Type document : *SRS*	*Projet RNTL MORSE*	**Date :** 13/06/2004
Sous-projet n° 2		**numéro de doc :** MORSE-SRS-041105-V0.1-FGI
tâche : 2.1		**rédacteur (s)** Frédéric Gilliers

BNF du langage L*f*P

Table des matières

Table des figures

1 Liste des modifications

date	Objet de la modification
05/11/2004	Première version présentée du document.

2 Introduction

Ce document présente la syntaxe textuelle du langage **L*f*P** développée dans le cadre de MORSE. L'objectif de ce format est de faciliter les échanges de spécifications **L*f*P** entre les partenaires du projet.La grammaire est présentée de manière contextuelle. Ce document n'est donc pas une documentation de l'implémentation du parseur. En particulier, il n'y a pas de correspondance directe entre les règles présentées ici et les règles effectivement implémentées dans le fichier de grammaire du parseur.

La section 3 rappelle les conventions utilisées dans les documents de syntaxe **L*f*P**. La section 6 présente la syntaxe de la partie déclarative du langage. Cette section définit principalement les déclarations de types du langage et la représentation du diagramme d'architecture. La section 7 présente la syntaxe des instructions du langage **L*f*P**.

3 Conventions de présentation

Ce document présente les règles de syntaxe en respectant les conventions suivantes :

- les mots clés seront écrits en rouge, et en minuscule ;
- les règles de syntaxe non terminales sont écrites en pourpre, et en minuscules ;

– les règles terminales du parseur sont écrites en bleu et en majuscules ;
– les caractères faisant partie de la syntaxe du lanagage **L*f*P** sont écrits en vert
– Les caractères en noir sont les caractères de description de la BNF, ils ne font pas partie du langage **L*f*P**.

Le mot ***composant*** sera utilisé pour désigner indifféremment une classe ou un média. Il est inutile de distinguer ces deux entités partout où ce mot est utilisé.

4 Éléments de base du langage

4.1 Casse des caractères

Le langage **L*f*P** n'est pas sensible à la casse des caractères. Par convention dans ce document, les mots réservés seront présentés en minuscules.

4.2 Mots réservés et symboles utilisés par le langage L*f*P

4.2.1 Mots réservés du langage

Les mots réservés utilisés par le langage **L*f*P** sont rassemblés sur la figure 1. L'utilisation d'un de ces lexème comme identificateur est interdite.

and	const	if	out	then
array	declare	in	opaque	trigger
asynchronous	else	inout	port	to
accept	elsif	is	procedure	type
bag	enum	label	range	while
begin	end	not	record	with
binder break	fifo	null	return	
channel	for	media	set	
circular	function	of	synchronous	
class	goto	or	static	

FIG. 1 – Liste des mots clés du langage **L*f*P**

4.2.2 Symboles utilisés par le langage

Les symboles utilisés par le langage lfp sont rassemblés sur la figure 2.

->	=>	:	=	'
<-	(	+	/=	/
<->	)	-	<=	*
<=>	[	*	>=	#
|->	]	/	:=	
<-|	;	>	.	
@	,	<	..	

FIG. 2 – Symboles utilisés par le langage **L*f*P**

4.3 Identificateurs valides

Le lexème correspondant aux identificateurs valides en **L*f*P** est présenté sur la figure 3. Un identificateur valide commence donc nécessairement par une lettre et peut être suivi d'un nombre quelconque de lettre, chiffre ou caractère souligné ("_").

$$IDENTIFIER \quad : \quad [\,a-z]+\ [\,a-z0-9\_]*$$

FIG. 3 – Lexème définissant un identificateur en **L*f*P**

4.4 Chiffres et nombres

Seuls les nombres entiers sont supportés par le langage **L*f*P**. Ils sont représentés par une suite de chiffres sans espaces. Ils sont définis par le lexème présenté sur la figure 4.

$$integer \quad : \quad [\,0-9]+$$

FIG. 4 – Lexème définissant un entier

5 Syntaxe des expressions

Cette section présente les opérateurs disponibles pour l'écriture des expression en **L*f*P**. Toute expression **L*f*P** doit retourner une valeur, les expressions **L*f*P** incluent donc les expressions arithmétiques, les appels de fonction, les variables, etc...

Le langage **L*f*P** définit les opérateurs suivants (classés par niveau croissant de priorité) :

1. < , > , <= , >= , = , /= représentent respectivement pour les éléments d'un interval donné :
   - inférieur strict ;
   - supérieur strict ;
   - inférieur ou égal ;
   - supérieur ou égal ;
   - égalité ;
   - différent (opérateur complémentaire de =).

   Les opérateurs < , <= , = , keyword/= sont également définis pour les ensembles. Il représentent alors respectivement les opérateurs suivants :
   - inclusion stricte ;
   - inclusion ;
   - égalité ;
   - différent (opérateur complémentaire de =).
2. + , - et or représentent respectivement l'addition, la soustraction et le *"ou"* booléen. Lorsqu'ils sont appliqués sur des ensembles, les opérateurs + et - représentent respectivement l'union et l'intersection ensembliste.
3. * , / et and représentent respectivement la multiplication, la division et le *"et"* booléen.
4. not , - , # représetent respectivement le *"non"* booléen, le moins unaire applicable sur les entiers et l'opérateur de sélection d'un élément dans un ensemble ou un multi-ensemble.

La figure 5 donne la BNF des expressions valides en **L*f*P**. La règle operator désigne l'ensemble des opérateurs listés dans les paragraphes précédents.

La règle 1 correspond à l'utilisation d'un des opérateurs binaires précédemment listé dans le cadre d'une expression arithmétique.

La règle 2 correspond à l'écriture d'une expression entre parenthèses.

La règle 3 correspond à l'utilisation d'un attribut de type :
- le premier identificateur est le nom du type dont l'attribut est exploité ;
- le deuxième identificateur est le nom de l'attribut utilisé ;
- l'expression entre parenthèses est l'expression sur laquelle l'attribut de type est appliqué.

Les noms des attributs ne sont pas des mots réservés du langage **L*f*P**, ils peuvent donc être utilisés comme identificateurs.

```
expression   :  expression operator expression                          (1)
             |  ( expression )                                          (2)
             |  IDENTIFIER ' ( expression )                             (3)
             |  expression @ IDENTIFIER ( [ expressionlist ] )          (4)
             |  IDENTIFIER ( [ expressionlist ] )                       (5)
             |  expression . IDENTIFIER                                 (6)
             |  appel_fonction                                          (7)

expressionlist  :  expression [ , expression ]*                         (8)
```

FIG. 5 – Syntaxe des expressions valides en **L*f*P**

La règle 4 correspond à l'appel d'une fonction externe sur un type opaque. L'expression à gauche du @ désigne l'instance de composant opaque sur lequel l'appel est effectué. L'identificateur est le nom de la méthode externe appelée. Le nom de la méthode est suivi de la liste de ses paramètres entre parenthèses.

La règle 5 a deux interprétations en fonction du type associé à l'identificateur :

1. si l'identificateur désigne le nom d'un tableau, il s'agit d'un accès à son contenu, et le ou les indices de l'élément sélectionné sont fournis entre parenthèses. Il y avoir au moins un indice de spécifié.
2. si l'identificateur correspond à un nom de trigger, il s'agit d'un appel de trigger retournant une valeur. La liste d'expression entre parenthèses correspond aux paramètres du trigger.

La règle 8 correspond à une liste d'expressions séparées par des virgules. Cette construction est très couramment utilisée en **L*f*P**, soit pour les paramètres des appels de fonction (comme pour la règle 4), soit pour la définition d'un élément de tableau (règle 5).

La règle 6 correspond à la dé-référenciation qui peut être utilisée dans deux contextes :

1. l'accès au champs d'une structure ;
2. l'accès à un port d'un composant.

La règle 7 correspond à un appel de fonction. La syntaxe de cette instruction sera présentée à section 7.12.

6 Syntaxe de la partie déclarative

Cette section présente la partie déclarative du langage **L*f*P**. Cette partie permet de définir les types de données du modèle. La partie statique d'un modèle **L*f*P** est constituée en deux parties :

- la partie déclarative globale du modèle ;
- la partie déclarative des composats.

La partie déclarative globale contient les déclarations anciennement définies sur le diagramme d'architecture, c'est à dire :

- les composants du modèle (clases et médias) ;
- les types et constantes partagés par les composants ;
- les binders ;
- les liaisons entre les composants de l'application.

La partie déclarative des composants permet de définir les attributs des composants, c'est à dire :

- les types privés ;
- les variables et constantes locales ;
- les méthodes (uniquement pour les classes).

Cette section présente d'abbord les déclarations des types de données valides en **L*f*P**, puis les déclarations des éléments spécifiques au diagrammes d'architecture (liens et binders).

6.1 Déclaration des types tableaux

La syntaxe des déclarations de types tableaux est donnée sur la figure 6.

type_tableau : *type IDENTIFIER is array* (*range_list*) *of IDENTIFIER* ; (9)

range : *expression* .. *expression* (10)

range_list : *range* [, *range*]∗ (11)

FIG. 6 – BNF de la déclaration d'un type tableau et des intervalles de valeurs

La règle 9 présente la BNF de la déclaration d'un type tableau :
- le premier identificateur correspond au nom du nouveau type ;
- la liste d'intervalle spécifiés entre parenthèses correspond à la définition des dimensions du tableau ;
- enfin le dernier identificateur à droite du mot clé of spécifie le type des éléments du tableau.

La règle 10 permet d'exprimer un intervalle de valeur, les deux expressions définissent respectivement les bornes inférieure et suppérieure de l'intervalle.

La règle 11 définit une liste d'intervalles séparés par des virgules, cette règle est utilisée pour définir les dimenstion des types tableaux.

6.2 Déclaration des types énumérés

La syntaxe de la déclaration des types énumérés est fournie par la figure 7.

type_enumere : *type IDENTIFIER is enum* (*identifier_list*) ; (12)

| *type IDENTIFIER is circular enum*
(*identifier_list*) ; (13)

identifier_list : *IDENTIFIER* [, *IDENTIFIER*]∗ (14)

FIG. 7 – Syntaxe des déclarations de types énumérés en **L*f*P**

La règle 12 définit un type énuméré. Le premier identificateur définit le nom du nouveau type. Il est suivi d'une liste d'identificateurs qui définissent toutes les valeurs du type.

La règle 13 définit un type énuméré circulaire, le mot réservé circular est rajouté à la règle 12 pourpréciser la nature du type.

La règle 14 définit une liste d'identificateurs séparés par des virgules, elle est utilisée pour spécifier l'ensemble des valeurs valides du type.

6.3 Déclaration des intervalles de valeurs

Un type définit par un intervalle de valeur définit un sous-type d'un type existant (type natif ou déjà déclaré par l'utilisateur). Il est possible de définir un sous-type d'un type entier, d'un type intervalle ou d'un type énuméré. La figure 8 donne la BNF de la déclaration d'un type défini par un intervalle.

La règle 15 définit un type intervalle. Le premier identificateur est le nom du nouveau type. La règle range définit l'intervalle de valeurs valide pour ce type, celle-ci a été définie à la figure 6. Enfin le dernier identificateur désigne le type parent du nouveau type.

La règle 16 définit un type interval circulaire. Seul le mot clé circular est rajouté à la règle 15 pour préciser la nature de l'intervalle.

type_range : *type IDENTIFIER is range* (*range*) *of IDENTIFIER* , (15)
| *type IDENTIFIER is circular range* (*range*) *of IDENTIFIER* (16)

FIG. 8 – BNF d'un type défini par intervalle de valeur

6.4 Déclaration des structures

La BNF de la déclaration d'un type record est donnée par la figure 9.

type_record : *type IDENTIFIER is record* [*champs*]+ *end* ; (17)

champs : *IDENTIFIER* [, *IDENTIFIER*]∗ : *IDENTIFIER* ; (18)

FIG. 9 – BNF de la déclaration d'un type record

La règle 17 correspond à la déclaration du nouveau type dont le nom est donné par l'identificateur. Le mot réservé record est suivi de la liste des champs déclarés.

La règle 18 donne la BNF de la déclaration d'un champs pour un type record. Chaque champs est défini par son nom suivi de son type séparés par le caractère " : ". Une syntaxe abrégée est fournie dans le cas ou plusieurs champs ont le même type, il est possible de les déclarer en une seule fois à l'aide d'une liste d'identificateurs séparés par des virgules.

6.5 Déclaration des types port

Les types port permettent de définir le type des messages envoyés vers les binders. La BNF de leur déclaration est donnée sur la figure 10

type_port : *type IDENTIFIER is port* ([*identifier*[, *IDENTIFIER*]∗]) ; (19)

FIG. 10 – BNF de la déclaration d'un type port

La règle 19 définit la BNF d'un type port. Le premier identificateur est le nom du nouveau type. La structure des discriminants associés à ce type est définie entre parenthèse : il s'agit d'une liste (potentiellement vide) d'identificateurs séparés par des virgules. Chacun de ces identificateurs désigne le type d'un élément du discriminant.

6.6 Déclaration de variables et de constantes

La figure 11 donne la BNF des déclarations de variables et de constantes. Les constantes peuvent être déclarées soit de manière globale (partagées par tous les composants du système, soit de manière locale comme attribut d'un composant (classe ou média).

La règle 20 définit la déclaration d'une constante :
- le premier identificateur est le nom de la variable ;
- le deuxième identificateur définit son type
- enfin l'expression définit sa valeur.

La règle 21 définit la déclaration d'une ou plusieurs variables. Il
- Le premier identificateur correspond au nom de la variable déclarée ;
- une syntaxe abrégée permet de déclarer et d'initialiser plusieurs variables de même type en utilisant des identificateurs séparés par des virgules

constante : *const IDENTIFIER* : *IDENTIFIER* := *expression* ; (20)

variable : *IDENTIFIER* [, *IDENTIFIER*]∗ : *IDENTIFIER*
[:= *expression*] ; (21)

FIG. 11 – BNF des déclarations de variables et constantes

– l'identificateur situé après le " : " définit le type de la ou des nouvelle(s) variable(s) ;
– enfin l'expression précise la valeur initiale des variables déclarées.

6.7 Déclaration des composants

Les composants du modèle sont les classes et médias qui définissent son comportement. Il existe trois types de déclaration des composants :
– la déclaration simple ;
– la déclaration d'interface ;
– la déclaration complète.

La déclaration simple permet simplement de déclarer l'existence du type. La déclaration d'interface déclare en plus les attributs et éventuellement les méthodes (dans le cas d'une classe) du type qui peuvent être utilisés avant sa déclaration complète. Enfin la déclaration complète du type spécifie la totalité des types locaux, attributs, ainsi que le corps des méthodes dans le cas d'une classe. La figure 12 donne la BNF d'une déclaration d'un type composant.

type_composant : *component_type IDENTIFIER* ; (22)
| *component_type IDENTIFIER is declarations end* ; (23)
| *component_type IDENTIFIER is declarations*
begin instructions end ; (24)

component_type : *media* (25)
| *class* (26)

FIG. 12 – BNF de la déclaration d'un type composant

La règle 22 correspond à la déclaration simple du composant, elle précise juste sa nature (média ou classe) et le nom du nouveau type de composant.

La règle 23 correspond à la déclaration de l'interface d'un composant, elle précise la nature du composant, son nom ainsi que les déclarations locales du composant. On peut y trouver :
– des déclarations de types ;
– des déclarations de constantes et variables ;
– des déclarations de triggers ;
– des déclarations de méthode (pour les classes seulement).

La règle 24 définit la déclaration complète d'un composant. La partie déclaration du composant est définie de la même manière que pour la déclaration d'une interface, mais les déclarations sont suivies du mot clé begin et d'une suite d'instruction correspondant à l'automate principal du composant.

6.8 Déclaration des méthodes des classes

La syntaxe des déclarations des méthodes des classes est faite dans la partie déclaration du composant. La figure 13 présente la BNF de la déclaration d'une méthode.

```
methode      : synchronisation procedure IDENTIFIER ( [ parametres ] )
               [ -> expression ] [ body ] ;                                (27)
             | function IDENTIFIER ( [ parametres ] )
               return IDENTIFIER [ -> expression ] [ body ] ;              (28)

parametres   : parametre [ ; parametre]*                                 (29)
synchronisation : synchronous                                            (30)
             | asynchronous                                              (31)
parametre    : IDENTIFIER : [ mode ] IDENTIFIER                          (32)

mode         : in                                                        (33)
             | inout                                                     (34)
             | out                                                       (35)
```

FIG. 13 – BNF de la déclaration d'une méthode **L*f*P**

La règle 27 définit la syntaxe d'une déclaration de procédure.
- la synchronisation de la procédure définit si l'appelant doit rester bloqué jusqu'à la fin de l'exécution de la procédure ;
- le premier identificateur donne le nom de la procédure déclarée ;
- la liste des paramètres est fournie entre parenthèses ;
- le caractère "-> est suivi de l'expression qui définit le port d'activation par défaut de la procédure ;
- la déclaration peut être suivie du corps de la procédure déclarée.

Les parenthèses sont obligatoires même si la liste de paramètre est vide.

La règle 28 définit la syntaxe d'une déclaration de fonction.
- le premier identificateur est le nom de la fonction ;
- il est suivi de la liste des paramètres entre parenthèses ;
- le mot clé return le type de donnée retourné par la fonction ;
- le symbole -> est suivi de l'expression qui définit le port d'activation par défaut de la fonction ;
- la déclaration de la fonction peut être suivie de son corps.

Les règles 30 et 31 définissent le mode de synchronisation d'une procédure.

La règles 29 définit la syntaxe des listes de paramètres, ils sont séparés par des points-virgules.

La règle 32 définit la syntaxe de la déclaration d'un paramètre d'une procédure ou d'une fonction :
- le premier identificateur est le nom du paramètre.
- le mode du paramètre détermine si celui-ci peut être lu, ou écrit ou les deux dans le corps de la fonction
- le deuxième identificateur spécifie le type de donnée du paramètre.

Les règles 33, 34, 35 définissent le mode de passage d'un paramètre :
- in le paramètre peut être lu dans le corps de la méthode ;
- inout le paramètre peut lu ou écrit dans le corps de la méthode, la nouvelle valeur est transmise à l'appelant ;
- out le paramètre peut uniquement être écrit dans le corps de la méthode, la valeur est transmise à l'appelant.

On rappelle les contraintes suivantes issues de la sémantique **L*f*P** :
- tous les paramètres d'une fonction doivent être en mode "in " ;
- une procédure dont au moins un paramètre est en mode inout ou out est obligatoirement synchrone ;
- une procédure dont tous les paramètres sont en mode in est par défaut asynchrone.

Les déclarations de procédures et de fonctions permettent également de définir leur corps. Cette solution permet de laisser plus ce choix à l'utilisateur sur la manière dont il souhaite structurer les déclarations de ses composants.

6.9 Déclaration des triggers

La syntaxe de la déclaration des triggers est donnée par la figure 14.

```
declaration_trigger : trigger IDENTIFIER ( parametres ) is
                      declarations begin instructions end ;                    (36)
                    | trigger IDENTIFIER ( parametres ) return IDENTIFIER is
                      declarations begin instructions end ;                    (37)
```

FIG. 14 – BNF de la déclaration d'un trigger

La règle 36 définit la syntaxe de la déclaration d'un trigger sans valeur de retour. Il s'agit d'une procédure privée utilisable uniquement dans le composant qui la définit. L'identificateur désigne le nom du trigger, la syntaxe des paramètres est la même que pour les déclarations de méthode ou de fonctions (cf. règles 29 et 32 de la figure 13.

La règle 37 donne la syntaxe de la déclaration d'un trigger qui retourne une valeur. Il s'agit d'une fonction privée utilisable uniquement dans le composant qui la définit :
- le premier identificateur est le nom du trigger ;
- la syntaxe des paramètres est la même que pour les déclarations de méthodes (cf. règles 29 et 32 de la figure 13) ;
- le mot clé return est suivi du nom du type de la valeur de retour.

Les appels de triggers retournant une valeur sont considérés comme des expressions, leur valeur de retour doit donc systématiquement être utilisée.

Tous les paramètres d'un trigger qui retourne une valeur doivent être en mode in

6.10 Déclaration des binders

Ces déclarations définissent la manière dont les binders correspondant aux ports des classes du modèle sont instanciés, ainsi que les valeurs de leurs attributs standards. La BNF de la déclaration des binders est donnée par la figure 15

```
binder    : binder IDENTIFIER . IDENTIFIER is ordering ( expression )
            conexion end ;                                                     (38)

ordering  : fifo                                                               (39)
          | bag                                                                (40)

conexion  : lecteurs ecrivains                                                 (41)

lecteurs  : IDENTIFIER |-> IDENTIFIER [ , IDENTIFIER ]* ;                      (42)
ecrivains : IDENTIFIER <-| IDENTIFIER [ , IDENTIFIER ]* ;                      (43)
```

FIG. 15 – BNF de la déclaration des binders

La règle 38 donne la BNF globale de la déclaration d'un binder en **L*f*P**
- le premier identficateur est le nom de la classe à laquelle le binder est rattaché ;
- Le second identificateur est le nom du port qui lui est associé ;
- ordering précise l'ordonancement des messages dans la file ;
- l'expression entre parenthèses précise la capacité du binder en nombre de messages ;

– elle est suivie de son schéma de connexion.

Les règles 39 et 40 définissent l'ordonnancement des messages dans un binder :
- fifo les messages doivent être consommés dans la file dans l'ordre de leur arrivée (la lecture sur le binder est bloquante si le premier message de la file ne peut être consommé) ;
- bag les message présents dans la file peuvent être consommés dans un ordre quelconque (la lecture sur le binder est bloquante si aucun message de la file ne peut être consommé).

La règle 41 permet de spécifier le schéma de connexion du binder, on spécifie toujours en premier les composants qui peuvent lire des messages dans le binder, puis les composants qui peuvent y écrire des messages.

La règle 42 permet de spécifier la liste des composants pouvant lire un message dans le binder :
- le premier identificateur rappelle le nom du binder (il doit être égal au deuxième identificateur de la règle 38 ;
- à droite du symbole |-> on trouve la liste des composants pouvant lire des messages dans le binder.

La règle 43 permet de spécifier la liste des composants pouvant écrire des messages dans le binder :
- le premier identificateur rappelle le nom du binder (il doit être égal au deuxième identificateur de la règle 38 ;
- à droite du symbole <-| on trouve la liste des composants pouvant écrire des messages dans le binder.

6.11 Déclaration de la topologie

La déclaration des binders et des composants qu'ils relient ne suffit pas à définir entièrement le diagramme d'architecture **L*f*P** original. Il est nécessaire de lui adjoindre la déclaration des liens. Cette déclaration spécifie les n-uplets de composants qu'un média est susceptible de relier. La BNF de la déclaration de la topologie de l'application est donnée sur la figure 16.

lien : *media IDENTIFIER component_list* ; (44)
| *media IDENTIFIER* (*component_list* [, *component_list*]+) ; (45)

component_list : (*IDENTIFIER* [, *IDENTIFIER*]+) ; (46)

FIG. 16 – BNF de la déclaration de la topologie du modèle

La règle 44 permet de spécifier que le média dont le nom est donné par le premier identificateur peut relier chacun des composants spécifiés dans la liste entre parenthèses.

La règle 45 est une extension de la précédente. Elle permet de spécifier plusieurs liens pour un seul média en une seule déclaration. Chacune des listes définit un lien possible pour le média, elles sont séparées par des virgules et la "liste de liste" est elle même entre parenthèses.

La règle 46 précise la syntaxe des listes de composants. Il s'agit d'une liste d'identificateur séparés par des virgules désignant une classe qui doit être déjà définie dans le modèle.

7 Syntaxe des instructions du langage L*f*P

Cette section présente la syntaxe des instructions du langage **L*f*P**. Les premières instructions traitées sont les instructions de structuration du langage, puis les instructions de communication.

7.1 Affectation

L'instruction d'affectation est définie par le symbole " := ". La BNF est donnée sur la figure 17.

La règle 47 définit la BNF d'une affectation :

affectation : *expression* \ := *expression* ; (47)

FIG. 17 – BNF de l'instruction d'affectation

– l'expression de gauche doit désigner une variable ou bien un paramètre en mode out ou inout défini dans l'environnement courant ;
– l'expression de droite désigne la valeur qui lui sera affectée.

Les deux expressions doivent être de types compatibles.

7.2 Les blocs d'instruction

Les blocs d'instruction permettent de regrouper des instructions dans un bloc délimité. La syntaxe de la déclaration d'un bloc nommé est présentée sur la figure 18.

bloc_instruction : *declare declarations begin instructions end* ; (48)
| : *IDENTIFIER declare declarations begin instructions end* ; (49)

FIG. 18 – Déclaration d'un bloc d'instructions

La règle 48 définit un bloc d'instructions. Il est constitué en deux parties : une partie déclarative permettant de définir des variables locales, et la partie contenant les instructions elles même.

La règle 49 définit la syntaxe d'un bloc d'instructions nommé. L'identificateur définit le nom du bloc ; ce nom peut servir de cible à une instruction de saut, le reste est identique à la règle 48.

7.3 Les boucles *"for"*

La figure 19 donne la BNF des boucles *"for"* en **L*f*P**.

boucle_for : [: *IDENTIFIER*] *for IDENTIFIER in expression .. expression*
begin instructions end ; (50)

FIG. 19 – BNF d'une boucle *"for"*

La règle 50 Définit la syntaxe d'une boucle for **L*f*P** :
– l'identificateur précédé du caractère " : " correspond au nom de la boucle, il peut être utilisé par une instruction break pour sortir explicitement d'une boucle ;
– l'identificateur suivant le mot clé for est le nom de la variable de boucle ;
– les deux expressions définissent l'intervalle de valeur parcouru par la variable de boucle ;
– le corps de la boucle est définit par les instructions situées entre les mots réservés begin et end .

les deux expressions définissant l'intervalle de valeurs de la variable de boucle doivent donc définir des valeurs d'un type énuméré ou défini par intervalle.

Le nom d'une boucle ne peut pas surcharger celui d'une variable déclarée. et ne peut pas servir de cible à une instruction de saut.

7.4 Les boucles *"while"*

La figure 20 donne la BNF des boucles *"while"* en **L*f*P**.

La règle 51 définit la BNF d'une boucle while en **L*f*P** :
– l'identificateur précédé du caractère " : " définit le nom de la boucle ;

boucle_while : [: *IDENTIFIER*] *while expression begin instructions end* ; (51)

FIG. 20 – BNF des boucles *"while"*

- l'expression définit la condition de sortie de boucle ;
- le corps de la boucle est donné entre les mots clés begin et end .

7.5 Instruction *"break"*

La syntaxe de l'expression break est donnée sur la figure 21.

instruction_break : *break* [*IDENTIFIER*] ; (52)

FIG. 21 – BNF de l'instruction break

La règle 52 définit une instruction break. Cette instruction doit être définie à l'intérieur du corps d'une boucle. Si l'identificateur optionel est omis, l'instruction provoque la sortie de la boucle courante. Si l'identificateur est précisé, il doit correspondre au nom d'une boucle, et l'instruction provoque la sortie du corps de cette boucle.

7.6 L'instruction d'alternative

La figure 22 donne la BNF de l'instruction d'alternative en **L*f*P**.

alternative : *if expression then instructions* (53)
[*elsif expressionthen instructions*]∗
[*else instructions*] *end* ;

FIG. 22 – BNF de l'instruction d'alternative

La règle 54 donne la BNF d'une instruction d'alternative **L*f*P**. On distingue trois types de branches :

1. la branche if principale ;
2. les branches elsif ;
3. la branche else .

La branche associée à l'instruction if est exécutée si l'expression est évaluée à true, sinon la première branche associée à une instruction elsif dont l'expression est évaluée à true est exécutée. Si aucune des expressions ne peut être évaluée à true, la branche else est exécutée.

7.7 Instruction de saut

La figure 23 présente la syntaxe de l'instruction de saut goto ainsi que la déclaration d'un label.

La règle 54 présente la BNF de l'instruction goto. L'identificateur désigne le label cible du saut.

La règle 55 présente la BNF de la déclaration d'un label en **L*f*P**. L'identificateur est le nom du label, il ne doit pas surcharger le nom d'une variable déclarée ou d'une boucle nommée.

instruction_goto : *goto IDENTIFIER ;* (54)

label : *: IDENTIFIER :* (55)

FIG. 23 – BNF de l'instruction goto et des labels

lecture : *discriminant (expression* [*, expression*]∗ *) <- expression*
[*with expression*] *;* (56)

discriminant : \[*expression* [*, expression*]∗ \] (57)

FIG. 24 – BNF de l'instruction de lecture de message

7.8 Lecture d'un message

Cette instruction permet de lire un message de donnée dans un binder et d'en fournir le contenu au composant qui effectue l'opération ; elle est présentée sur la figure 24. En fonction de la nature du composant, cette instruction prend des formes différentes.

La règle 56 décrit la syntaxe d'une opération de lecture :
- lorsque l'instruction est effectuée depuis un média, le discriminant peut être lu (cf. règle 57) ;
- la liste d'expression entre parenthèse définit l'ensemble des variables ou paramètres en mode out ou inout qui vont recevoir les données contenues dans le message ;
- l'expression après le symbole "<- " définit le port sur lequel le message est lu ;
- il est possible de spécifier une garde sur le contenu du message en utilisant le mot clé with suivi d'une expression booléenne.

La règle 57 définit la syntaxe d'un discriminant. Il s'agit d'une liste entre crochets d'expression séparées par des virgules.

7.9 Attente d'activation de méthode

L'instruction d'attente d'activation de méthode peut uniquement être utilisée dans une classe. Elle correspond à la lecture d'un message d'activation pour une méthode donnée sur un port déterminé. La figure 25 présente la BNF d'une instruction d'attente d'activation de méthode.

attente_activation : *IDENTIFIER <- expression ;* (58)
| *IDENTIFIER <- ( ) ;* (59)

FIG. 25 – BNF d'une instruction d'attente d'activation de méthode

La règle 59 définit la BNF d'une attente d'activation de méthode :
- l'identificateur donne le nom de la méthode activable ;
- l'expression donne le port sur lequel le message d'activation devra être lu ;

La règle 59 définit la BNF d'une attente d'activation de méthode sur le port par défaut. L'identificateur désigne le nom de la méthode activable. Le message sera lu sur le port d'activation par défaut spécifié lors de la déclaration de la méthode.

7.10 Multiplexage des opérations de réception de messages

L'instruction accept permet de multiplexer les opérations de lecture de messages ; la syntaxe de cette instruction est donnée sur la figure 26.

instruction_accept : *accept lecture* [, *lecture*] *end* ; (60)

FIG. 26 – BNF de l'instruction accept

La règle 60 présente la BNF d'une instruction accept. Chaque opération de lecture peut être soit une lecture d'un message de donnée, soit une opération d'attente d'activation de méthode. Les syntaxes utilisées pour spécifier ces opérations sont celles vues en 7.8, et 7.9 sans le " ; " final. Les opérations multiplexées sont séparées entre elles par une virgule. Dans le cas d'une instruction de lecture, les expressions doivent définir des variables, et l'instruction doit être dans média.

7.11 Envoi d'un message de données

L'instruction d'envoi d'un message de donnée est présenté sur la figure 27.

envoi_message : *discriminant* (*expression* [, *expression*]∗)
-> *expression* ; (61)

FIG. 27 – BNF de l'instruction d'envoi d'un message de donnée

La règle 61 définit l'envoi d'un message de donnée :
- le discriminant du message peut être spécifié en utilisant la syntaxe définie à la règle 57 de la figure 24 ;
- les données du message sont situées entre les parenthèses après le discriminant du message ;
- l'expression située après le symbole -> spécifie le port sur lequel le message est envoyé.

Dans le cas d'une écriture, le discriminant peut contenir toute expression, y compris une valeur immédiate. Un envoi de message ne peut contenir un discriminant que si l'instruction est spécifiée depuis une classe.

7.12 Appel d'une méthode d'un composant

On distingue en fait trois instructions d'appel de méthodes :
- l'appel de procédure asynchrone ;
- l'appel de procédure synchrone ;
- l'appel de fonction.

Les deux premiers types d'appels sont des instructions simples, mais l'appel de fonction retourne une valeur, il s'agit donc d'une instruction et d'une expression dont la valeur de retour doit être utilisée. L'instruction d'appel de fonction peut donc être insérée partout où une expression retournant une valeur immédiate est acceptée (discriminant, contenu d'un message, paramètre d'un autre appel, etc…). La figure 28 donne la syntaxe d'un appel de méthode **L*f*P**.

designation : [*discriminant*] [*IDENTIFIER* :]
IDENTIFIER ([*parametres*]) (62)
parametres : *expression* [, *expression*]∗ (63)
appel_methode : *designation* -> *expression* ; (64)
| *designation* <-> *expression* ; (65)
| *designation* <=> *expression* ; (66)

FIG. 28 – BNF d'un appel de méthode en lfp

La règle 62 définit la manière dont la méthode appelée est définie :
- le discriminant associé à l'appel de méthode est spécifié de la même manière que pour les envois de messages (cf. 7.11) ;
- le premier identificateur permet de préciser le type du composant définissant la méthode appelée ;
- l'identificateur donne le nom de la méthode ;
- les paramètres sont donnés entre parenthèses

La règle 63 donne la syntaxe de la liste des paramètres. Ils sont séparés par des virgules. Toutes les expressions sont acceptées, mais les expressions retournant des valeurs immédiates sont considérées comme invalides pour les paramètres en mode out .

Les règles 64, 65 et 66 spécifient respectivement les appels de procédures asynchrones, de procédures synchrones et de fonctions. L'expression située après les symboles -> , <-> ou <=> définissent le port dans lequel le message d'activation est écrit.