 Liste de signaux

On peut définir des listes de signaux, afin d' :

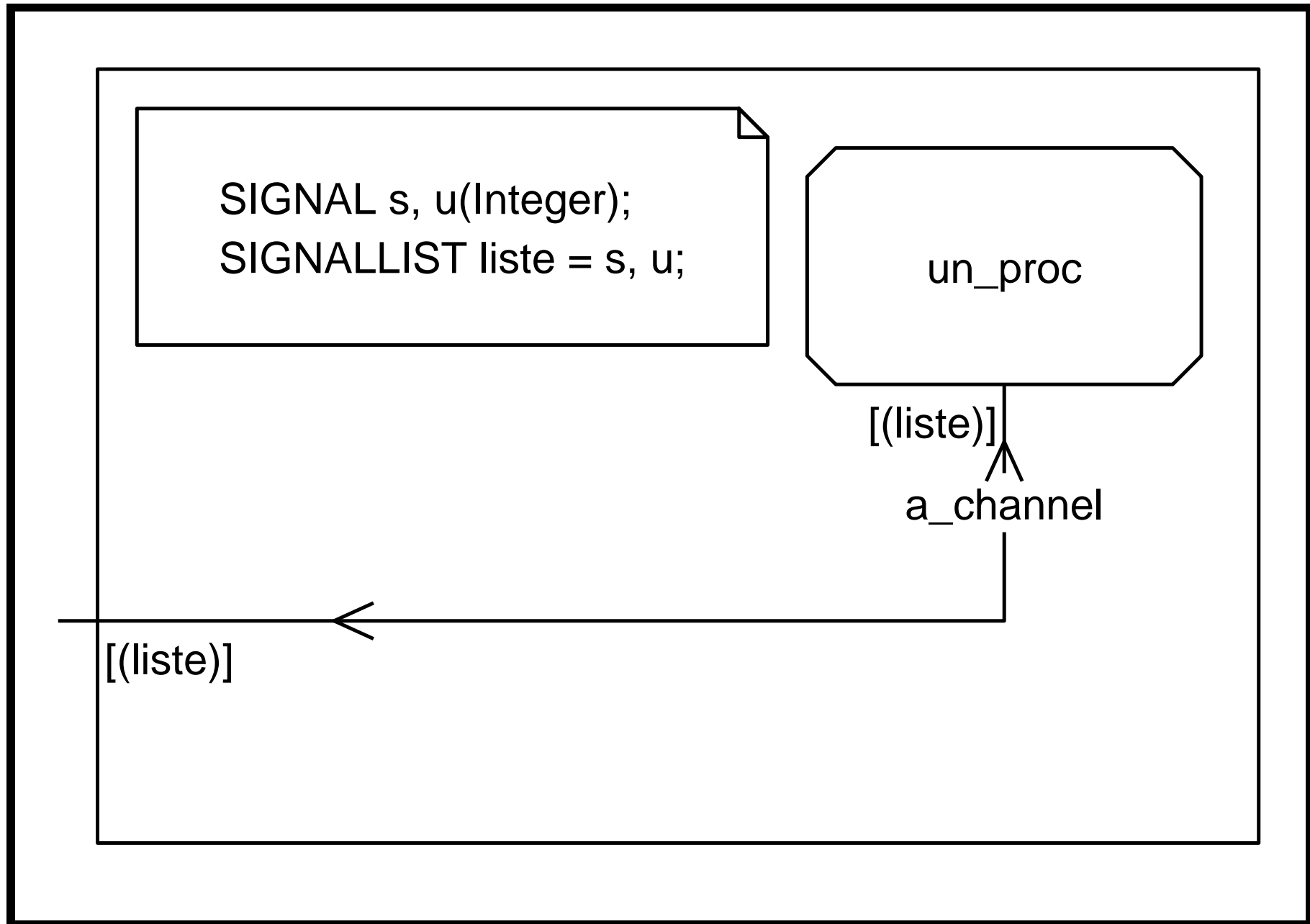
- alléger les écriture (quand on type les canaux)
- abstraire une communication (on ne connaît pas encore le détail des messages échangés)

SIGNALLIST <nom liste> = <element> (, <element>)* ;

avec

<element> := <nom signal> | (<nom liste>)

et dans les diagrammes, on écrit (<nom liste>)



10.2 Nouveau type : NEWTYPE

SDL permet de définir des types de façon très riche, en définissant la liste des opérations possibles sur le type (assez proche d'un objet en fait), mais nous n'allons en voir qu'une petite partie.

10.3 Type énuméré : NEWTYPE LITERALS

NEWTYPE <type name>

LITERALS <ident> (, <ident>)*****;

ENDNEWTYPE;

process enumeration

```
NEWTYPE logique  
  LITERALS vrai, faux, inconnu;  
ENDNEWTYPE ;
```

```
NEWTYPE Etats  
  LITERALS inactif, actif, signale;  
  OPERATORS ORDERING ;  
ENDNEWTYPE ;
```

10.4 Type enregistrement : NEWTYPE STRUCT

La même chose que le struct C.

NEWTYPE <type name>

STRUCT (<ident> <type> ;)*

ENDNEWTYPE ;

process test_struct

NEWTYPE PDU

STRUCT

seqNum Natural;

payload Integer;

last Boolean;

ENDNEWTYPE ;

DCL p PDU;

p!last:= false

On accède aux champs d'une variable de type enregistrement avec l'opérateur !.

10.5 Sous-type contraint : SYNTYPE

On peut définir un sous-type contrainte à partir d'un autre type ordonné (supportant la comparaison).

```
SYNTYPE <new type> = <parent type> [DEFAULT  
<val>] CONSTANTS <range cond> ENDSYNTYPE;
```

où la mot clé **DEFAULT** définit une nouvelle valeur par défaut pour le type, et où la condition d'intervalle <range cond> a trois formes possibles :

- <constante> : <constante>
- <constante>
- <opérateur comparaison> <constante>

Une variable de sous-type peut être utilisée partout où une variable du type parent était attendu, et inversement si on reste dans la plage de valeurs.

On peut omettre la contrainte.

Exemples :

```
SYNTYPE Indice = Integer CONSTANTS 0:7 ENDSYNTYPE;
```

```
SYNTYPE PosStrict = Natural DEFAULT 1 CONSTANTS > 0  
ENDSYNTYPE;
```

10.6 Utilisation de types génériques

Le langage SDL permet de définir des types génériques.

Pour utiliser des listes (**String**), des tableaux associatifs (**Array**), des multi-ensembles (**PowerSet**), il faut *instancier* des types génériques.

10.7 Liste : String

NEWTYPE <s-type>

String (<e-type>, <ident-s-empty>)

ENDSYNTYPE

Définit un nouveau type <s-type> qui est une liste d'éléments de type <e-type>, et la constante <ident-s-empty> qui est la liste vide de ce type.

Les opérateurs possibles sont :

MkString	: <e-type>	→ <s-type>	Liste d'un seul élément
Length	: <s-type>	→ Integer	Taille
First, Last	: <s-type>	→ <type>	Premier, Dernier
//	: <s-type>, <s-type>	→ <s-type>	Concaténation
(.)	: <s-type>, Integer	→ <type>	Accès à un élément
Substring	: <s-type>, Integer, Integer	→ <s-type>	Sous liste

Remarques :

- Les index commencent à 1 !
- Il existe un type **CharString**, pour les chaînes de caractères.

Exemple :

Dans un bloc de déclaration :

DCL nom **CharString**;

DCL c **Char**;

Dans un block tâche

nom := 'Boyer',

c := nom(1)

Limitation RTDS 5.0

Le constructeur de type **String** n'existe que partiellement dans RTDS 5.0
mais **CharString** existe.

10.8 Array

Le type **Array** n'est pas un tableau "à la C", mais un tableau *associatif* dont l'index peut être quelconque !

C'est comme les Map/HashMap de Java.

NEWTYPE <a-type>

Array (<key-type>, <content-type>)

ENDNEWTYPE

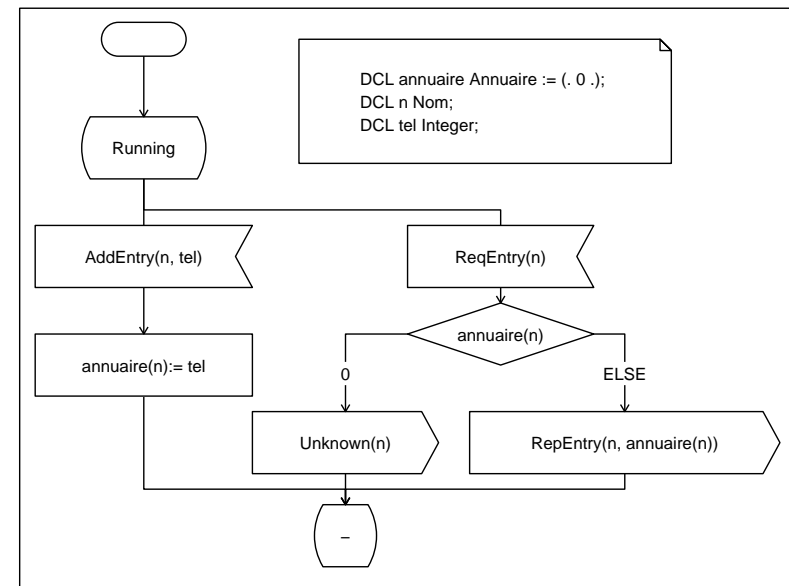
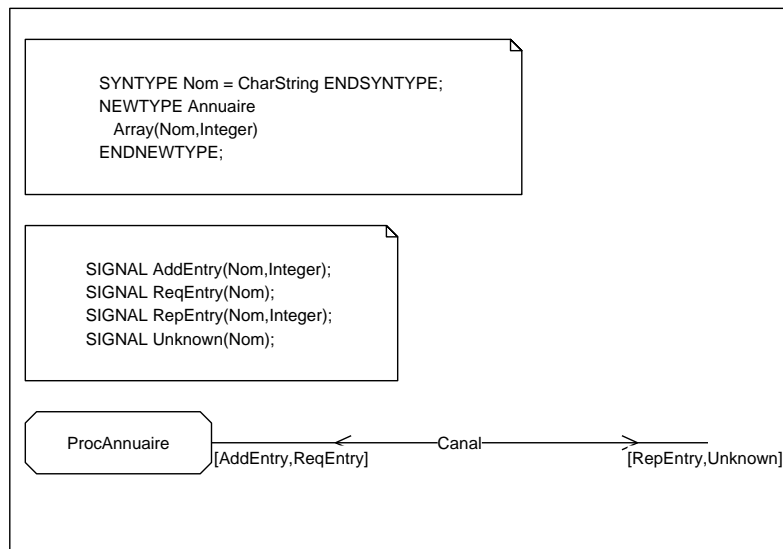
Particularités :

- L'accès aux éléments se fait avec $(-)$ (ex : $a(x) := 1$).
- Il existe une valeur par défaut (toute clé a une valeur).
- On peut initialiser le tableau à une valeur avec $(. \text{ <val> } .)$
ex : $a := (. \ 0 \ .)$

On peut faire des tableaux de tableau.

On ne peut pas mettre de structure en index †avec RTDS 5.0.

Exemple : annuaire



Second Exemple : revenir aux tableaux avec index entier finis

```
SYNTYPE LittleIndex = Integer CONSTANTS 0:4 ENDSYNTYPE;  
NEWTYPE LittleTable  
    Array(LittleIndex, Boolean)  
ENDNEWTYPE;
```

10.9 ANY

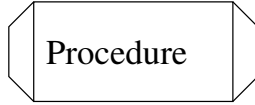
Avec les types constraints, on peut faire un choix aléatoire : **ANY**.

Exemple :

```
DCL x Integer;
```

On peut utiliser l'expression `x := ANY (Indice)`

11 – Procédures

- Il n'y a que des procédures en SDL, mais elles peuvent retourner une valeur.
- Une procédure est définie dans un bloc ou un processus .
- La signature de la procédure est définie dans un bloc de texte



avec la syntaxe :


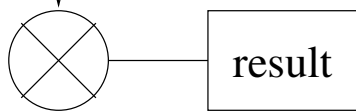

FPAR

<param> (, <param>)* ;

[RETURNS <type> ;]

avec

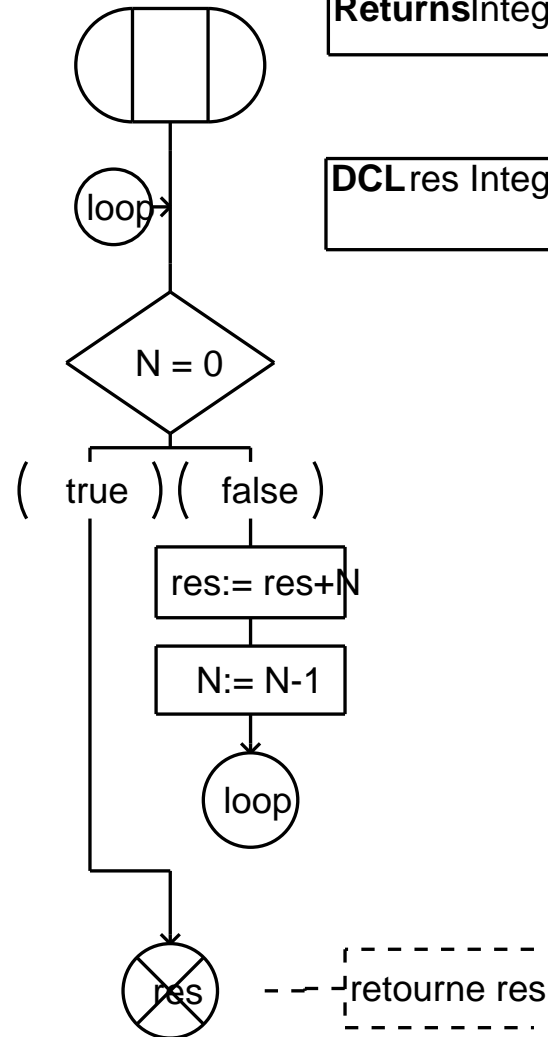
<param> := **[IN | IN/OUT]** <ident> <type>

- On décrit une procédure comme un processus, hormis
 - l'état initial 
 - la terminaison  (le texte associé est inutile si la procédure n'a pas de valeur de retour)
- L'appel de procédure se fait
 - dans n'importe quelle expression en la faisant précéder de **call**
(c-à-d on écrit `x := call f(y, z)` là ou en C/Java/... on aurait écrit `x = f(y, z) ;`)
 - dans un bloc spécifique 

procedure somme_1_N

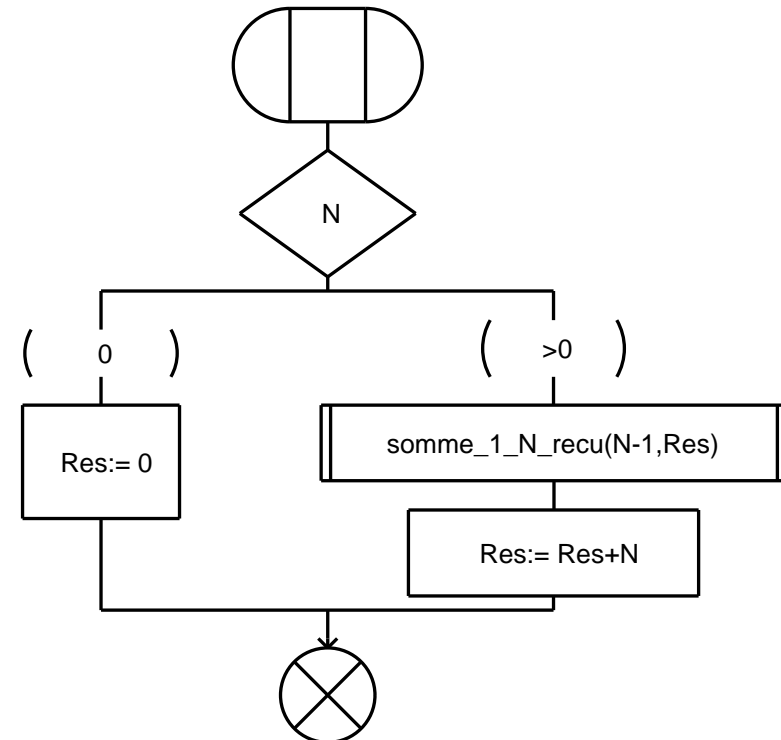
FPAR N Integer;
Returns Integer;

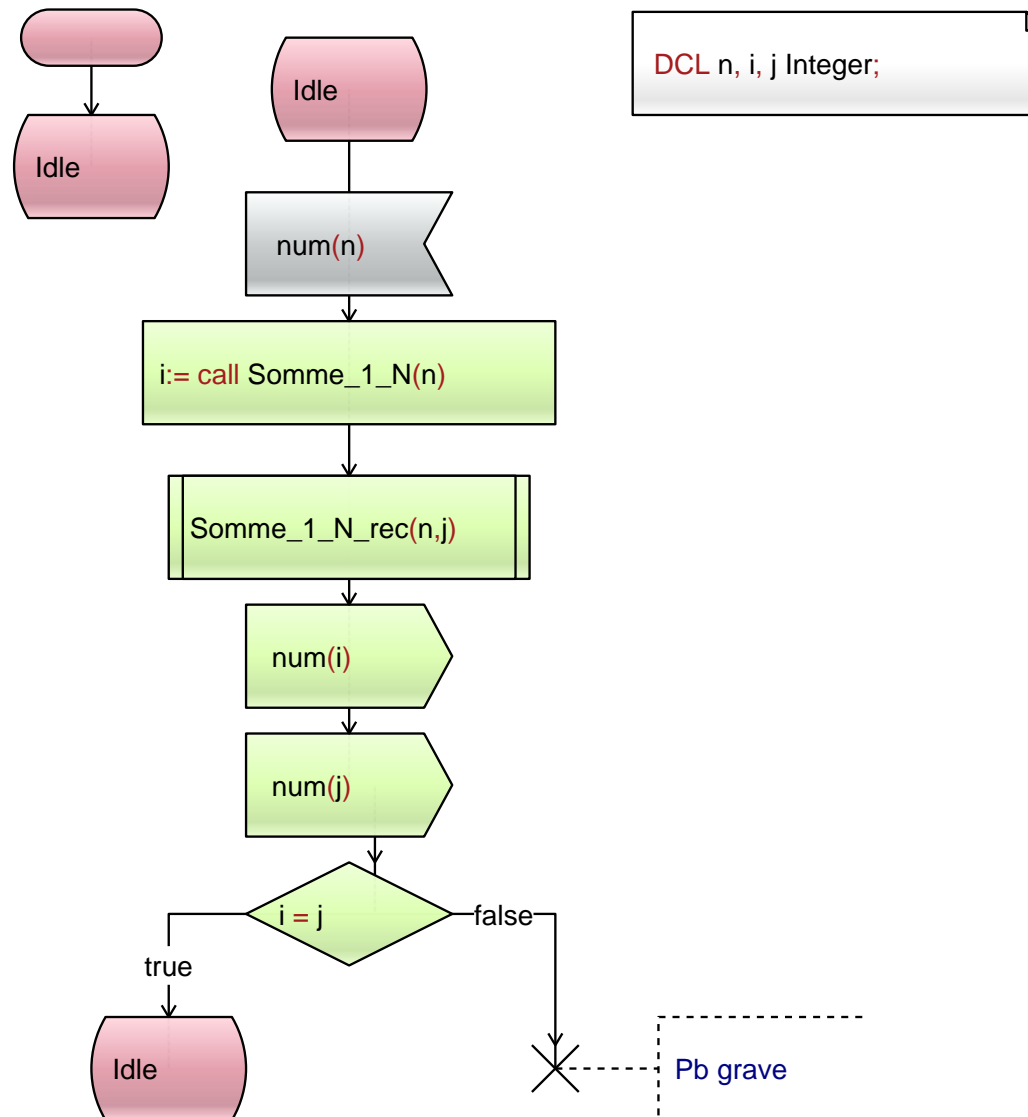
DCL res Integer := 0;



procedure somme_1_N_recu


FPAR
IN N Integer,
IN/OUT Res Integer;





12 – Processus dynamiques

Un processus peut exister en plusieurs instances.

Quand on crée un bloc processus , on peut spécifier deux valeurs, le nombre *initial* et le nombre *maximal* d'instances.

Comme une procédure, un processus peut avoir des paramètres formels (avec la même syntaxe **FPAR** qu'une procédure). Les valeurs sont indéterminées pour des processus dont des instances sont créées avec le système.

Un processus est toujours créé par un autre processus^a.

a. hormis les *initiaux*, créés avec le système

La création d'un processus se fait avec le symbole :

Type proc.

 . On peut mettre le mot clef **THIS** pour spécifier un processus de même type que le créateur.

Chaque processus possède plusieurs variables implicites de type **pid** pour manipuler des processus :

- **offspring** dans laquelle est stocké le PID du dernier processus créé, qui vaut **Null** si la création a échoué ;
- **self**, explicite
- **parent**, PID du processus créateur, **Null** si processus initial
- **sender**, PID du dernier processus dont on a lu un message

13 – Chemins de données

Dans les exemples vus, chaque processus avait une seule route en sortie, une seule en entrée et inversement, chaque route n'avait qu'un seul processus (ou connecteur) à chaque extrémité.

Dans la réalité, c'est plus complexe, et on peut donc préciser le destinataire d'une émission, avec deux mots-clés :

TO : sélectionne le(s) destinataire(s),

VIA : sélectionne la/les route(s)

Par défaut, si rien n'est spécifié, chacun prend une valeur possible parmi les choix offerts ^a.

a. définis par les routes de signaux incidentes

TO <ident> : sélectionne le(s) destinataire(s), suivant <ident>

- un PID : le récepteur est le processus avec ce PID
- un type de processus : le récepteur est un processus de cette classe
- **this** : le récepteur est un processus de la même classe que l'émetteur

VIA <ident> : sélectionne la/les route(s), suivant <ident>

- un nom de route : le signal est émis sur cette route

Il n'existe pas de *broadcast* en SDL (émission vers tous) mais uniquement une forme de *multicast* lié à la topologie.

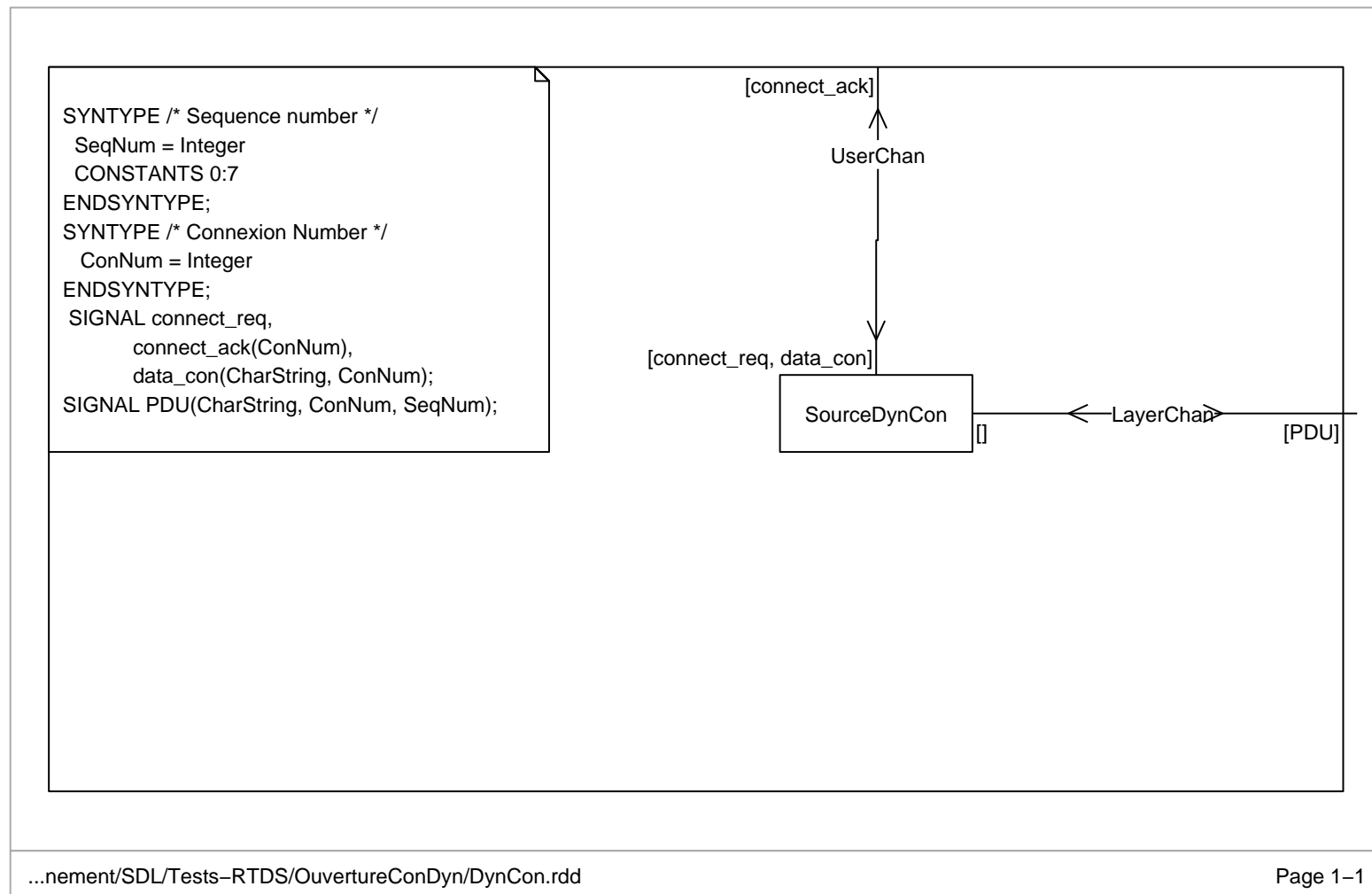
Conséquence : il est impossible d'envoyer un signal à toutes les instances dynamiques d'un processus.

14 – Exemple complet : gestion ouverture de connexions

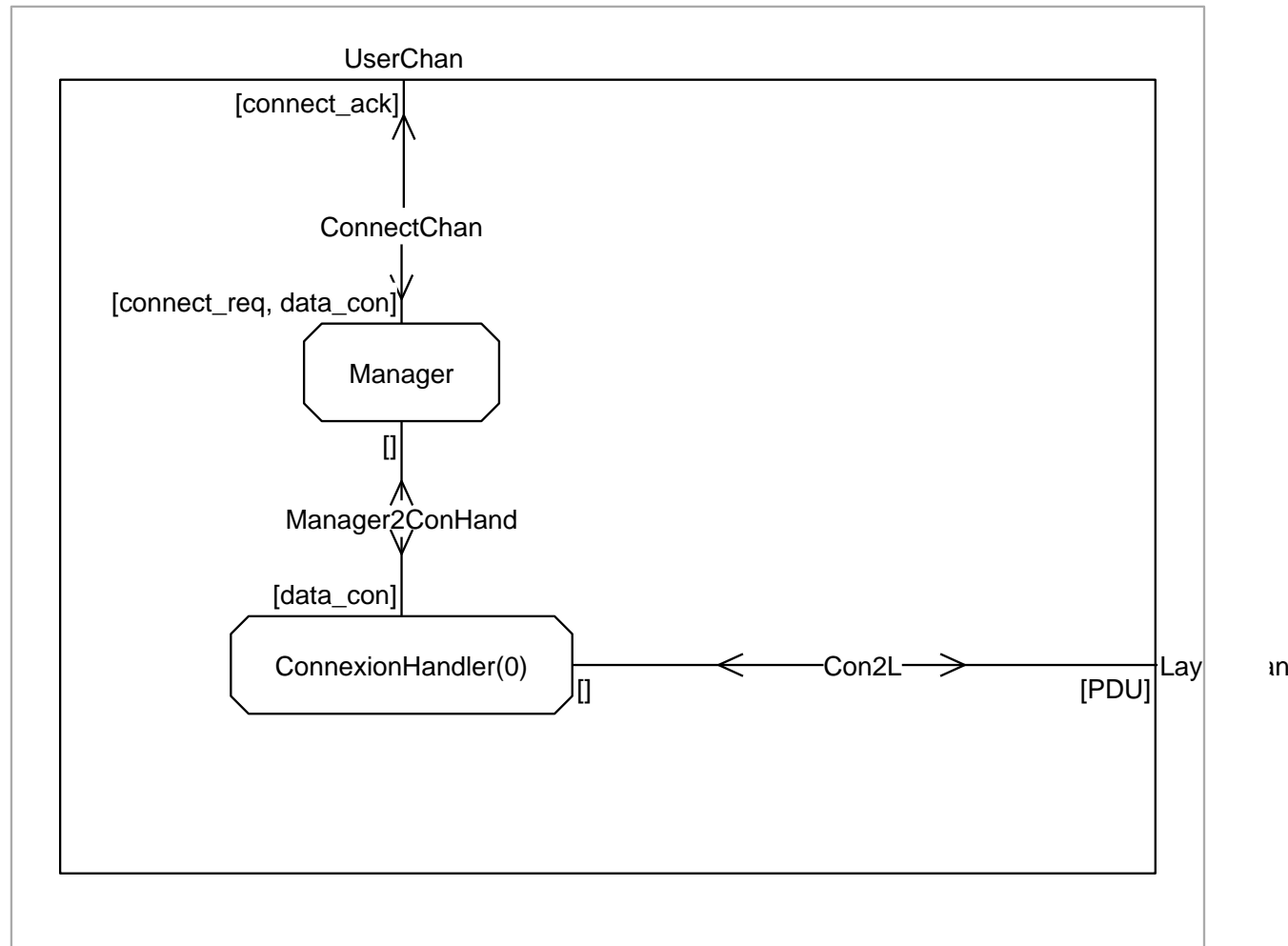
Principe :

- l'utilisateur envoie des demandes de connexion, et reçoit en retour un numéro de connexion
- l'utilisateur envoie ensuite les données en précisant le numéro de connexion
- lors de la création d'une connexion, un "Manager" crée un processus de gestion de cette connexion
- ce processus ajoute des numéros de séquence

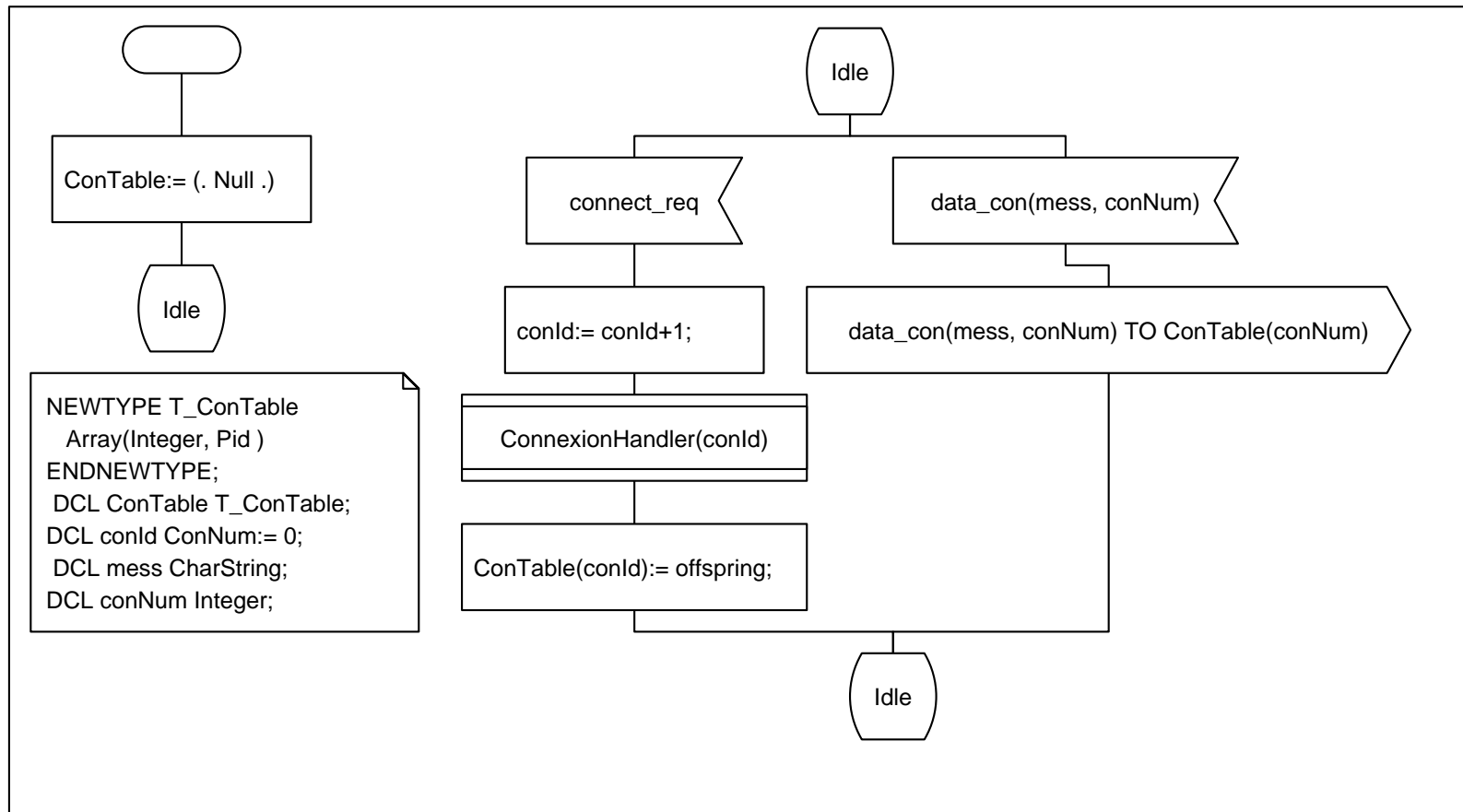
14.1 Le niveau Systeme



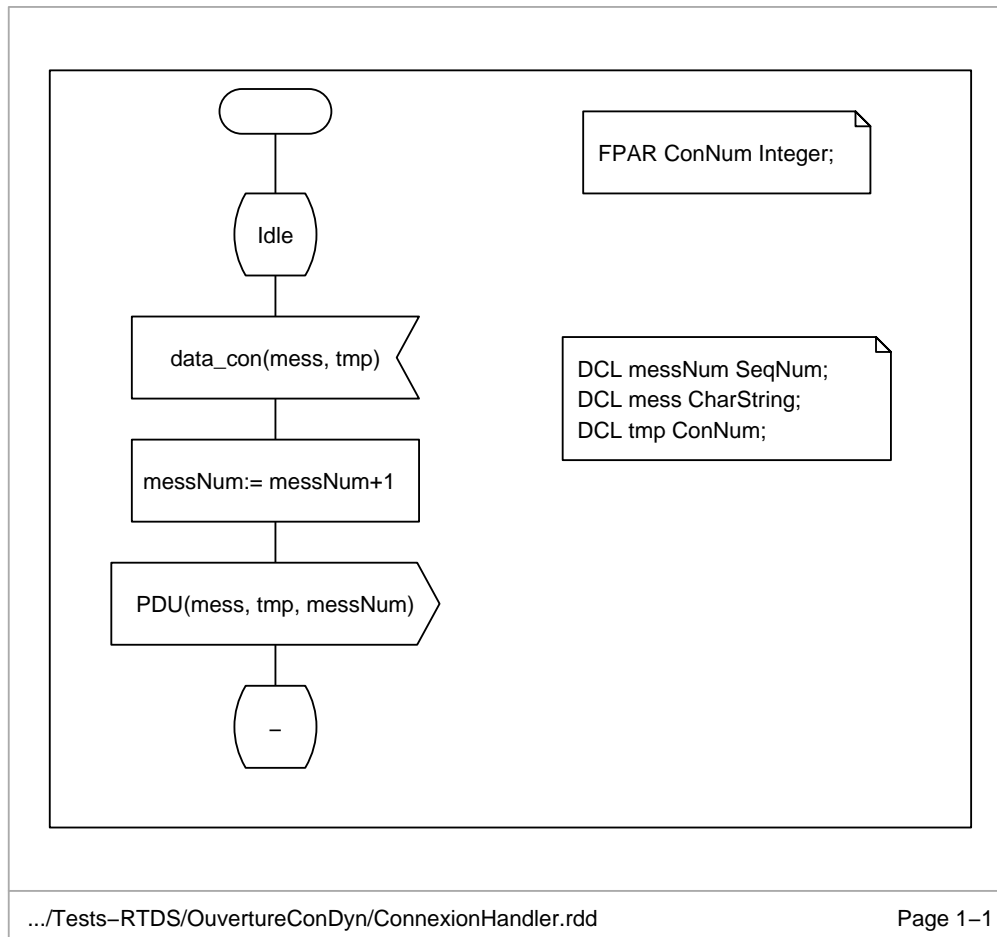
14.2 Le niveau Bloc



14.3 Le processus Manager



14.4 Le processus ConnectionHandler



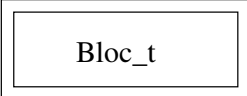
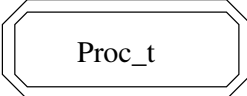
14.5 Il reste deux erreurs

Il reste dans cet exemple des erreurs de conception, et des maladresses...

15 – Type de bloc et processus

15.1 Déclaration et instantiation

Quand un élément (bloc ou processus) se répète à l'identique à différents endroits du système, on peut définir un *type* dont on créera ensuite des *instances*.

Les symboles graphiques sont les mêmes, mais avec un double trait (bloc type , processus type ).

On déclare des instances de ces types en donnant un nom de la forme :

`<nom instance> : <nom element type>`

15.2 Entrées/Sorties des types de blocs : portes (*gate*)

Quand on connecte un lien interne à un bloc à un canal externe, par un connecteur, on doit nommer le dit canal dans la définition du bloc^a.

Dans la définition d'un bloc type, on ne connaît pas encore le connecteur (puisque différentes instances seront sûrement connectées à différents canaux).

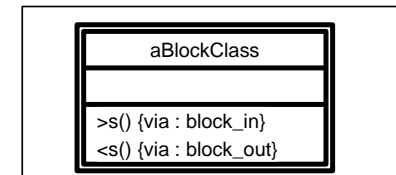
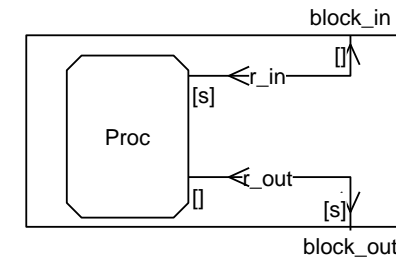
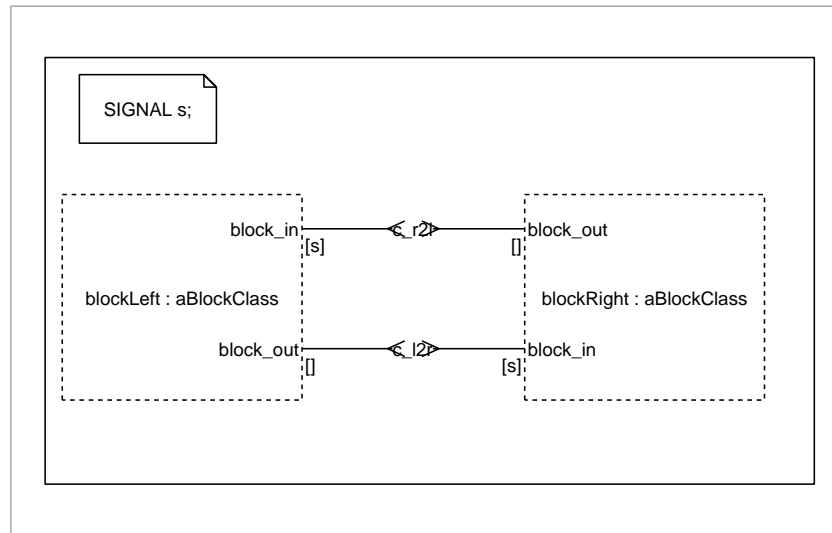
On utilise alors des portes (*gates*) au lieu des connecteurs.

Principe :

- une porte possède un nom, et des listes de signaux en entrée et sortie
- lors de l'édition d'un type de bloc, on crée des portes
- lors de l'instantiation d'un bloc de ce type, on crée des portes de même nom, et y relie les canaux

a. cf. l'erreur du débutant, T. 19

Exemple



Bug : RTDS 5.0 demande à ce qu'on crée un *Diagramme UML* associé à un *Block type* (cf dessin).

Rq : si un type signal est émis/reçu par plusieurs *gate*, la syntaxe est "s () {via:gate1, via:gate2}"

Bug2 : RTDS 5.0 n'aime pas les espaces dans la partie *via*.

15.3 Paquetages

SDL offre la possibilité de faire des “paquetages”. C’est en général là qu’on définit les types de blocs et de processus.

On dit à un système d’*utiliser* un paquetage XXX en mettant dans un bloc texte la commande `USE XXX;`.

16 – Critères de conceptions

La conception “système” est un problème extrêmement complexe.

De nombreux critères s’opposent :

- logiciel vs matériel
 - performances
 - volatilité du logiciel
- décomposition vs intégration
 - séparation des fonctions
 - besoin d’interfaces claires
 - synchronisation, exclusion mutuelle
 - séparation des responsabilités
 - maintenance

List of Slides

1 – Introduction	2
1.1 – Historique	4
1.2 – Les caractéristiques principales	5
1.3 – Le problème de la conception	6
1.4 – Phases d'utilisation dans le cycle en V	7
1.5 – Quel SDL dans ce cours ?	11
2 – Un langage graphique <i>et</i> textuel	12
3 – SDL et UML	13
A - Un SDL minimal	
4 – Règles lexicales	15
5 – Architecture d'un modèle	16

5.1 – La décomposition	16
5.2 – Déclaration de signaux (sans paramètre)	17
5.3 – Un premier exemple	18
6 – Ajoutons des comportements : processus	20
6.1 – Exemple : processus du client	21
6.2 – Exemple : processus du serveur	22
6.3 – Gestion des signaux dans les files	24
7 – Ajoutons quelques données	26
7.1 – Types de base	26
7.2 – Déclaration de variables	27
7.3 – Déclaration de constantes	27
8 – Les comportements	28

8.1 – Atomicité	28
8.2 – Signaux transportant des valeurs	29
8.3 – Signal Continu	31
8.4 – Tâche (action)	32
8.5 – Décision	34
8.6 – Expression conditionnelle	35
8.7 – Branchements	36
8.8 – Timers	37
9 – Exercice	42
 B - SDL avancé	
10 – Types de données utilisateur	44
10.1 – Liste de signaux	44

10.2 – Nouveau type : NEWTYPE	45
10.3 – Type énuméré : NEWTYPE LITERALS	47
10.4 – Type enregistrement : NEWTYPE STRUCT	48
10.5 – Sous-type contraint : SYNTYPE	49
10.6 – Utilisation de types génériques	50
10.7 – Liste : String	51
10.8 – Array	53
10.9 – ANY	56
11 – Procédures	57
12 – Processus dynamiques	61
13 – Chemins de données	63
14 – Exemple complet : gestion ouverture de connexions	65

14.1 – Le niveau Systeme	66
14.2 – Le niveau Bloc	67
14.3 – Le processus Manager	68
14.4 – Le processus ConnectionHandler	69
14.5 – Il reste deux erreurs	70
15 – Type de bloc et processus	71
15.1 – Déclaration et instantiation	71
15.2 – Entrés/Sorties des types de blocs : portes (<i>gate</i>)	72
15.3 – Paquetages	74
16 – Critères de conceptions	75