Type document : *SRS*	*Projet RNTL MORSE*	**Date :** 13/06/2004
Sous-projet n° 2		**numéro de doc :** MORSE-SRS-031015-V0.20-FGI
tâche : 2.1		**rédacteur (s)** Frédéric Gilliers

BNF de la grammaire des attributs des diagrammes L*f*P

Table des matières

Table des figures

1 Liste des modifications

date	Objet de la modification
22/08/2003	Modification de la section sur l'allocation dynamique pour prendre en compte le fait qu'on peut maintenant instancier dynamiquement les médias.
25/08/2003	Ajout des figures de l'exemple. Quelques corrections dans les commentaires qui doivent encore être améliorés.
28/08/2003	Remaniement complet pour donner un ordre logique à l'enchainement des sections.
29/08/2003	Ré-écriture de la section d'introduction pour la mettre en accord avec le remaniement, et surtout en faire une vraie intro pour la grammaire **L*f*P**
29/O8/2003	Ajouté une petite introduction aux sections sur les diagrammes d'architecture et de comportement
17/09/2003	– Correction d'erreurs dans les règles de syntax . – Prise en compte de la déclaration des ports dans les documents. – Mise à jour de l'exemple.

date	Objet de la modification
18/09/2003	Correction de la date du document, et correction d'une référence de figure erronée
29/09/2003	Corrections en fonction des remarques de Fabrice. Correction de déclarations de variables dans l'exemple des horloges.
8/10/2003	Rajout de la signification des opérateurs + et − pour les ensembles et multi-ensembles.
18/12/2003	Modification de la sous-section sur les envois de messages pour prendre en compte l'envoi de messages "simples" par les classes (i.e. : des messages qui ne sont pas des activations de méthode). Supression des crochets inutiles pour les messages qui n'ont pas de discriminant.
22/12/2003	Mise du document au style "MORSE" Passe complète sur le document : – Retiré plusieurs remarques d'ordre sémantique qui sont maintenant beaucoup mieux traitées et plus détaillées dans le document de sémantique. – rajouté des figures pour les exemples des transitions de communication. – corrigé plusieurs erreurs de présentation dans les règles de syntaxe. – mis à jour la syntaxe des envois de messages.
05/01/2004	Petites modifications de présentation pour coller plus au style des documents MORSE.
27/05/2004	Intégré les types opaques dans le document. Modifié la sémantique de l'opération de déréférenciation pour les composants. Quelques modifications de typographie et erreurs mineurs dans les figures.

2 Présentation

Ce document présente la syntaxe des attributs des diagrammes du langage **L*f*P**. 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.

Après les conventions de présentation énumérées dans la section 3, ce document énumère les règles du langage **L*f*P** proprement dit dans les sections 4 à 6. La section 4 énumère les éléments disponibles dans tous les diagrammes **L*f*P** ; la section 5 présente les attributs des composants du diagramme d'architecture ; la section 6 présente les attributs des éléments présents dans les diagrammes de comportement. En fin de document, la section 7 présente un petit exemple de modèle **L*f*P** basé sur les horloges de Matern.

3 Conventions

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 syntaxes 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 communs à tous les diagrammes

4.1 Eléments de base

Les éléments de base concernent la définition des identificateurs valides du langage, ainsi que la liste des mots réservés.

4.1.1 Casse des caractères

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

4.1.2 Identificateurs valides

Le lexème définissant un identificateur valide en **L*f*P** est donnée sur la figure 1. Un identificateur valide en **L*f*P** commence donc nécessairement par une lettre, et est suivi d'un nombre quelconque de lettre, chiffre, ou caractère souligné ("_").

$IDENTIFIER : [a-zA-Z] + [\_a-zA-Z0-9]*$

FIG. 1 – Lexème définissant un identificateur en **L*f*P**.

4.1.3 chiffres et nombres

Seuls les nombres entiers sont supportés par le langage. Ils sont représentés par une suite de chiffres sans espaces. Ils sont définis par le lexème de la figure 2.

$INTEGER : [0-9]+$

FIG. 2 – Lexème définissant un nombre

4.1.4 Composant

Il existe deux types de composant en **L*f*P** les classes et les médias. Le terme *composant* représente indifféremment une classe ou un média.

4.2 Syntaxe des expressions

Cette section présente les opérateurs disponibles pour les expressions. Schématiquement, on considère comme expression tout ce qui "retourne une valeur", ce qui inclut les expressions arithmétiques, les variables etc...
Pour écrire les expressions, on dispose des opérateurs suivants :

1. `< > <= >= =` représentent respectivement pour les éléments d'un interval ordonné :
   – inférieur strict ;
   – supérieur strict ;
   – inférieur ou égal ;
   – supérieur ou égal ;
   – égalité.
   Les opérateurs `<` , `<=` et `=` sont également valables pour des ensembles et représentent respectivement l'inclusion stricte, l'inclusion et l'égalité entre ensembles.
   Tous ces opérateurs retournent des valeurs booléennes.
2. `+ - or` respectivement pour l'addition, la soustraction et le "ou" booléen, lorsqu'ils sont appliqués sur des ensembles ou des multi-ensembles, les opérateurs `+` et `-` représentent respectivement l'union et l'intersection ensembliste ;
3. `* / and` respectivement pour la multiplication, la division et le "et" booléen ;

4. not - # représentent respectivement le "non" booléen, le moins unaire applicable sur les entiers et l'opérateur permettant de sélectionner n'importe quel élément d'un ensemble ou d'un multi-ensemble. Ces trois opérateurs sont unaires.

Les opérateurs ci-dessus sont associatifs à gauche, et donnés par ordre de priorité croissante en fonction de la ligne (tous les opérateurs sur une ligne ont la même priorité).

Les règles de syntaxe des expressions sont données sur la figure 3 et leur signification est explicité ci-dessous (la règle operator désigne l'un des opérateurs binaire évoqué ci-dessus).

expression	:	*expression operator expression*	(1)
	|	**(** *expression* **)**	(2)
	|	**|** *expression* **[** " *expression* **]** **|**	(3)
	|	*expr_variable*	(4)
	|	*INTEGER*	(5)
	|	*op_unaire expression*	(6)
	|	**(** *IDENTIFIER* => *expression* **[** , *IDENTIFIER* => *expression* **]** **)**	(7)
	|	**(** *others* => *expression* **)**	(8)
	|	**{** **[** *expression* **[** , *expression* **]** ∗ **]** **}**	(9)
	|	*IDENTIFIER* ' *IDENTIFIER* **[** **(** *expressions* **)** **]**	(10)
	|	*expr_variable* @ *IDENTIFIER* **[** **(** *expression* **[** , *expression* **]** ∗ **)** **]**	(11)
expr_variable	:	*expr_variable* . *IDENTIFIER*	(12)
	|	*expr_variable* **(** *expression* **[** , *expression* **]** **)**	(13)
	|	*IDENTIFIER*	(14)

FIG. 3 – Syntaxe d'une expression valide en **L*f*P**.

La règle 1 représente une expression obtenue en utilisant les opérateurs binaires définis précédemment.

La règle 2 définit une expression entre parenthèses.

La règle 3 [1] permet d'obtenir le cardinal d'un ensemble ou le cardinal d'un multi-ensemble pour un élément donné :
- l'expression à gauche du " ' " correspond à la désignation de l'ensemble ou du multi-ensemble dont on souhaite obtenir le cardinal.
- l'expression à droite du " ' " doit désigner une valeur contenue par un multi-ensemble afin de déterminer combien d'éléments semblables le multi-ensemble contient.

La règle 4 désigne l'utilisation d'une variable.

La règle 5 correspond à une valeur entière.

La règle 6 correspond à une expression préfixée d'un opérateur unaire : not ou - (moins unaire).

La règle 8 définit un agrégat permettant de définir un type record en spécifiant une partie ou la totalité de ses champs. L'ordre de définition des champs est libre.

La règle 8 permet de définir la valeur d'un tableau. Pour le moment, on est obligé d'initialiser toutes les cases du tableau à la même valeur. Cela permet de faciliter l'analyse des expressions (qui devient extrêmement difficile dans le cas de tableaux à plusieurs dimensions.

La règle 9 permet de définir une valeur immédiate de type ensemble en énumérant les valeurs contenues dans le nouvel ensemble. Cette règle ne peut être utilisée que pour initialiser une variable de type ensemble.

[1] Comme beaucoup d'opérateurs sur les ensembles, cette règle n'est pas implémentée par l'analyseur sémantique. L'arbre produit ne pourra donc pas être vérifié.

La règle 10 permet de représenter les accès aux attributs d'un type. Le premier IDENTIFIER est le nom du type dont on utilise un attribut ; le deuxième IDENTIFIER est le nom de l'attribute ; si nécessaire, une expression entre parenthèse spécifie l'expression sur laquelle l'attribut est appliqué.

La règle 11 permet de représenter un appel à une fonction d'un type externe. Seuls les appels à des fonctions sont considérés comme des expressions (ils retournent une valeur). La partie à gauche du @ désigne la variable opaque sur laquelle la fonction est appelée. La partie à droite désigne la fonction appelée (IDENTIFIER) et ses éventuels paramètres (liste d'expressions séparées par des virgules).

La règle 12 correspond à la déréférenciation. Elle peut être utilisée dans deux contexte :

1. l'accès au champs d'un type record ;
2. l'accès aux binders d'une classe.

La règle 13 permet d'accéder aux éléments d'un tableau. Elle est constituée de la désignation du tableau, suivie de la liste d'expression qui donnent les index de la case à accéder.

La règle 14 désigne le nom d'une variable.

4.3 Exemples d'expressions

```
type rec is record champ1 : integer ; end ;

type tableau is array (1..54) of rec ;

tab : tableau ;

5 + tab(5).champ1 -- expression valide.
```

4.4 Syntaxe des déclarations de types

4.4.1 types instanciés

On peut déclarer des types de données dans la partie déclaration de n'importe quel diagramme **L*f*P**. On peut définir sept sortes de types :

- les types *range* définis par restriction de type (règle 1 de la figure 4) ;
- les types range définis par énumération (règle 2 de la figure 4) ;
- les types tableau (règle 3 de la figure 4) ;
- les types ensemble (règle 4 de la figure 4) ;
- les types multi-ensembles (règle 5 de la figure 4) ;
- les types "record" (règle 6 de la figure 4) ;
- les types "port" (règle 7 de la figure 4) ;

La règle 1 permet de définir un type énuméré par restriction d'un autre type énuméré. Le nouveau type considéré est en fait un "sous-type" du type initial.

- le premier IDENTIFIER est le nom du nouveau type ;
- si nécessaire, le mot clé circular précise que le nouveau type doit être circulaire ;
- la règle range_rule permet de définir les bornes de l'intervalle définissant le type (cf. description des règles 8 et 9) ;
- le dernier IDENTIFIER permet de spécifier le nom du type initial.

La règle 2 permet de définir un type énuméré.

- le premier IDENTIFIER permet de définir le nom du type ;
- si nécessaire, le mot clé circular permet de rendre le type circulaire ;
- enfin la déclaration de type se termine par la liste de ses valeurs, c'est à dire une liste d'identificateurs séparés par des virgules.

La règle 3 permet de définir des types tableau :

- le premier IDENTIFIER correspond au nom du nouveau type ;
- celui-ci est suivi d'un ou plusieurs intervalles séparés par des virgules qui définissent les dimensions du tableau ;
- enfin, le dernier IDENTIFIER est le type des éléments contenus dans le tableau.

```
type_rule      :  type IDENTIFIER is [ circular ] range range_rule of IDENTIFIER ;      (1)
               |  type IDENTIFIER is [ circular ] range ( IDENTIFIER
                  [ , IDENTIFIER ]* );                                                  (2)
               |  type IDENTIFIER is array ( range_rule
                  [ , range_rule ]* ) of IDENTIFIER ;                                   (3)
               |  type IDENTIFIER is set of IDENTIFIER ;                               (4)
               |  type IDENTIFIER is bag of IDENTIFIER ;                               (5)
               |  type IDENTIFIER is record
                  [ IDENTIFIER : IDENTIFIER [ := expression ] ; ]+
                  end ;                                                                 (6)
               |  type IDENTIFIER is port [ ( IDENTIFIER [ , IDENTIFIER ]* ) ] ;       (7)
               |  type IDENTIFIER is opaque [ exten_method ]* end                      (8)
extern_method  :  function IDENTIFIER [ parameters ] return IDENTIFIER                 (9)
               |  procedure IDENTIFIER [ parameters ]                                  (10)

range_rule     :  expr_variable .. expr_variable                                       (11)
               |  IDENTIFIER                                                           (12)
```

FIG. 4 – Règles de syntaxe pour les déclarations de types instanciés.

Les règles 4 et 5 permettent de définir respectivement des types ensembles et des multi-ensembles. Le premier IDENTIFIER correspond au nom du nouveau type ; le second correspond au type des éléments de l'ensemble ou du multi-ensemble.

La règle 6 permet de définir des types articles (ou structures) :
- le premier IDENTIFIER correspond au nom du nouveau type ;
- vient ensuite la liste des champs :
  - l'IDENTIFIER avant le " : " correspond au nom du champ ;
  - l'IDENTIFIER après le " : " correspond au type du champ ;
  - ces deux identificateurs sont éventuellement suivis de " := " et d'une expression donnant la valeur initiale du champ.

  Chaque déclaration de champ du type est suivie d'un " ; ".

La règle 7 permet de définir les types des ports des composants :
- le premier IDENTIFIER est le nom du nouveau type ;
- il peut être éventuellement suivi d'une liste de noms de types qui définissent la liste des types du discriminant.

La règle 8 définit un type opaque. Les types opaques permettent de représenter les composants métier, extérieurs à la spécifications **L***f***P**. L'interface associée à ce type est définie par la règle extern_method .

Les règles 9 et 10 définissent la syntaxe des déclarations de l'interface d'un type opaque. Elle est composée de méthodes (procédures ou fonctions). L'IDENTIFIER définit le nom de la méthode. La règle parameters définit les paramètres des méthodes. Elle est identique à celle définie sur la figure 10 page 11.

Les règles 11 et 12 permettent de définir un interval. Il existe deux manières de définir un intervalle :
- en définissant ses bornes explicitement à l'aide d'une expression (règle 8) ;
- en considérant l'intégralité des valeurs d'un type énuméré donné par son nom (règle 9).

4.4.2 Exemples de déclarations de types

Les déclarations de type suivantes sont valides en **L***f***P** :

```
-- une série de déclaration de type : type caractere is range 0..255
of integer ; type type_2 is circular range (valeur1, valeur2, valeur3)
; type tableau is array (1..240, type2) of caractere ;

type ensemble_1 is set of integer ; type m_ensemble is bag of tableau
; type essai is record champ1 : tableau ; champ2 : integer := 8 ;
champ4 : integer ; end ;

-- exemple de type port type my_port is port (integer, essai) ; --
définition des ports toto, tutu : my_port ;
```

Ces définitions correspondent respectivement à :

1. la définition d'un type "intervalle" par restriction d'un type existant ;
2. la définition d'un type "intervalle" par énumération des valeurs ;
3. la définition d'un type tableau (ici à deux dimensions), on notera deux manières de définir l'intervalle correspondant à une dimension du tableau :
   – par définition des bornes de l'intervalle (première dimension du tableau) ;
   – en utilisant toute la plage permise par un type donné (deuxième dimension du tableau).
   Il n'est pas possible d'utiliser des tableaux de taille dynamique en **L*f*P**.
4. la définition d'un type ensemble d'entiers ;
5. la définition d'un type multi-ensemble contenant des tableaux ;
6. la définition d'un type record ayant trois champs respectivement de type tableau, integer (innitialisé à la valeur 8) et integer non initialisé.
7. la déclaration d'un type de port dont le discriminant doit contenir deux valeurs : une valeur de type `integer`, et un valleur de type `essais`.
8. la déclaration de deux ports du type précédemment défini. Les messages des discriminants transitant par ces ports depuis les classes vers les médias devront donc contenir deux valeurs : la première de type entier, et la seconde de type "essai".

4.5 déclaration de variables et de constantes

Les variables d'un type quelconque peuvent être définies dans l'attribut global de tous les diagrammes **L*f*P**. La syntaxe permettant de définir une variable est donnée par la figure 5.

variable : *IDENTIFIER* [, *IDENTIFIER*]* : *IDENTIFIER* [:= *expression*] ; (1)

constant : *const IDENTIFIER* [, *IDENTIFIER*]* : *IDENTIFIER* := *expression* ; (2)

FIG. 5 – déclaration de variables et de constantes

Les règles de la figure 5 permettent de définir respectivement des variables et des constantes du modèle. La définition des variables commence par la liste des noms des nouvelles variables, suivie de leur type et éventuellement d'une valeur initiale donnée sous la forme d'une expression.

Dans le cas des constantes, la règle est préfixée par le mot clé const . De plus, une constante doit toujours être initialisée.

4.5.1 Variables désignant des composants L*f*P

Toute variable dont le type correspond à un composant **L*f*P** (classe ou média) est implicitement une référence vers un objet de ce type (ce mécanisme est le même que celui de java). Si elle n'est pas initialisée, elle est considérée comme invalide (valeur "null" par défaut). Ainsi créér une variable d'un type composant ne crée pas une instance de ce composant ; seul l'opérateur d'instanciation présenté à la section 6.7 permet de créer une nouvelle instance de composant.

4.5.2 Exemples de déclarations de variables et constantes

Les définitions suivantes sont valides en LfP :

```
-- définition et initialisation d'une variable entière.
entier : integer := 45 ;
-- définition et initialisation d'une variable ensemble.
e : ensemble :=  {1,2, 567, 7, -8 }
-- variable tableau non initialisée.
t : tableau ;
-- définition d'un type tableau, et initialisation d'une variable
-- de ce type
type int_t is array(1..240) of caractere ;
 -- voir la section consacrée aux expressions pour la définition
 -- de la syntaxe de la valeur initiale
te : int_t := (others => 0) ;
-- une constante de type "integer"
const b : integer := 4 ;
```

5 Le diagramme d'architecture

Ce diagramme permet de spécifier les composants initiaux du modèle **L*f*P** et sa topologie. L'attribut de déclaration du diagramme d'architecture permet de déclarer :

- les constantes globales du modèle ;
- les instances des composants initialement présents dans le modèle (instances statiques) ;
- les dépendances pour l'instanciation automatique des médias ;
- les types glaubaux du modèle.

La syntaxe des déclarations de constantes est donnée dans la section 4. Dans cette section, nous étudierons successivement les syntaxes pemettant de définir :

- les instances statiques des composants du modèle (spécifiées dans l'attribut global du modèle) ;
- les attributs des binders ;
- les dépendances des médias.

5.1 Déclaration des instances statiques

Les règles de syntaxes présentées par la figure 6 concernent les déclarations des instances statiques des composants du modèle. Elles ne sont acceptées que dans le diagramme d'architecture.

static_instanciation : *IDENTIFIER* **[** , *IDENTIFIER* **]** : *IDENTIFIER*
with agregat
agregat : **(** *IDENTIFIER* => *expression* **[** , *IDENTIFIER* => *expression* **]** ∗ **)**

FIG. 6 – Déclaration des composants du modèle (réservées au diagramme d'architecture)

La règle static_instanciation permet de définir une instance statique d'un composant.

- elle commence par le nom de la variable à définir ou la liste des noms des variables à définir séparés par des virgules.
- l'IDENTIFIER après le " : " correspond au nom du type des nouvelles variables (un nom du composant).
- enfin, on trouve si nécessaire, la liste des variables de la classe à initialiser sous forme d'un agrégat.

la règle agregat correspond à l'association entre les variables de la classes et leur valeur initiale.

- IDENTIFIER le nom de la variable à initialiser.
- expression l'expression qui permet de calculer la valeur initiale de la variable.

5.2 Définition des caractéristiques des binders

Les caractéristiques des binders dont définies à l'aide des quatre attributs suivants :

- multiplicity définit la multiplicité du binder, il peut prendre deux valeurs :
  - 1 : un binder doit être instancié à chaque fois qu'une instance de la classe est créé, et cette instance de binder est uniquement liée à l'instance de classe qui a provoqué l'instanciation ;
  - all : Une seule instance de binder est créée lors de l'initialisation du système, et les nouvelles instances de classes y sont reliées lors de leur création.
- binding : cet attribut relie les instances de binders aux ports des classes. Sa syntaxe est donnée par la figure 7, le premier identificateur désigne le nom de la classe à laquelle est reliée le binder, le deuxième désigne le nom du port auquel il correspond.
- capacity : Définit la capacité du buffer (nombre de messages qu'il peut contenir). Cet attribut doit obligatoirement contenir un entier. Actuellement le parseur ne supporte pas l'utilisation d'une expression (même constante) pour la définition de cet attribut.
- ordering : Cet attribut définit la politique d'ordonnancement du binder. Il doit contenir un des mots clés suivants :
  - fifo qui définit une politique de fifo pour le binder ;
  - bag qui indique que les messages sont traités sans ordre particulier.

binding : *IDENTIFIER* . *IDENTIFIER*

FIG. 7 – association des binders

La figure 8 représente la déclaration d'un binder entre une classe `C1` et un média `M1`. Ce binder est de type all ; son instance est donc partagée entre toutes les instances de la classe `C1`. Il dispose d'une capacité de 6 messages, et utilise une politique fifo.

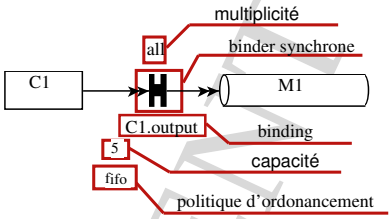


FIG. 8 – Un exemple de déclaration de binder

La signification du binding est la suivante :

1. A l'intérieur de la classe, ce binder est désigné par le nom `output` ;
2. A l'intérieur du média, le même binder sera désigné sous le nom `input`

6 Le diagramme de comportement

Ce diagramme permet de spécifier le comportement des composants **L*f*P**. On doit cependant distinguer l'attribut global d'un diagramme de comportement d'une classe de ceux des sous-diagrammes de cette classe.

L'attribut global d'un diagramme de comportement d'un composant doit contenir les déclarations de types et de variables (ou constantes) locales du composant, ainsi que les déclarations de ses triggers et méthodes. A contrario, les déclarations des sous-diagrammes d'un composant ne peuvent contenir que des déclarations de variables (ou constantes) et de type.

Cette section présentera successivement les syntaxes permettant :

- de déclarer les méthodes et triggers d'un composant ;
- de définir les attributs globaux des triggers, méthodes et sous-diagrammes d'un composant ;

– d'envoyer et de recevoir des messages ;
– de définir les instructions portées par une transition ;
– d'instancier dynamiquement des composants du modèle.

6.1 Déclaration des triggers

Un trigger est un sous-diagramme nommé utilisable dans tout le diagramme de comportement d'un composant. La déclaration doit être faite dans l'attribut `déclaration` du diagramme principal du composant, sa syntaxe est donnée sur la figurereffig :triggerDef

trigger : *trigger IDENTIFIER* ;

FIG. 9 – Déclaration des triggers

6.2 Déclaration des méthodes d'une classe

Les règles de la figure 10 permettent de définir les méthodes exportées par une classe. Ces règles ne peuvent être utilisées que dans l'attribut global du diagramme principal de la classe.

method	:	*function IDENTIFIER* **[** *parameters* **]** *return IDENTIFIER* ;	(1)
	|	**[** *synchrony* **]** *procedure IDENTIFIER* **[** *parameters* **]** ;	(2)
parameters	:	(*IDENTIFIER* \ **:** **[** *param_mode* **]** *IDENTIFIER*	
		[, *IDENTIFIER* \ **:** **[** *param_mode* **]** *IDENTIFIER* **]** ∗)	(3)
synchrony	:	*synchronous*	(4)
	|	*asynchronous*	(5)

FIG. 10 – Déclarations des méthodes d'une classe

La règle 1 correspond à la déclaration d'une méthode avec une valeur de retour (fonction). La règle 2 correspond à la déclaration d'une méthode sans retour de valeur (procédure). La règle 3 correspond à la déclaration d'un trigger.

une fonction définit une communication synchrone et est déclarée de la manière suivante (règle(1)) :
– le mot clé function ;
– le nom de la fonction ;
– éventuellement une liste de paramètres qui doivent tous être en mode in ;
– le mot clé return ;
– le type de retour de la fonction.

une procédure peut définir une communication synchrone ou asynchrone et est déclarée de la manière suivante (règle (2)) :
– Le mode de synchronisation de l'appel : synchronous (procédure synchrone) ou asynchronous (procédure asynchrone). Une procédure dont au moins un paramètre est en mode out ou inout doit être synchrone. Une déclaration de procédure asynchrone avec des paramètres en mode out ou inout est incorrecte. Dans le cas où le mode de synchronisation de la méthode n'est pas spécifiée, il est déduit de la déclaration des paramètres de la procédure :
  – S'il existe au moins un paramètre en mode out ou inout , la procédure est obligatoirement synchrone.
  – Si tous les paramètres de la procédure sont en mode in et que la synchronisation n'est pas définie explicitement, la procédure est déclarée asynchrone.
– Le nom de la procédure.
– Eventuellement une liste de paramètre.

les paramètres d'une méthode sont définis de la manière suivante (règle 3) :

- le nom du paramètre (IDENTIFIER) ;
- éventuellement le mode du paramètre :
  - in : le paramètre est passé par valeur, il n'est pas mis à jour au retour de l'appel
  - out : la valeur passée en paramètre est ignorée, la valeur du paramètre est mise à jour lors du retour de la fonction, le paramètre effectif ne doit pas être une constante ou une expression statique[2] ;
  - inout : la valeur du paramètre est transmise à la fonction appelante, et est susceptible d'être modifiée par celle-ci, le paramètre effectif ne doit donc pas être une constante ou une expression statique[3].
- Le nom du type du paramètre (un IDENTIFIER).

Il est important de noter que les modes out et inout des paramètres sont traités par "copy / restore". Il ne s'agit pas d'un passage par référence (inapplicable dans le cas d'une application distribuée). Ces deux comportements produisent des résultats très différents lors d'un traitement d'exception, ou le traitement d'une interruption (time-out sur l'attente d'une réponse).

Toute variable désignant un composant étant forcément une référence vers l'instance de ce composant, il est donc impossible de passer un composant par valeur en paramètre d'une méthode. seule sa référence peut être passée en paramètre. Dans le cas où cette référence est passée en mode out ou inout , elle est traitée par "copy / restore" comme tous les paramètres.

6.3 Syntaxe des attributs globaux des sous-diagrammes

Cette section va présenter la syntaxe des `declaration` des sous diagrammes lfp. On distingue trois cas :

- le sous diagramme est une transition hiérarchique ;
- le sous diagramme est le corps d'une méthode de la classe ;
- le sous diagramme est le corps d'un trigger de la class ;

Dans tous les cas la syntaxe des éléments déclarés est celle déjà évoquée précédemment. En revanche, l'entête et la fin des déclaration n'est pas identique.

Dans le reste de cette section consacrée aux sous-diagrammes, on utilisera la règle declarations pour désigner une suite de déclaration de types, variables ou constantes telle que présentée sur la figure 11.

declarations : **[** *declaration* **]**+
declaration : *type_rule*
| *variable*
| *constant*

FIG. 11 – syntaxe d'une suite de déclaration

6.3.1 Sous diagrammes hiérarchiques

Dans ce cas il n'y a pas d'entête de déclaration, on se contente de déclarer les éléments (types, variables, etc...) comme par exemple pour le diagramme d'architecture. La syntaxe de cet attribut est donc directement donné par la règle declarations de la figure 11.

6.3.2 Déclaration du corps d'un trigger

Les déclarations du corps d'un trigger doivent être précédés du rappel de la déclaration du trigger et terminées par le mot clé end ; la BNF est donnée par la figure 12. l'IDENTIFIER est le nom du trigger dont le corps est donné par le sous diagramme.

[2] Le parseur ne vérifie pas cette contrainte
[3] Le parseur ne vérifie pas cette contrainte

trigger_body : *trigger IDENTIFIER is* **[** *declarations* **]** ∗ *end* ;

FIG. 12 – déclaration du corps d'un trigger

6.3.3 Déclaration du corps d'une méthode

De même que pour les triggers, une méthode doit rappeler sa déclaration dans son attribut "déclarations". Cette syntaxe est donnée sur la figure 13 :

- IDENTIFIER désigne le nom de la méthode
- parameters rappelle la liste des paramètres formels de la méthode, tels qu'ils ont été définis lors de sa déclaration.

method_body : *procedure IDENTIFIER* **[** *parameters***]** *is declaration end* ;
| *function IDENTIFIER* **[** *parameters***]** *return IDENTIFIER is declarations end* ;

FIG. 13 – Déclaration du corps d'une méthode

6.4 Attributs des transitions simples

Une transition simple est définie par ses instructions et sa garde. La syntaxe des instructions attachées à une transition est définie sur la figure 14. La syntaxe des gardes des transitions est définie en 6.6 après avoir étudié l'ensemble des transitions du langage **L*f*P**

instructions : **[** *instruction* **]** ∗ (1)

instruction : *if expression then instructions* **[** *else instructions* **]** *end* ; (2)
| *for IDENTIFIER in range_rule loop instructions end* ; (3)
| *while expression loop instructions end* ; (4)
| *variable* := *expression* ; (5)
| *expr_var* @ *IDENTIFIER* **[** (*expression* **[** , *expression*) **]** ∗ **]** (6)

FIG. 14 – Instructions disponibles sur les transitions **L*f*P**.

La règle (1) définit les suites d'instructions. Il n'est pas obligatoire qu'une transition porte des instructions.

La règle (2) correspond à une alternative. L'expression suivant le mot clé if doit avoir une valeur booléenne. La sémantique correspond à celle rencontrée dans les langages de programmation classiques. Pour le moment, aucune forme en "elsif" à la Ada n'est implémentée, mais le mot "elsif" est réservé au cas où l'on souhaiterai incorporer cette fonctionnalité.

La règle (3) correspond à une boucle `"for"` à la Ada, mais en plus limitée : on doit systèmatiquement définir les bornes de l'intervale parcouru par la variable de boucle (l'IDENTIFIER de la règle). La signification de range_rule est la même que pour les définitions de type range donnée sur la figure 4 page 7.

La règle (4) correspond à une boucle while qui correspond au comportement classique d'un langage de programmation : tant que l'expression booléenne est vraie, on exécute le bloc d'instruction défini entre loop et end .

La règle (5) correspond à l'affectation d'une valeur à une variable. Les types de la variable et de l'expression doivent être compatibles. La règle variable correspond à celle définie pour les expressions sur

la figure 3. Il est possible d'affecter directement des valeurs immédiates "complexes" comme des ensembles ou des types record. En revanche, on ne peut pas appliquer d'opérateurs sur ces valeurs immédiates.

La règle (6) correspond à l'appel d'une procédure externe définie par l'interface d'un type opaque. La règle expr_variable done la variable contenant le composant externe sur lequel l'appel est effectué ; l'IDENTIFIER est le nom de la procédure appelée ; la liste d'expression entre parenthèse corresopnd aux paramètres effectifs de l'appel.

6.4.1 Exemple de code possible pour une transition

```
-- On suppose qu'on dispose des déclarations suivantes
-- dans le contexte courant :
-- sum : integer := 0 ;
-- type t_grades  is array (1..nbr_of_grades) of integer ;
--  et que le tableau a été précédemment initialisé.

--  on peut alors écrire :
sum := 0 ;
for i in 1..nbr_of_grades
loop
        sum := sum + grades(i) ;
end ;
sum := sum / nbr_of_grades ;
```

6.5 Opérations autorisées sur les transitions de communication

Les opérations permises sur les binders varient en fonction du contexte, mais dans tous les cas, le binder sur lequel l'opération doit être effectuée est désigné par le port du composant réalisant l'opération.

Nous séparons donc les opérations sur les binders depuis les classes (appels de méthode et envois de messages) et les opérations réalisées depuis les médias (traitement des messages enovoyés par les classes). Ces opérations sont situées dans l'attribut `message` des transitions dédiées à la communication. On dispose de trois types de transitions de communication présentées sur la fiure 15.

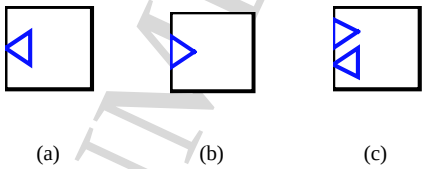


FIG. 15 – Les trois types de transitions d'envoi / réception de méthode

La transition 15(a) est une transition d'envoi de messages : elle ne peut être utilisée que pour les opérations ne nécessitant aucune lecture d'information sur le binder.

La transition 15(b) permet de lire des informations dans les binders spécifiés. La transition de la figure 15(c) permet de lire et d'écrire des informations dans le binder, elle ne peut être utilisées que dans les diagrammes de comportement des classes.

6.5.1 Opérations sur les binders depuis les classes

La syntaxe des opérations sur les binders au niveau des classes est définie sur la figure 16. Pour toutes les règles de cette figure, on convient d'appeler *discriminant* la partie entre crochets. Cette partie contient des informations destinées au média pour le routage des messages.

```
message_operations : [ message_operation ]+                                    (1)
message_operation  : method_call                                               (2)
                   | return_statement                                          (3)
                   | message                                                   (4)
                   | reception                                                 (5)
                   | controll_msg                                              (6)

method_call        : & IDENTIFIER [ discriminant ] [ \ : IDENTIFIER ] \ :      (7)
                     IDENTIFIER [ ( expressions ) ; ]
                   | expr_variable := & IDENTIFIER [ discriminant ] [ \ : IDENTIFIER ] \(8)
                     IDENTIFIER [ ( expressions ) ; ]
return_statement   : & IDENTIFIER [ discriminant ] return ( expressions ) ;    (9)
activation         : & IDENTIFIER ;                                            (10)
message            : & IDENTIFIER [ discriminant ] : ( expressions )            (11)
                   | & IDENTIFIER ( expr_variable [ , expr_variable ] ) ;      (12)
                     [ , expr_variable ]* ) ;
controll_msg       : & IDENTIFIER discriminant ;                               (13)

discriminant       : \[ expressions \]                                         (14)
expressions        : expression [ , expression ]*                              (15)
```

FIG. 16 – Opérations réalisables sur les ports d'une classes **L*f*P**

Règle 1 : cette règle définit l'attribut message d'une transition de communication. Cet attribut est constitué d'une suite d'opération à réaliser sur les ports de la classe. La règle de la figure 16 ne prend pas en compte les types de transition sur lesquelles chaque opération est aurorisées. Ces limitations sont précisées dans le commentaire qui leur est associé.

Règles 2, 3, 4, 5, 6 : Ces règles définissent l'ensemble des opérations qu'il est possible d'associer à une transition de communication :

- appel de méthodes ;
- envoi de la valeur de retour d'une méthode (uniquement valable dans le corps d'une fonction) ;
- envoi ou réception de messages ;
- activation d'une méthode ;
- envoi de "messages de contrôles" au média.

Une transition donnée ne peut porter qu'un seul type d'interaction, on ne peut donc mélanger sur une transition d'envoi des appels de procédure et des envois de messages.

Règle 7 : Cette règle donne la syntaxe d'un appel de procédure :

- Le premier identificateur est le port (référence du binder dans la classe) par lequel est envoyé l'activation.
- Le discriminant contient les paramètres de routage de l'appel vers le composant cible.
- L'identificateur optionnel suivant le discriminant permet de spécifier le nom de la classe du composant cible.
- Le dernier IDENTIFIER désigne le nom de la procédure appelée. Il peut être suivi d'une liste d'expressions entre parenthèses qui sont les paramètres de l'appel.

Règle 8 : cette règle donne la syntaxe des appel de fonctions ; elle est presque identique à l'appel de procédure, mais elle oblige à stocker la valeur de retour de la fonction appelée dans une variable (la règle expr_variable est définie sur la figure 3).

Règle 9 : cette règle permet de représenter l'envoi de la valeur de retour d'une fonction. Le premier IDENTIFIER désigne le port vers lequel le message de retour est envoyé ; l'expression entre parenthèses après le mot clé return est la valeur retournée par la fonction.

Règle 10 : cette règle permet de spécifier le port d'activation d'une méthode. L'identificateur est le port sur lequel la méthode est en attente d'activation il s'agit de la référence du binder devant contenir le message d'activation. Une transition d'activation doit toujours être la première transition qui suit l'état initial d'une méthode.

Règle 11 : cette règle représente les envois de messages depuis une classe. Un envoi de message n'est pas une activation de méthode, il s'agit juste d'un tranfert de données typées. Cette règle ne peut être utilisée que sur une transition d'envoi (fig 15(a)). La structure de cette règle est la suivante :
- Le premier IDENTIFIER désigne le port dans lequel le message est écrit ou lu ;
- le discriminant définit les paramètres de routage du message ;
- La liste d'expressions entre parenthèses désigne le contenu du message envoyé.

Règle 12 : cette règle représente une lecture de message sur un port. L'identificateur désigne le port sur lequel le message est attendu, il est suivi de la liste des variables dans lesquelles seront stockées le contenu du message.

Règle 13 : cette règle définit la syntaxe des envois de messages dits "de contrôle". Il s'agit de messages envoyés par la classe à destination du média. Le premier identificateur contient le port dans lequel le message est déposé ; et le discriminant contient le message destiné au média.

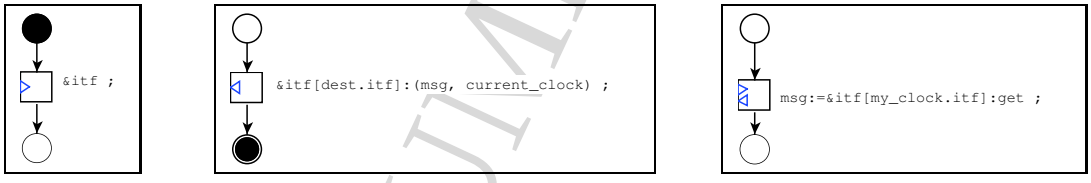
Règle 14 : la règle discriminant permet de définir les paramètres de routage des messages. Ils sont exprimés par une suite d'expressions séparées par des virgules et entre crochets[4].

Règle 15 : la règle expressions désigne une suite d'expressions arithmétique séparées par des virgules.La règle expression est définie sur la figure 3 (page 5).

6.5.2 Exemples d'opérations réalisables depuis une classe

La figure 17 donne trois exemples d'utilisation de transitions de communications dans une classe. La figure 17(a) présente un exemple de transition d'activation de méthode. Cette méthode peut donc être activée par via le binder désigné par ce port. La figure 17(b) représente un envoi de message vers une autre classe. Le message est constitué des deux variables entre parenthèses, et le discriminant du message (qui ne sera transmi qu'au média) est constitué d'une seule variable entre crochet. Enfin, la figure 17(c) représente un l'appel de la fonction `get` via le port `itf` de la classe courante. Le discriminant est constitué du port `itf` du composant `my_clock` ; la valeur de retour est stockée dans la variable `msg`

–



(a) (b) (c)

FIG. 17 – Exemples d'opérations de communication depuis les classes **L*f*P**

6.5.3 Syntaxe pour les opérations sur les binders depuis les médias

La syntaxe pour les réceptions de messages est présentée sur la figure 18. Ces deux règles peuvent être utilisées en réception et en émission, c'est à dire respectivement sur les transitions des figures 15(b) et 15(a).

Règle 1 : cette règle précise la syntaxe des réceptions de messages de contrôle dans les médias. Cette règle ne peut être utilisée que sur une transition de réception.

[4] Attention : les crochets préfixés par \ font partie de la syntaxe du langage.

message_operation : & *IDENTIFIER discriminant* ; (1)
| & *IDENTIFIER* **[** *discriminant* **]** \ **:** *expr_variable* (2)
| & *IDENTIFIER* \ **:** *expr_variable* (3)

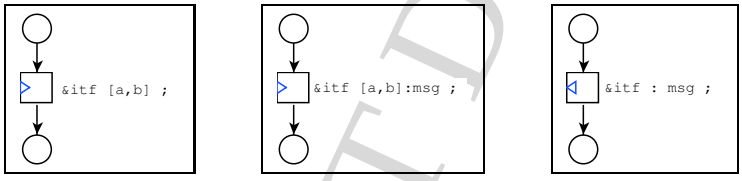
FIG. 18 – Opérations réalisables sur les ports d'un média **L*f*P**

Règle 2 : cette règle précise la syntaxe des réceptions de messages. Cette règle est constituée du port de réception (référence du binder dans lequel le message doit être lu), du discriminant du message (se reporter à la figure 16 pour la syntaxe des discriminants), et enfin de la désignation de la variable qui contiendra le message à transmettre par le média.

Règle 3 : cette règle donne la syntaxe des envois de messages depuis les médias, elle est constituée du port dans lequel le message est envoyé, et de la variable qui contient le message à transmettre.

Les variables servant à stocker les messages à transmettre par le média doivent être de type message ; il s'agit d'un type prédéfini **L*f*P** visible uniquement dans les médias[5]. Il permet de représenter de manière "opaque" le message à transmettre à la classe destinataire. Un message peut être soit un appel de méthode, soit le transport de la valeur de retour.

6.5.4 Exemples d'opérations réalisées depuis un média



(a) (b) (c)

FIG. 19 – Exemples d'opérations de communication depuis les médias **L*f*P**

La figure 19 donne trois exemples d'opérations de communication depuis les médias. La figure 19(a) illustre la réception d'un message de contrôle, le conenu du message est envoyé dans les variables a et b. La figure 19(b) illustre une réception de message avec discriminant. La figure 19(c) illustre l'envoi d'un message depuis un média.

6.6 Garde des transitions

Toutes les transitions peuvent avoir une garde (y compris les transitions d'envoi / réception de messages. La syntaxe des gardes est donnée sur la figure 20. La garde d'une transition est une expression entre crochets qui qui doit avoir une valeur booléenne. La garde peut porter sur toute variable définie dans le contexte courant.

guarde : \[*expression* \] (1)

FIG. 20 – Syntaxe des gardes des transitions

Une transition ne peut être franchie que lorsque sa garde est vraie. Si aucune garde n'est évaluée à true en sortie d'un état :

[5] Le parseur ne sait pas vérifier que le type message n'est utilisé que dans les médias

– si au moins une garde porte sur une réception de message, on attend le message suivant, et on teste à nouveau la garde, jusqu'à ce qu'on trouve un message qui la satisfait ;
– si aucune garde ne porte sur un message, le composant est bloqué dans l'état courant, une erreur doit être levée.

Les transitions de réception de message des médias peuvent présenter un cas particulier pour les gardes. Les gardes portent en effet sur le contenu des variables après lecture du contenu du discriminant du message. Ce mécanisme permet d'accepter ou non le message en entrée. Si la garde n'est pas vérifiée, le message n'est pas consommé, et la transition ne peut être franchie (pas de modification de l'état du média).

6.7 Instanciation dynamique de composants

6.7.1 Présentation de la syntaxe

La syntaxe des instanciations dynamiques est donnée par la figure 21.

instanciation : *expr_variable* **:=** *IDENTIFIER* **(** *IDENTIFIER* **=>** **expression** **;**
[**,** *IDENTIFIER* **=>** **expression** **]** $*$ **)** (1)
| *expr$_v$ariable* **:=** *IDENTIFIER* **;** (2)

FIG. 21 – paramètres d'instanciation d'une classe

La règle (1) correspond à l'instanciation du composant avec initialisation de tout ou partie de ses variables :
– expr_variable correspond à la variable qui contiendra la référence de l'instance produite.
– Le premier IDENTIFIER correspond au nom du type du composant à instancier.
– Les éléments entre parenthèse correspondent à l'initialisation des attributs du composant :
  – IDENTIFIER correspond au nom de l'attribut ;
  – expression correspond à la valeur initiale de la variable.

la règle (2) correspond à la création d'une instance d'un composant sans initialiser ses attributs.

En **L*f*P**, il n'y a pas de distinction explicite pour la visibilité des attributs d'un composant. En revanche, les types définis dans un composant ne sont pas visibles depuis le reste du modèle. Lors d'une instanciation dynamique, il est donc impossible d'initialiser les attributs dont le type est défini dans le composant instancié.

6.7.2 Exemple

Soit une classe `C1` définissant deux attributs `A1` et `A2` de type integer , et une variable `V` de type `C1`, on peut écrire :

```
-- instanciation définissant tous les attributs
-- exportés de la classe.
V := C1(A1 => 3, A2 =>4) ;
```

7 Un exemple commenté : les horloges

Cette section est dédiée à la présentation d'un exemple commenté : une modélisation simplifiée des horloges de Matern (horloges vectorielles). La sous-section 7.1 présente le système en se basant sur le diagramme de classe UML. Les sections suivantes présentent la modélisation **L*f*P** du système en commençant par le diagramme d'architecture (sous-section 7.2, puis en se focalisant sur chacun des composants **L*f*P** du modèle (sous-sections 7.3 à 7.6).

7.1 Présentation du système modélisé

On souhaite modéliser le mécanisme des horloges vectorielles. On considère un nombre fixe de participant (pas de pannes, ni de nouvel arrivant pendant l'exécution).

La figure 22 donne le diagramme de classe UML du système. On distingue principalement quatre classes :

- `application_class` représente les classes utilisant le système d'horloges, dans le cadre de cette modélisation, toutes les classes utilisant les horloges doivent disposer de la méthode `receive` qui est rendue nécessaire par le fait que `fifo_link` implémente une liaison unidirectionnelle ;
- `clock` implémente la gestion des horloges vectorielles, elle dispose pour cela de deux méthodes `send_message` et `get_message` permettant à l'application d'envoyer et de recevoir des messsages.
- `clock_manager` sert de superviseur pour le système, il permet l'initialisation de l'application ;
- `fifo_link` est la classe de liaison entre les composants du système, elle implémente une fifo unidirectionnelle, sans perte de message.

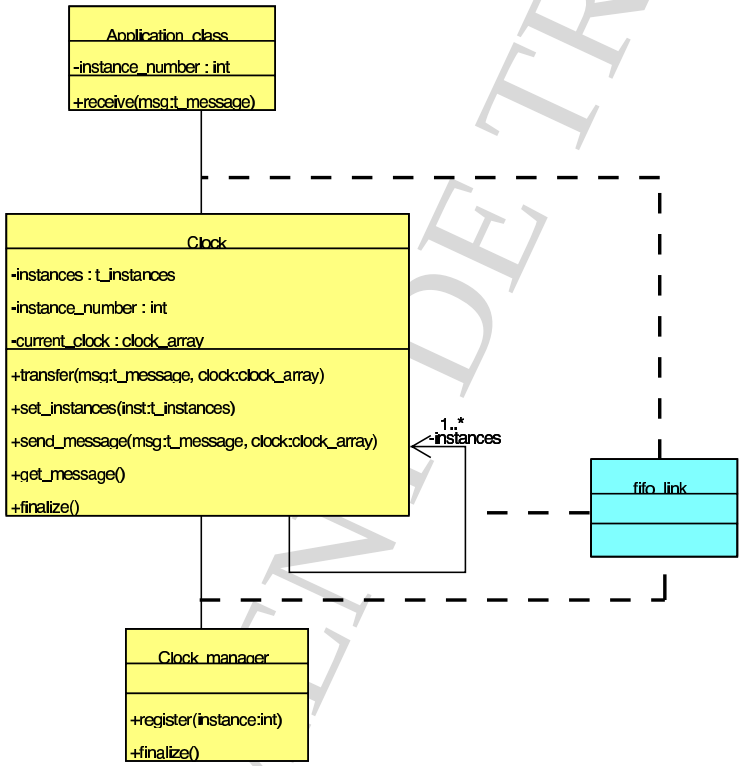


FIG. 22 – Diagramme de classe du système des horloges vectorielles

7.1.1 La classe `application_class`

`Application_class` est une classe "vide" qui permet de représenter artificiellement une application utilisant le système d'horloges. On note que la classe doit contenir au moins un attribut : `instance_number` qui identifie chaque instance de la classe. Toutes les instances des classes utilisant les horloges doivent avoir un numéro unique. La méthode `receive` permet de recevoir les messages. L'application peut demander un message en appelant la méthode `get_message` de la classe `clock` qui sera vue plus un peu plus loin.

7.1.2 La classe `clock`

`Clock` représente le mécanisme d'horloge. Parmis ses attributs, on trouve :

– `instance_number` de type entier est le numéro de l'horloge. Ce numéro est égal à celui de l'instance de `application_class` que l'horloge doit gérer. C'est également l'indice de cette instance dans l'horloge vectorielle.
– `instances` : un tableau de référence vers les autres horloges participant à l'application. L'indice d'une référence dans ce tableau est toujours égal à son numéro d'instance.
– `current_clock` est la valeur de l'horloge de l'instance à l'instant courant.

Les méthodes de `clock` sont :
– `set_instances` est appelée par `clock_manager` lorsque toutes les instances d'horloges se sont enregistrées. Elle prend en paramètre la liste des horloges participant à l'application.
– `send_message` est appelée par l'instance de `application_class` qui est reliée à l'horloge. Cette méthode est appelée pour envoyer un message (paramètre `msg`) à un destinataire désigné par son numéro d'instance (paramètre `destination`).
– `get_message` est appelée par `application_class`) lorsqu'elle veut récupérer un message. Le message n'est pas retourné directement (la liaison fifo est unidirectionnelle).
– `transfer` prend un message dans son binder de réception et l'envoie à `application_class` après avoir mis à jour son horloge.
– `finalize` cette méthode (qui n'est pas utilisée pour le moment...) autorise la destruction de l'horloge.

Lors de leur création, les instances de la classe clock doivent :
– créer les liens de communication vers `clock_manager` ;
– s'enregistrer auprès de `clock_manager`

7.1.3 La classe `clock_manager`

`clock_manager` permet de gérer l'initialisation de l'application[6]. C'est sur cette classe que les horloges s'enregistrent lors de leur création. Lorsque la classe a enregistré tous les participants, elle lance la partie applicative proprement dite en appelant la méthode `set_instances` de la classe `clock` sur toutes les instances enregistrées.
`clock_manager` dispose de deux méthodes :
– `register` permet à une horloge de s'enregistrer, et lorsque le nombre d'horloge requis est atteint, elle lance l'application proprement dite ;
– `finalize` termine l'instance courante.

7.1.4 La classe de liaison `fifo_link`

Cette classe représente une communication unidirectionnelle fifo fiable. Ce comportement correspond aux carractéristiques nécessaires pour représenter les liens entre horloges. Par extension, et pour simplifier l'exemple, c'est le moyen de communication qui est retenu pour l'ensemble des communications du modèle.

7.1.5 Présentation du fonctionnement de l'exemple

Le diagramme de séquence de la figure 23 présente un scénario possible d'initialisation du système. Les messages sont présentés au niveau des classes applicatives (les classes d'interactions sont ignorées).
On considère initialement :
– deux instances d'`application_class` ;
– une instance de `clock_manager`.

Le scénario considéré est le suivant :
– la première instance d'`application_manager` crée l'horloge qui lui est associée ;
– la nouvelle instance de `clock` s'enregistre sur `clock_manager` ;
– la deuxième instance d'`application_manager` crée l'horloge qui lui est associée ;
– cette deuxième instance de `clock` s'enregistre sur `clock_manager` ;
– toutes les instances attendues sont enregistrées, `clock_manager` envoie donc deux messages asynchrones `set_instances`.

Lors du fonctionnement "courrant" de l'application, on a un schéma d'interaction pouvant correspondre au diagramme de séquence présenté par la figure 24. Sur ce diagramme, on ne représente pas les liens de communication (classe `fifo_link` qui assurent l'envoi des messages asynchrones ; on considère deux

[6] Si le modèle était réellement complet, elle devrait aussi gérer sa terminaison

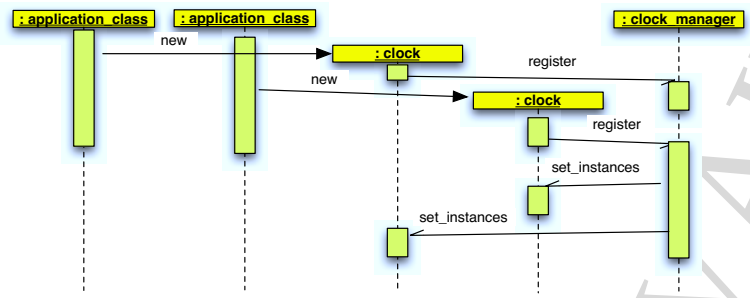


FIG. 23 – séquence d'initialisation de l'application pour deux instances d'`application_class`

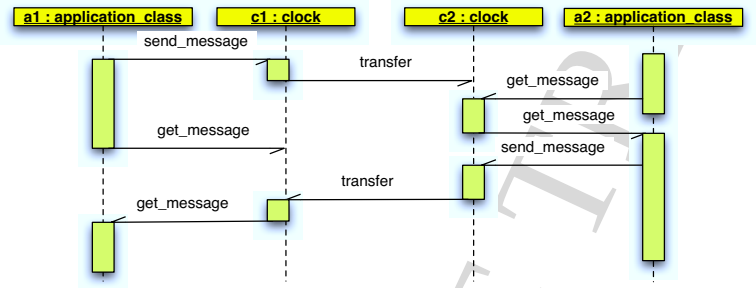


FIG. 24 – Un scénario de fonctionnement de l'exemple avec deux instances d'`application_class`

instances de `application_class` `a1` et `a2` et les deux instances de `clock` qui leurs sont respectivement associées : `c1` et `c2`.

Le scénario se déroule comme suit :

- `a1` envoie un message à `a2` via `c1` ;
- `c1` marque le message avec l'horloge courante associée à `a1`, puis le transmet à `c2` via la méthode `transfer` ;
- `c2` conserve le message jusqu'à ce que `a2` lui demande un message (appel de `get_message` ;
- `c2` transmet alors le message à `a2` et met à jour sa propre horloge ;

Afin de Décrire le système plus précisément, les sections suivantes présentent la modélisation de ce système dans le langage **L*f*P**

7.2 Le diagramme d'architecture

La figure 25 présente le diagramme d'architecture du système. On y retrouve les classes vues précédemment. La classe `fifo_link` du diagramme UML est implémentée sous la forme d'un média **L*f*P** reliant les autres classes du modèle. Les instances seront créées dynamiquement.

On trouve en plus sur ce diagramme un ensemble de définitions :

- Une définition de constante (`Nbr_of_instances` qui correspond au nombre d'instances participant à l'application.
- un ensemble de définition de types :
  - `clock_array` permet de représenter les horloges vectorielles ;
  - `t_instances` qui permet de représenter un tableau de références vers des instances de la classe `clock`.
- `t_message` est le type des messages envoyés. Pour cet exemple simple, on a définit un type énuméré réduit à un seul élément. Mais il pourrait s'agir d'un type beaucoup plus complexe (tableau, ou type `record`).
- les instances statiques du modèle :
  - `inst1`, `inst2`, `inst3` qui sont les instances de `application_class` participant à l'application. Leur attribut `instance_number` est initialisé de manière statique.

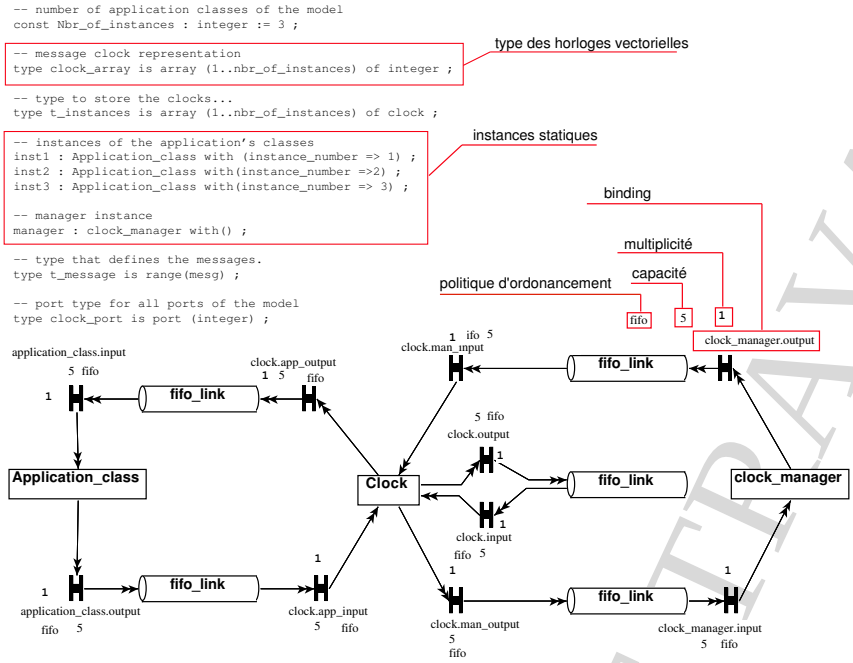


FIG. 25 – Diagramme d'architecture du modèle des horloges

– `manager` qui servira de superviseur pour l'application (instance de clock_manager). Aucun attribut n'est initialisé.

Dans le cadre de cet exemple, on s'est limité à trois instances d'`application_class`. Pour étendre le nombre de sites participant, il suffit de modifier la constante `nbr_of_instances`, et le nombre d'instances statiques de `application_class`.

On note sur chaque binder les attributs permettant de définir l'association entre les ports de la classe et les binders, la cardinalité, la capacité du buffer, et la politique d'ordonnancement.

7.3 La classe `application_class`

Le diagramme de comportement de cette classe est présenté sur la figure 26. On retrouve les déclarations des attributs définis dans le diagramme UML :

– `my_clock` : l'instance de `clock` associée à l'instance courante ;
– `instance_number` : le numéro de l'instance courante ;

On trouve également de nouvelles variables rendues nécessaires par les choix de modélisation :

– `the_fifo` qui stocke les valeurs retournées par les instanciations de média. Cette valeur n'est pas utilisée dans ce modèle, mais la valeur de retour d'une instanciation doit toujours être écrite dans une variable ;
– `destination` qui représente le numéro d'instance de la destination d'un message de l'application.

On trouve également la déclaration de la méthode `receive` illustrée par la figure 27. Elle permet la réception d'un message sur le port d'entrée. C'est une procédure asynchrone, dont l'unique paramètre (en mode `in`) contient le message.

D'un point de vue comportemental, la classe commence par créer l'instance d'horloge qui lui est associée, et les deux liens de communications les reliant ensemble. Ensuite, elle envoie, et reçoit alternativement des messages à des destinataires choisis dans un ordre important peu pour l'exemple.

La méthode `receive` permet de recevoir un message (passé en paramètre). Dans le cadre de cet exemple, aucun traitement n'est prévu, le message est juste accepté et retiré de la liste des messages en attente.

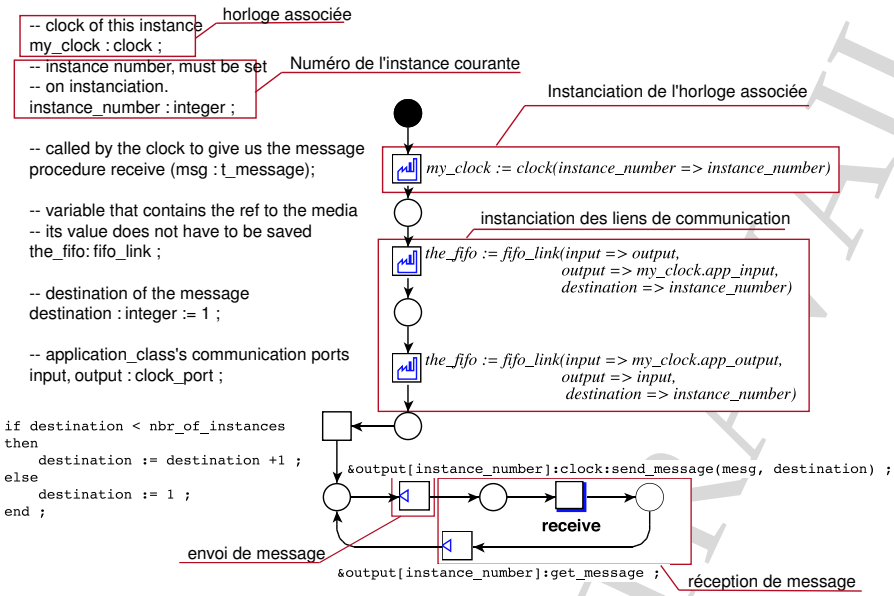


FIG. 26 – Diagramme de comportement de la classe `application_class`

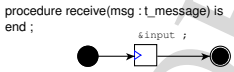


FIG. 27 – Diagramme de comportement de la méthode `receive` de la classe `application_class`

7.4 La classe `clock`

La classe clock est chargée de la gestion des horloges. Son diagramme de comportement est donné par la figure 28. Le comportement de la classe est simple : Lors de son lancement, elle appelle le sous diagramme init dont le comportement est donné par la figure 29. Ce sous-diagramme appelle la méthode `register` de `manager` en lui fournissant son numéro d'instance.

Puis la classe attend qu'on appelle la méthode `set_instances` dont le comportement est spécifié sur la figure 30. Le paramètre formel de cette méthode est un tableau contenant l'ensemble des références des instances de `clock` du modèle. L'objectif de cette méthode est d'attendre que le manager fournisse les références vers les autres instances de `clock` participant à l'application. Cet ensemble de valeurs est utilisé par le sous-diagramme `set_links` présenté sur la figure 31 pour créer un lien fifo entre l'instance courante de `clock` et chacune des autres instances. A chaque fois, on précise au média le numéro d'instance de son destinataire (`clock_manager` garantit que le numéro d'instance est égal à l'indice de la référence dans le tableau).

Une fois que la phase d'initialisation est passée, trois méthodes sont accessibles :

- `get_message` (cf. figure 32) ;
- `send_message` cf. figure 34) ;
- `finalize` (cf. figure 35).

`get_message` est appelée par l'instance de `application_class` reliée à l'horloge, lorsqu'elle souhaite lire un message. Cette méthode est "vide" et sert simplement à faire changer la classe d'état. Après exécution de cette méthode, la classe ne peut exécuter que la méthode `transfert`.

Cette méthode est présentée sur la figure 33. Elle lit un message sur le binder d'entrée `input` reliant les horloges entre elles, et le transmet à `application_class` en appelant la méthode `receive`. La classe clock reste bloquée tant qu'un message ne lui est pas parvenu.

REMARQUE : Au niveau de `application_class`, il faut s'assurer que tout appel à `get_message` est bien suivi d'une attente sur la méthode `receive` ; cette caractéristique pourait faire partie des propriétés à vérifier sur le modèle.

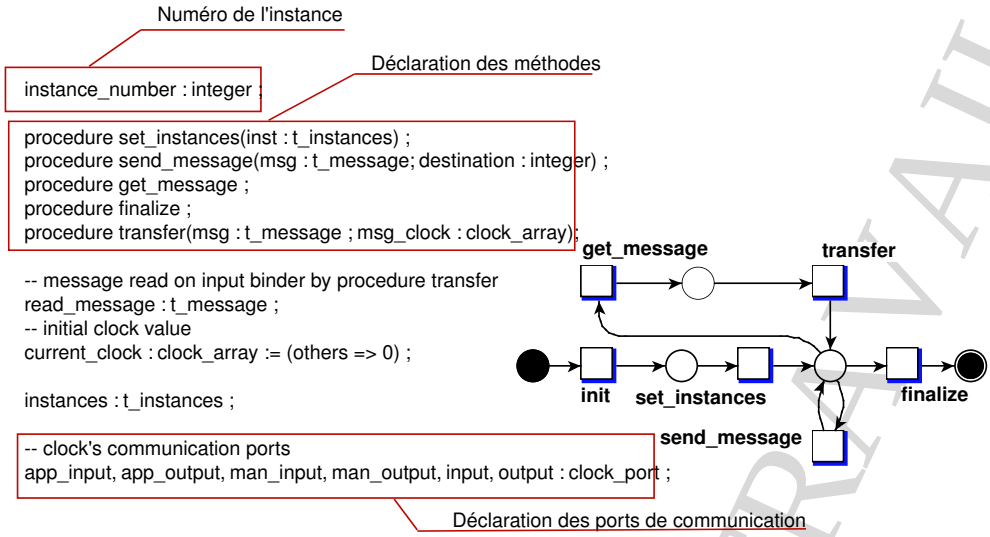


FIG. 28 – Diagramme de comportement de la classe `clock`

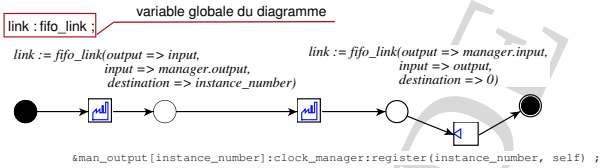


FIG. 29 – Diagramme de comportement du sous-diagramme `init` de la classe `clock`

La méthode `send_message` (cf. figure 34) est appelée par `application_class` pour envoyer un message. Le message et son destinataire sont passés en paramètres ; comme cette procédure est asynchrone, l'appelant poursuit son exécution pendant l'envoi du message.

La méthode `finalize` doit être appelée lorsqu'on souhaite détruire l'instance d'horloge. Son seul effet est de la laisser aller dans son état terminal.

7.5 la classe `clock_manager`

Le diagramme de cette classe, présenté sur la figure 36 dispose que de deux méthodes. La méthode `finalize` (cf.figure 37 permet de terminer l'instance du gestionnaire.La méthode `register` présentée sur la figure 38 est utilisée par les instances d'horloges lors de leur initialisation. Elle stocke les références d'horloges lorsque celles-ci s'enregistrent. Une fois que toutes les horloges se sont enregistrées (`nbr_registered` = `number_of_instances`, cette méthode lance l'application proprement dite. Cette opération est réalisée par le sous diagramme `start_appli`.

Le diagramme de comportement de ce sous-diagramme est donné par la figure 39. On peut noter sur cette figure que la syntaxe de l'attribut global est différent de ceux des triggers et méthodes qui doivent reprendre leurs prototype ce qui est inutile dans le cas d'un sous-diagramme.

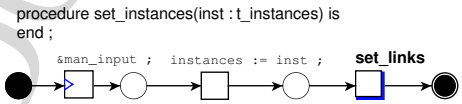


FIG. 30 – Diagramme de comportement de la méthode `set_instances` de la classe `clock`

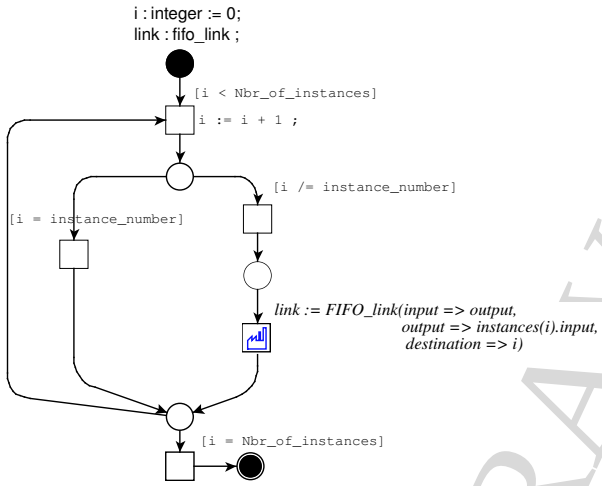


FIG. 31 – Diagramme de comportement du sous-diagramme `set_links`

FIG. 32 – diagramme de comportement de la méthode `get_message` de la classe `clock`

7.6 Le média `fifo_link`

Ce média, représenté sur la figure 40 implémente une fifo unidirectionnelle sans perte de message. Le “buffer” de la fifo est assuré par les ports aux extrémités. Le média en lui même se contente de prendre un message sur son entrée, et de le transférer sur la sortie.

Ce média ne lit que les messages qui sont destinés à l’instance portant le numéro conservé dans l’attribut `instance_number`.

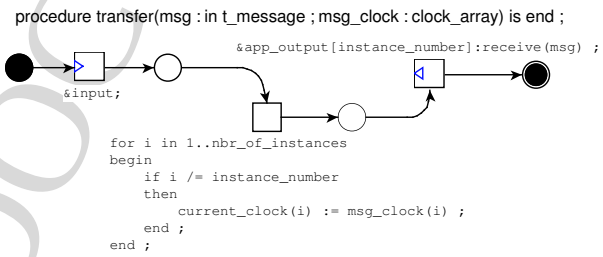


FIG. 33 – Diagramme de comportement de la méthode `transfer` de la classe `clock`

FIG. 34 – Diagramme de comportement de la méthode `send_message` de la classe clock

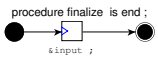


FIG. 35 – Diagramme de comportement de la méthode `finalize` de la classe `clock`

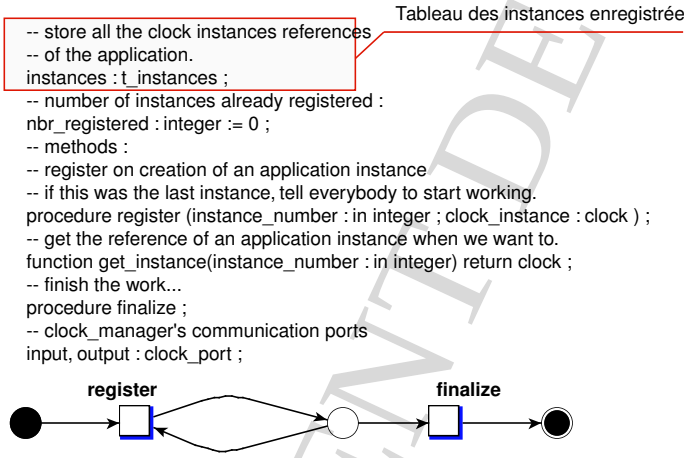


FIG. 36 – Diagramme de comportement de la classe `clock_manager`

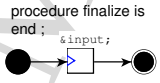


FIG. 37 – Diagramme de la méthode `finalize` de `clock_manager`

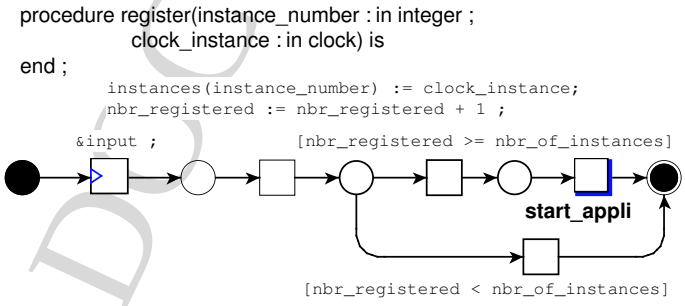


FIG. 38 – Diagramme de comportement de la méthode `register` de la classe `clock_manager`

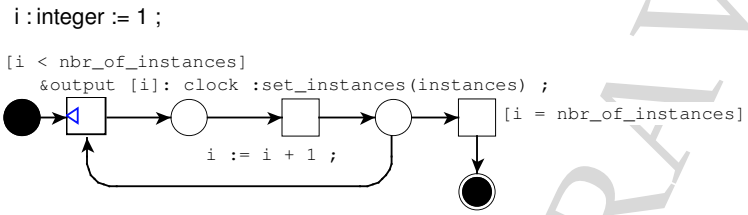


FIG. 39 – Sous diagramme `start_appli` de la classe `clock_manager`

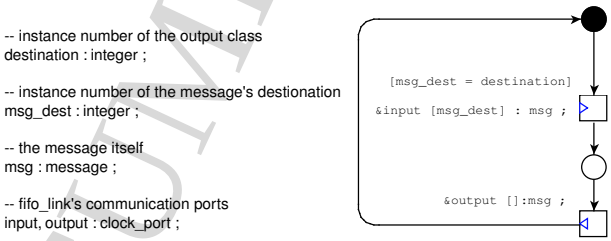


FIG. 40 – Diagramme de comportement du média `fifo_link`