Type document : *SRS*	*Projet RNTL MORSE*	**Date :** 12/06/2003
Sous-projet n° 2		**numéro de doc :** MORSE-SRS-031114-V0.14-FGI
tâche : 2.1		**rédacteur (s)** Frédéric Gilliers

Description de la sémantique du langage L*f*P

Résumé

Ce document a pour objectif de décrire de manière aussi précise que possible la sémantique du langage **L*f*P**. Il s'appuie sur le travail préliminaire réalisé par Denis Poitrenaud et Emmanuel Paviot-Adet dans le cadre de la vérification de modèles **L*f*P** à l'aide de DDD.

Table des matières

1 Introduction **2**
- 1.1 Structure d'une application spécifiée avec **L*f*P** . . . 2
- 1.2 Démarche Méthodologique . . . 3
- 1.3 Structuration du document . . . 3

2 Structuration des modèles L*f*P **4**
- 2.1 Le diagramme d'architecture . . . 5
- 2.2 Les diagrammes de comportement des composants . . . 5

3 Sémantique des types L*f*P **5**
- 3.1 Les types de base du langage . . . 5
  - 3.1.1 Le type `integer` . . . 6
  - 3.1.2 Le type `message` . . . 6
  - 3.1.3 le type `semaphore` . . . 6
- 3.2 Les types "ensemble" (set) . . . 6
  - 3.2.1 Les types multi-ensemble (bag) . . . 7
- 3.3 Les types énumérés . . . 7
- 3.4 Les types définis par restriction de type . . . 8
- 3.5 Les types "port" . . . 9
- 3.6 Les références vers les composants du modèle . . . 10
- 3.7 Les types article (record) . . . 10
- 3.8 Les types tableau . . . 10

4 Sémantique des éléments du diagramme d'architecture **11**
- 4.1 Sémantique statique des composants **L*f*P** . . . 11
- 4.2 Sémantique des binders . . . 11
  - 4.2.1 Multiplicité des binders . . . 11
  - 4.2.2 Capacité des binders . . . 12
  - 4.2.3 Ordonancement des messages . . . 12
  - 4.2.4 Liaison entre les binders et les composants . . . 12
  - 4.2.5 Ajout de message dans un binder . . . 13
  - 4.2.6 Lecture d'un message dans un binder . . . 13
- 4.3 Sémantique des arcs . . . 14

DOCUMENT DE TRAVAIL

5 Sémantique des diagrammes de comportement **14**
5.1 définitions 14
5.2 Éléments communs 15
5.2.1 Définition des composants **L*f*P** 15
5.2.2 Sémantique générale des états 16
5.2.3 Sémantique des sous-diagrammes 16
5.2.4 Sémantique des triggers 17
5.2.5 Opérations sur les sémaphores 17
5.2.6 Sémantique des transitions simples 17
5.2.7 Instanciation dynamique de composants 18
5.3 Diagramme de comportement d'une classe 18
5.3.1 Sémantique des transitions de communication 19
5.3.2 Sémantique des méthodes 19
5.3.3 Sémantique des états des classes 20
5.3.4 Appel de procédure synchrone 20
5.3.5 Appel de fonction 21
5.3.6 Appel de procédure asynchrone 22
5.3.7 Envoi et réception de messages entre classes 22
5.3.8 Envoi de messages de contrôle aux médias 23
5.4 Diagramme de comportement d'un média 23
5.4.1 Sémantique des transitions de communication 23
5.4.2 Réception de messages dans les médias 23
5.4.3 Envoi de messages par les médias 24

1 Introduction

L'objectif de ce document est de fournir une documentation aussi précise que possible de la sémantique du langage **L*f*P**. Il trouve ses racines dans le travail effectué par Dan Regep[1] pour la définition du langage **L*f*P**, Denis Poitrenaud[2] et Emmanuel Paviot-Adet[3] dans le cadre de la vérification de modèles **L*f*P** à l'aide de DDD.

1.1 Structure d'une application spécifiée avec L*f*P

Le langage **L*f*P** est dédié à la description de la partie contrôle des systèmes distribués. Cela signifie que les spécifications exprimées doivent décrire précisément la manière dont les composants de l'application communiquent et interagissent. La partie "calculatoire" de l'application n'est pas modélisable par ce biais. Une liaison entre des composants "métier" et la spécification **L*f*P** doit être établie pour obtenir une représentation globale de l'application.

Les interactions entre les composants "métier", et la spécification **L*f*P** est illustrée sur la figure 1. Les classes **L*f*P** appellent des fonctions appartenant à des bibliothèques pré-définies. La production de ces bibliothèques n'est pas abordée par le langage **L*f*P** et sort du cadre du projet MORSE.

Chaque composant métier est constitué d'un ensemble de composants de base interagissant pour remplir une fonction de l'application. Ces interactions doivent respecter un ensemble de contraintes garantissant qu'elles n'influent pas sur la partie contrôle de l'application. Un composant métier ne doit donc pas :

- modifier directement l'état courant d'un composant **L*f*P**, ou une de ses variables.
- appeler une méthode d'un composant **L*f*P** ;
- communiquer directement avec une instance de composant manipulée par un autre composant **L*f*P**.

[1]dan.regep@lip6.fr
[2]denis.poitrenaud@lip6.fr
[3]emmanuel.paviot-adet@lip6.fr

DOCUMENT DE TRAVAIL

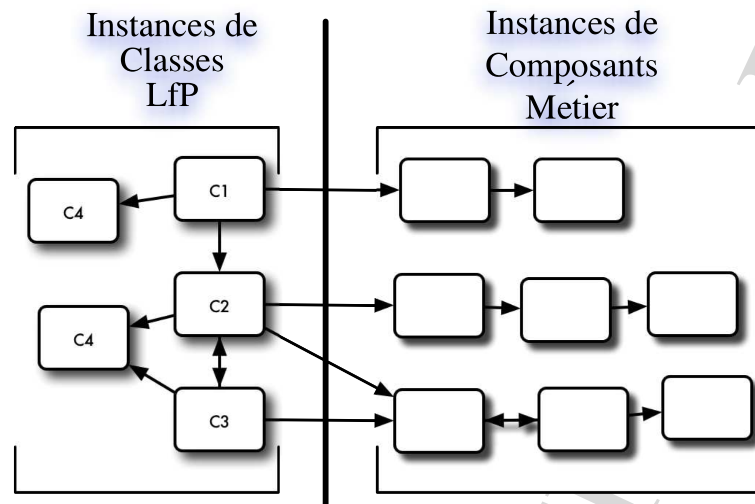


FIG. 1 – Interaction entre les componsants "métier" et la spécification **L*f*P**

DOCUMENT DE TRAVAIL

De même, les composants **L*f*P** doivent transmettre les messages circulant entre les composants “métier” sans altérer leur contenu. L’interaction avec les composants métiers doit être limitée à l’interface prévue par l’implémentation. Si ces contraintes ne sont pas totalement respectées, il sera impossible de vérifier formellement le comportement du système. Un composant de l’application sera donc constitué de l’agrégation d’un composant **L*f*P** et de l’ensemble des composants métiers avec lesquels il communique directement.

1.2 Démarche Méthodologique

Du point de vue de la méthodologie de développement de l’application, l’impact de **L*f*P** est illustré par la figure 2. Les parties “contrôle” et “métier” de l’application sont spécifiés et implémentés de manières différentes : ils

L’objectif du projet MORSE est de produire la chaîne de traitement de la partie “contrôle” de l’application répartie. La première étape consiste à la traduire une spécification de haut niveau (UML) en une spécification **L*f*P**. Une étape de vérification formelle est ensuite appliquée pour s’assurer que la solution retenue respecte les exigences exprimées. Puis le code correspondant est généré. Le programme réparti ainsi produit utilise les bibliothèques de composants externes pour réaliser les parties “métier” de l’application. Le générateur de code produisant le code de “contrôle” de l’application ne fait aucune hypothèse sur la manière dont sont produites ces bibliothèques. Les appels aux fonctions de bibliothèque sont spécifiés dans la description du contrôle de l’application sous la forme “d’appels externes”.

1.3 Structuration du document

Ce document définit la sémantique du langage **L*f*P**, il passe donc en revue l’ensemble des structures et opérateurs du langage et en décrit le comportement en fonction du contexte d’exécution. Il est structuré comme suit :

1. présentation de la structure des modèles **L*f*P** ;
2. sémantique des types reconnus par le langage ;
3. sémantique du diagramme d’architecture :
   - sémantique des composants ;
   - sémantique des binders ;
4. sémantique des diagrammes de comportement ;
   - sémantique des méthodes ;

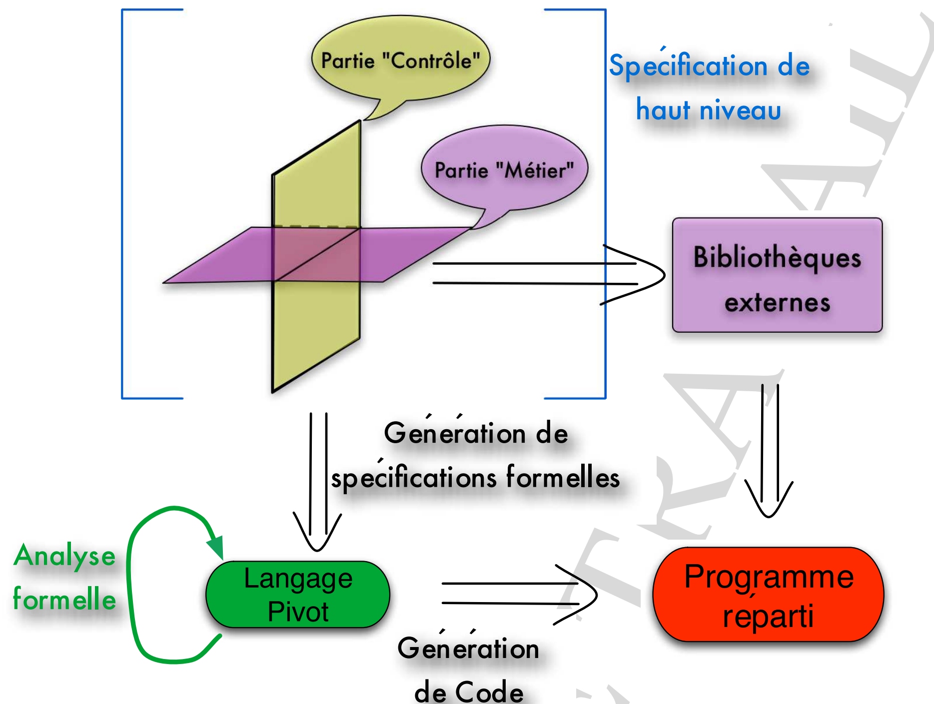


FIG. 2 – Méthodologie de développement avec **L*f*P**

– sémantique des sous-diagrammes ;
– sémantique des triggers ;
– opérations sur les sémaphores ;
– structures de contrôle graphiques ;
– structures de contrôle textuelles ;
– instanciation dynamique de composants.

Le premier point est étudié à la section 2 qui présente la structuration générale des modèles **L*f*P**. Elle introduit les principaux concepts du langage et les présente succintement.

Le deuxième point est étudié à la section 3 qui présente l'ensemble des types reconnus par le langage **L*f*P**. Chaque sorte de type est décrite avec l'utilisation qui peut en être faite ainsi que les opérations primitives qui y sont associées.

Le troisième point est abordé à la section 4 qui présente la sémantique des éléments du diagramme d'architecture. Cette section présente donc en détail la sémantique statique des classes, médias et binders **L*f*P**.

Le quatrième point est abordé à la section 5, il concerne la sémantique d'exécution des composants du modèle. Cette section décrit donc les opérateurs structurant l'exécution des composants.

2 Structuration des modèles L*f*P

Les spécifications **L*f*P** prennent la forme d'un ensemble de diagrammes hiérarchiques représentant le comportement des composants du modèle. Chaque modèle **L*f*P** utilise deux types de diagrammes :

– le diagramme d'architecture spécifie de manière statique la topologie de l'application, et les liens de communication existant entre les composants ;
– les diagrammes de comportement des composants spécifient le comportement dynamique de chaque composant de manière hiérarchique : chaque transition du diagramme

DOCUMENT DE TRAVAIL

peut être elle même décomposée en un diagramme de comportement de niveau inférieur.

2.1 Le diagramme d'architecture

Les principaux éléments consitutant un diagramme d'architecture sont les suivants :
- les `classes` sont les composants applicatifs du modèle ;
- les `médias` sont les composants logiciels permettant d'assurer les liens de communication entre les classes ;
- les `binders` définissent les points d'interaction entre classes et médias.

Les classes et les médias sont spécifiés en **L*f*P** à l'aide de diagrammes de comportement. Les binders sont des buffers de messages permettant d'assurer la connexion entre classes et médias. La manière dont les liaisons sont réalisées dépend de la multiplicité du binder ; il est possible de spécifier la politique d'ordonnancement des messages de chaque binder. Ces points, seront abordés en détail par la section 4 consacrée au diagramme d'architecture.

2.2 Les diagrammes de comportement des composants

Les diagrammes de comportement permettent de spécifier hiérarchiquement le contrat d'exécution de chaque composant. Ils sont principalement constitués des éléments suivants :
- Les `états` qui correspondent à des points de choix entre (un ou) plusieurs comportement possibles (alternatives) ;
- Les `transitions` qui permettent de spécifier les actions à réaliser par le composant ;
- Les `opérations sur les sémaphores` ;
- Les `triggers` qui correspondent à un sous diagramme de comportement qui peut être réutilisé en plusieurs emplacements dans le diagramme de comportement d'un composant ;
- Les opérations `d'envoi et de réception` de messages.
- Pour les classes, des `méthodes` qui correspondent à des points d'activation (elles sont exécutées sur réception du message correspondant) ;
- Pour les classes, des `appels de méthodes` distants entre classes.

La structure globale de ces diagrammes correspond à un automate état / transition. Bien que relativement similaires, les diagrammes de comportement d'une classe et d'un média n'ont pas la même structure. Ces différences, ainsi que les définitions complètes des opérations utilisables seront abordées dans la section 5.

3 Sémantique des types L*f*P

Cette section présente le système de typage de **L*f*P** ainsi que la sémantique des types du langage et leur utilisation. Cette section étudie donc successivement les types suivants :
- les types de base du langage ;
- les types définis par restriction de type ;
- les types énumérés ;
- les types "ports" ;
- les références vers les composants du modèle ;
- les types ensemble (set) ;
- les types "articles" (record) ;
- les types tableau.

3.1 Les types de base du langage

Les types de base connus du langage **L*f*P** sont les suivants :
- le type `integer` ;
- le type `message` ;
- le type `semaphore` ;

DOCUMENT DE TRAVAIL

DOCUMENT DE TRAVAIL

3.1.1 Le type `integer`

Le type integer représente les entiers signés. Il n'y a pas de borne théorique à ce type. Les bornes doivent être fixées par l'implémentation et / ou la vérification formelle. Ce n'est pas un type circulaire : si un résultat entier ne fait pas partie de l'intervalle choisi pour l'implémentation, une erreur doit être levée.

Les opérations définies sur le type `integer` sont les suivantes :

- successeur : retourne le successeur de l'entier (`integer`) passé en argument ;
- prédécesseur : retourne le prédécesseur de l'entier (`integer`) passé en argument ;
- somme : La somme de deux entiers a et b retourne le b^{ieme} successeur de a
- différence : la différence de deux entiers a et b retourne le b^{ieme} prédécesseur de a.
- produit : le produit de a par b est défini par : $\sum_{i=1}^{b} a$
- quotient : le quotient q de a par b est défini par : $q \times b = a$ où $q \times b$ désigne le produit de q par b.
- opérateurs de comparaison : les relations d'ordre classiques sont définies sur les types entiers ($<$, $\leq$, $>$, geq, $\neq$).

3.1.2 Le type `message`

Le type `message` est utilisé pour représenter les messages de l'application. Il n'est exploitable que dans les médias : on ne peut manipuler explicitement des messages dans les classes.

Ce type est utilisé dans les médias pour stocker le contenu des messages à transmettre à l'instance de destination du message. La seule opération définie pour ce type est l'affectation de la valeur du message à une variable ; elle est réalisée par recopie du contenu du message.

3.1.3 le type `semaphore`

Les sémaphores sont des mécanismes de synchronisation permettant la définition de sections critiques durant l'exécution des composants.

Lors de sa déclaration une variable sémaphore, doit être initialisée avec une valeur entière qui défini le nombre de composant pouvant initialement entrer dans la section critique.

On défini trois opérations sur les sémaphores :

- **initialisation** : initialise le compteur du sémaphore à une valeur entière, supérieure ou égale à 0.
- **`P`** : si le compteur est strictement supérieur à 0, décrémente le compteur du sémaphore. Si le compteur est égal à 0, l'opération est bloquante jusqu'à ce que le compteur redevienne strictement positif et puisse être décrémenté.
- **`V`** : incrémente le compteur du sémaphore, si des opérations **`P`** étaient en attente sur ce sémaphore, la plus ancienne peut se terminer.

3.2 Les types "ensemble" (set)

Les types ensemble permettent de stocker des valeurs différentes de même type. Un ensemble ne peut contenir deux valeurs identiques. Le nombre d'élément d'un ensemble n'est théoriquement pas limité.

Opérations disponibles sur les ensembles :

- Ajout d'une valeur : ajoute une valeur à l'ensemble. Si la valeur n'est pas déjà contenue par l'ensemble, elle est ajoutée. Si la valeur est déjà présente dans l'ensemble, il n'est pas modifié.
- Retirer un élément d'un ensemble : retire une valeur d'un ensemble et retourne cette valeur. La valeur est choisie aléatoirement dans l'ensemble.
- Cardinalité d'un ensemble : retourne le nombre d'élément contenus dans l'ensemble.
- union : retourne un ensemble égal à l'union ensembliste de deux ensembles.
- intersection : calcule l'intersection de deux ensemble (ensemble des éléments communs aux deux ensembles)

– l'inclusion(resp. inclusion stricte) : retourne un booleen égal à true si l'ensemble à gauche de l'opérateur est inclu (resp. strictement inclu) dans l'ensemble situé à droite de l'opérateur.

3.2.1 Les types multi-ensemble (bag)

Les types multi-ensemble permettent de stocker une suite de valeur de même type. Un multi-ensemble peut contenir plusieurs d'occurences d'une même valeur.

Opérations disponibles sur les multi-ensembles :
– Ajout d'une valeur : ajoute une valeur au multi-ensemble.
– Retirer une valeur : retire une occurence d'une valeur du multi-ensemble et retourne cette valeur.
– Cardinalité d'une valeur du multi-ensemble : retourne le nombre d'occurence d'une valeur donnée dans le multi-ensemble.
– Cardinalité du multi-ensemble : retourne le nombre total d'occurences de valeurs présentes dans le multi-ensemble.
– union : retourne l'union de deux multi-ensembles : soient les multi-ensemble $me_1 = \{a_1, a_1, a2, a_3\}$ et $me_2 = \{a_1, a_4\}$, l'union de me_1 et me_2 est : $me_1 + me_2 = \{a_1, a_1, a_1, a_2, a_3, a_4\}$.
– intersection : conserve les éléments communs à deux multi-ensembles, avec la cardinalité la plus faible pour chaque valeur : soient deux multi-ensembles $me_1 = \{a_1, a_1, a_2, a_3\}$ et $me_2 = \{a_1, a_2, a_4\}$ leur intersection est le multi-ensemble suivant : $\{a_1, a_2\}$.
– l'inclusion stricte : retourne un booleén égal à true si le multi-ensemble à gauche de l'opération est inclu strictement dans le multi-ensemble à droite de l'opérateur. Un multi-ensemble a_1 est inclu strictement dans un multi-ensemble a_2 si et seulement si tous les cardinaux des valeurs de a_2 sont supérieurs ou égaux aux cardinaux des valeurs de a_1 et que a_1 et a_2 sont différents(i.e. : au moins un cardinal d'une valeur de a_1 est strictement inférieur à celui de a_2 ou bien a_2 contient des valeurs qui ne sont pas contenues dans a_1).
– l'inclusion : retourne un booléen égal à true si le multi-ensemble à gauche de l'opérateur est inclu dans le multi-ensemble à droite de l'opérateur, ou s'ils sont égaux.

3.3 Les types énumérés

Les types énumérés sont définis comme des ensembles de valeur de même type. Les valeurs des types énumérés sont implicitement ordonnées dans l'ordre de leur déclaration. Un type énuméré peut être circulaire.

Opérations définies pour les types énumérés :
– successeur : retourne l'élément suivant du type. Si le type est circulaire, le successeur du dernier élément défini est le premier élément défini.
– prédécesseur : retourne l'élément précédant du type. Si le type est circulaire, le prédécesseur du premier élément déclaré est le dernier élément déclaré.
– relation d'ordre : la relation d'ordre est définie en fonction de l'ordre de déclaration des valeurs du type. Si une valeur v_1 est déclarée avant v_2, alors : $v_1 < v_2$ est vrai. Cette relation d'ordre n'est donc pas affectée par le fait que le type soit circulaire ou non.
  Les autres opérateurs de relation sont définies à partir de cette relation d'ordre :
  – $v_1 \leq v_2 \Leftrightarrow (v_1 < v_2 \; or \; v_1 = v_2)$
  – $v_1 > v_2 \Leftrightarrow v_2 < v_1$
  – $v_1 \geq v_2 \Leftrightarrow (v_2 < v_1 \; or \; v_2 = v_1)$
– L'accès à un attribut du type : cet opérateur permet d'accéder à un attribut de type pré-défini, pour les types énumérés, on dipose des attributs suivants :
  – `first` : retourne le premier élément déclaré du type énuméré ;
  – `last` : retourne le dernier élément du type énuméré ;
  – `pred` : cet attribut implémente l'opérateur prédécesseur précédemment décrit ;
  – `succ` : cet attribut implémente l'opérateur successeur précédemment décrit.

DOCUMENT DE TRAVAIL

3.4 Les types définis par restriction de type

Les types définis par restriction de types définissent un sous ensemble de valeur d'un type. Ainsi tout élément d'un type défini par restriction est une valeur valide du type parent (type restreint).

Il n'est possible de restreindre que des types discrets, dans le cadre de **L*f*P**, on ne peut donc restreindre que le type integer ou un type énuméré, ou une restriction de l'un de ces types. L'ensemble des types définis par restriction à partir d'un type initial (integer ou un type énuméré) forme un arbre de sous-typage.

Les opérations associées aux types définis par restriction sont les mêmes que celles définies pour leurs types parents. En revanche, les bornes considérées pour le calcul du résultat sont celles du type restreint ; enfin le résultat peut être différent selon que le type restreint est circulaire ou non.

Opérations définies sur les types définis par restriction :
- successeur : retourne le successeur dans l'intervalle de définition du type de la valeur passée en argument ;
- prédécesseur : retourne le prédécesseur dans l'intervalle de définition du type de la valeur passée en argument ;
- addition : (uniquement pour les types définis par restriction sur les entiers) additionner a et b de même type définit par restrictions revient à prendre b fois le successeur de a.
- soustraction : (uniquement pour les types définis par restriction sur les entiers) soustraire b à a de même type définit par restriction revient à prendre b fois le prédécesseur de a.
- produit : (uniquement pour les types définis par restriction sur les entiers) le produit de a et b est défini par $\sum_{i=1}^{b} a$.
- quotient : (uniquement pour les types définis par restriction sur les entiers) le quotient q de a par b est défini par $a = q \times b$.
- affectation : L'affectation affecte à une variable d'un type énuméré, une valeur d'un type énuméré compatible et réalise la conversion de type si nécessaire. Les types des opérandes de l'affectation doivent avoir un parent commun : par exemple, on ne peut pas affecter une valeur d'un type énuméré à une variable de type "integer". L'opération de conversion peut lever une erreur lors de l'exécution si la valeur à convertir ne fait pas partie de l'intervalle de définition du type cible.
- relation d'ordre : La relation d'ordre sur les types définis par restriction est héritée du type père, mais son intervalle de définition est restreint aux valeurs contenues dans le nouveau type. La relation d'ordre pour un type circulaire est la même que pour le type équivalent non-circulaire. On doit toujours utiliser les opérateurs de relation avec des opérandes dont les types ont au moins un ancètre commun dans l'arbre de sous-typage.
- accès à un attribut de type : cet opérateur permet d'accéder à un attribut du type restreint :
  - `first` : retourne le premier élément déclaré du type restreint ;
  - `last` : retourne le dernier élément du type restreint ;
  - `pred` : retourne l'élément prédécesseur d'un élément donné pour le type restreint, cet attribut implémente l'opérateur prédécesseur précédemment décrit ;
  - `succ` : retourne l'élément suivant un élément donné pour le type restreint, cet attribut implémente l'opérateur successeur précédemment décrit.

On peut utiliser des opérations arithmétiques sur des opérandes de type différent, l'opérateur utilisé est celui associé au premier père commun (en remontant vers la racine) des deux types des opérandes. Ce type père commun est le type de la valeur retournée par l'opération.

Dans le cas où le type de la variable recevant le résultat n'est pas du même type que le résultat lui même, ce dernier est converti dans le type de la variable. Cette opération est suceptible de lever une erreur si le résultat n'est pas dans l'intervalle de valeurs défini pour le type cible. Ce cas ne peut se produire que si le type de la variable n'est pas un type parent dans l'arbre de sous-typage.

Dans le cas où il n'y a pas de type parent commun pour les opérandes, l'expression n'est pas correcte (cas d'un type restriction d'entier et d'un type restriction d'un type énuméré). Cette erreur peut être déterminée lors de l'analyse statique du modèle.

DOCUMENT DE TRAVAIL

Par exemple :

```
type t1 is circular range 0..256 of integer ;
type t2 is circular range -1..345 of integer ;
type t3 is circular range -1..300 of t2 ;
type t4 is circular range 300..345 of t2 ;

a : t1 ;
b : t2 ;
c : t3 ;
d : t4 ;
e : t4 ;
i : integer ;

-- les expressions suivantes sont toujours valides :
i := a + b ;
b := c + d ;
d := d + e ;

-- l'expression suivante peut générer une erreur à l'exécution
-- suivant les valeurs de a et c
-- a + c : de type integer. la valeur est convertie vers le type
-- t4 après calcul.
e := a + c ;
```

Les types restreints peuvent être circulaires, dans ce cas les opérations définies sur le type ne déclenchent jamais de débordement de capacité, en effet le successeur du dernier élément de l'intervalle est le premier élément, et le prédécesseur du premier élément est le dernier élément de l'intervalle. En revanche, les types définis par restriction ne fournissent aucune garantie lors des conversions : si une valeur convertie dans un type défini par restriction ne fait pas partie de l'intervalle de définition du type, une erreur est levée. Cette erreur ne peut être vérifiée que dynamiquement et doit être signalée lors de l'exécution.

3.5 Les types "port"

Les types "port" servent à définir les références vers les binders associés à une instance d'un composant **L*f*P** (classe ou média). Un port est donc une référence vers une instance de binder. Les types port sont définis en fonction du contenu du discriminant des messages envoyés dans le binder. Ils permettent de vérifier les types des valeurs contenues dans les discriminants des messages. Tous les composants qui référencent un même binder doivent le faire via des ports de même type.

Pour les types port, on définit trois opérations :
- l'envoi de message dans un port ;
- la lecture de message dans le port ;
- l'affectation.

Les sémantiques des opérations de lecture et d'envoi de message sont celles associées aux opérations réalisées sur le binder sous-jacent.

L'affectation permet de définir les binders d'un composant en fonction des binders d'un autre composant. Cela permet d'initialiser les valeurs des ports des médias et de les relier aux instances des classes.

Opérations sur les ports :
- Les ports sont des références sur les binders, ils sont utilisés en lieu et place des binders dans les opérations d'envoi et de réception de messages.
- L'affectation change le binder référencé par un port, et mets à jour l'attribut self[4] qui gère la référence sur le composant courant.

[4] Se reporter à la section 5.2.1 pour la description de cet attribut

DOCUMENT DE TRAVAIL

DOCUMENT DE TRAVAIL

3.6 Les références vers les composants du modèle

Les références vers les composants sont des types permettant de désigner explicitement une instance d'un composant **L*f*P**. Ces références sont typées dans le sens où une référence sur un composant ne peut désigner qu'une seule classe de composant.

Toute variable dont le type correspond au nom d'un composant **L*f*P** (classe ou média) est implicitement une référence vers une instance de ce composant (ce mécanisme est le même que celui de java). Si une référence n'est pas initialisée, elle est considérée comme invalid (valeur "null" par défaut). Créer une variable d'un type composant ne crée pas une instance de ce composant, l'opérateur d'instanciation dynamique présenté en section 5 est le seul à créer de nouvelles instances de composant.

Un type référence est associé implicitement à toute classe ou tout média défini dans le modèle. Une référence vers un composant contient les éléments suivants :
- la désignation complète du composant (localisation et numéro d'instance).
- les références des binders auxquels le composant est connecté (ports). Une

Les seules opérations réalisables sur un type référence en **L*f*P** sont l'affectation et la déréférenciation :
- l'affectation permet de dupliquer la référence, ou de récupérer la référence d'une instance nouvellement créée.
- la déréférenciation permet d'accéder aux références des binders du composant référencé.

REMARQUE : Les références des binders contenues dans une référence de composants **ne sont pas** mises à jour si les ports des composants sont modifiés après la création de sa référence. De même si le contenu d'une référence sur un composant est modifié "mantuellement" par l'utilisateur, le port du composant référencé n'est pas modifié.

3.7 Les types article (record)

Les types articles (record) permettent de rassembler en une seule variable un ensemble d'éléments de types distincts. Chaque élément distinct est appelé un "champ". N'importe quel type (y compris de taille variable tel qu'un ensemble) peut être un champ d'un type record.

On définit les opérations suivantes sur les types record :
- affectation : le contenu de chacun des champs du record à droite de l'affectation est recopié dans les champs correspondants de celui à gauche de l'affectation.
- déréférenciation : cette opération permet d'accéder à chacun des champs du record par son nom.

3.8 Les types tableau

Les types tableaux permettent de rassembler un nombre défini d'élément de même type. Tout type de donnée (y compris des types tableaux) est susceptible d'être un élément d'un type tableau. Le langage **L*f*P** ne supporte que les tableaux de taille fixe. En revanche, le nombre maximal de dimension n'est pas fixé. L'implémentation du générateur de code ou de l'outil de vérification peuvent le limiter en cas de besoin.

On définit les opérations suivantes sur les types tableaux :
- affectation : on définit l'affectation sur les tableaux par recopie de la totatlité du contenu du tableau à droite de l'affectation vers le tableau à gauche de l'affectation. Les deux tableaux doivent être des instances du même type.
- accès à une case : permet d'accéder en lecture ou en écriture à un élément contenu dans le tableau en fonction de sa position (son indice).
- accès à toutes les cases : permet d'accéder en écriture à la totalité des cases d'un tableau (surtout utile pour l'initialisation des variables tableau).

4 Sémantique des éléments du diagramme d'architecture

L'objectif de cette section est de présenter la sémantique des éléments disponibles dans le diagramme d'architecture, c'est à dire :
- classes ;
- média ;
- binders ;
- arcs de liaison.

4.1 Sémantique statique des composants L*f*P

Les composants **L*f*P** sont les classes et les médias. Les classes **L*f*P** représentent les composants de l'application. Les médias **L*f*P** représentent les composants de communication entre les classes.

Chaque instance de composant assure les propriétés suivantes :
- exécution "active" : chaque instance de classe ou de média dispose d'un support d'exécution dédié (peu différent d'une thread) ;
- exécution séquentielle : aucun parallélisme n'est supporté à l'intérieur d'une classe ou d'un média.

4.2 Sémantique des binders

Les binders sont les buffers assurant la transmission des messsages entre les classes et les médias.

Les binders sont définis par les attributs suivants :
- *multiplicity* ;
- *capacity* ;
- *ordering* ;
- *binding*.

Les opérations réalisables sur les binders sont les suivantes :
- ajout de message ;
- lecture de message.

Il existe deux types de binders (synchrones et asynchrones) suivant le comportement désiré lors de l'ajout de message. Les différences entre ces deux types de binders seront vues lors de la présentation des opérations d'ajout et de lecture de messages.

La figure 3 présente un binder reliant une classe `C1` à un média `M1` et illustre les attributs des binders. La suite de cette section présente les atributs statiques puis les opérations dynamiques associées aux binders.

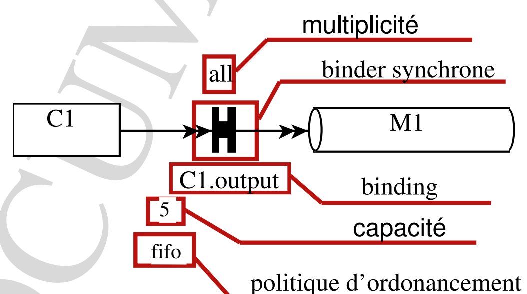


FIG. 3 – exemple de binder dans un diagramme d'architecture

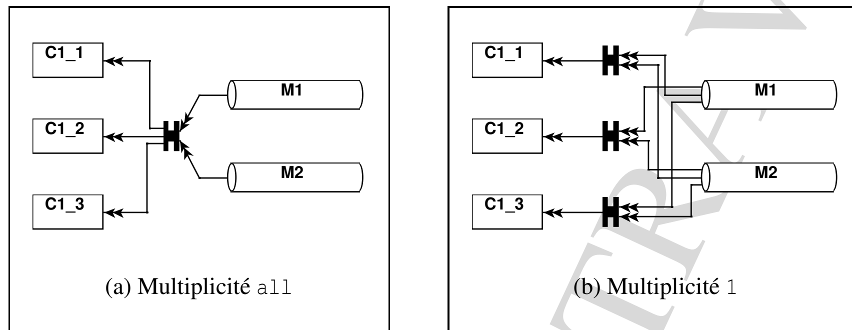
4.2.1 Multiplicité des binders

La multiplicité du binder (attribut *multiplicity*) définit les instances de classes auxquelles le binder est connecté. Il y a deux valeurs possibles :

DOCUMENT DE TRAVAIL

– `1` : le binder n'est accessible qu'à une seule instance de la classe qui lui est associée. Il est instancié dynamiquement lors de la création d'une nouvelle instance de la classe.
– `all` : le binder est partagé entre toutes les instances de la classe à laquelle il est relié ; il est instancié statiquement lors de l'initialisation du modèle.

La multiplicité n'a aucune influence sur la manière dont le binder est connecté au(x) média(s). Les différences entre multiplicité `all` et `1` sont illustrées sur la figure 4 qui représente toutes les communications possibles entre trois instances d'une classes `C1` et deux médias `M1` et `M2`.



(a) Multiplicité `all` (b) Multiplicité `1`

FIG. 4 – Différences sémantiques entre les multiplicité `all` et `1`

Dans le cas d'un binder de multiplicité `all` (figure 4(a)), une instance de buffer est partagée entre toutes les instances de la classe C1 (C1_1, C1_2 et C1_3). De même, toutes les instances de média reliées à ce binder sur le diagramme d'architecture peuvent accéder au binder.

Dans le cas d'un binder de multiplicité `1` (figure 4(b)), chaque instance de la classe C1 dispose de sa propre instance de binder. Chaque instance de média connectée au binder sur le diagramme d'architecture peut accéder à toutes les instances de ce binder.

4.2.2 Capacité des binders

La capacité du binder définit le nombre maximum de messages que peut contenir le binder. Les binders sont des buffers bi-directionnels : ils sont en fait constitués de deux files de messages de même capacité : une file contient les messages émis depuis la classe vers les médias, l'autre contient les messages en provenance des médias à destination de la classe. Sur l'exemple de la figure 3, le binder relie le port "output" de la classe `C1` au port "input" du média `M1`

4.2.3 Ordonancement des messages

La politique d'ordonancement du binder est spécifiée par l'attribut *ordering* qui définit la politique d'ordonancement des messages. Deux politiques sont considérées par défaut :
– `fifo` : les messsages sont fournis par le binder dans l'ordre de leur arrivée ;
– `bag` : les messages sont fournis par le binder dans un ordre quelconque ;

4.2.4 Liaison entre les binders et les composants

Lors de la création d'une instance de classe ses ports sont automatiquement initialisés à partir des informations présentes sur le diagramme d'architecture et des attributs `binding` et `multiplicity`. L'attribut binding permet de définir quels ports de la classe doivent être initialisés : seuls les ports reliés à un binder sur le diagramme d'architecture sont initialisés. Cela signifie que si des variables de type "`port`" sont définie dans la classe, qu'elles n'apparaissent pas dans l'attribut `binding` d'un binder, et qu'aucune valeur initiale n'est précisée dans de leur déclaration, elles ne seront pas initialisées.

L'attribut multiplicity détermine de quelle manière le port est initialisé :

DOCUMENT DE TRAVAIL

- si la multiplicité est `1`, une nouvelle instance de binder est créée ;
- si la multiplicité est `all`, le port est initialisé à partir de la référence d'une instance de binder partagée entre toutes les instances de la classe.

La liaison entre les binders et les médias est laissée à la responsabilité de l'utilisateur. Ce dernier dispose de trois méthodes :

- en fournissant une valeur initiale pour les attributs de type `port` lors de l'instanciation du média ou de la déclaration d'une instance statique ;
- en utilisant des messages de contrôle (msgs envoyés par la classe à destination du média) ;
- en utilisant les discriminants des messages que le média doit router.

Dans tous les cas ces liens doivent respecter les contraintes statiques définies sur le diagramme d'architecture : pour permettre l'envoi de message, un port ne doit pas désigner un binder d'un composant auquel le média n'est pas relié sur le diagramme d'architecture.

4.2.5 Ajout de message dans un binder

L'opération d'ajout de message dans un binder peut être réalisée depuis un média ou une classe. Seule la structure du message ajouté diffère dans ces deux cas. Cette opération transfère un message dans le buffer de manière atomique pour l'instance de composant et le binder.

Cas d'un binder synchrone

Dans le cas d'un binder synchrone, l'opération d'ajout de message ne peut être réalisée que lorsqu'un emplacement est disponible dans le binder. Le composant qui demande cette opération est bloqué jusqu'à ce qu'elle puisse être réalisée.

Cas d'un binder asynchrone

Dans le cas d'un binder asynchrone, l'opération d'ajout de message est toujours possible, mais lorsque le binder est plein, les messages ajoutés sont simplement "perdus" : ils ne sont plus pris en compte dans la suite de l'exécution. Cette opération n'est jamais bloquante.

4.2.6 Lecture d'un message dans un binder

La lecture d'un message dans un binder revient à prendre un message fourni par le binder. Dans le cas d'un binder `fifo`, il s'agit du message le plus ancien se trouvant dans le binder. Dans le cas d'un binder `bag`, il s'agit d'un message choisi aléatoirement dans le binder.

L'opération de lecture sur un binder est bloquante tant qu'un message n'a pas été consommé par le composant. Jusqu'à la consommation effective du message, l'opération de lecture peut être abandonnée. Cette propriété est nécessaire dans le cas de lectures simultanées sur plusieurs binders : seul un message sera effectivement consommé, mais plusieurs peuvent être lus. Une opération de lecture peut se terminer de deux manières différentes (cf. figure 5) :

- le composant annule la lecture (il a trouvé un message valide dans un autre binder) ;
- le composant consomme le message .

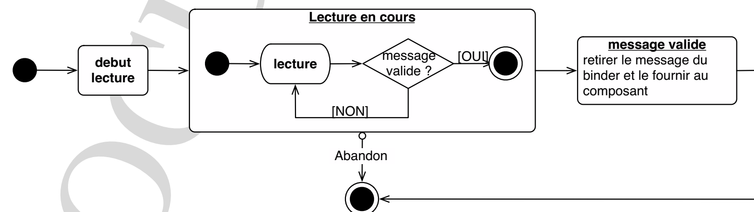


FIG. 5 – Déroulement d'une lecture sur un binder

Un message ne peut être consommé que lorsqu'il est valide pour le composant qui initie la lecture (pour la définition d'un message valide, se reporter à la section 5). Le message est alors retiré du binder, et son emplacement est à nouveau disponible.

DOCUMENT DE TRAVAIL

DOCUMENT DE TRAVAIL

Dans le cas d'un binder `bag`, si le premier message proposé n'est pas valide pour le composant, les autres messages du binder sont essayés dans un ordre aléatoire.

Dans le cas où le composant abandonne la lecture, l'état du binder n'est pas modifié, comme l'exécution d'un composant qui demande un message est suspendue jusqu'à réception d'un message valide, cela signifie qu'il en a trouvé un dans un autre binder.

4.3 Sémantique des arcs

Les arcs des diagrammes d'architecture n'ont pas d'équivalent "logiciel", ils expriment simplement la possibilité d'existence d'une communication entre deux entités du diagramme. Ce sont ces liens qui permettent le tranfert de données entre les composants du diagramme d'architecture.

La durée de vie effective du lien dépend de l'instanciation des composants qu'il relie : un lien ne peut effectivement relier deux composants que s'ils ont été instanciés tous les deux et que les deux instances sont encore valides.

Si aucun lien ne relie deux entités du diagramme d'architecture, ces deux entités ne peuvent *jamais* communiquer directement (échanger des messages via un binder).

Les arcs sont orientés dans le sens du tranfert de données supporté par ce lien. On définit deux orientations possibles pour les arcs du diagramme d'architecture :

- les arcs unidirectionnels qui ne permettent le passage des données que dans un seul sens ;
- les arcs bidirectionnels qui permettent le passage des données dans les deux sens.

Si on se reporte à nouveau à la figure 3, on peut voir que les arcs sont orientés respectivement depuis la classe `C1` vers le binder, et depuis le binder vers le média `M1`. Cela signifie que la classe ne peut qu'émettre des messsages vers le binder. De même, le média ne peut que lire des données dans ce dernier.

5 Sémantique des diagrammes de comportement

Cette section présente la sémantique des diagrammes de comportement **L*f*P**. Ces diagrammes permettent de représenter de manière hiérarchique le comportement des composants du modèle. On doit distinguer les classes des médias.

Dans le cas des classes, le diagramme de comportement est séparé en deux niveaux sémantiques. Le premier niveau sémantique est appelé *diagramme principal* de la classe. Il permet de définir les actions exécutées par la classe, et l'enchainement des méthodes : une classe lfp est un fil d'exécution qui peut se mettre en attente d'activation de méthode (attendre qu'une de ses méthodes soit appelée). Un diagramme de comportement principal peut être défini sur plusieurs niveaux hiérarchique en utilisant des triggers et des sous-diagrammes.

Le deuxième niveau sémantique d'une classe correspond au corps de ses méthodes. Ces diagrammes de comportement sont appelés *diagrammes de méthode*. Ils décrivent les actions associées à chaque méthode de la classe, ce qui introduit quelques contraintes (étudiées en 5.3.2) par rapport au diagramme principal de la classe.

Enfin, on doit distinguer les sous-diagrammes et les triggers qui permettent d'introduire respectivement un aspect hiérarchique pour améliorer la lisibilité des diagrammes et un moyen de réutiliser un sous-diagramme à l'intention du diagramme de comportement d'un composant.

Cette section commence par quelques définitions, puis elle présente les éléments communs des diagrammes de comportement. Ensuite, elle présente les éléments spécifiques aux diagrammes de comportement d'une classe, puis ceux des diagrammes de comportement des médias.

5.1 définitions

Transition franchissable : une transition est dite franchissable si toutes les préconditions nécessaires à son exécution sont remplies. Les conditions de franchissabilité de chaque

type de transition sont définies dans les sections les décrivant.

Garde d'une transition : la garde d'une transition est une condition booléenne déterminant si la transition est franchissable ou non. Une garde "vide" (non spécifiée) est une garde qui s'évalue toujours à "true".

Etat initial : un état initial d'un diagramme de comportement est l'état dans lequel il se trouve lors de son activation.

Etat final : un état final d'un diagramme de comportement marque la fin de l'exécution du diagramme. Toutes les ressources qui sont liées à ce diagramme sont automatiquement libérées.

Transition de sortie : une transition de sortie d'un état est une transition potentiellement exécutable depuis l'état courant.

Etat de sortie : l'état de sortie d'une transition est l'état dans lequel se trouve l'automate après exécution de la transition. Une transition ne peut avoir qu'un seul état de sortie.

transition hiérarchique : une transition hiérarchique est une transition pouvant contenir un sous-diagramme. Si une transition hiérarchique ne contient pas de sous-diagramme, il s'agit d'une référence vers un trigger ou une méthode. La référence est résolue à partir du nom de la transition. Le trigger ou la méthode doit avoir été déclaré(e) dans l'attribut déclaration du diagramme de comportement principal du composant. La définition du trigger ou de la méthode est assurée par une transition hiérarchique portant son nom et contenant le diagramme de comportement qui lui est associé.

transition de communication : une transition de communication est une transition permettant à un composant d'accéder ou de modifier le contenu d'un binder.

transition terminale : une transition terminale est une transition qui ne peut pas contenir de sous-diagramme, et qui n'est pas une transition de communication. Elles permettent de spécifier textuellement un ensemble de traitements locaux.

Contexte d'exécution : Le contexte d'exécution d'un diagramme est l'ensemble des déclarations visibles lors de l'exécution du diagramme.

Sous-diagramme : un sous-diagramme est un diagramme de comportement défini dans un autre diagramme de comportement. Son contexte d'exécution est constitué du contexte d'exécution du diagramme dans lequel il est inséré, augmenté de l'ensemble des déclarations locales.

Trigger : un trigger est un diagramme de comportement particulier pouvant être référencé depuis n'importe quel diagramme de comportement du composant dans lequel il est déclaré.

Méthode : une méthode est un sous-diagramme d'une classe définissant les actions entreprises par la classe lors de la réception d'un message particulier. Une méthode peut être référencée dans tout le diagramme principal de la classe.

Attribut d'un composant : un attribut d'un composant est une variable déclarée dans l'attribut `declaration` du diagramme de plus haut niveau du composant.

5.2 Éléments communs

5.2.1 Définition des composants L*f*P

Le mot composant désigne indifféremment une classe ou un média, ils partagent les caractéristiques communes suivantes :

Attributs des composants : Les attributs sont les variables définies dans le diagramme de plus haut niveau hiérarchique du composant. Leur visibilité depuis l'extérieur du composant est définie comme suit :
- les attributs de type port sont inclus par recopie dans toutes les références vers le composant, si le type du port qui leur est associé est défini dans le diagramme d'architecture ;

DOCUMENT DE TRAVAIL

- les attributs dont le type est défini dans le composant sont toujours invisibles depuis l'extérieur ;
- les attributs dont le type est déclaré dans le diagramme d'architecture peuvent recevoir une valeur initiale lors de l'instanciation de ce composant.

Attribut `SELF` : Tous les composants **L*f*P** possèdent un attribut appelé "`SELF`" défini par le langage. L'attribut "`SELF` d'une instance de composant contient la référence vers cette instance. Ce nom est un mot réservé du langag **L*f*P** et ne peut être utilisé comme identificateur.

5.2.2 Sémantique générale des états

Les états des diagrammes de comportement définissent des alternatives : ils sont suivis d'une ou plusieurs transition. Lorsqu'un composant atteint un état du diagramme, il doit choisir la prochaine transition à exécuter.

Lorsqu'un état est suivi de plusieurs transitions, les arcs peuvent porter une priorité définie sous la forme d'un entier. Si un arc de sortie d'un état porte une priorité, tous les arcs doivent porter une priorité.

La franchissabilité des transitions de sortie d'un état est testée en suivant l'ordre croissant des priorités. Si deux arcs ont la même priorité, l'ordre d'évaluation des transitions de sortie qui leur est associé n'est pas spécifié.

On définit plusieurs types d'états en fonction de la nature des transitions de sortie. Le comportement des états suivis de transitions de communications sont traités dans les sections spécifiques dédiées aux classes et aux médias car leur comportement dépend de la nature du composant auquel elles appartiennent.

En revanche, le comportement des états suivis uniquement de transitions terminales ou de sous-diagrammes ou de triggers est identiques que le composant soit un média ou une classe :

1. déterminer la prochaine transition à exécuter ;
2. exécuter entièrement la transition ;
3. passer à l'état de sortie de la transition exécutée.

La prochaine transition à exécuter est déterminée en testant la franchissabilité de chacune des transitions de sortie de l'état courant dans l'ordre défini par les priorités associées aux arcs. La première transition franchissable doit alors être exécutée. Dans le cas général, la franchissabiligé d'une transition est définie par l'évaluation de sa garde :

- Si la garde est évaluée à `true`, la transition est franchissable ;
- si la garde est évaluée à `false`, la transition n'est pas franchissable.

Si aucune transition franchissable n'est trouvée en sortie de l'état courant, une erreur doit être levée.

L'exécution de la transition signifie l'exécution de toutes les instructions qui lui sont associées, ou du sous-diagramme correspondant dans le cas d'une transition hiérarchique.

Enfin, le passage à l'état de sortie de la transition termine le franchissement de la transition et le cycle recommence en fonction du nouvel état courant du composant.

5.2.3 Sémantique des sous-diagrammes

Un sous-diagramme est la description d'une transition sous la forme d'un diagramme de comportement, il ne peut être utilisé qu'une fois dans le modèle, et ne peut pas être référencé. On peut y trouver tous les éléments disponibles dans le diagramme dans lequel il est inséré (y compris les déclarations de types si applicable). Les opérations réalisables dans un sous-diagramme dépendent donc du contexte dans lequel celui-ci est défini.

Il s'agit principalement d'une facilité d'écriture et de structuration permettant de regrouper une partie du diagramme de comportement. D'un point de vue sémantique, cette construction se rapproche du "bloc d'instruction" défini dans les langages de programmation.

L'exécution du sous-diagramme peut être conditionné par une garde ; celle-ci est évaluée dans le contexte du diagramme où le sous-diagramme est défini. La transition définissant le sous-diagramme est franchissable si et seulement si la garde s'évalue à true.

DOCUMENT DE TRAVAIL

DOCUMENT DE TRAVAIL

Le contexte d'exécution d'un sous-diagramme est constitué du contexte d'exécution du diagramme de comportement parent (diagramme dans lequel le sous-diagramme est défini), augmenté de l'ensemble des déclarations faites dans le sous-diagramme lui même.

5.2.4 Sémantique des triggers

Les triggers sont des sous-diagrammes réutilisables à plusieurs endroit du diagramme dans lequel ils sont définis. Le contexte d'exécution d'un trigger est défini par :
- les déclarations du diagramme d'architecture ;
- les déclarations du premier niveau hiérarchique du diagramme principal du composant ;
- les déclarations locales (déclarations faites dans le trigger lui même).

Propriétés des triggers :
- Les triggers doivent être déclarés dans les déclarations du diagramme principal d'un composant ;
- le contexte d'exécution d'un trigger ne dépend pas de l'endroit où son diagramme de comportement est déclaré. Il s'agit toujours du contexte associé au diagramme principal du composant.
- On ne peut pas définir de trigger local à un sous diagramme.
- Un trigger peut être utilisé dans tout le diagramme de comportement du composant où il est défini.
- Un trigger défini dans une classe ne peut contenir que des opérations autorisées dans le corps des méthodes (cf. 5.3.2), il peut en effet être appelé dans tout le diagramme de comportement de la classe, y compris dans le corps des méthodes.

La définition d'un trigger est faite par une transition hiérarchique contenant le diagramme de comportement du trigger. Cette définition peut apparaitre n'importe où dans le diagramme de comportement du composant. La définition du trigger est associée à sa déclaration par le nom de la transition hiérarchique.

Pour référencer un trigger, on utilise une transition hiérarchique qui ne contient pas de sous-diagramme et qui porte le nom du trigger.

5.2.5 Opérations sur les sémaphores

Les opérations sur les sémaphores peuvent être réalisées depuis n'importe où dans les diagrammes de comportement des composants du moment que la déclaration du sémaphore utilisée est visible. Ces transitions couvrent les opérateurs définis dans la description du type `semaphore` à la section 3.

L'initialisation ne peut être réalisée que lors de la déclaration du sémaphore. Les opérations `P` et `V` sont disponibles dans n'importe quel diagramme ou sous-diagramme de comportement.

5.2.6 Sémantique des transitions simples

Les transitions terminales portent des instructions textuelles permettant de spécifier des traitements locaux (pas d'opérations de communication). Leur contexte d'exécution est celui défini pour le diagramme qui contient la transition.

while : cette construction correspond à la boucle while classique des langages de programmation : le corps de la boucle est exécuté tant que l'expresion booléenne associée s'évalue à true.

for : cette construction correspond à la boucle for des langages de programmation ("à la Ada"). Une variable de boucle définie dans l'instruction elle même parcours un intervalle discret de valeurs, le corps de la boucle est exécuté pour toutes les valeurs de l'intervalle, ce dernier est parcouru depuis la borne inférieure vers la borne supérieure. Le contexte d'exécution d'une boucle `for` est augmenté de la déclaration de la variable de boucle ; cette variable n'est accessible qu'en lecture, et sa portée est réduite au corps de la boucle.

alternative (if ... then ... else) : Cette construction a la sémantique classique définie dans les langages de programmation. Si l'expression booléenne associée s'évalue à true, le bloc d'instruction définie après le `then` est exécuté, sinon, c'est le bloc d'instruction défini après le `else` qui est exécuté.

affectation : Les particularités de l'affectation associée à chaque type sont spécifiées dans leur présentation (se reporter dans la section 3 à la description du type de donnée utilisé). La valeur à gauche de l'affectation doit toujours être une variable, ou un paramètre en mode "inout" ou "out".

La séquence d'exécution d'une transition est la suivante :
- exécution séquentielle des instructions portées par la transition ;
- saut à l'état de sortie de la transition.

5.2.7 Instanciation dynamique de composants

Le langage **L*f*P** supporte la création dynamique d'instances de composants. Cette opération est réalisable dans tout diagramme de comportement **L*f*P**. Lorsqu'un composant est instancié dynamiquement, on doit créer l'ensemble des éléments nécessaires à son exécution. Dans le cas d'un média, il s'agit principalement du support d'exécution du média. Dans le cas d'une classe, il faut également initialiser les ports de la classe présents sur le diagramme d'architecture, et donc instancier l'ensemble des binders qui lui sont connectés. On peut se reporter au point 4.2.4 qui présente les règles d'instanciation des binders lors de l'instanciation des composants **L*f*P**.

L'instruction d'instanciation dynamique permet de spécifier des valeurs initiales pour les attributs de la nouvelle instance de composant. Seuls les attributs dont le type est visible dans le contexte d'exécution de l'instruction peuvent être initialisés. Si des valeurs initiales sont spécifiées pour des attributs du composant lors de l'instanciation dynamique, elles écrasent les valeurs par défaut spécifiées lors de la déclaration des attributs.

5.3 Diagramme de comportement d'une classe

Le diagramme de comportement de la classe définit son comportement. Pour les classes, on distingue deux niveaux sémantiques pour le diagramme de comportement :
- le diagramme principal de la classe qui définit son contrat d'exécution et l'ensemble des services qu'elle offre ;
- les diagrammes des méthodes qui définissent le comportement des méthodes de la classe.

Le diagramme de comportement d'une classe peut contenir les éléments suivants :
- déclaration de méthodes (uniquement dans les déclarations du premier niveau hiérarchique du diagramme principal) ;
- méthodes (uniquement dans le diagramme principal) ;
- déclarations de types, variables et constantes ;
- état initial ;
- état final
- transition ;
- sous-diagrammes ;
- états ;
- états d'attente d'activation de méthode (uniquement dans le diagramme principal) ;
- instanciation dynamique de composants.

Le diagramme principal d'une classe est donc constitué du premier diagramme de la classe (portant les déclarations de ses attributs), et de l'ensemble des sous-diagrammes qui y sont définis. Les "corps" des méthodes, ainsi que ceux des triggers définissent le deuxième niveau sémantique du diagramme de comportement et ne font donc pas partie du diagramme principal.

DOCUMENT DE TRAVAIL

5.3.1 Sémantique des transitions de communication

La séquence d'exécution d'une transition de communication dans le diagramme de comportement d'une classe est la suivante :
- évaluation de la garde ;
- exécution des instructions de communications de la transition ;
- saut à l'état de sortie de la transition.

Une transition de communication peut porter une ou plusieurs instruction de communication qui seront exécutées séquentiellement. Il existe trois types de transitions de communication pour les diagrammes de comportement des classes :
- Les transitions de réception qui sont réservées pour préciser le port d'activation d'une méthode et la réception de message ;
- Les transitions d'envois qui permettent l'appel de procédures asynchrones et l'envoi de message ;
- les transition d'envoi et réception qui sont utilisées pour les appels de fonction et les appels de procédures synchrones.

Les instructions portées par les transitions de communication seront présentées par la suite.

5.3.2 Sémantique des méthodes

Une méthode définit le traitement à effectuer lors de la réception du message d'activation correspondant. Les opérations disponibles depuis le diagramme d'une méthode sont les suivantes :
- transitions simples ;
- appels de méthodes avec ou sans valeur de retour ;
- opérations sur les sémaphores ;
- déclarations de types valables sur toute la méthode (diagramme principal de méthode).
- définition de sous-diagrammes ;
- appel et définition de triggers ;
- instanciation dynamique de composants.

Dans une méthode, on ne peut pas insérer :
- des déclarations de méthodes ;
- des déclarations de triggers ;
- des états d'attente d'activation de méthode.

Les déclarations des triggers et des méthodes doivent être faites dans le premier diagramme principal de la classe, elles ne peuvent donc être faite dans le diagramme d'une méthode. De plus, une instance de classe ne peut pas avoir deux méthodes activées simultanément. C'est pour cette raison qu'il est impossible d'avoir un état d'attente d'activation de méthode dans le corps d'une méthode.

La première transition d'une méthode doit être une transition de réception de message. Elle sert à préciser le binder sur lequel est attendu le message d'activation.

Les paramètres formels de la méthode et leur mode de passage sont définis dans le diagramme principal de la classe ; ils sont rappelés par l'attribut "déclaration" du diagramme principal de la méthode. Les paramètres effectifs de la méthode sont associés aux paramètres formels lors de l'activation. On distingue trois mode de passage pour les paramètres d'une méhode :
- `in` : la valeur passée en paramètre effectif ne peut être que lue dans la méthode. Au niveau de l'appel de méthode, on peut passer un paramètre effectif qui soit une constante, ou une expression.
- `out` : la valeur passée en paramètre ne peut être qu'écrite dans la méthode. Au niveau de l'appel de méthode, on doit passer une variable capable de stocker la nouvelle valeur.
- `inout` : la valeur passée en paramètre peut être lue et écrite dans la méthode. Au niveau de l'appel de méthode, on doit passer une variable capable de stocker la nouvelle valeur.

La sémantique des modes `inout` et `out` est de type copy / retore : les valeurs ne sont mises à jour qu'à la fin de l'exécution de la méthode appelée.

On distingue plusieurs types de méthodes selon le schéma d'interaction désiré :

DOCUMENT DE TRAVAIL

– Les fonctions qui retournent une valeur. Tous les paramètres d'une fonction doivent être en mode `in`. Les appels de fonction sont bloquants tant que la valeur de retour n'a pas été reçue.
– Les procédures synchrones : il s'agit soit de procédures déclarées explicitement comme `synchronous`, soit des procédures ayant au moins un paramètre en mode `out` ou `inout`. L'appel est bloquant tant que les paramètres effectifs n'ont pas été mis à jour.
– les procédures asynchrones qui sont toutes les procédures dont les paramètres sont tous en mode "in" et qui ne sont pas explicitement déclarées `synchronous`. L'appel de procédures asynchrones est non bloquant : le composant appelé continue son exécution dès que le message d'activation a été écrit dans le binder d'envoi.

5.3.3 Sémantique des états des classes

Dans le diagramme de comportement d'une classe, on doit distinguer deux types d'états :
– les états dont **aucune** transition de sortie n'est une méthode ;
– les états dont **toutes** les transitions de sortie sont des méthodes.

Dans le premier cas, on applique la sémantique générale des états, la franchissabilité des transitions de sortie est directement déterminée par l'évaluation de leur garde. Les transitions de sortie de l'état peuvent être :
– des transitions simples ;
– des sous-diagrammes ;
– des triggers ;
– des transitions de communication ;
– des opérations sur un sémaphores.

Le deuxième cas n'est valide que dans le diagramme principal d'une classe ou dans un de ses sous-diagrammes, elle définit le . ***Elle n'est pas valide dans le diagramme de comportement d'une méthode***.

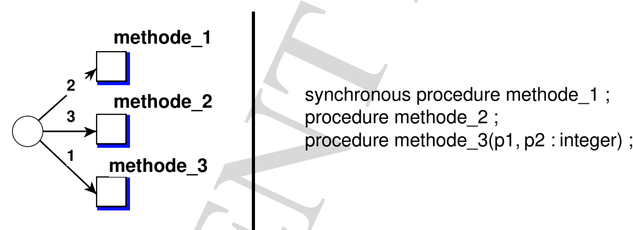


FIG. 6 – Etat de choix de la prochaine méthode à exécuter

Les opérations réalisées lors du choix de la prochaine méthode à exécuter sont :
– évaluation des gardes des transitions de sortie ;
– lecture des binders correspondant aux transitions franchissables ;
– si aucun message valide n'est reçu (absence de message ou messages invalides pour l'état courant), mise en attente d'un message valide ;
– si un message valide est reçu, il est consommé ;
– si plusieurs messages valides sont disponibles, on utilise les priorités portées par les arcs de sortie de l'état pour déterminer le message qui sera effectivement consommé ;
– exécution de la méthode correspondant au message consommé.

Si des priorités spécifiées sur les arcs de sortie de l'état, les binders sont lus dans l'ordre spécifié par les priorités. Dans le cas où aucune priorité n'est précisée, l'ordre de lecture des messages n'est pas spécifié, et dépend de l'implémentation.

5.3.4 Appel de procédure synchrone

Lors d'un appel à une procédure synchrone, la classe appelante suspend son exécution pendant toute la durée de l'appel. Cette opération doit être portée par une transition de communication de type "envoi et réception".

DOCUMENT DE TRAVAIL

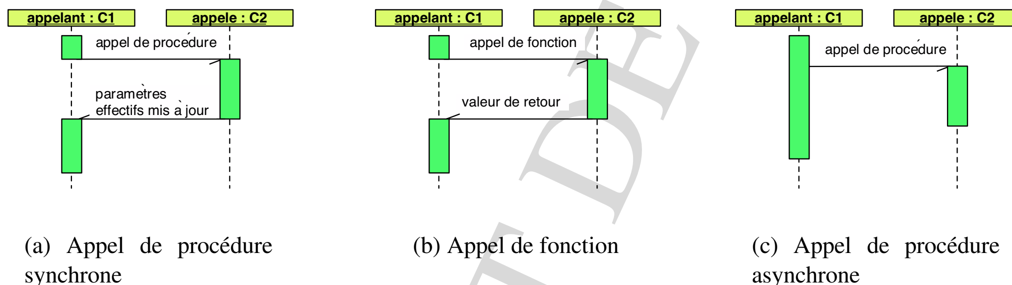
Un appel de procédure synchrone contient les éléments suivants :
- le binder cible ;
- Un discriminant ;
- éventuellement le nom du type du composant appelé ;
- le nom de la méthode appelée ;
- les paramètres effectifs de l'appel.

Le binder cible est le binder dans lequel le message sera ajouté pour être ensuite routé par un média. Le discriminant est un ensemble de paramètres passés au média pour assurer le routage du message. Les paramètres du discriminant sont systématiquement en mode “in”.

Le type du composant appelé permet de faire la vérification statique de l'existence d'une méthode ayant le prototype demandé par l'appel sur le type de composant cible. Dynamiquement, il peut être utilisé pour vérifier que le composant recevant le message est bien du type attendu.

Le nom de la méthode appelée permet de faire l'aiguillage vers la méthode à exécuter. **L***f***P** ne proposant pas de polymorphisme, l'aiguillage vers la méthode à exécuter est donc fait uniquement en fonction de ce nom.

Les paramètres effectifs sont les valeurs des paramètres passés à la méthode. Ils doivent respecter les contraintes explicitées précédemment en fonction du mode de passage du paramètre.



(a) Appel de procédure synchrone

(b) Appel de fonction

(c) Appel de procédure asynchrone

FIG. 7 – Les trois types d'appels de méthode

Un appel de procédure synchrone effectue les opérations suivantes :
- construction d'un message à partir des éléments précédemment cités ;
- envoi du message sur le binder cible (opération d'ajout de message sur le binder) ; la file de message est celle à destination des médias.
- lecture bloquante sur l'autre file de message du même binder pour récupérer les valeurs des paramètres mis à jours lors de l'exécution de la méthode distante.
- mise à jour des paramètres effectifs passés en mode `inout` et `out`.

Le composant suspend donc son exécution pendant l'attente du message de retour (cf. figure 7(a)). L'exécution de l'appel de procédure est terminé lorsque les paramètres effectifs ont été mis à jour en fonction des valeurs reçues dans les message de retour.

5.3.5 Appel de fonction

En plus des éléments spécifiés pour les appels de procédures synchrones, un appel de fonction doit spécifier la variable dans laquelle sera stockée la valeur de retour de la fonction. Cette opération doit également se trouver sur une transition d'envoi et réception.

Le principe d'un appel de fonction est proche de celui d'un appel de procédure synchrone. Seule la manière dont est transmise la valeur de retour de la méthode :dans le cas d'un appel de fonction, la valeur de retour est écrite dans une variable indépendamment des paramètres effectifs. Les paramètres effectifs d'une fonction sont toujours en mode `in`, ils ne sont donc pas modifiés.

Un appel de fonction effectue les opérations suivantes :

DOCUMENT DE TRAVAIL

- envoi de message dans la file de sortie ;
- réception du message de retour dans la file d'entrée ;
- si le premier message lu dans le binder après l'envoi du message d'appel n'est pas la valeur de retour de la fonction, le comportement est indéterminé (modèle erroné) ;
- stockage de la valeur de retour contenue dans le message de retour dans la variable prévue à cet effet ;
- saut à l'état suivant du diagramme de comportement.

Cette opération peut être protégée par une garde, qui porte sur la valeur des variables de la classe **avant** l'appel.

5.3.6 Appel de procédure asynchrone

Un appel de procédure asynchrone contient les mêmes éléments qu'un appel de procédure synchrone. C'est un mécanisme relativement proche de l'envoi de message, il est illustré sur la figure 7(c). Cette opération ne peut se trouver que sur une transition d'envoi.

Les opérations suivantes sont réalisées lors de l'appel de procécure asynchrone :
- création du message à partir du discriminant, des paramètres de la procédure, de son nom et du nom du type du composant appelé s'il est fourni ;
- envoi du message sur le binder cible ;
- saut à l'état suivant la transition d'appel de procédure.

L'exécution du composant continue dès que l'opération d'ajout du message sur le binder cible s'est terminée (cf. figure 7(c)). Aucun message de retour n'est lu.

5.3.7 Envoi et réception de messages entre classes

Les classes peuvent également communiquer par envoi de message (sans activation de méthode). Les opérations liées aux messages sont autorisées dans tout le diagramme de comportement des classes (diagramme principal, diagrammes de classes trigger, sous-diagrammes).

Structure des messages

Un message **L*f*P** est constitué d'un ensemble de valeurs typées.

Typage des messages

Le type des messages est déterminé par l'ensemble des types des valeurs qu'il contient.

Lecture des messages

Une opération de lecture de message est constituée :
- du nom du binder dans lequel le message doit être lu ;
- d'un ensemble de variable qui recevront les valeurs transportées par le message

La lecture des messages dans les classes correspond à affecter le contenu du message contenu dans le binder. Une lecture est correctement typée si l'ensemble des types des variables utilisées pour la lecture est le même que celui du message.

L'opération de lecture est bloquante si le binder dans lequel le message est lu est synchrone et ne contient pas de message correctement typé.

Lors d'une lecture dans un binder asynchrone, si aucun message ne peut être lu, le contenu des variables utilisées pour la lecture n'est pas modifié.

Une opération de lecture de message ne peut se trouver que sur une transition de réception.

Envoi de message

L'envoi de message correspond à l'écriture du contenu du message dans le binder. Cette opération ne peut être portée que par une transition d'envoi. Cette opération nécessite les éléments suivants :
- le port dans lequel le message doit être écrit ;
- un discriminant optionnel permet de spécifier des paramètres de routage utilisés par le média ;
- un ensemble de valeurs correspondant au contenu du message.

DOCUMENT DE TRAVAIL

Cette opération est bloquante si le binder est synchrone et qu'il n'y a pas d'emplacement libre dans le binder ; si le binder est asynchrone et qu'il n'y a pas d'emplacement libre, le message est perdu ; s'il y a de la place dans le binder, le message est envoyé.

Traitement du message par les médias

Les messages envoyés par les classes sont traités par les médias de la même manière que les activations de méthode. Le média ne peut pas déterminer la nature du message qu'il transporte.

Un envoi de message n'est bloquant que pendant l'écriture du message dans le binder cible. Le composant envoyant le message reprend ensuite son exécution.

5.3.8 Envoi de messages de contrôle aux médias

Les messages de contrôle sont des messages envoyés par la classe pour un média auquel il est relié. Il s'agit en fait de messages réduits à un discriminant. Ils ne contienent pas de partie à transmettre, et sont donc simplement traités par les médias.

5.4 Diagramme de comportement d'un média

Le diagramme de comportement d'un média est relativement différent de celui d'une classe : il ne peut contenir de méthode. La sémantique des médias est en effet de plus "bas niveau" : ils manimulent directement les message. Comme ces derniers ne sont pas "traités" mais seulement "routés" vers leur destination, leur contenu n'est pas visible à l'intérieur du média.

La vision des messages par les médias est bridée à une variable de type "message" (type **L*f*P** prédéfini pour représenter les appels de méthodes). Seul le discriminant du message est visible (paramètres supplémentaires passés par le composant appelant lors de l'appel). Le routage se fait donc en fonction du discriminant.

Les éléments utilisables dans un diagramme de comportement d'un média sont les suivants :
- réception de message ;
- envoi de message ;
- états ;
- transitions ;
- triggers ;
- sous-diagrammes ;
- instanciation de composants **L*f*P** ;
- instanciation dynamique de composants.

5.4.1 Sémantique des transitions de communication

La sémantique des transitions de communication est différente dans les médias. En premier lieu, le seul shéma d'interaction reconnu par les médias est l'envoi de message, on ne fera donc pas de distinction entre une activation de méthode et un message au niveau des médias.

Les transitions de communication utilisables dans les médias sont :
- Les transitions d'envoi de message ;
- les transitions de réception de message.

L'exécution des transitions d'envoi de message est la même que dans le cadre des classes :
- évaluation de la garde ;
- envoi des messages associés à la transition ;
- saut à l'état de sortie de la transition.

5.4.2 Réception de messages dans les médias

La réception de messages dans les médias est associée à la transition de réception de message, il ne peut donc y avoir qu'une seule instruction de lecture par transition de réception de message.

DOCUMENT DE TRAVAIL

Une instruction de lecture sur un binder réalise les opérations suivantes de manière atomique pour le média :

- Sauvegarde des variables utilisées pour stocker le discriminant ;
- Lecture du discriminant du message.
- Évaluation de la garde
- Si la garde s'évalue à `true`, le message est effectivement consommé :
  - si c'est une lecture d'un message de contrôle, l'opération de lecture est terminée.
  - si c'est une lecture d'un message ou d'une activation de méthode, le contenu est sauvegardé dans une variable de type `message`.
- Si la garde s'évalue à `false`, le message n'est pas consommé :
  - les valeurs des variables utilisées pour lire le discriminant sont restaurées à leur ancienne valeur ;
  - le média reste en attente sur ce binder jusqu'à ce qu'un message valide soit lu.

Si un état du diagramme de comportement d'un média est suivi d'au moins une transition de réception de message, il ne peut être suivi que par des transitions de réception de messages. Lorsqu'un état est suivi de plusieurs transitions de réception de message :

- l'ordre d'évaluation est donné par les priorités portées par les arcs ;
- dès qu'un message valide est reçu, les autres lectures sont abandonnées ;
- la transition qui a retourné le message valide est "franchie", et le nouvel état du média est l'état de sortie de cette transition.

5.4.3 Envoi de messages par les médias

Les instructions d'envoi de message dans les médias est portée par une transition d'envoi. Il est possible d'avoir une ou plusieurs de ces instructions sur une transition d'envoi.

Les médias transmettent les messages aux classes par l'intermédiaire des binders. L'envoi du message se fait par écriture dans le binder de la variable qui contient le message. Le binder est désigné par le port correspondant dans le média. Le message est inséré dans le binder en utilisant la fonction d'insertion de message, le comportement de cette opération est donc dépendant du type du binder.

On doit donc distinguer trois cas :

- Binder synchrone sans emplacement disponible pour le nouveau message : Cette opération est bloquante tant que le message ne peut pas être déposé dans le buffer, le composant reste bloqué sur la transition tant que le message n'a pas été déposé dans le buffer. Une fois l'opération terminée, le composant va dans l'état de sortie de la transition.
- Binder asynchrone sans emplacement disponible pour le nouveau message : le message est perdu, cette opération n'est pas bloquante, le composant va directement dans l'état de sortie de la transition.
- Binder synchrone ou asynchrone avec un emplacement disponible pour le nouveau message : le message est déposé dans le binder, et le composant passe directement dans l'état de sortie de la transition.

Dans tous les cas, le message ne peut être envoyé que si la garde de la transition est évaluée à true.

DOCUMENT DE TRAVAIL