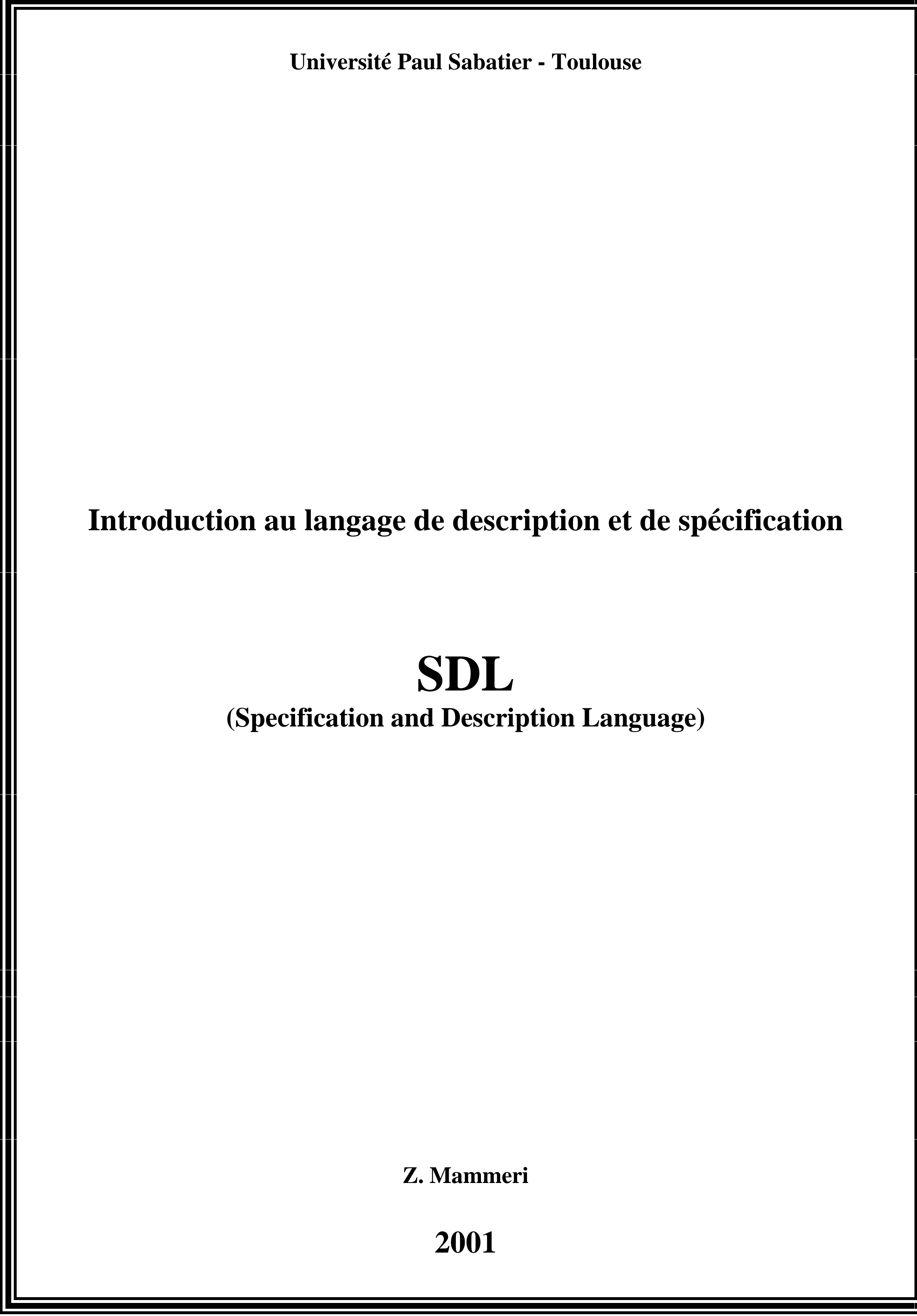

Université Paul Sabatier - Toulouse

Introduction au langage de description et de spécification

SDL

(Specification and Description Language)

Z. Mammeri

2001

1. Généralités

1.1 SDL : c'est quoi ?

En 1972, le CCITT (Comité Consultatif International Téléphonique et Télégraphique ou « Consultative Committee for International Telegraph and Telephony »), appelé ITU (International Telecommunication Union) depuis 1993, a mis en place un groupe de travail pour la définition d'un langage de spécification pour l'industrie des télécommunications. Le langage (et la méthode aussi) proposé par ce groupe de travail fût appelé SDL (Specification and Description Language). La première version de SDL a vu le jour en 1976, suivie ensuite par une nouvelle version tous les quatre ans. La dernière version de SDL date de 2000 ; elle est référencée norme **ITU-T Z.100**.

SDL ou LDS en français (Langage de Description et Spécification) est l'une des trois techniques de description formelle ou FDT (Formal Description Technique) adoptées par les instances de normalisation internationales ITU et ISO. Les deux autres techniques sont LOTOS (Language of Temporal Ordering Specification) et ESTELLE.

SDL est fondé sur les principes de l'orienté objet. L'existence d'outils (tels que *ObjectGeode*, commercialisé par la société Verilog en France) permettant d'aller de la spécification à la génération de code, rend SDL plus attractif dans le milieu industriel contrairement à d'autres méthodes formelles où les outils font défaut.

Le but poursuivi, en recommandant l'utilisation de SDL, est d'avoir un langage permettant de spécifier et de décrire sans ambiguïté le comportement de systèmes englobant des activités parallèles et communicantes. Les spécifications et les descriptions faites à l'aide de SDL doivent être formelles dans ce sens qu'il doit être possible de les analyser et de les interpréter sans ambiguïté. Les termes spécification et description sont utilisés dans le sens ci-après :

- la spécification d'un système est la description du comportement souhaité de celui-ci,
- la description d'un système est la description du comportement réel de celui-ci.

Etant donné qu'il n'est pas fait de distinction entre l'utilisation de SDL pour la spécification et son utilisation pour la description, le terme spécification est utilisé dans la norme SDL (Z.100) pour désigner à la fois le comportement souhaité et le comportement réel d'un système.

1.2 Objectifs

Les objectifs généraux qui ont été pris en compte lors de la définition de SDL sont de fournir un langage :

- facile à apprendre, à utiliser et à interpréter ;
- permettant l'élaboration de spécifications dépourvues d'ambiguïté pour faciliter la soumission des offres et le partage des commandes ;
- extensible pour permettre un développement ultérieur ;
- permettant l'application de plusieurs méthodologies de spécification et de conception de système, sans supposer a priori l'une quelconque de ces méthodologies.

1.3 Domaine d'applications

Le domaine d'application principal de SDL est la description du comportement des d'applications composées d'activités concurrentes. Ces applications comprennent notamment :

- les systèmes de télécommunication (le traitement des appels, signalisation téléphonique, comptage aux fins de taxation, maintenance et relève des dérangements),
- les fonctions de gestion des réseaux de communication,
- les systèmes réactifs ou temps réel et systèmes critiques,
- les protocoles de communication et les systèmes distribués.

SDL est un langage particulièrement riche qui peut être utilisé à la fois pour des spécifications de haut niveau informelles (et/ou formellement incomplètes), des spécifications partiellement formelles et des spécifications détaillées. L'utilisateur doit choisir les parties appropriées de SDL en fonction du niveau de communication souhaité et de l'environnement dans lequel le langage sera utilisé. Selon l'environnement dans lequel une spécification est utilisée, certains éléments qui relèvent du simple bon sens pour l'émetteur et le destinataire de la spécification ne seront pas explicités. Ainsi, SDL peut être utilisé pour :

- établir les spécifications d'une installation,
- établir les spécifications d'un système,
- établir des recommandations de l'ITU,
- établir des spécifications de conception d'un système,
- établir des spécifications détaillées,
- concevoir un système (à la fois globalement et dans le détail),
- effectuer des essais d'un système.

1.4 Spécification d'un système

SDL décrit le comportement d'un système sous la forme de stimulus/réaction, étant admis que les stimuli aussi bien que les réactions sont des entités discrètes et contiennent de l'information. En particulier, la spécification d'un système est vue comme étant la séquence de réactions associée à une séquence de stimuli. Le modèle de spécification d'un système est fondé sur la notion de machine à états finis étendue. Un système SDL est composé d'un ensemble de **blocs** (figure 1). Les blocs sont connectés entre eux et à l'environnement par des **canaux**. A l'intérieur de chacun des blocs, il y a un ou plusieurs **processus**.

Un processus est une machine d'états finis étendue fonctionnant de manière concurrente et autonome avec les autres processus. Les processus communiquent de manière **asynchrone** par l'intermédiaire de **signaux**. Aucune synchronisation n'est requise entre l'émetteur d'un signal et son consommateur. Un processus peut envoyer (resp. recevoir) des signaux vers (resp. de) l'environnement ou les autres processus d'un même système. Les signaux reçus par un processus sont stockés dans une file de taille supposée non bornée.

SDL est fondé sur les machines d'états finis étendues et sur les types abstraits de données ou ADT (Abstract Data Type). Le concept de types abstraits de données (qui est utilisé aussi par LOTOS) permet d'avoir des spécifications de données rigoureuses et avec plusieurs niveaux d'abstraction.

SDL fait appel à des concepts structurels qui facilitent la spécification des grands systèmes et des systèmes complexes. Il est ainsi possible de subdiviser la spécification d'un système en unités faciles à gérer qui peuvent être traitées et comprises de manière indépendante.

1.5 Grammaires de SDL

SDL offre le choix entre deux formes syntaxiques différentes pour représenter un système : une représentation graphique (SDL/GR : « SDL Graphical Representation ») et une représentation textuelle (SDL/PR : « SDL Phrase Representation »). Comme ces formes sont toutes les deux des représentations concrètes de la même sémantique de SDL, elles sont équivalentes du point de vue sémantique. Un sous-ensemble de SDL/PR est commun avec SDL/GR. Ce sous-ensemble est appelé grammaire textuelle commune.

Remarque : ce document s'intéresse particulièrement à la grammaire textuelle.

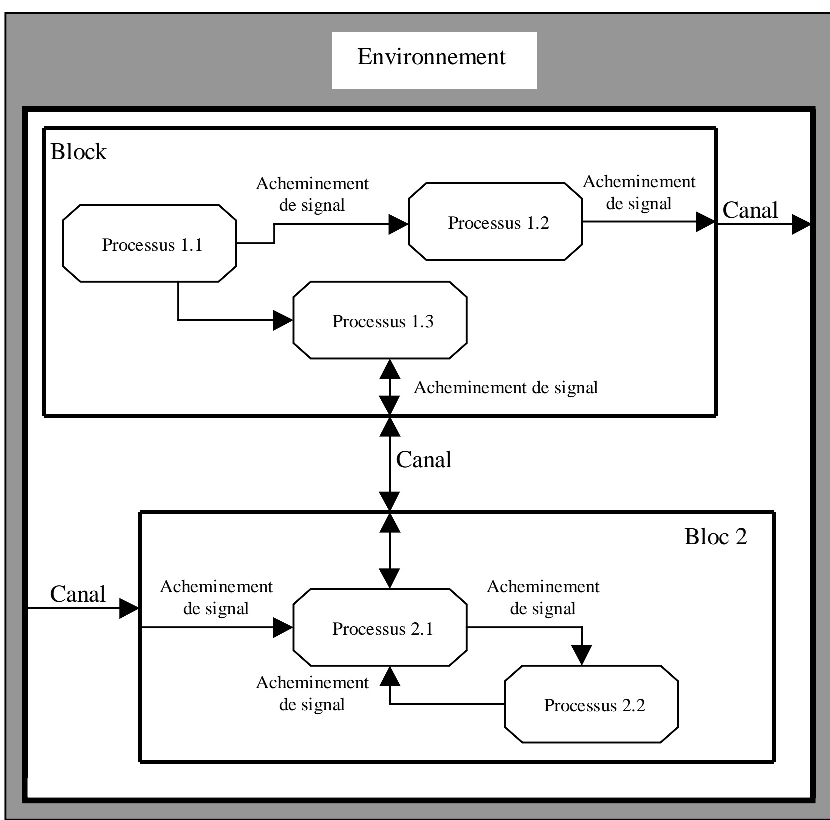

Figure 1. *Structure générale d'une spécification de système en SDL*

1.6 Définitions fondamentales

Un **type** (par exemple : un type de processus), se décrit par des définitions, lesquelles définissent toutes les propriétés associées à ce type. Un type peut être décomposé en un nombre quelconque d'**instance**s. Toute instance d'un type particulier possède toutes les propriétés définies pour ce type. Ce schéma s'applique à plusieurs concepts de SDL, c'est-à-dire qu'il existe des définitions de système et des instances de système, des définitions de processus et des instances de processus, etc.

Les systèmes qui sont spécifiés en SDL réagissent d'après les stimuli qu'ils reçoivent du monde extérieur. Ce monde extérieur est appelé **environnement** du système en cours de spécification. On suppose qu'il y a une ou plusieurs instances de processus dans l'environnement et, par conséquent, les signaux circulant de l'environnement en direction du système ont des identités associées à ces instances de processus.

2 SDL de base

Dans cette section, nous allons présenter les règles syntaxiques pour décrire des spécifications SDL. Nous utilisons les règles syntaxiques suivantes :

`un mot en gras`	:	désigne un mot-clé du langage SDL,
`un terme entre < et >`	:	désigne un non-terminal,
`|`	:	désigne une alternative,
`[ ]`	:	désigne une option,
`{ }*`	:	désigne un groupe de termes pouvant être vide,
`{ }+`	:	désigne un groupe de termes contenant au moins un élément.

2.1 Règles générales

Les règles lexicales définissent des unités lexicales. Les unités lexicales sont les symboles terminaux de la grammaire textuelle.

```
<unité lexicale> ::=
       <mot> | <chaîne de caractères> | <spécial>
       | <spécial composé> | <note> | <fin> | <mot clé>

<mot> ::=
               { <alphanumérique> | <point> }* <alphanumérique>
               { <alphanumérique> | <point> }*

<alphanumérique> ::=
       <lettre> | <chiffre décimal> | <national>

<lettre> ::=
       A | B | C | D | E | F | G | H | I | J | K | L | M |
       N  | O | P | Q | R | S | T | U | V | W | X | Y | Z
       | a | b | c | d | e | f | g | h | i | j | k | l | m
       | n  | o | p | q | r | s | t | u | v | w | x | y | z

<chiffre décimal> ::=
       0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<national> ::=
       #  | `  | @  |  \ |  ~ |  ^ |  [  | ] |  {  | }
       | <ligne verticale>  |  <point>  |  <souligné> | <autres caractères>

<ligne verticale> ::= |

<point> ::=  .

<souligne> ::= _

<chaîne de caractères> ::=
       <apostrophe> { <alphanumérique> | <autre caractère>
       | <spécial> | <point> | <souligné> | <espace>
       | <apostrophe> <apostrophe }* <apostrophe>

<apostrophe> ::=  ’

<autre caractère> ::= ? | & | %

<spécial> ::= + |  - | !  |  /  | > | * |  (  |  )  |  "  |  , | ; |  < |  =  |  :

<spécial composé> ::= << | >> | <= | ==> | /= | == | >= |  // | := | => | -> | (. |  .)

<fin> ::= ; | <commentaire> ;

<commentaire> ::= comment <chaîne de caractères>

<note> ::=  /* <texte> */

<texte> ::=
       { <alphanumérique> | <autre caractère> | <spécial> | <point> | <souligné>
       | <espace> | <apostrophe> }*
```

```
<mot clé> ::=
      active              | adding            | all
      | alternative       | and               | any
      | as                | atleast           | axioms
      | block             | call              | channel
      | comments          | connect           | connection
      | constant          | constants         | create
      | dcl               | decision          | default
      | else              | endalternative    | endblock
      | endchannel        | endconnection     | enddecision
      | endgenerator      | endmacro          | endnewtype
      | endoperator       | endpackage        | endprocedure
      | endprocess        | endrefinement     | endselect
      | endservice        | endstate          | endsubstructure
      | endsyntype        | endsystem         | env
      | error             | export            | exported
      | external          | fi                | finalized
      | for               | fpar              | from
      | gate              | generator         | if
      | import            | imported          | in
      | inherits          | input             | interface
      | join              | literal           | literals
      | macro             | macrodefinition   | macroid
      | map               | mod               | nameclass
      | newtype           | nextstate         | nodelay
      | noequality        | none              | not
      | now               | offspring         | operator
      | operators         | or                | ordering
      | out               | output            | package
      | parent            | priority          | procedure
      | process           | provided          | redefined
      | referenced        | refinement        | rem
      | remote            | reset             | return
      | returns           | revealed          | reverse
      | save              | select            | self
      | sender            | service           | set
      | signal            | signallist        | signalroute
      | signalset         | spelling          | start
      | state             | stop              | struct
      | substructure      | synonym           | syntype
      | system            | task              | then
      | this              | timer             | to
      | type              | use               | via
      | view              | viewd             | virtual
      | with              | xor
```

2.2 Concepts de base liés aux données

2.2.1 Définitions des types de données

SDL utilise la notion de types abstraits de données pour la définition de données manipulées par les processus. Il permet de spécifier le comportement d'un type indépendamment de la manière dont ce type sera implanté. Un type de données (ou **sorte**) définit :

- un ensemble de valeurs,
- un ensemble d'opérateurs qui peut être appliqué à ces valeurs,
- et un ensemble de règles algébriques (équations) qui définissent le comportement de ces opérateurs lorsqu'ils sont appliqués aux valeurs. Les valeurs, les opérateurs et les règles algébriques définissent tous ensemble les propriétés du type de données.

2.2.2 Variables

Les variables sont des objets qui peuvent être associés à une valeur par une affectation. Quand on accède à la variable, on renvoie la valeur associée.

2.2.3 Valeurs et littéraux

Les opérateurs sont définis à partir et vers les valeurs des sortes. Par exemple, l'application de l'opérateur somme (+) à partir et vers les valeurs de la sorte entière est valable, tandis que la somme de la sorte booléenne ne l'est pas. Toutes les sortes ont au moins une valeur. Chaque valeur appartient a une sorte unique, c'est-à-dire que les sortes n'ont jamais de valeurs en commun.

Pour la plupart des sortes, il y a des formes **littérales** qui décrivent les valeurs de la sorte (par exemple, pour les entiers, *2* est utilisé de préférence à *1+1*). Il peut y avoir plusieurs littéraux qui décrivent la même valeur (par exemple, *12* et *012* peuvent être utilisés pour décrire la même valeur entière). La même description littérale peut être utilisée pour plus d'une sorte ; par exemple, 'A' est à la fois un caractère et une chaîne de caractères de longueur 1. Certaines sortes peuvent ne pas avoir de littéraux ; par exemple, une valeur composée n'a souvent pas de littéraux en propre mais a des valeurs qui sont définies par des opérations de composition sur les valeurs de ses composantes.

2.2.4 Expressions

Une expression décrit une valeur. Si une expression ne contient pas de variables, par exemple, si elle est un littéral d'une sorte donnée, chaque occurrence de l'expression décrira toujours la même valeur. Une expression qui contient des variables peut être interprétée comme différentes valeurs durant l'interprétation d'un système SDL selon les valeurs associées aux variables.

2.3 Spécification des composants d'un système

2.3.1 Progiciels et définitions référencées

Une `<spécification SDL>` peut être décrite comme une définition de système monolithique ou comme une collection de `<package>` et de `<définition référencée>`. Un `<package>` permet la réutilisation de définitions dans différents contextes. Une `<définition référencée>` est une définition qui a été supprimée de son contexte de définition pour accroître la visibilité. Elle est similaire à une macro, mais elle est insérée à un seul endroit en utilisant une référence.

Les définitions qui font partie d'un progiciel définissent des types, des générateurs de données, des listes de signaux, des spécifications distantes et des synonymes.

Avant de pouvoir analyser une `<définition de système concret>`, chaque référence doit être remplacée par la `<définition référencée>` correspondante. Dans cette substitution, l'`<identificateur>` de la `<définition référencée>` est remplacé par le `<nom>` dans la référence.

Grammaire textuelle

```
<spécification SDL> ::=
        <liste de progiciels>
        | [ <définition de système> { <définition référencée> }* ]
```

```
<liste de progiciels> ::= { <définition de progiciel> { <définition référencée> }* }*

<définition de progiciel> ::=
        { <clause de référence de progiciel> }*
        package <nom de progiciel>
                [ <interface> ] <fin>
                {
                        <définition de type de système>
                        | <référence de type de système>
                        | <définition de type de bloc>
                        | <référence de type de bloc>
                        | <définition de type de processus>
                        | <référence de type de processus>
                        | <définition de procédure>
                        | <définition de procédure distante>
                        | <référence de procédure>
                        | <définition de signal>
                        | <définition de liste de signaux>
                        | <définition de type de service>
                        | <référence de type de service>
                        | <définition de sélection>
                        | <définition de variable distante>
                        | <définition de donnée>
                        | <définition de macro> }*
        endpackage [ <nom de progiciel> ] <fin>

<clause de référence de progiciel> ::=
        use <nom de progiciel> [ / <liste de sélection de définitions> ] <fin> }*

<liste de sélection de définitions> ::=
        <sélection de définition> { , <sélection de définition> }*

<sélection de définition> ::= <type d'entité> <nom>

<type d'entité> ::=
        system | type block | type process | type service | procedure
        | type signal | newtype | signallist | generator | synonym | remote

<interface ::= interface <liste de sélection de définitions>

<définition référencée> ::= <définition> | <diagramme pour la forme graphique>

<définition de système> ::=
        { <définition de système> | <diagramme de système }

<définition> ::=
        <définition de type de système>
        | <définition de type de bloc>
        | <définition de bloc>
        | <définition de type de processus>
        | <définition de processus>
        | <définition de procédure>
        | <définition de sous-structure de bloc>
        | <définition de sous-structure de canal>
        | <définition de type de service>
        | <définition de service>
        | <définition de macro>
        | <définition d'opérateur>
```

2.3.2 Système

Décrire un système revient, d'habitude, à décrire :

- le nom du système,
- les signaux échangés entre les processus du système et entre les processus du système et l'environnement,
- les canaux pour la connexion entre les blocs et entre les blocs et l'environnement,
- les données distantes manipulées par les processus du système,
- les blocs composant le système.

L'interprétation d'une description de système consiste à créer des instances de processus et à lancer leur exécution.

La communication entre le système et l'environnement ou entre les blocs intérieurs au système ne peut se faire qu'au moyen de signaux. A l'intérieur d'un système, ces signaux sont véhiculés par des canaux. Les canaux relient les blocs entre eux ou à la frontière du système (c'est-à-dire l'environnement). Le comportement de l'environnement est par définition non déterministe. Le système et son environnement interagissent selon un protocole connu des deux parties. Des processus de l'environnement peuvent envoyer ou recevoir des signaux. Ces processus doivent avoir des identificateurs différents de ceux utilisés pour les processus du système.

Un système peut être décrit pour gérer tous les comportements de l'environnement, ou au contraire être focalisé sur certains aspects de l'environnement seulement, les autres sont ignorés.

Une `<définition de système>` est la représentation en SDL d'une spécification ou de la description d'un système.

Grammaire textuelle

```
<définition de système> ::=
       { <clause de référence de progiciel> }*
       system <nom de système> <fin>
              { <entité dans système> }+
       endsystem [ <nom de système> ] <fin>

<entité dans système> ::=
       <définition de bloc>
       | <référence de bloc>
       | <définition de canal>
       | <définition de signal>
       | <définition de liste de signaux>
       | <définition de sélection>
       | <définition de macro>
       | <définition de donnée>
       | <définition de variable distante>
       | <référence de type de bloc>
       | <définition de type de bloc>
       | <définition de type de processus>
       | <référence de type de processus>
       | <définition de procédure>
       | <définition de procédure distante>
       | <référence de procédure>
       | <définition de type de service>
       | <référence de type de service>

 <référence de bloc> ::= block <nom de bloc> referenced <fin>
```

2.3.3 Bloc

Une <définition de bloc> contient une ou plusieurs définitions de processus d'un système et éventuellement une sous-structure de bloc. La définition de bloc permet de regrouper les processus qui tous ensemble assurent une certaine fonction, soit directement soit par l'intermédiaire d'une sous-structure de bloc.

Une <définition de bloc> assure une interface statique de communication par laquelle les processus peuvent communiquer avec d'autres processus. De plus, elle donne la portée pour les définitions de processus.

Interpréter un bloc consiste à créer les processus initiaux dans le bloc.

Décrire un bloc revient, d'habitude, à décrire :

- le nom du bloc,
- les signaux visibles à l'intérieur du bloc,
- les routes pour interconnecter les processus du bloc,
- les connexions des routes à des canaux pour l'interconnexion de canaux externes au bloc et les routes internes au bloc,
- les processus composant le bloc.

Grammaire textuelle

```
 <définition de bloc> ::=
      block { <nom de bloc> | <identificateur de bloc> } <fin>
             {      <définition de signal>
                    | <définition de liste de signaux>
                    | <définition de processus>
                    | <référence de processus>
                    | <définition de route>
                    | <définition de macro>
                    | <définition de variable distante>
                    | <définition de donnée>
                    | <définition de sélection>
                    | <définition de type de processus>
                    | <référence de type de processus>
                    | <définition de type de bloc>
                    | <référence de type de bloc>
                    | <définition de procédure>
                    | <référence de procédure>
                    | <définition de procédure distante>
                    | <définition de type de service>
                    | <référence de type de service>
                    | <connexion de canal vers route>
             }*
             [ <définition de sous-structure de bloc>
             | <référence de sous-structure de bloc> ]
      endblock [ <nom de bloc> | <identificateur de bloc> ] <fin>

<référence de processus> ::=
      process <nom de processus> [ <nombre d'instances> ] referenced <fin>
```

2.3.4 Processus

Une `<définition de processus>` introduit le type d'un processus qui est destiné à représenter un comportement dynamique.

Une instance de processus est une machine à états finis étendue communicante qui assure un certain nombre d'actions, appelées **transitions**, suite à la réception d'un signal donné, chaque fois qu'elle se trouve dans un certain **état**. La réalisation de la transition aboutit à l'attente du processus dans un autre état, qui n'est pas nécessairement différent de l'état d'origine.

Le concept de machine à états finis a été étendu en ce sens que l'état résultant d'une transition, à côté du signal provoquant la transition, peut être affecté par des décisions prises à partir de variables connues du processus.

Plusieurs instances du même type de processus peuvent exister au même instant et s'exécuter de manière asynchrone et en parallèle les unes aux autres, et avec d'autres instances de différents types de processus dans le système. Dans le `<nombre d'instances>`, la première valeur représente le nombre d'instances du processus qui existe à la création du système, la deuxième valeur représente le nombre maximal d'instances simultanées du type de processus. Si le `<nombre initial>` est omis, le `<nombre initial>` est *1*. Si le `<nombre maximal>` est omis, le `<nombre maximal>` n'est pas limité. Le `<nombre initial>` d'instances doit être inférieur ou égal au `<nombre maximal>`.

Lorsqu'un système est créé, les processus initiaux sont créés dans un ordre aléatoire. La communication des signaux entre les processus ne commence que lorsque tous les processus initiaux ont été créés.

Les instances de processus existent à partir du moment ou un système est créé ou peuvent être créées par une action de demande de création qui lance les processus à interpréter (opération **`create`**) ; leur interprétation commence lorsque l'action de départ (**`start`**) est interprétée ; des actions d'arrêt (**`stop`**) peuvent arrêter leur existence.

Les signaux reçus par les instances de processus sont appelés signaux d'entrée, et les signaux envoyés sont appelés signaux de sortie.

Les signaux peuvent être traités par une instance de processus seulement lorsque celle-ci se trouve dans un certain état. L'ensemble complet de signaux d'entrée valides est l'union de l'ensemble des signaux se trouvant dans tous les trajets des signaux conduisant au processus, de l'`<ensemble de signaux d'entrée valides>`, des signaux implicites et des signaux de réveil (temporisation).

Un accès d'entrée unique est associé à chaque instance de processus. Lorsqu'un signal d'entrée parvient au processus, il est appliqué à l'accès d'entrée de l'instance de processus.

Un processus peut être soit mis en attente, en occupant un état, soit en activité, en effectuant une transition. Pour chaque état, il existe un ensemble de signaux sauvegardés. Dans le cas d'attente dans un état, le premier signal entrant dont l'identificateur ne figure pas dans l'ensemble des signaux sauvegardés est extrait de la file d'attente, et traité par le processus.

L'accès d'entrée peut retenir un nombre quelconque de signaux, de sorte que plusieurs signaux entrants sont mis dans une file d'attente pour le processus. L'ensemble des signaux retenus est ordonné dans la file d'attente, selon l'ordre d'arrivée. Si deux ou plusieurs signaux arrivent simultanément, ils sont ordonnés arbitrairement. La file d'attente associée à un processus est dite aussi port d'entrée du processus.

Lorsqu'un processus est créé, on lui attribue un accès d'entrée vide, et il y a création de variables locales auxquelles on affecte des valeurs.

Les paramètres formels d'un processus sont des variables qui sont créées soit lorsque le système est créé (mais aucun paramètre réel ne lui est transmis et par conséquent ces paramètres ne sont pas initialisés), soit lorsque l'instance de processus est dynamiquement créée.

Pour désigner un processus, on utilise une expression de type prédéfini **`PId`** (Process Identifier). Une valeur de PId identifie un et un seul processus à un instant. A la création d'un processus, le système détermine l'identificateur du processus créé. Pour toutes les instances de processus, on peut utiliser quatre expressions produisant un **`PId`** :

- `self` qui fournit le PId du processus exécutant,
- `parent` qui fournit le PId du processus créateur,
- `offspring` qui fournit le PId de l'instance de processus la plus récente créée par le processus exécutant,
- et `sender` qui fournit le PId de l'instance de processus en provenance de laquelle le dernier signal entrant a été utilisé.

Pour toutes les instances de processus qui se trouvent présentes au moment de l'initialisation du système, l'expression prédéfinie `parent` présente toujours la valeur `null`. Pour toutes les instances de processus nouvellement créées, les expressions prédéfinies `sender` et `offspring` ont la valeur `null`.

Décrire un processus revient, d'habitude, à décrire :

- le nom du processus,
- les paramètres formels du processus,
- les variables du processus,
- les réveils manipulés par le processus,
- la description, à l'aide d'états et transitions, du comportement du processus.

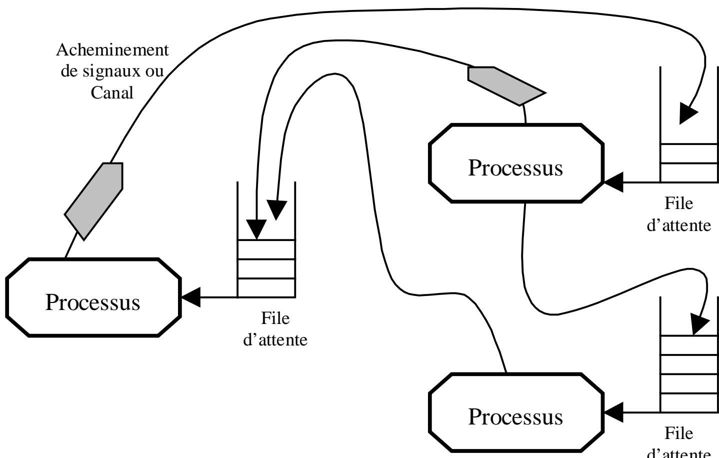

Figure 2. *Principe de communication entre processus*

Grammaire textuelle

```
<définition de processus> ::=
      process { <identificateur de processus> | <nom de processus> }
            [ <nombre d'instances> ] <fin>
            [ <paramètres formels> <fin> ]
            [ <ensemble des signaux d'entrée valides> ]
            {      <définition de signal>
                  | <définition de liste de signaux>
                  | <définition de procédure>
                  | <référence de procédure>
                  | <définition de procédure distante>
                  | <spécification de procédure importée>
                  | <définition de macro>
                  | <définition de variable distante>
                  | <définition de donnée>
                  | <définition de variable>
                  | <spécification de variable importée>
```

```
                | <définition de vue>
                | <définition de sélection>
                | <définition de réveil>
                | <définition de route>
                | <définition de service>
                | <référence de service>
                | <définition de type de service>
                | <référence de type de service>
                | <connexion de route à route>
        }*
        [ <corps de processus> ]
    endprocess [ <nom de processus> | <identificateur de processus> ] <fin>

<référence de procédure> ::=
    <préambule de procédure> procedure <nom de procédure> referenced <fin>

<référence de service> ::= service <nom de service> referenced <fin>

<ensemble de signaux d'entrée valides> ::=  signalset [ <liste de signaux> ] <fin>

<corps de processus> ::=  <départ> { <état> | <action> }*

<paramètres formels> ::= fpar <paramètres> { , <paramètres> }*

<paramètres> ::= <nom de variable> { , <nom de variable> }* <sorte>

<nombre d'instances> ::=  ( [ <nombre initial> ] [ , [ <nombre maximal> ] ] )

<nombre initial> ::= <entier naturel>

<nombre maximal> ::= <entier naturel>
```

2.3.5 Service

Un processus peut être décrit par une seule machine d'états finis ou par plusieurs machines, ce qui permet, dans certains cas, d'augmenter la lisibilité d'un processus. Chaque service peut définir un comportement partiel d'un processus.

Dans une instance de processus, il existe une instance de service pour chaque `<définition de service>`. Les instances de services sont des composantes d'un processus et ne peuvent pas être manipulées comme des objets séparés. Les services partagent l'accès d'entrée de signaux de l'instance du processus auxquels ils sont rattachés.

Une instance de service est une machine d'états. Lorsqu'une instance de processus est créée, les nœuds de départ (c'est-à-dire **start**) des services sont exécutés dans n'importe quel ordre. Aucun état d'aucun service n'est interprété avant que tous les nœuds de départ de tous les services d'une instance de processus ne soient terminés. A un instant donné, un seul service exécute une transition. Lorsque le service exécutant atteint un état, le signal suivant à l'accès d'entrée est donné au service qui est capable de le traiter. Lorsqu'un service cesse d'exister, les signaux d'entrée liés à ce service sont écartés. Lorsque tous les services cessent d'exister, l'instance de processus cesse d'exister.

Grammaire textuelle

```
<définition de service> ::=
    service { <nom de service> | <identificateur de service> } <fin>
        [ <ensemble de signaux d'entrée valides> ]
        {     <définition de variable>
```

```
                | <définition de donnée>
                | <définition de vue>
                | <spécification de variable importée>
                | <spécification de procédure importée>
                | <définition de sélection>
                | <définition de macro>
                | <définition de procédure>
                | <référence de procédure>
                | <définition de réveil>
        }*
        <corps de service>
    endservice { <nom de service> | <identificateur de service> } <fin>

<corps de service> ::= <corps de processus>
```

2.3.6 Procédure

Les procédures de SDL ont le même sens que les procédures dans les langages de programmation tels que C, ADA ou Pascal. Une procédure est un moyen de donner un nom à un assemblage d'objets et de le représenter par une référence unique.

Des paramètres sont associés à un appel de procédure : on les emploie à la fois pour fournir les valeurs et pour limiter la portée des variables pour l'exécution de la procédure. Les paramètres formels d'une procédure peuvent être de type «appel par valeur» (désigné par le mot-clé **`in`**) ou de type «appel par référence» (désigné par le mot-clé **`in/out`**).

Une procédure se décrit comme un processus, sauf qu'elle se termine par **`return`** (au lieu de **`stop`**, pour un processus).

Dans une `<définition de procédure>`, la `<définition de variable>` ne peut contenir les expressions **`revealed`**, **`exported`**, **`revealed exported`**, **`exported revealed`** `<nom de variable>`, c'est-à-dire qu'une variable de procédure est une variable locale à la procédure qui ne peut apparaître, être visible, être exportée ou être importée. Elle est créée lors de l'interprétation du nœud de départ de procédure et cesse d'exister lors de l'interprétation du nœud de retour de procédure.

Une `<définition de procédure>` est interprétée seulement lorsqu'une instance de processus l'appelle, et l'interprétation est faite par l'instance de processus.

`exported` dans un préambule de procédure permet à la procédure d'être appelée en tant que procédure distante (cf. §2.3.8.10).

Grammaire textuelle

```
<définition de procédure> ::=
    <préambule de procédure>
    procedure { <identificateur de procédure> | <nom de procédure> }
        [ <paramètres formels de procédure> <fin> ]
        [ <résultat de procédure> <fin> ]
        {     <définition de donnée>
                | <définition de variable>
                | <référence de procédure>
                | <définition de procédure>
                | <définition de sélection>
                | <définition de macro>
        }*
        [ <corps de procédure> ]
    endprocedure [ <nom de procédure> | <identificateur de procédure> ] <fin>
```

```
<préambule de procédure> ::= [ exported [ as <identificateur de procédure distante> ] ]

<paramètres formels de procédure> ::=
        fpar <paramètre formel de procédure> { , <paramètre formel de procédure> }*

<paramètre formel de procédure> ::=
        [ in/out | in ] <nom de variable>{ , <nom de variable> } <sorte>

<corps de procédure> ::= <corps de processus>

<résultat de procédure>  ::=  returns [ <nom de variable> ] <sorte>
```

2.3.7 Communication

La communication de signaux s'effectue via des canaux et des routes. Les routes servent à la communication entre processus d'un même bloc, et les canaux à la communication entre processus de blocs différents.

2.3.7.1 Signal

Une instance de signal est un flot d'informations entre des processus. Une instance de signal peut être envoyée par l'environnement ou par un processus, elle se dirige vers un processus ou vers l'environnement.

On associe à chaque instance de signal deux valeurs de type `PId` indiquant les processus de départ et de destination du signal. Un signal peut contenir ou non des données. Si le signal doit véhiculer des données, les sortes de ces données doivent être spécifiées dans la définition du signal.

Grammaire textuelle

```
<définition de signal> ::=
        signal <définition d'item de signal> { , <définition d'item de signal> }* <fin>

<définition d'item de signal> } ::=
        <nom de signal> [ <paramètres formels> ] [ ( <sorte> { , <sorte> }* ) ]
        [ <raffinement de signal> ]
```

2.3.7.2 Définition de liste de signaux

La notion de liste de signaux est un moyen abrégé pour désigner un ensemble de signaux. Un `<identificateur de liste de signaux>` peut être utilisé dans une `<définition de canal>`, une `<définition de route>`, une `<définition de liste de signaux>`, un `<ensemble de signaux d'entrée valides>` ou une `<liste de sauvegarde>`.

Grammaire textuelle

```
<définition de liste de signaux> ::=
        signallist <nom de liste de signaux> = <liste de signaux> <fin>

<liste de signaux> ::= <item de liste de signal> { , <item de liste de signaux> }*

<item de liste de signaux> ::=
        <identificateur de signal> | (<identificateur de liste de signaux>)
        | <identificateur de réveil>
```

2.3.7.3 Canal

Un canal est un moyen de transport pour les signaux. Un canal peut être considéré comme étant un ou deux trajets de canal unidirectionnels indépendants entre deux blocs ou entre un bloc et son environnement.

Les signaux acheminés par les canaux sont véhiculés jusqu'à l'extrémité de destination. A l'extrémité de destination d'un canal, les signaux se présentent dans le même ordre que celui des signaux au point d'origine. Si deux ou plusieurs signaux arrivent simultanément sur le canal, ils sont ordonnés arbitrairement.

Un canal peut retarder l'acheminement des signaux sur le canal. Cela signifie qu'une file d'attente du type premier-entré-premier-sorti se trouve associée à chaque direction dans un canal. Quand un signal se présente sur le canal, il est placé dans la file d'attente. Après un intervalle de temps non déterminé et non constant, la première instance de signal dans la file d'attente est libérée et appliquée à l'un des canaux ou à l'un des trajets de signaux qui se trouvent connectés au canal. Plusieurs canaux peuvent exister entre deux extrémités. Le même type de signal peut être acheminé sur des canaux différents.

Le mot-clé **`nodelay`** permet de spécifier des canaux qui ne causent pas de retard dans la communication de signaux.

Grammaire textuelle

```
<définition de canal> ::=
       channel <nom de canal> [ nodelay ]
       <trajet de canal> [ <trajet de canal> ]
       [ <définition de sous-structure de canal>
       | <référence de sous-structure de canal> ]
       endchannel [ <nom de canal> ] <fin>

<trajet de canal> ::=
       from <extrémité de canal> to <extrémité de canal> with <liste de signaux> <fin>

<extrémité de canal> ::=
       { <identificateur de bloc> | env } [ via <route> ]
```

2.3.7.4 Routes

Une routeest un moyen de transport pour les signaux. Il peut être considéré comme étant un ou deux trajets d'acheminement de signal unidirectionnels et indépendants entre deux processus ou entre un processus et son environnement. Une route n'apporte aucun retard dans le transport des signaux.

Plusieurs trajets de signaux peuvent exister entre deux extrémités. Le même type de signal peut être acheminé sur différentes routes.

Grammaire textuelle

```
<définition de route> ::=
       signalroute <nom de route> <trajet de route>
       [ <trajet de route> ]

<trajet de route> ::=
       from <extrémité de route>
       to < extrémité de route>
       with <liste de signaux> <fin>

<extrémité de route> ::=
       { <identificateur de processus> |  <identificateur de service> |  env }
       [ via <porte> ]
```

2.3.7.5 Connexion

La mise en correspondance entre les canaux, permettant d'interconnecter des blocs différents, et les routes, permettant d'interconnecter des processus, se fait via une opération de connexion explicite (**connect**).

Grammaire textuelle

```
<connexion de canal à route> ::=
       connect <identificateurs de canal>
       and <identificateurs de route> <fin>

<identificateurs de canal> ::= <identificateur de canal> { , <identificateur de canal> }*

<identificateurs de route> ::=
       <identificateur de route>
       { , <identificateur de route> }*

<connexion de route à route> ::=
       connect <identificateurs de route externe> and
       <identificateurs de route> <fin>

<identificateurs de route externe> ::=
       <identificateur de route>
       { , <identificateur de route> }*
```

2.3.8 Comportement de système

Un processus est une machine d'états étendue dont on doit spécifier les variables locales et comportement (par des affectations de variables, des tests, des opérations de réception ou d'émission de signaux, etc.).

2. 3.8.1 Début d'un processus (départ)

Tout processus commence par exécuter la transition définie dans son état de départ.

```
<départ> ::= start <fin> <transition>
```

2.3.8.2 Les états d'un processus

Un état représente une condition particulière dans laquelle un processus peut traiter une instance de signal en exécutant une transition. Si l'état n'a ni transitions spontanées, ni signaux continus et s'il n'y a pas d'instances de signaux en attente à l'entrée autres que celles mentionnées dans la sauvegarde, le processus est bloqué dans l'état jusqu'à ce qu'une instance de signal soit reçue.

Dans la spécification d'une `<liste d'états>`, **'*'** désigne tous les états possibles autres que ceux cités explicitement et '**-**' désigne le même état (c'est-à-dire que le processus reste dans le même état).

Grammaire textuelle

```
<état> ::=
       state <liste d'états> <fin>
              { <partie entrée> | <entrée prioritaire> | <partie sauvegarde>
              | <transition spontanée> | <signal continu> }*
       [ endstate [ <nom d'état> ] <fin> ]
<liste d'états> ::= <nom d'état> { , <nom d'état> }* | *
```

Exemple

```
state * (S1, S2) ;  /* Etat dans lequel on attend tous les signaux sauf S1 et S2 */
nextstate - ;       /*rester dans le même état */
```

2.3.8.3 Entrée de signaux et déclenchement d'une transition

Les transitions d'un processus sont déclenchées principalement suite à la réception de signaux (on dit aussi entrée de signaux).

Une entrée permet le traitement de l'instance de signal spécifiée. Le traitement du signal d'entrée rend l'information véhiculée par le signal disponible au processus. Aux variables associées à l'entrée, on affecte les valeurs acheminées par le signal utilisé.

La réception d'un signal provoque immédiatement une transition, sauf si une `<condition de validation>` est spécifiée. Dans ce cas, la transition ne peut être déclenchée que si le signal attendu est arrivé et que la `<condition de validation>` est vraie.

L'identité du processus émetteur du signal peut être obtenue grâce à **`sender`** (qui est de type PId).

`<partie entrée>` définie dans la spécification d'`<état>` spécifie un signal dont les instances sont attendues par le processus.

Grammaire textuelle

```
<partie entrée> ::=
       input <liste d'entrées> <fin> [ <condition de validation> ] <transition>

<liste d'entrées> ::= * | <stimulus> { , <stimulus> }*

<stimulus> ::=
       { <identificateur de signal> | <identificateur de réveil> }
       [ ( [ <variable> ] { , [ <variable> ] }*) ]

<condition de validation> : := provided <expression booléenne> <fin>
```

2.3.8.4 Sauvegarde de signaux

Lorsqu'un processus se trouve dans un état, tous les signaux qui ne sont pas explicitement attendus par le processus dans cet état sont ignorés (ils sont donc implicitement consommés dès qu'ils arrivent). Cela signifie qu'un signal n'est mis en file d'attente pour être traité par le processus que si ce processus a demandé d'attendre explicitement ce signal. Un processus attend explicitement un signal `Sig`, s'il mentionne le nom `Sig` dans un **`input`** ou bien s'il utilise un **`*`** (c'est-à-dire tous les états valides) dans un **`input`** sans exclure explicitement le signal `Sig`. Pour éviter de perdre des signaux, tout processus doit demander (en utilisant **`save`**) la sauvegarde des signaux qu'il souhaite traiter dans un état suivant.

Les signaux sauvegardés sont bloqués à l'accès d'entrée dans l'ordre de leur arrivée. La sauvegarde n'a d'effet que sur les états auxquels la sauvegarde est associée. Dans l'état suivant, les instances de signaux sauvegardées sont traitées comme des instances normales de signaux.

`<partie sauvegarde>` définie dans la spécification d'`<état>` spécifie la liste des instances de signaux qui ne peuvent être traitées par le processus dans l'état auquel la sauvegarde est associée et qui nécessitent une sauvegarde pour être traitées ultérieurement.

```
<partie sauvegarde> ::= save { <liste de signaux> | * } <fin>
```

2.3.8.5 Entrée prioritaire de signaux

Dans certains cas, on peut exprimer que la réception d'un signal est plus prioritaire par rapport à d'autres signaux. Cela s'exprime par :

```
<entrée prioritaire> ::= priority  input <liste de signaux> <fin> <transition>
```

2.3.8.6 Transition spontanée

Une transition spontanée permet l'activation d'une transition sans qu'aucun stimulus ne soit présenté au processus (c'est-à-dire sans réception de signal). Il n'y a pas de priorité entre les transitions activées par des signaux et les transitions spontanées.

Une transition spontanée peut être sujette à une `<condition de validation>`. Si c'est le cas, la transition ne peut être franchie que si sa condition de validation est vraie.

Grammaire textuelle

```
<transition spontanée> ::= input none <fin> [ <condition de validation> ] <transition>
```

2.3.8.7 Signal continu

Dans certaines situations, on aimerait décrire une transition causée par la valeur vraie d'une `<expression booléenne>` et non par l'occurrence d'un signal. On parle dans ce cas de *signal continu.*

Si un processus est en attente de plusieurs signaux dont certains sont continus, et si plusieurs de ces signaux sont présents simultanément, le choix du signal retenu dépend de la priorité des signaux quand celle-ci est spécifiée. Dans le cas où les priorités ne permettent pas de trancher entre des signaux, le choix se fait de manière aléatoire entre les signaux considérés.

Grammaire textuelle

```
<signal continu> ::=
        provided <expression booléenne> <fin>
        [ priority <nombre entier> <fin> ] <transition>
```

2.3.8.8 Transition

Une transition réalise une suite d'actions. Pendant la transition, les variables du processus peuvent être modifiées et les signaux peuvent être envoyés. La transition s'arrête quand le processus change d'état.

Grammaire textuelle

```
<transition> ::=
        { { <instruction d'action> }+ [ <instruction de terminaison> ] }
        | <instruction de terminaison>

<instruction d'action> ::= [ <étiquette> : ] <action> <fin>

<action> ::=
        <tâche>
        | <émission de signal> | <création de processus>| <décision>
        | <initialisation de réveil> | <réinitialisation de réveil>
        | <exportation> | <appel de procédure> | <appel de procédure distante>
```

1. Tâches

Le mot-clé **`task`** permet de spécifier une ou plusieurs opérations. Ces opérations peuvent être formelles (c'est le cas des affectations de valeurs à des variables) ou informelles (dans ce cas on spécifie un texte entre **'** et **'**). Un texte informel dans une action permet au spécifieur d'indiquer des choses qu'il souhaite formaliser plus tard.

Grammaire textuelle

```
<tâche> ::= task { <instruction d'affectation> { , <instruction d'affectation> }* }
       | { <texte informel> { , <texte informel> }* }
```

2. Création de processus

Une instance de processus peut créer de nouvelles instances de processus dans le même bloc (grâce à **`create`**). Le fait qu'un processus ne peut créer d'autres processus que dans le même bloc, conduit à spécifier au moins un processus, par bloc, avec un nombre initial d'instances différent de zéro. La première instance de processus (instance qui est créée automatiquement en même temps que le système à interpréter) sert à créer les autres instances de processus. Les expressions **`parent`**, **`self`** et **`offspring`** sont modifiées par les actions de création de processus. Lorsqu'une instance de processus est créée, les variables sont créées et les paramètres formels sont remplacés par les paramètres effectifs. L'instance processus créée commence son interprétation au point indiqué par **`start`**. L'instance processus créée se déroule de manière parallèle et asynchrone avec les autres processus. Si un processus tente de créer plus d'instances de processus que la valeur maximale d'instances spécifiée lors de la définition du processus, la valeur **`null`** est renvoyée dans **`offspring`** et aucune instance de processus n'est créée.

Grammaire textuelle

```
<création de processus> ::=
       create <identificateur de processus> [ <paramètres effectifs> ]

<paramètres effectifs> ::= ( [ <expression> ] { , [expression> ] }* )
```

3. Appel de procédure

`call` permet d'appeler et d'interpréter une procédure avec ou sans paramètres.

Grammaire textuelle

```
<appel de procédure> ::=
       call <identificateur de procédure> [ <paramètres effectifs> ]
```

4. Emission de signaux

Un processus peut envoyer (en utilisant **`output`**) un ou plusieurs signaux à un ou plusieurs processus via des chemins implicitement ou explicitement spécifiés. Un signal peut transporter ou non des informations (c'est-à-dire un message).

Si un signal peut être envoyé sur plusieurs routes, le mot clé **`via`** permet, le cas échéant, d'indiquer le ou les routes à utiliser. Les mots-clés **`via all`** constitue un acheminement multidestinataire. Si aucun chemin à emprunter n'est spécifié dans la clause **`via all`** et qu'aucun destinataire n'est spécifié, tout processus pour lequel il existe un trajet de communication reçoit le signal envoyé. Cela veut dire que SDL permet d'adresser un message explicitement à un ensemble de processus ou de le diffuser à tous les récepteurs potentiels.

La valeur du PId du processus émetteur du signal est incorporée au signal permettant à tout récepteur de savoir d'où vient tout signal qu'il reçoit. Lorsque la clause **`to`** `<destination>` n'est pas spécifiée, cela peut

entraîner un non déterminisme explicite pour la communication de signaux ; cela conduit le système d'interprétation à déterminer les processus destinataires en se basant sur la description statique des canaux et routes.

Grammaire textuelle

```
<émission de signal> ::=
       output <identificateur de signal> [ <paramètres effectifs> ]
       { , <identificateur de signal> [ <paramètres effectifs> ] }*
       [ to <destination> ] [ via [ all ] <chemins à emprunter> ]

<destination> ::= <PId de processus> | <identificateur  de processus>

<chemins à emprunter> ::= <chemin à emprunter> { , <chemin à emprunter> }*

<chemin à emprunter> ::=
       <identificateur de route>
       | <identificateur de canal> | <identificateur de porte>
```

Exemple (figure 3)

```
       signal S1(integer) ;
       . . .
       process P1 ;
       
       . . .
       output S1(100) to P2
```

```
       process P2 ;
       dcl integer v1 ;
       . . .
       input S1(v1)
```

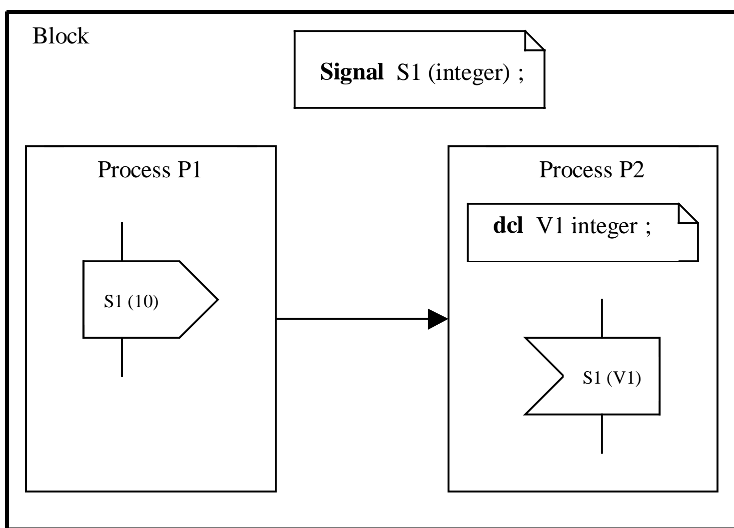

Figure 3. *Exemple d'échange de signaux entre processus*

5. Décision (ou test)

Comme dans les langages de programmation, SDL offre la possibilité d'effecteur des tests (par **`decision`**), ce qui permet à un processus de choisir la prochaine action en fonction de la réponse à un test. Un ensemble de réponses possibles à la question est défini, chacune d'elle spécifiant un ensemble d'actions à interpréter

pour ce choix. Une des réponses peut être le complément des autres, c'est le cas quand on spécifie la partie **else**. L'utilisation de **any** signifie n'importe quelle valeur de la sorte spécifiée.

Grammaire textuelle

```
<décision> ::= decision  <question> <fin> <corps de décision> enddecision

<corps de décision> ::=
       { <partie condition vraie> <partie condition fausse> }
       |  { <partie condition vraie> { <partie condition vraie> }+
       [ <partie condition fausse> ] }

<partie condition vraie> ::= ( [ <réponse> ] ) : [ <transition> ]

<réponse> ::= <valeur dans un intervalle> | <texte informel> | true | false

<partie condition fausse> ::= else : [ transition> ]

<question> ::= <condition booléenne> | <texte informel> |  any
```

Exemple

```
decision compteur ;
     (1) : . . .
     (2) : . . .
     (3) : . . .
     else : . . .
enddecision ;

decision compteur > 10 ;
     (true) : . . .
     (false) : . . .
enddecision ;

decision reponse ;
     ('O') : . . .
     ('oui') : . . .
     ('N') : . . .
     ('non') : . . .
enddecision ;
```

6. Terminaison d'une transition

Toute transition de processus (de service ou de procédure) doit se terminer par l'une des trois possibilités suivantes :

- en spécifiant (par **nextstate**) le prochain état vers lequel évoluera le processus, la procédure ou le service une fois la transition courante terminée. Si `<nom d'état suivant>` est égal à **-**, cela signifie que le processus reste dans le même état.
- en effectuant un saut (par **join**) vers une étiquette (c'est-à-dire un point de branchement) préalablement définie dans le processus, la procédure ou le service.
- en mettant fin (par **stop**) à l'interprétation du processus ou du service courant.

En plus des trois possibilités précédentes, les transitions - ou généralement la dernière transition - d'une procédure se terminent par **return** (pour spécifier la fin de la procédure et retour au processus ou au service appelant). Après le retour, toutes les variables créées par la procédure appelée sont détruites.

Grammaire textuelle

```
<instruction de terminaison> ::= [ <nom d'étiquette> : ] <terminateur> <fin>

<terminateur> ::=
        nextstate { <nom d'état suivant> | - }  | join <nom d'étiquette>
        | stop  | return [ <expression> ]
```

2.3.8.9 Gestion du temps

Pour permettre la gestion du temps nécessaire aux applications où il existe des contraintes temporelles, SDL offre les opérations permettant de :

- lire la valeur du temps courant (via l'expression **now**),
- définir des réveils (via **timer**),
- armer ou annuler des réveils (via **set** et **reset**).

Deux sortes sont prédéfinies dans SDL : **time** (pour désigner le temps) et **duration** (pour désigner une durée de temps). Ces deux sortes sont des réels positifs. L'unité de temps n'est pas définie dans SDL.
Un réveil fonctionne comme un signal, sauf que le signal n'est pas envoyé par un processus défini par l'utilisateur, mais par le système lorsque le réveil arrive à échéance.

Un réveil peut être actif ou inactif. Lorsqu'un réveil inactif est initialisé, par **set**, une valeur de temps est associée à ce réveil. Si la valeur d'initialisation d'un réveil, `<expression de durée>`, est inférieure à l'instant courant (obtenu par **now**), le système provoque une génération immédiate de signal de réveil. Ainsi, `<expression de durée>` est souvent fixée à **now** plus une durée. Lorsqu'un réveil arrive à échéance, un signal est envoyé au processus qui attend l'expiration du réveil (dans ce cas, l'expression **sender** prend la valeur de **self**). Initialiser un réveil déjà initialisé conduit à écraser la valeur actuelle du réveil par la valeur spécifiée par `<expression de durée>`.

Lorsqu'un réveil actif est annulé, par **reset**, l'association avec la valeur de temps est perdue. De plus, s'il y a un signal correspondant à ce réveil qui est retenu dans l'accès à l'entrée du processus ayant initialisé le réveil, ce signal est supprimé, et le réveil devient inactif. L'annulation d'un réveil inactif n'a aucun effet.

L'association d'une `<liste de sortes>` à un réveil permet, comme dans le cas des signaux normaux, de transmettre des informations en même temps que le signal de réveil. Cette possibilité permet, par exemple, d'utiliser un seul réveil pour plusieurs raisons différentes et d'associer au réveil, à chaque fois qu'il est armé, une valeur qui permet de savoir plus tard la raison de chaque déclenchement de ce réveil. Les informations véhiculées par un signal de réveil sont fixées au moment de l'armement du réveil par **set**.

La fonction booléenne **active**`(<nom de réveil>)` permet de savoir si un réveil est actif ou non.

Grammaire textuelle

```
<définition de réveil> ::=
        timer <définition de réveil> { , <définition de réveil> }* <fin>

<définition de réveil> ::=
        <nom de réveil> [ <liste de sortes> ] [ := <expression de durée> ]

<initialisation de réveil> ::=
        set (<initialiser réveil> { , <initialiser réveil> }*)

<initialiser réveil> ::=
        ( [ <expression de temps>, ] <identificateur de réveil>
        [ ( <liste d'expressions> ) ] )

<annulation de réveil> ::= reset (<annuler réveil> { ,<annuler réveil> }*)
```

```
<annuler de réveil> ::= <identificateur de réveil> [ ( < liste d'expressions> ) ]
```

Exemple

```
/* Définition et utilisation d'un réveil */
dcl timer t1 ;
…
set (now+100, t1) ;
state attente_réveil ;
 input t1 ;
     . . .
```

2.3.8.10 Déclaration et partage de variables

Une ou plusieurs variables peuvent être déclarées, et éventuellement initialisées, par le mot-clé **dcl**.

Dans SDL, une variable appartient toujours à une instance de processus dont elle est une variable locale. Elle peut être visible uniquement pour ce processus ou bien être accessible à d'autres instances de processus du même bloc ou d'autres blocs. Selon que l'accès est autorisé pour des processus appartenant à un même bloc ou des blocs, SDL offre deux mécanismes. Dans le premier cas, on utilise un mécanisme dit « **view**/**revealed** », dans le second, un mécanisme dit « **import**/**export** ».

1. *Partage de variable à l'intérieur d'un même bloc*

   Pour qu'une variable d'un processus puisse être lue par un autre processus d'un même bloc, elle doit être déclarée, dans son processus d'appartenance, avec l'attribut **revealed**. Un processus qui désire lire une variable déclarée **revealed** dans un autre processus, doit déclarer une `<définition de vue>` sur cette variable, par **viewed**. Pour accéder effectivement à la variable, le processus lecteur utilise le mot-clé **view** avec le nom de la variable à lire.

2. *Partage de variables entre processus appartenant à des blocs différents*

   Pour qu'une variable d'un processus puisse être lue par un autre processus appartenant à un autre bloc d'un même système, elle doit être déclarée, dans son processus d'appartenance, avec l'attribut **exported**. Un processus qui désire lire une variable déclarée **exported** dans un autre processus, doit déclarer cette variable comme étant importée, par **imported**. La valeur courante d'une variable exportée est rendue visible par le processus propriétaire par **export**. Cette valeur est lue par un processus autorisé (c'est-à-dire qui a importé la variable) par **import**.

   Une opération d'exportation est l'exécution d'un **export** par lequel le processus exportateur divulgue la valeur courante d'une variable exportée. Une opération **export** provoque le stockage de la valeur courante de la variable exportée dans sa copie implicite.

   Une opération d'importation est l'exécution de **import** permettant au processus importateur d'accéder à la valeur d'une variable exportée. La valeur importée est stockée dans la variable d'import implicite.

Les mécanismes **view**/**revealed** ou **import**/**export** ne permettent pas à un processus de modifier la valeur d'une variable déclarée dans un autre processus. Le mécanisme **view**/**revealed** permet d'accéder à la valeur courante d'une variable (on dit que la variable est visible en continu), alors que le mécanisme **import**/**export** permet au processus lecteur de lire seulement la dernière valeur de variable rendue visible par le processus propriétaire (par **export**).

Grammaire textuelle

```
<déclaration de variable> ::=
      dcl [ revealed | exported | revealed exported | exported revealed ]
      <variables de sorte> { , <variables de sorte> }* <fin>
```

<variable de sorte> ::=
<nom de variable> [<exportée>] { , <nom de variable> [<exportée>] }*
<sorte> [**:=** <expression>]

<exportée> ::= **as** <identificateur de variable distante>

<définition de vue> ::=
viewed <nom de vue> { , <nom de vue> }* <sorte>
{ , <nom de vue> { , <nom de vue> }* <sorte> }* <fin>

<lecture d'une variable d'un autre processus du même bloc> ::=
view (<nom de variable>,
[<PId de l'instance de processus où se trouve la variable à lire>]

<définition de variable distante> ::=
remote <nom de variable distante>
{ , <nom de variable distante> }* <sorte> [**nodelay**]
{ , <nom de variable distante>
{ , <nom de variable distante> }* <sorte> [**nodelay**] }* <fin>

<spécification de variable importée> ::=
imported <identificateur de variable distante>
{ , <identificateur de variable distante> }* <sorte>
{ , <identificateur de variable distante>
{ , < identificateur de variable distante> }* <sorte> }* <fin>

<expression d'importation> ::=
import **(** <identificateur de variable distante> [, <destination>] **)**

<exportation de valeur de variable> ::=
export (<identificateur de variable> { , <identificateur de variable> }*)

Exemple

```
/* Les processus P1 et P2 appartiennent au même bloc. Le processus
P3 appartient à un autre bloc.
Le processus P1 permet au processus P2 d'accéder à la variable v3 et
au processus P3 d'accéder à la variable v4 */

process P1
dcl integer v1 ;
dcl integer v2 :=10 ;
dcl revealed  v3 integer ;
dcl exported  v4 real ;
. . .
task v3 := 20 ;
task v4 := 12.5 ;
task export(v4) ;
. . .

process P2
dcl integer y ;
viewed v3 integer ;
. . .
task y := view(v3) ;
. . .

process P3
imported v4 real ;
dcl real x ;
. . .
task x := import(v4) ;
. . .
```

2.3.8.11 Procédures distantes

Le concept de procédure distante permet à un processus d'appeler une procédure définie dans un autre processus.

Une procédure exportée est une procédure ayant l'attribut **`exported`** dans sa déclaration. La signature d'une procédure désigne les types de paramètres d'appel.

Un appel de procédure distante par un processus demandeur conduit ce dernier à attendre que le processus serveur (celui où est définie la procédure appelée) exécute la procédure. Les signaux envoyés par le processus serveur pendant l'attente sont sauvegardés. Le processus serveur exécute la procédure demandée dans l'état suivant où la sauvegarde de la procédure n'est pas spécifiée, selon l'ordre normal de réception des signaux.

Les appels de procédures distantes sont modélisés par des échanges de signaux véhiculés par des canaux ou des routes implicites. **nodelay**, dans la définition d'une procédure distante, désigne des canaux sans retard de communication.

Grammaire textuelle

```
<définition de procédure distante> ::=
        remote procedure <nom de procédure distante> [ nodelay ] <fin>
        [ <signature de procédure> <fin> ]

<spécification de procédure importée> ::=
        imported procedure <nom de procédure distante> <fin>
        [ <signature de procédure> <fin> ]

<appel de procédure distante> ::=
        call <identificateur de procédure distante> [ <paramètres effectifs> ]
        [ to <destination> ]

<transition d'entrée de procédure distante> ::=
        input procedure <liste d'identificateurs de procédures distantes> <fin>
        [ <condition de validation> ] <transition>

<sauvegarde de procédure distante> ::=
        save procedure <liste d'identificateurs de procédures distantes> <fin>
```

2.4. Subdivision de blocs et canaux

Les mécanismes de subdivision fournis par SDL permettent d'appréhender plus facilement les systèmes vastes ou complexes. En effet, lorsqu'on est amené à spécifier un système vaste ou complexe, on doit subdiviser la spécification du système global en unités que l'on puisse traiter et comprendre indépendamment les unes des autres. On obtient ainsi un système global avec une structure hiérarchique. Parallèlement à l'opération de subdivision, il est nécessaire d'ajouter de nouveaux détails au comportement d'un système lorsque l'on descend vers les niveaux inférieurs de la structure hiérarchique d'un système (cette opération s'appelle **affinage**).

On peut subdiviser un bloc en un ensemble de sous-blocs, de canaux et de sous-canaux. On peut aussi subdiviser un canal en un ensemble de blocs et sous-canaux (figure 4).

2.4.1. Subdivision de blocs

Une sous-structure de bloc est similaire à une structure de système. Si `<nom de sous-structure de bloc>` est omis, il est le même que celui de `<définition de bloc>` ou `<définition de type de bloc>` englobante.

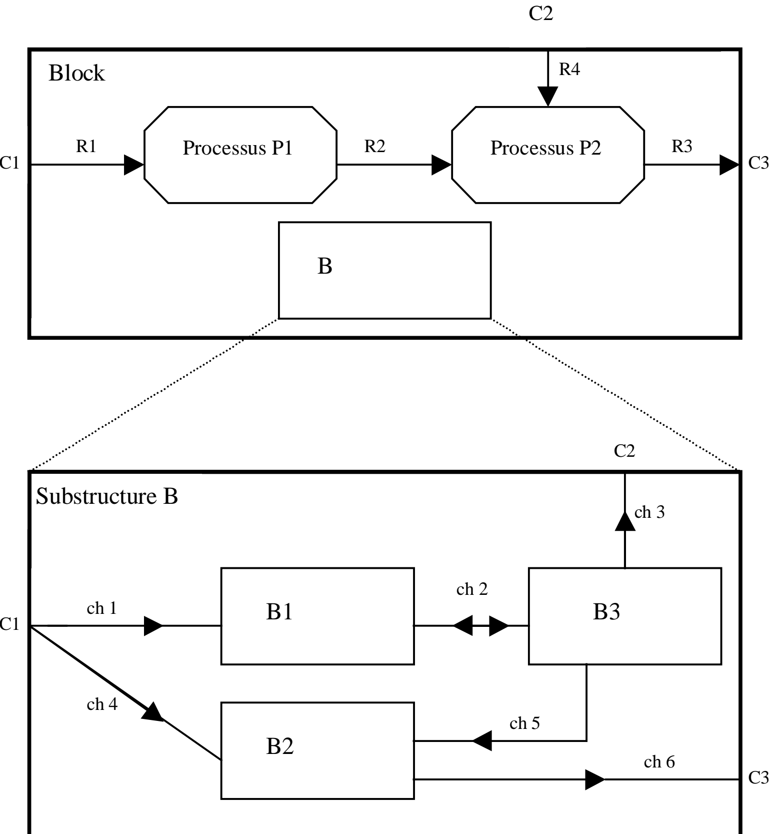

Figure 4. *Exemple de subdivision de bloc*

Grammaire textuelle

```
<définition de sous-structure de bloc> ::=
       substructure
              { [ <nom de sous-structure de bloc> ]
              | <identificateur de sous-structure de bloc> }
              <fin>
              { <entité dans système> | <connexion de canal> }+
       endsubstructure [ { <nom de sous-structure de bloc> |
                     <identificateur de sous-structure de bloc> } ]<fin>

<référence de sous-structure de bloc> ::=
       substructure <nom de sous-structure de bloc> referenced <fin>

<connexion de canal> ::=
       connect <identificateurs de canal> and <identificateurs de sous-canal> <fin>

<identificateurs de sous-canal> ::= <identificateurs de canal>
```

2.4.2 Subdivision de canaux

Pour avoir une représentation exacte de la manière dont un signal est véhiculé, dans certains cas, il est nécessaire de représenter le comportement du canal de communication (une connexion ATM, par exemple). Ceci est réalisé en considérant le canal comme une entité propre ayant comme environnement les blocs qu'il interconnecte. Ainsi, un canal peut, à son tour, être décomposé en blocs et canaux.

Si <nom de sous-structure de canal> est omis, il est le même que celui de <définition de canal> ou <nom de canal> englobante.

Les deux extrémités de la <définition de canal> subdivisé doivent être distinctes et pour chaque extrémité on doit avoir exactement une <extrémité de connexion de canal>.

Grammaire textuelle

```
<définition de sous-structure de canal> ::=
        substructure
               { [ <nom de sous-structure de  canal> ]
               | <identificateur de sous-structure de canal> } <fin>
               { <entité dans système> | <extrémité de connexion de canal> }+
        endsubstructure [ { <non de sous-structure de canal> |
               <identificateur de sous-structure de canal> } ] <fin>

<référence de sous-structure de canal> ::=
        substructure <identificateur de sous-structure de canal> referenced <fin>

<extrémité de connexion de canal> ::=
        connect { <identificateur de bloc> | env }
        and <identificateurs de sous-canal> <fin>
```

2.4.3 Raffinement de signaux

Le mécanisme de raffinement de signaux a été introduit dans SDL comme moyen de cacher les signaux de bas niveau pour les niveaux hauts d'abstraction, et permettre une description « top-down » du comportement d'un système. Le raffinement de signaux permet de partitionner un signal en un ensemble de sous-signaux. Ainsi, à l'intérieur de la description d'un signal, on peut décrire un ensemble de nouveaux signaux appelés signaux raffinés ou sous-signaux. Le raffinement peut être répété un nombre quelconque de fois, donnant une structure hiérarchique des définitions de signal et de leurs définitions de sous-signal (directes ou indirectes).

Le raffinement de signaux est étroitement lié à la subdivision de bloc, car seuls les signaux qui sont transportés par un canal connecté à un bloc subdivisé peuvent être raffinés. Quand un signal est raffiné, il est remplacé, dans le bloc subdivisé, par ses sous-signaux. Le canal transporte automatiquement tous les sous-signaux du signal raffiné.

Lorsqu'un signal est défini pour être transporté par un canal, le canal sera automatiquement le moyen de transport utilisé pour tous les sous-signaux du signal. Le raffinement peut avoir lieu lorsque le canal est subdivisé ou connecté à des sous-canaux d'une sous-structure. Dans ce cas, les sous-canaux transporteront les sous-signaux à la place du signal raffiné. Le sens d'un sous-signal est donné par le sous-canal qui le transporte. Un sous-signal peut avoir une direction opposé au signal raffiné, ce qui est indiqué par le mot clé **reverse**. Les signaux ne peuvent être raffinés qu'aux <connexion de canal> et <extrémité de connexion de canal>.

Les signaux doivent être utilisés au même niveau de raffinement dans les processus qui communiquent. Cela signifie que pour chaque ensemble d'instances de processus qui peut acheminer un signal à un autre ensemble d'instances de processus via des trajets de communication, le signal doit être utilisé au même niveau de raffinement dans les deux ensembles.

Le raffinement est fait de manière statique et il n'y a aucun lien dynamique entre un signal et ses sous-signaux.

Grammaire textuelle

```
<raffinement de signal> ::= refinement { <définition de sous-signal> }+ endrefinement

<définition de sous-signal> } ::= [ reverse ] <définition de signal>
```

2.5 Règles de visibilité et identificateurs

Certains objets peuvent être désignés avec un `<nom>` seulement, d'autres avec un `<nom>` ou un `<identificateur>`. Un `<identificateur>` est composé d'un `<qualificatif>` et d'un `<nom>`.
Tout nom (sauf les états d'un processus et les noms d'étiquettes de branchement) est visible dans la structure où il est défini et dans toutes les structures qu'elle englobe.

Le `<qualificatif>` reflète la structure hiérarchique à partir du niveau du système vers le contexte de définition, et de manière telle que le niveau du système est la partie textuelle la plus à gauche. Toutefois, on peut omettre certains `<élément de chemin>` les plus à gauche, lorsque le premier `<élément de chemin>` restant le plus à gauche dans le `<qualificatif>` est unique à l'intérieur de toute la `<définition de système>`.

Lorsque tout le `<qualificatif>` est omis et que le `<nom>` désigne une entité de la classe d'entité contenant des variables, des synonymes, des littéraux et des opérateurs, l'association du `<nom>` avec une définition doit pouvoir être résolue par le contexte réel. Dans d'autres cas, l'`<identificateur>` est associé à une entité qui a son contexte de définition dans l'unité de portée englobante la plus proche dans laquelle le `<qualificatif>` de l'`<identificateur>` est le même que la partie la plus à droite du `<qualificatif>` complet désignant cette unité de portée. Le `<qualificatif>` dans un `<identificateur>` spécifie uniquement le contexte de définition du `<nom>`.

Un sous-signal doit être qualifié par ses signaux parents, à moins qu'aucun autre signal visible n'existe à cet endroit qui porte le même `<nom>`.

Seuls les identificateurs visibles peuvent être utilisés, à l'exception de l'`<identificateur de variable>` dans une `<définition de vue>` et de l'`<identificateur>` utilisé à la place d'un `<nom>` dans une `<définition référencée>` utilisée dans la `<définition de système>`.

Une liste de définitions est associée à une unité de portée. Chaque définition décrit une entité appartenant à une certaine classe d'entités et ayant un nom associé. Pour une `<définition partielle de type>`, la liste de définitions associée comprend les `<signature d'opérateur>`, les `<signature de littéral>` et toutes `<signature d'opérateur>` et `<signature de littéral>` provenant d'une sorte parente, d'un générateur d'instance ou imposé par l'utilisation d'abréviations tel le mot-clé **`ordering`**.

Chaque entité est dite avoir son contexte de définition dans l'unité de portée qui la définit. Les entités sont référencées au moyen d'`<identificateur>`. Un `<identificateur>` est dit être visible dans une unité de portée si l'une des conditions suivantes est satisfaite :

- la partie nom de l'`<identificateur>` a son contexte de définition dans cette unité de portée,
- il est visible dans l'unité de portée qui définit cette unité de portée,
- l'unité de portée contient une `<définition partielle de type>` dans laquelle l'`<identificateur>` est défini,
- l'unité de portée contient une `<définition de signal>` dans laquelle l'`<identificateur>` est défini.

Grammaire textuelle

```
<identificateur> ::=  [ <qualificatif> ] <nom>

<nom> ::=  <mot>{ <souligné> <mot> }*

<qualificatif> ::=
       <élément de chemin> { /<élément de chemin }*
       | << <élément de chemin> { /<élément de chemin }* >>

<élément de chemin> ::=
       <genre d'unité de portée> { <nom> | <opérateur entre guillemets> }

<genre d'unité de portée> ::=
       PACKAGE | SYSTEM TYPE | SYSTEM | BLOCK TYPE | BLOCK | SUBSTRUCTURE | SIGNAL
       | PROCESS TYPE | PROCESS  | PROCEDURE | TYPE | SERVICE TYPE | SERVICE | OPERATOR
```

2.6 Représentation graphique

La grammaire graphique est décrite dans la norme ITU-T Z.100. Nous présentons, dans les figures 5-a et 5-b, les différents symboles graphiques pour permettre la compréhension de spécification de systèmes SDL décrits graphiquement.

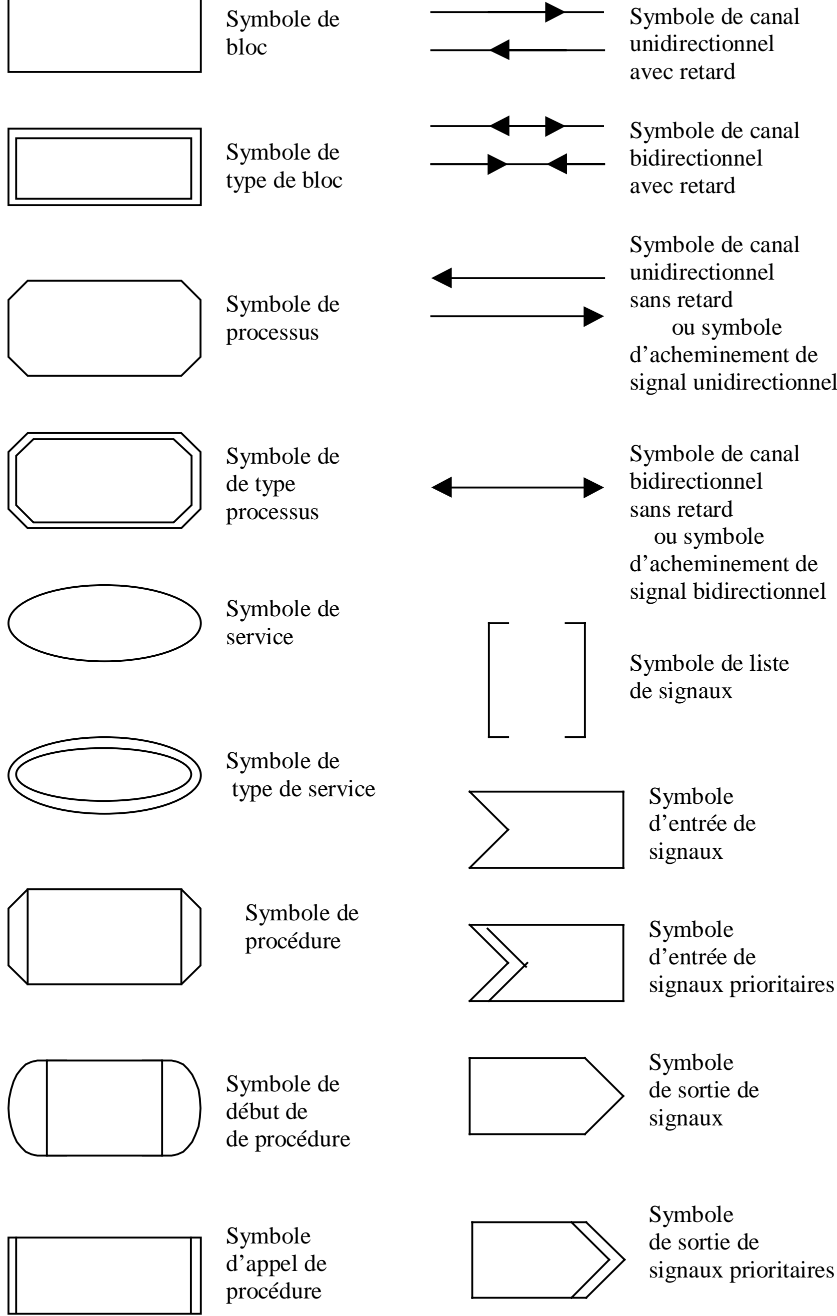

Figure 5-a. *Symboles graphiques de SDL*

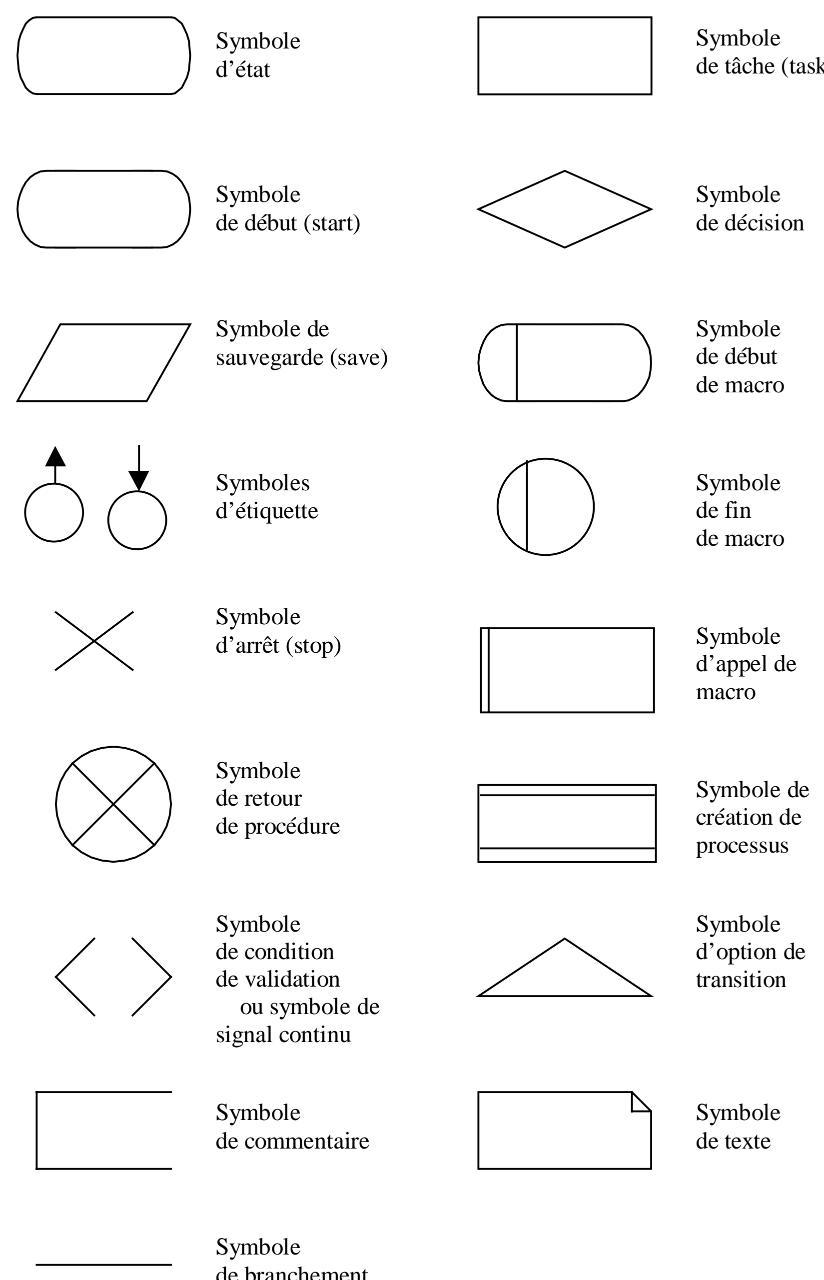

Figure 5-b. *Symboles graphiques de SDL*

3. Description de données

3.1 Introduction

Le concept de type de données de SDL est fondé sur la notion de type abstrait de données et il est conceptuellement équivalent à ACT ONE retenu aussi par LOTOS.

Dans SDL, il faut décrire formellement les types de données du point de vue de leur comportement, plutôt que de les composer à partir de primitives fournies, comme dans un langage de programmation. SDL est basé sur la notion de types abstraits, ce qui permet à l'utilisateur de se concentrer sur le comportement de son système, et il a le choix entre plusieurs implémentations plus tard.

Toute valeur appartient à une seule **sorte**. Une sorte définit un ensemble de valeurs. Ces valeurs sont construites par des littéraux (**literals**) et opérateurs (**operators**). La sémantique des opérateurs est exprimée par des axiomes (**axioms**) qui décrivent des règles d'équivalence entre des termes obtenus en appliquant les opérateurs d'une sorte. Un type abstrait de données est composé de sortes, d'opérateurs et d'axiomes.

Type abstrait de données = sortes + opérateurs + axiomes

Pour faciliter la tâche des utilisateurs familiers aux langages de programmation, SDL offre un ensemble de types prédéfinis que l'on trouve dans la plupart des langages de programmation, à savoir les types **boolean**, **integer**, **natural**, **real**, **character**, **charstring**, **duration**, **time** et **PId**. SDL permet aussi de déclarer des tableaux (grâce au générateur de type prédéfini **array**), des ensembles (grâce au générateur de type prédéfini **powerset**) et des chaînes (grâce au générateur de type prédéfini **string**). Ces types et générateurs prédéfinis sont présentés dans les tableaux 1 et 2.

```
Nom de la sorte : boolean
Littéraux : true, false
Opérateurs :
      not    : boolean -> boolean ;            /* NON logique */
      and    : boolean, boolean -> boolean ;   /* ET logique */
      or     : boolean, boolean -> boolean ;   /* OU logique */
      xor    : boolean, boolean  -> boolean ; /* OU logique exclusif */
      =>     : boolean, boolean  -> boolean ; /* implication logique */
```

Tableau 1.1. *Littéraux et opérateurs associés au type prédéfini* **boolean**

```
Nom de la sorte : integer
Littéraux : nombres entiers
Opérateurs :
      -      : integer -> integer ; /* moins unaire */
      +      : integer , integer -> integer ; /* addition */
      -      : integer , integer -> integer ; /* moins binaire */
      *      : integer , integer -> integer ; /* multiplication */
      /      : integer , integer -> integer ; /* division */
      mod    : integer , integer -> integer ; /* modulo */
      rem    : integer , integer -> integer ; /* reste de la division entière */
      >      : integer , integer -> boolean ; /* supérieur à */
      <      : integer , integer -> boolean ; /* inférieur à */
      >=     : integer , integer -> boolean ; /* supérieur ou égal à */
      <=     : integer , integer -> boolean ; /* inférieur ou égal à */
      float  : integer -> real ; /* conversion d'un entier en réel */
      fix    : real -> integer ; /* troncature d'un réel en un entier */
```

Tableau 1.2. *Littéraux et opérateurs associés au type prédéfini* **integer**

Nom de la sorte : **`natural`**
Littéraux : `entiers positifs ou nuls`
Opérateurs : `mêmes opérateurs que le type integer`

Tableau 1.3. *Littéraux et opérateurs associés au type prédéfini* **`natural`**

Nom de la sorte : **`real`**
Littéraux : `réels`
Opérateurs :

```
-      : real -> real;           /* moins unaire */
+      : real, real -> real;     /* addition */
-      : real, real -> real;     /* moins binaire */
*      : real, real -> real;     /* multiplication */
/      : real, real -> real;     /* division */
>      : real, real -> boolean ; /* supérieur à */
<      : real, real -> boolean ; /* inférieur à */
>=     : real, real -> boolean ; /* supérieur ou égal à */
<=     : real, real -> boolean ; /* inférieur ou égal à */
```

Tableau 1.4. *Littéraux et opérateurs associés au type prédéfini* **`real`**

Nom de la sorte : **`character`**
Littéraux : `un caractère entre guillemets (‘ ’)`
Opérateurs :

```
>      : character, character -> boolean ; /* supérieur à */
<      : character, character -> boolean ; /* inférieur à */
>=     : character, character -> boolean ; /* supérieur ou égal à */
<=     : character, character -> boolean ; /* inférieur ou égal à
chr    : integer -> character ;  /* conversion de numérique à caractère */
num    : character -> integer ;  /* conversion de caractère à numérique */
```

Tableau 1.5. *Littéraux et opérateurs associés au type prédéfini* **`character`**

Nom de la sorte : **`charstring`**
Littéraux : `caractères entre guillemets (‘ ’)`
Opérateurs :

```
mkstring   : character -> charstring ;
             /* conversion d'un caractère en chaîne de caractères */
length     : charstring -> integer ;
             /* longueur d'une chaîne de caractères */
first      : charstring -> character ;
             /* premier caractère d'une chaîne de caractères */
last       : charstring -> character ;
             /* dernier caractère d'une chaîne de caractères */
//         : charstring, charstring -> charstring ;
             /* concaténation de deux chaînes de caractères */
substring  : charstring, integer, integer -> charstring ;
             /* extraction d'une sous-chaîne de caractères */
extract!   : charstring, integer -> character ;
             /* extraction d'un caractère d'index donné dans une chaîne */
modify!    : charstring, integer, character  -> charstring ;
              /* modification d'un caractère d'index donné dans une chaîne */
```

Tableau 1.6. *Littéraux et opérateurs associés au type prédéfini* **`charstring`**

Nom de la sorte : **duration**
Littéraux : `les mêmes que real`
Opérateurs :

```
      duration!    : real -> duration ; /* conversion d'un réel en une durée */
      +      : duration, duration -> duration ; /* addition de deux durées */
      -      : duration -> duration;            /* moins unaire */
      -      : duration, duration -> duration ; /* soustraction entre durées */
      *      : duration, real -> duration       /* multiplication de durée */
      *      : real, duration -> duration       /* multiplication de durée */
      /      : duration, real -> duration       /* division de durée */
      >      : duration, duration -> boolean ;  /* supérieur à */
      <      : duration, duration -> boolean ;  /* inférieur à */
      >=     : duration, duration -> boolean ;  /* supérieur ou égal à */
      <=     : duration, duration -> boolean ;  /* inférieur ou égal à */
```

Tableau 1.7. *Littéraux et opérateurs associés au type prédéfini* **`time`**

Nom de la sorte : **time**
Littéraux : `les mêmes que real`
Opérateurs :

```
      time!  : real  -> time ; /* conversion d'un réel en un temps */
      +      : duration, time -> time ; /* addition d'une durée et d'un temps */
      +      : time, duration -> time ; /* addition d'un temps et d'une durée */
      -      : time, duration -> time ; /* soustraction d'une durée d'un temps */
      -      : time, time -> duration ; /* soustraction entre deux temps */
      >      : time, time -> boolean ;  /* supérieur à */
      <      : time, time -> boolean ;  /* inférieur à */
      >=     : time, time -> boolean ;  /* supérieur ou égal à */
      <=     : time, time -> boolean ;  /* inférieur ou égal à */
```

Tableau 1.8. *Littéraux et opérateurs associés au type prédéfini* **`duration`**

Nom de la sorte : **PId**
Littéraux : `Null`
Opérateurs :

```
     unique! : PId -> PId ; /* cet opérateur est défini seulement pour montrer
               qu'il existe une infinité de valeurs de PId. Il ne peut être
                utilisé que dans des axiomes */
```

Tableau 1.9. *Littéraux et opérateurs associés au type prédéfini* **`PId`**

Nom du générateur : **array** (**type** type_d_index**,** **type** type_d_element)
Littéraux : `aucun`
Opérateurs :

```
      make!      : type_d_element -> array ; /* initialisation de toutes
                   les cases d'un tableau avec la même valeur */
      extract! : Array, type_d_index -> type_d_element
                 /* extraction d'un élément d'un tableau */
      modify!  : Array, type_d_index, type_d_element -> array ;
                 /* modification d'un élément d'un tableau */
```

Tableau 2.1. *Littéraux et opérateurs associés au générateur prédéfini* **`array`**

```
Nom du générateur : powerset (type type_d_element)
Littéraux : empty /* ensemble vide */
Opérateurs :
      "in"   : type_d_element, powerset -> boolean ;  /* test d'appartenance */
      incl   : type_d_element, powerset -> powerset ; /* adjonction d'élément */
      del    : type_d_element, powerset -> powerset ; /* suppression d'élément */
      "and"  : powerset, powerset -> powerset ; /* intersection d'ensembles */
      "or"   : powerset, powerset -> powerset ; /* union d'ensembles */
      ">"    : powerset, powerset -> boolean ;
               /* test si un ensemble inclut un autre ensemble */
      "<"    : powerset, powerset -> boolean ;
               /* test si un ensemble est inclus dans un autre ensemble */
      ">="   : powerset, powerset -> boolean ;
          /* test si un ensemble inclut un autre ensemble ou s'il lui est égal */
      "<="   : powerset, powerset -> boolean ;
          /* test si un ensemble inclut un autre ensemble ou s'il lui est égal */
```

Tableau 2.1. *Littéraux et opérateurs associés au générateur prédéfini* **`powerset`**

```
Nom du générateur : string (type type_d_element, literal emptystring)
Littéraux : emptystring /* chaîne vide */
Opérateurs :
      mkstring     : type_d_element -> string ;
                     /* conversion d'un élement en une chaîne */
      length       : string -> integer ;
                     /* longueur d'une chaîne */
      first        : string -> type_d_element;
                     /* premier élément d'une chaîne */
      last         : string -> type_d_element;
                     /* dernier élément d'une chaîne */
      //           : string, string -> string ;
                     /* concaténation de deux chaînes */
      substring    : string, integer, integer -> string ;
                     /* extraction d'une sous-chaîne */
      extract!     : string, integer -> type_d_element;
                     /* extraction d'un élément d'index donné dans une chaîne */
      modify!      : string, integer, type_d_element -> string ;
                     /* modification d'un élément d'index donné dans une chaîne */
```

Tableau 2.1. *Littéraux et opérateurs associés au générateur prédéfini* **`string`**

Remarques sur les sortes et générateurs prédéfinis

1. Les opérateurs = (égal) et /= (non égal) sont implicitement définis pour tous les types.
2. Les opérateurs avec point d'exclamation (c'est-à-dire `make!`, `extract!` et `modify!`) ne peuvent être utilisés que dans les axiomes.
3. Le générateur **`powerset`** est utilisé pour définir des ensembles. Le terme **`powerset`** (qui désigne littéralement un super ensemble) est retenu pour éviter la confusion avec le mot-clé **`set`** (qui désigne l'initialisation d'un réveil).
4. Le générateur **`array`** est utilisé pour définir des tableaux. Contrairement aux langages de programmation, où des restrictions sont faites sur les index qui doivent être le plus souvent des entiers naturels, SDL permet de définir des index de n'importe quel type.
5. Le générateur **`string`** est utilisé pour définir des listes de valeurs de n'importe quel type. Il ne doit pas être confondu avec le type prédéfini **`charstring`** qui définit des listes de caractères. Un type **`charstring`** peut toujours être défini par un **`string`**, le contraire n'est pas vrai. Le générateur **`string`** a un paramètre formel (`emptystring`) désignant la chaîne vide, le type **`charstring`** n'en a pas.
6. A part l'utilisation d'un symbole de texte, il n'y a pas de grammaire graphique pour la description de données.

3.2 Définition de nouveaux types

Le spécifieur a la possibilité (en utilisant **newtype**) de définir de nouveaux types dont il décrit certaines propriétés (on parle de définition partielle de type). Une <définition partielle de type> ne produit pas toutes les propriétés du type de données, mais définit partiellement des propriétés. On peut obtenir les propriétés complètes d'un type par combinaison de toutes les définitions partielles de type qui s'appliquent à l'intérieur de l'unité de portée contenant la définition du type.

L'ensemble des sortes (resp. des opérateurs ou axiomes) d'un type de données est l'union de l'ensemble des sortes (resp. des opérateurs ou axiomes) introduites (resp. introduits) dans l'unité de portée courante.

Grammaire textuelle

```
<définition partielle de type> ::=
       newtype <nom de sorte>
              [ <propriétés étendues> ]
              <expression de propriétés>
        endnewtype [ <nom de sorte> ]

<expression de propriétés> ::=
       <opérateurs> { <propriétés internes> | <propriétés externes> }
       [ <initialisation par défaut> ]

<propriétés internes> ::=
       [ <définitions d'opérateur> ]
       axioms <axiomes> ] [ <mise en correspondance de littéral> ]
```

3.2.1 Littéraux et opérateurs paramétrés

Les noms des sortes et des opérateurs définis dans un type déterminent la signature de ce type. Une signature d'opérateur définit la façon dont l'opérateur peut être utilisé dans les expressions. C'est la signature de l'opérateur qui indique si une expression utilisant cet opérateur est correcte ou non.

Un opérateur sans argument est appelé littéral et représente une valeur fixe appartenant à la sorte résultante de l'opérateur.

Grammaire textuelle

```
<opérateurs> ::= [ <liste de littéraux> ] [ <liste d'opérateurs> ]

<liste de littéraux> ::=
       literals <signature de littéral> { , <signature de littéral> }* [ <fin> ]

<signature de littéral> ::= <nom d'opérateur littéral> | <nom de littéral étendu>

<liste d'opérateurs> ::=
       operators <signature d'opérateur> { <fin>  <signature d'opérateur> }* [ <fin> ]

<signature d'opérateur> ::=
       <nom d'opérateur> : <liste d'arguments> -> <résultat> | ordering | noequality

<nom opérateur> ::= <nom d'opérateur> | <nom d'opérateur étendu>

<liste d'arguments> ::= <sorte d'argument> { , <sorte d'argument> }*
```

```
<sorte d'argument> ::= <sorte étendue>

<résultat> :: = <sorte étendu>

<sorte étendue> ::= <sorte> | <générateur de sorte>

<sorte> ::= <identificateur de sorte>  | <syntype>
```

3.2.2. Axiomes et propriétés de type

Les axiomes déterminent quels termes représente quelle valeur. A partir des axiomes, on détermine les relations entre les valeurs des résultats des opérateurs. Les axiomes sont donnés soit sous la forme de booléens (**`true`** ou **`false`**) soit sous la forme d'équations algébriques d'équivalence.

Chaque équation exprime l'équivalence algébrique de termes. Le terme de droite d'une équation peut être substitué au terme de gauche à chaque fois que ce dernier terme apparaît dans une expression.

La quantification (par **`for`**), dans un axiome, permet de préciser explicitement les valeurs de la sorte utilisée.

Les identificateurs de valeurs sont introduits dans les axiomes pour représenter toutes valeurs de données appartenant aux sortes de quantification.

Une `<équation conditionnelle>` permet la spécification d'équations qui ne sont valides que lorsque certaines restrictions existent. Ces restrictions sont écrites sous la forme d'équations simples.

Un axiome contenant un texte informel n'est pas interprété par SDL.

Attention, dans les axiomes, le symbole '==' définit une équation d'égalité, alors que le symbole '=' désigne un opérateur booléen (test d'égalité). Dans les axiomes, l'utilisateur peut, grâce au mot-clé **`error`**, spécifier des comportements erronés (c'est-à-dire des comportements qui, s'ils sont interprétés, conduisent à un comportement indéterminé du système). Par exemple, pour interdire la division par zéro, on utilise l'axiome :

```
for all x in real   (x/0)== Error ! ;
```

Le mot-clé **`ordering`** permet de spécifier explicitement les opérateurs de relation d'ordre et un ensemble d'équations d'ordre pour une définition partielle de type.

Grammaire textuelle

```
<axiomes> ::= <équation> { <fin> <équation> }* [ <fin> ]

<équation> ::=
       <équation non quantifiée> | <équations quantifiées>
       | <équation conditionnelle> | <texte informel>

<équations quantifiées> ::= <quantification> ( <axiomes> )

<quantification> ::= for all <nom de valeur> { , <nom de valeur> }* in <sorte étendue>

<équation non quantifiée> ::= <terme> == <terme>

<terme> ::= <terme clos> | <terme composé> | <terme d'erreur> | <terme relais>

<terme composé> ::=
       <identificateur de valeur>
       | <identificateur d'opérateur> ( <liste de termes composés> )
       | ( <terme composé> ) | <terme composé étendu>

<liste de termes composés> ::=
       <terme composé> { , <terme> }* | <terme>, <liste de termes  composés>
```

```
<terme clos> ::=
       <identificateur de littéral>
       | <identificateur d'opérateur> ( <terme clos> { , <terme clos> }* )
       | ( <terme clos> )  | <terme clos étendu>

<identificateur de littéral> ::=
       <identificateur d'opérateur littéral> | <identificateur de littéral étendu>

<équation conditionnelle> ::= <restriction> { , <restriction> }* ==> <équation réduite>

<équation réduite> ::= <équation non quantifiée>

<restriction> ::= <équation non quantifiée>
```

Exemple :

```
newtype pair literals 0 ; /* définition d'un type pair */
       operators
              plus_p_p : pair, pair -> pair ;
              plus_i_i : impair, impair -> pair ;
       axioms
              plus_p_p(a,0) == a ;
              plus_p_p(a,b) == plus_p_p(b,a) ;
              plus_i_i(a,b) == plus_i_i(b,a) ;
endnewtype pair ;

newtype impair literals 1 ; /* définition d'un type impair */
       operators
              plus_i_p : impair, pair -> impair ;
              plus_p_i : pair, impair -> impair ;
       axioms
              plus_i_p(a,0) == a ;
              plus_p_i(a,b) == plus_i_p(b,a) ;
endnewtype pair ;
```

Pour éviter la division par zéro, on utilise une équation conditionnelle du type :

```
       (x /=0) == true ==> (y/x)*x == y
```

3.3 Définitions de types étendus

Les définitions de types étendues concernent les règles d'héritage, l'utilisation de générateurs de types et la définition de structures (ou enregistrements).

Un nom d'opérateur défini avec **!** a la sémantique normale, mais il n'est visible que dans les axiomes et dans les `<liste d'héritage>`.

Un opérateur infixe ou monadique, entre guillemets, a la sémantique normale mais il est, syntaxiquement, mis entre guillemets.

Grammaire textuelle

```
<propriétés étendues> ::=
       <règle d'héritage>
       | <transformations de générateur>
       | <définition de structure>
```

```
<terme composé étendu> ::=
        <identificateur d'opérateur étendu> ( <liste de termes composés> )
        | <terme composé> <opérateur infixe> <terme>
        | <terme> <opérateur infixe> <terme composé>
        | <opérateur monadique> <terme composé>
        | <terme composé conditionnel>

<terme clos étendu> ::=
        <identificateur d'opérateur étendu> ( <terme clos> { , <terme clos> }* )
        | <terme clos> <opérateur infixe> <terme clos>
        | <opérateur monadique> <terme clos>
        | <terme clos conditionnel>

<identificateur d'opérateur étendu> ::=
        <identificateur d'opérateur> !
        | <générateur de nom formel>
        | [ <qualificatif> ] <opérateur entre guillemets>

<nom d'opérateur étendu> ::=
        <nom d'opérateur> !
        | <générateur de nom formel>
        | <opérateur entre guillemets>

<liste de noms étendus> ::=
        <littéral de chaîne de caractères>
        | <générateur de nom formel>
        | <littéral de classe de nom>

<opérateur entre guillemets> ::= "<opérateur infixe>" | "<opérateur monadique>"

<opérateur infixe> ::=
          =>  |   or   |   xor   |   and   |   in   |   mod
        | /=  |   =    |   >     |   <     |   <=   |   rem
        | >=  |   +    |   /     |   *     |   //   |   -

<opérateur monadique> ::= - | not

<identificateur de littéral de chaîne de caractères> ::=
        [ <qualificatif> ] <littéral de chaîne de caractères>

<littéral de chaîne de caractères> ::= <chaîne de caractères>

<terme conditionnel composé> ::=  <terme conditionnel>

<terme clos conditionnel> ::= <terme conditionnel>

<terme conditionnel> ::= if <condition> then <terme> else <terme> fi

<terme d'erreur> ::= error !
```

3.3.1 Les structures (ou enregistrements)

Des structures de données composées (ou enregistrements) peuvent être construites à partir d'autres données. Une `<définition de structure>` définit une sorte de structure dont les valeurs sont composées à partir d'une liste de valeurs de champs appartenant à des sortes données.

Grammaire textuelle

```
<définition de structure> ::= struct <liste de champs> <fin> [ adding ]

<liste de champs> ::= <champs> { <fin> <champs> }*

<champs> ::= <nom de champ> { , <nom de champ> }* <sorte de champ>

<sorte de champ> ::= <sorte>
```

Exemple

```
/* définition d'un type s composé de trois champs */
newtype s struct
      b boolean ;
      i integer ;
      c character ;
endnewtype s ;
```

3.3.2 Héritage

Comme dans les langages orientés objet, le concept d'héritage de SDL permet d'éviter de réutiliser des spécifications déjà existantes. Les valeurs, les opérateurs et les équations (dans les axiomes) peuvent être hérités par une sorte-fille à partir d'une sorte-parente. Des littéraux, des opérateurs et des équations peuvent être ajoutés à la sorte-fille par le mot-clé **`adding`**. Le mot clef adding est optionnel ; il permet d'accroître la lisibilité de spécification.

Grammaire textuelle

```
<règle d'héritage> ::=
      inherits <sorte de type> [ <renommage de littéraux> ]
      [ [ operators ] { all | (<liste d'héritage>) } [ <fin> ] ] [ adding ]

<liste d'héritage> ::= <opérateur hérité> { , <opérateur hérité> }* [ , noequality ]

<opérateur hérité> ::= [ <nom d'opérateur> = ] <nom d'opérateur hérité>

<nom d'opérateur hérité> ::= <nom d'opérateur de base>

<renommage de littéraux> ::= literals <liste de renommage de littéraux> <fin>

<liste de renommage de littéraux> ::=
      <paire de renommage de littéral> { , <paire de renommage de littéral> }*

<paire de renommage de littéral> ::
      <signature de renommage de littéral> =
      <signature de renommage de littéral de base>


<signature de renommage de littéral> ::=
      <nom d'opérateur littéral> | <littéral de chaîne de caractères>
```

Exemple

```
newtype bit
  inherits boolean
         literals 1 = true, 0= false ;
         operators ("not", "and", "or")
  adding
  operators
         exor : bit, bit -> bit ;
  axioms
         exor(a,b) == (a and (not b)) or ((not a) and b) ;
endnewtype bit ;
```

3.3.3 Générateurs

Il arrive parfois que les différences entre deux ou plusieurs de sortes soient mineures. Dans ce cas, on peut définir un type partiel paramétré qui peut être instancié ensuite avec des valeurs de paramètres permettant d'avoir plusieurs sortes. Par exemple, les propriétés d'un ensemble sont relativement indépendantes de celles des éléments qui le composent. Sans la paramétrisation de type, chaque ensemble doit être spécifié séparément. Une `<définition de générateur>` permet de définir un type paramétré.

Grammaire textuelle

```
<définition de générateur> ::=
       generator <nom de générateur>
              ( <liste de paramètres du générateur> )
              <texte du générateur>
       endgenerator [ <nom de générateur> ]

<texte de générateur> ::= [ <transformations de générateur> ] <expression de propriétés>

<liste de paramètres du générateur> ::=
       <paramètre de générateur> { , <paramètre de générateur> }*

<paramètre de générateur> ::=
       { type | literal | operator | constant }
       <nom formel de générateur> { , <nom formel de générateur> }*

<transformations de générateur> ::=
       { <transformation de générateur> [ <fin> ] [ adding ] }+

<transformation de générateur> ::=
       <identificateur de générateur> ( <liste de générateurs actuels> )

<liste de générateurs actuels> ::= <générateur actuel> { , <générateur actuel> }*

<générateur actuel> ::=
       <sorte étendue> | <signature de littéral> | <nom d'opérateur> | <terme clos>
```

Exemple :

```
       /* Définition d'un type ensemble d'entiers */
       newtype ensemble_entier
              literals ensemble_vide_entier ;
              operators
                     add : ensemble_entier, integer -> ensemble_entier ;
                     est_dans : ensemble_entier, integer -> boolean ;
              axioms
                     for all m, n in integer, s in ensemble_entier
                     ( est_dans(ensemble_vide_entier, m) == false ;
```

```
                  est_dans(ajouter(s, m), n) == (m = n) or (est_dans(s, n)) ;
                  m=n and ajouter(add(s,m),n) == add(s,m) ;
                  m/=n and ajouter(ajouter(s,m), n) == ajouter(ajouter(s, n), m) ;
                 ) ;
endnewtype ensemble_entier ;
```

Si on veut avoir un ensemble de réels (`ensemble_reel`), il faut réécrire les mêmes choses que pour `ensemble_entier` en remplaçant `entier` par `reel`. On peut éviter une telle tâche fastidieuse en définissant et en utilisant un générateur de type `ensemble` de la manière suivante :

```
/* Définition d'un générateur d'ensemble */
generator ensemble (type element, literals ensemble_vide)
      literals ensemble_vide ;
      operators
            add : ensemble, integer -> ensemble ;
            est_dans : ensemble, integer -> boolean ;
      axioms
            for all m, n in element, s in ensemble
            ( est_dans(ensemble_vide, m) == false ;
              est_dans(ajouter(s, m), n) == (m = n) or (est_dans(s, n)) ;
              m=n and ajouter(add(s,m),n) == add(s,m) ;
              m/=n and ajouter(ajouter(s,m), n) == ajouter(ajouter(s, n), m) ;
             ) ;
endgenerator ;

/* Utilisation du générateur de type */
newtype ensemble_entier ensemble(integer, ensemble_vide_entier) endnewtype ;
newtype ensemble_reel ensemble(real, ensemble_vide_reel) endnewtype ;
```

3.4 Utilisation des données

L'accès à une variable s'effectue en utilisant le nom de cette variable. La valeur d'une variable peut être utilisée dans une expression ou modifiée par affectation. L'affectation est désigné par **`:=`**.

L'utilisation des opérateurs est la même que dans les langages de programmation.

L'accès aux éléments d'un tableau s'effectue de la manière suivante :

```
<accès à un élément de tableau> ::= <nom de tableau>(<liste des indices>)
```

L'accès aux champs d'une structure peut s'effectuer de deux manières :

```
<accès à un champ de structure> ::= <nom de structure>!<nom de champ>

<accès à tous les champs d'une structure> ::= <nom de structure>
             (. <liste d'expressions> .)
```

La première forme permet d'accéder à un champ d'une structure pour le lire ou le modifier. La seconde permet d'initialiser une structure.

Exemple

```
/* Définition d'un type T_couleur */
newtype T_couleur
      literals Rouge, Jaune, Bleu, vert, Noire, Blanc ;
endnewtype T_couleur ;
```

```
/*  définition du type voiture */
newtype voiture
       struct
              marque  charstring,
              annee natural,
              couleur T_couleur,
              numero_licence natural,
              proprietaire T_personne ;
       adding
       literals Laguna ;
       axioms marqueExtract!(Laguna) == Renault ;
endnewtype voiture ;

/* utilisation du type voiture */
dcl vehicule1 voiture, vehicule2 voiture, Martin T_personne Jean T_personne ;
dcl couleurX T_couleur ;
task vehicule1!proprietaire := Martin ;
task vehicule1!couleur := Rouge ;
task vehicule2 := (. Peugeot, 1999, Bleu, 1345, Jean .) ;
task couleurX := vehiculeX!couleur ;
```

Lorsqu'on définit les axiomes associés à un type **struct**, les opérateurs suivants sont prédéfinis :

make! : pour construire une valeur à partir de valeurs de champs individuelles,

<nom de champ>**Modify**! : pour modifier la valeur d'un champ,

<nom de champ>**Extract**! : pour obtenir la valeur d'un champ.

Remarque : Make!, **Modify!** et **Extract!** ne sont utilisables que dans les axiomes.

Une expression conditionnelle permet de désigner une expression parmi deux selon la valeur d'une condition. Elle est de la forme :

```
if  <expression booléenne>
       then <expression>
       else <expression>
fi
```

Par exemple : `task maxAB := if A > B then A else B fi ;`

Expressions particulières

1. Les opérateurs impératifs fournissent des valeurs de l'état du système. Il s'agit des opérateurs **now**, **self**, **parent**, **offspring** et **sender** que nous avons déjà présentés.

Exemple :
```
task MonPere := parent ;
task t1 := now + 100 ;
```

2. Une expression de vue, en utilisant **view** (cf. §2.3.8.10), permet à un processus d'obtenir la valeur d'une variable d'un autre processus du même bloc.

Exemple : `task X := view(VarRevelee) + 5 ;`

3. Une expression d'importation **import**(<variable importée>) permet à un processus d'obtenir la valeur exportée par un autre processus d'un autre bloc.

Exemple : `task Y := import(VarImporte)/a;`

4. L'expression **`active`**`(<identificateur de réveil>)`, vue précédemment, permet d'avoir une expression booléenne dont la valeur dépend de l'état du réveil testé.

Exemple : **task** A := **if active**(reveil2) **then** A **else** C **fi** ;

5. L'expression **`any`**`(<sorte>)` permet de spécifier une valeur quelconque d'une `<sorte>` donnée. L'opérateur **`any`** permet spécifier des comportements non-déterministes.

Données externes

SDL permet d'intégrer, dans une spécification de système, des données externes, c'est-à-dire des données non décrites formellement par la syntaxe SDL. Cela permet d'introduire, à un niveau de la spécification, des données qui seront explicitées plus tard. Une `<spécification de propriétés externes>` se fait via le mot-clé **`alternative`**.

Grammaire textuelle

```
<spécification de propriétés externes> ::=
       alternative
              <nom de formalisme externe> [ , <mot> ] <fin>
              <description de données externes>
       [ endalternative ] [ <fin> ]

<nom de formalisme externe> ::= <texte>

<description de données externes> ::= <texte>
```

3.5 Autres constructions de définitions de données

1. *Syntype*

Le mot-clé **`syntype`** permet d'associer plusieurs noms à une même sorte pour augmenter la lisibilité des spécifications selon le contexte d'utilisation.

Grammaire textuelle

```
<définition de syntype> ::=
       syntype
              <nom de syntype> = <sorte>
              [ <initialisation par défaut> ] [ constants <condition d'intervalle> ]
       endsyntype [ <nom de syntype> ]

<condition d'intervalle> ::=
       { <intervalle fermé> | <intervalle ouvert> }
       {, {<intervalle fermé> | <intervalle ouvert> }}*

<intervalle fermé> ::=  <constante> : <constante>

<intervalle ouvert> ::= <constante> | { = | =/ | > | < | >= | <= } <constante>
```

Exemple :

```
syntype compteur = integer constants 1 : 99 endsyntype ;
syntype rayon = real endsyntype ;
```

2. *Synonyme*

Le mot-clé **`synonym`** permet de définir des constantes.

Exemple :

```
        synonym taille_maximale integer = 100 ;
```

Grammaire textuelle

```
<définition de synonyme> ::=
        synonym <item de définition de synonyme> { , <item de définition de synonyme> }*

<item de définition de synonyme> ::= <nom de synonyme>  [ <sorte> ] = <expression close>
```

3. *Mise en correspondance de littéral*

Les `<mise en correspondance de littéral>` (effectuées par le mot-clé **map**) sont des notations abrégées pour faire la correspondance entre des littéraux et des valeurs.

Grammaire textuelle

```
<mise en correspondance de littéral> ::=
        map <équation littérale> { <fin> <équation littérale> }* [ <fin> ]

<équation littérale> ::=
        <quantification littérale> ( <axiomes littéraux>
        { <fin> <axiomes littéraux> }* [ <fin> ] )

<axiomes littéraux> ::= <équation> | <équation littérale>

<quantification littérale> ::=
        for all <nom de valeur> { , <nom de valeur> }* in <sorte étendue> literals
```

4. *Définition d'opérateur*

Des `<définition d'opérateur>` (effectuées par **operator**) permettent de définir des opérateurs de manière semblable aux procédures renvoyant une valeur.

Grammaire textuelle

```
<définition d'opérateur> ::=
        operator
                { <identificateur d'opérateur> | <nom d'opérateur> } <fin>
                <paramètres formels> <fin> <résultat d'opérateur> <fin>
                {       <définition de données> | <définition de variable>
                        | <définition de macro> | <définition de sélection>
                }*
                <début>  { <action> }*
        endoperator
                [ { <identificateur d'opérateur> | <nom d'opérateur> } ] <fin>

<résultat d'opérateur>::= returns [ <nom de variable> ] <sorte étendue>
```

Exercices

Exercice 1

```
/* Exemple d'une spécification d'un système composé d'un seul bloc. Lequel bloc est
composé de deux processus P1 et P2.  Il y a une seule instance pour chacun des deux
processus. Le processus P1 envoie des messages à P2 et attend un acquittement. Il arme
une temporisation pour éviter d'attendre indéfiniment l'ACK. Si l'ACK arrive avant le
déclenchement de la tempo, P1 envoie le message suivant. Sinon il renvoie le même message
Nb_max fois au plus. De manière aléatoire, le processus P2, renvoie un ACK ou non, cela
permet de simuler les pertes de messages ou d'ACK. Le nombre maximum de valeurs qui
peuvent être traitées par P2 est de 10. Lorsque la valeur contenue dans le message
dépasse un certain seuil, P2 s'arrête. Lorsque P1 veut s'arrêter, il envoie un signal
SIG_FIN à P1. */

system S1;
  block B1 referenced;
endsystem S1;

block B1;
  signal SIG_ACK, SIG_message(Integer), SIG_FIN;
  signalroute R1 from P1 to P2 with SIG_message;
  signalroute R2 from P2 to P1 with SIG_ACK, SIG_FIN;
  process P1 (1, 1) referenced;
  process P2 (1, 1) referenced;
endblock B1;

process P1 ;
  timer tim1;
  dcl Val_message Integer;
  dcl delai Duration ;
  dcl Nb_retransmissions Integer ;
  synonym Nb_max = 10 ; /* Nombre maximum de retransmissions */
 start;
    task Val_message := 5 ;
    task delai := 10;
    task Nb_retransmissions := 0;
    output SIG_message(Val_message) to P2;
    set(NOW+delai,tim1);
    nextstate attente;
 state attente;
    input SIG_ACK;
         task Nb_retransmissions := 0;
         task Val_message := Val_message*2;
         output SIG_message(Val_message) to P2 ;
         set(NOW+delai,tim1);
         nextstate attente;
     input tim1;
         task Nb_retransmissions := Nb_retransmissions + 1;
         decision Nb_retransmissions > Nb_Max ;
              (true) : stop ;
              (false) :
                 /* Apres expiration du timer, renvoyer la même valeur */
                 output SIG_message(Val_message) to P2;
                 set(NOW+delai,tim1);
                 nextstate attente ;
           enddecision;
      input SIG_FIN ;
         stop;
endprocess P1;

process P2 ;
   dcl x, cpt Integer;
   start;
     task cpt := 1;
     nextstate attente;
```

```
    state attente;
     input SIG_message(x);
             decision any ; /* Envoi de l'ACK de manière aléatoire. */
                () : nextstate attente ;
                () : task cpt := cpt + 1;
                     decision x>100 or cpt > 10;
                        (true):
                           output SIG_FIN to P1;
                           stop;
                        (false):
                           output SIG_ACK to P1 ;
                     enddecision;
                     nextstate attente;
              enddecision;
endprocess P2;
```

Questions :

Changer la spécification du système précédent pour tenir compte des besoins suivants :

1. Les deux processus `P1` et `P2` ne communiquent pas directement, mais par l'intermédiaire d'un processus que nous appelons `Reseau` (qui permet de simuler le réseau de communication). Aucun des deux processus ne perd de message. Par contre, le réseau peut perdre des messages (de données ou d'acquittement).

2. Pour communiquer les deux processus passent par une phase d'établissement de connexion. De manière inconnue à l'avance, un des deux processus peut demander l'établissement de connexion. En cas de perte de message, un initiateur de connexion ne peut retransmettre le message de demande de connexion que cinq fois au maximum.
Les deux processus peuvent demander simultanément l'établissement de connexion.
La connexion peut être rompue par l'un ou l'autre des deux interlocuteurs.
Si `P1` termine de transmettre toutes ses données, il ferme la connexion.
Si `P2` reçoit une valeur qui dépasse un seuil donné, il ferme la connexion.

3. Les deux processus utilisent un mécanisme de contrôle de flux avec une fenêtre d'anticipation de `N` (`N` fixé) messages.

Exercice 2
On s'intéresse à la spécification simplifiée d'un système multimédia composé de deux sources de données (une source `S_audio` qui échantillonne du son à une fréquence `F1`) et une source `S_video` (qui échantillonne de l'image à une fréquence `F2`). Les fréquences `F1` et `F2` peuvent être ajustées en fonction de la qualité audiovisuelle souhaitée.
Chaque échantillon de la source `S_audio` est représenté un entier et chaque échantillon de la source `F2` est représenté par une matrice d'entiers `N*M` (`N` est le nombre de lignes et `M` le nombre de colonnes de l'image).
Les données échantillonnées par les deux sources sont envoyées vers un récepteur `S_animation` qui les utilise pour animer une scène (ou projeter un film).
Les données des échantillons en provenance de la source `S_audio` sont envoyées (par `S_animation`) vers un organe périphérique audio et celles en provenance de la source `S_video` sont envoyées vers un organe vidéo. Il y a une synchronisation entre les deux flux : chaque échantillon de la source `S_audio` est présenté simultanément avec `k` échantillons de `S_audio`. La valeur de `k` est paramétrable.

La première phase du projet consiste à spécifier en SDL les trois équipements audiovisuels en supposant qu'ils communiquent par un canal SDL sans retard.

La deuxième phase du projet consiste à faire communiquer les trois équipements via un réseau de type bus à jeton. Le réseau considéré est composé de quatre stations seulement `S_audio`, `S_video`, `S_animation` et une station `Y` dont le seul objectif est de générer un trafic aléatoire qui fait varier la charge du réseau.

Pour modéliser en SDL le protocole du bus à jeton implanté sur les quatre stations, on fera les hypothèses suivantes :

- l'anneau logique est préconfiguré de la manière suivante `S_audio` → `S_video` → `S_animation` → `Y` → `S_audio`.
- à l'initialisation du système, le jeton se trouve dans la station `Y`.
- Il n'y a pas d'opérations de maintenance de l'anneau logique (pas d'insertion ou de retrait de station, pas de perte du jeton, pas de duplication de jeton, pas d'erreur de transmission, …) .

Après avoir modélisé le réseau et les équipements multimédia, on veut analyser la gigue de communication correspondant à chacun des deux flux, pour éventuellement effectuer des réglages du système. Pour cela, l'équipement `S_animation` affiche sur un organe périphérique la gigue de chacun des deux flux de manière périodique.