THÈSE

présentée à

L'UNIVERSITÉ PIERRE ET MARIE CURIE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ PARIS VI

Spécialité

Informatique

soutenue par

Frédéric GILLIERS

le 28 Septembre 2005

Développement par prototypage et Génération de Code à partir de L*f*P, un langage de modélisation de haut niveau

Directeur de thèse : Professeur Fabrice KORDON

Jury

Président	Jacques	MALENFANT	Professeur à l'Université Pierre et Marie Curie.
Rapporteurs	Laurent	PAUTET	Professeur à l'Ecole Nationale Supérieure des Télécommunications.
	Pascal	ESTRAILLIER	Professeur à l'Université de la Rochelle.
Examinateurs	Fabrice	KORDON	Professeur à l'Univeristé Pierre et Marie Curie.
	Marie Pierre	GERVAIS	Professeur à l'Université de Paris X.
	Didier	BUCHS	Professeur à l'Université de Genève.
	Jean-Michel	COUVREUR	Professeur à l'Université d'Orleans.
	Jean-Pierre	VELU	Expert de la direction technique SAGEM.

UNIVERSITE PIERRE & MARIE CURIE
LA SCIENCE A PARIS

Systèmes Répartis et Coopératifs

Table des matières

1 Introduction Générale **13**
- 1.1 Problématique . . . 13
- 1.2 Objectifs de ce mémoire . . . 14
- 1.3 Contexte du déroulement de cette thèse . . . 15
- 1.4 Organisation de ce mémoire . . . 15
  - 1.4.1 Modélisation et développement . . . 16
  - 1.4.2 Exploitation des modèles et génération de code . . . 16
  - 1.4.3 Expérimentation et perspectives . . . 16

2 La modélisation des systèmes répartis **19**
- 2.1 Introduction . . . 19
- 2.2 Critères d'évaluation des approches présentées . . . 21
  - 2.2.1 Critères méthodologiques . . . 21
  - 2.2.2 Intégration de composants existants . . . 22
  - 2.2.3 Description de l'architecture de l'application . . . 22
- 2.3 L'incontournable UML . . . 22
  - 2.3.1 Présentation de la notation . . . 22
  - 2.3.2 Critères méthodologiques . . . 23
  - 2.3.3 Description de l'architecture de l'application . . . 24
  - 2.3.4 Synthèse sur UML . . . 24
- 2.4 Le langage SDL . . . 25
  - 2.4.1 Présentation du langage . . . 25
  - 2.4.2 Eléments Méthodologiques . . . 26
  - 2.4.3 Utilisation de composants existants . . . 27
  - 2.4.4 Description de l'architecture de l'application . . . 27
  - 2.4.5 Conclusion sur SDL . . . 27
- 2.5 Le langage CO-OPN . . . 28
  - 2.5.1 Présentation du langage . . . 28
  - 2.5.2 Critères méthodologiques . . . 29
  - 2.5.3 Conclusion sur CO-OPN . . . 29
- 2.6 Les langages basés sur l'algèbre des processus . . . 29
  - 2.6.1 Présentation du langage . . . 30
  - 2.6.2 Critères Méthodologiques . . . 31
  - 2.6.3 Utilisation de composants existants . . . 31
  - 2.6.4 Description de l'architecture . . . 31
  - 2.6.5 Conclusion sur les langages de la famille LOTOS . . . 32
- 2.7 Les langages de description d'architecture . . . 32
  - 2.7.1 Présentation du langage AADL . . . 32
  - 2.7.2 Critères Méthodologiques . . . 34

2.7.3 Intégration de composants existants . . . 35
2.7.4 Description de l'architecture . . . 35
2.7.5 Conclusion sur AADL . . . 35
2.8 La notation **L*f*P** . . . 36
2.8.1 Présentation de la notation . . . 36
2.8.2 Critères méthodologiques . . . 37
2.8.3 Description de l'architecture . . . 37
2.8.4 Intégration de composants extérieurs . . . 37
2.8.5 Conclusion sur **L*f*P** . . . 38
2.9 Synthèse des approches proposées . . . 38
2.9.1 synthèse sur les aspects *langages* . . . 40
2.9.2 Synthèse sur les aspects méthodologiques . . . 40
2.10 Conclusion . . . 41

3 La méthodologie et le langage L*f*P **43**
3.1 Introduction . . . 43
3.2 Conception d'une méthodologie de développement pour **L*f*P** . . . 43
3.2.1 Modélisation de l'application . . . 44
3.2.2 Génération du modèle formel . . . 45
3.2.3 Vérification formelle . . . 46
3.2.4 Génération automatique de code . . . 46
3.2.5 La méthodologie **L*f*P** dans le cadre du MDA . . . 47
3.3 Présentation du langage . . . 48
3.3.1 Le diagramme d'architecture . . . 48
3.3.2 Les diagrammes de comportement des composants . . . 50
3.3.3 Les types de données du langage **L*f*P** . . . 52
3.4 Sémantique des types du langage et de leurs opérateurs . . . 53
3.4.1 Présentation des types **L*f*P** . . . 53
3.4.2 Les types de données . . . 54
3.4.3 Les types structurés . . . 59
3.4.4 Les types composants et leurs références . . . 62
3.4.5 Sémantique des binders . . . 66
3.5 Sémantique des éléments statiques du langage . . . 70
3.5.1 Définition de la topologie de l'application . . . 70
3.5.2 Partie déclarative du diagramme d'architecture . . . 71
3.6 Sémantique comportementale du langage **L*f*P** . . . 71
3.6.1 Instructions internes . . . 71
3.6.2 Instructions de contrôle du flot d'exécution . . . 71
3.6.3 Instanciation dynamique . . . 72
3.6.4 Instructions de communication des classes . . . 73
3.6.5 Instructions de communication des médias . . . 77
3.6.6 Multiplexage d'opérations de lecture . . . 78
3.7 Conclusion . . . 79

4 Déploiement **81**
4.1 Introduction . . . 81
4.2 Architecture de placement . . . 82
4.3 Partitionnement de l'application . . . 83
4.3.1 Structuration des exécutables . . . 83

4.3.2 Spécialisation des exécutables . . . 84
4.3.3 Démarrage des exécutables . . . 85
4.4 Placement des exécutables . . . 85
4.5 Déploiement des instances dynamiques . . . 86
4.6 Structure d'une application **L*f*P** . . . 86
4.7 Conclusion . . . 88

5 L'exécutif L*f*P **89**
5.1 Introduction . . . 89
5.2 Fonctionnalités de l'exécutif **L*f*P** . . . 90
5.2.1 Structure de l'exécutif . . . 90
5.2.2 Structure de la bibliothèque de composants . . . 91
5.3 Implémentation de l'exécutif . . . 93
5.3.1 Architecture du prototype de l'exécutif . . . 94
5.3.2 Implémentation de la bibliothèque de composants de l'exécutif . . . 96
5.3.3 Interface externe de l'exécutif . . . 99
5.4 Format des informations de démarrage . . . 101
5.4.1 Informations nécessaire à la définition d'une instance statique . . . 101
5.4.2 Structure du fichier de déploiement . . . 101
5.4.3 Exemple de fichier de déploiement . . . 102
5.5 Perspectives de développement de l'exécutif . . . 102
5.5.1 Perspective dans les environnements contraints . . . 102
5.5.2 Liaison avec des environnements standards . . . 103
5.6 Conclusion . . . 103

6 Règles de transformation pour la génération de code **105**
6.1 Introduction . . . 105
6.2 La transformation de modèles . . . 106
6.2.1 Catégories de transformations de modèles . . . 106
6.2.2 Langages de transformation de modèles . . . 107
6.3 Transformation de modèle et génération de code pour **L*f*P** . . . 108
6.3.1 Processus de génération de code . . . 108
6.3.2 Structuration des représentations des modèles . . . 109
6.4 Règles de transformation pour les types **L*f*P** . . . 110
6.4.1 Le type `boolean` . . . 110
6.4.2 Les types opaques . . . 110
6.4.3 Les types composants . . . 111
6.4.4 Les types énumérés et leurs types restreints . . . 113
6.4.5 Le type integer et ses types restreints . . . 114
6.4.6 Les types structurés . . . 115
6.5 Règles de génération des éléments statiques du langage . . . 117
6.5.1 Partie déclarative du diagramme et déploiement de l'application . . . 117
6.5.2 Partie architecturale du diagramme . . . 117
6.6 Présentation des règles de génération des éléments dynamiques . . . 118
6.6.1 Code généré pour une structure d'alternative . . . 118
6.6.2 Code généré pour les structures de boucles . . . 119
6.6.3 Code généré pour un envoi de message de données . . . 119
6.6.4 Code généré pour une réception de message de données . . . 119
6.6.5 Code généré pour un appel de méthode . . . 120

6.6.6 Activation de méthode et lecture de message par les médias 120
6.6.7 L'instanciation dynamique 123
6.6.8 Terminaison de l'exécution d'un composant 124
6.7 Le code Java généré 124
6.7.1 Principes de génération des types de données 125
6.7.2 Code généré pour les composants **L*f*P** 125
6.7.3 Initialisation de l'application 128
6.8 Conclusion 128

7 Expérimentation : le projet MORSE **131**
7.1 Introduction 131
7.2 Présentation de l'exemple 131
7.3 Partie statique du modèle : le diagramme d'architecture 133
7.3.1 Définition des composants 133
7.3.2 Définition des types de données 134
7.4 Partie dynamique du modèle : comportement des composants 135
7.4.1 Modélisation des médias 135
7.4.2 La réception des messages du côté client 136
7.4.3 L'envoi des messages par le client 137
7.4.4 La gestion du groupe : la classe `group` 138
7.4.5 Le serveur de gestion de groupes 141
7.5 Déploiement de l'exemple 144
7.5.1 Le site du serveur de groupes 144
7.5.2 Les sites clients 144
7.6 Présentation du code généré 145
7.6.1 Implémentation des appels de fonction 145
7.6.2 Code généré pour une instruction select 146
7.7 Conclusion 148

8 Perspectives et retours sur expériences **149**
8.1 Introduction 149
8.2 Vérification formelle de spécifications **L*f*P** 149
8.3 Évolutions du langage **L*f*P** 150
8.3.1 Liaisons entre les composants 151
8.3.2 Système de traitement des exceptions 153
8.3.3 Ajout de temporisation pour le traitement des messages 153
8.3.4 Gestion des ports d'activation des méthodes 153
8.3.5 Syntaxe du langage **L*f*P** 154
8.4 Optimisation du code généré 154
8.5 Conclusion 155

9 Conclusion Générale **159**
9.1 Apports méthodologiques 160
9.2 Définition du langage **L*f*P** 160
9.3 La génération de code 161
9.4 Perspectives 162

A BNF du langage L*f*P sous sa forme graphique **165**
A.1 Présentation . . . 165
A.2 Conventions . . . 165
A.3 Éléments communs à tous les diagrammes . . . 165
A.3.1 Eléments de base . . . 165
A.3.2 Syntaxe des expressions . . . 166
A.3.3 Syntaxe des déclarations de types . . . 168
A.3.4 déclaration de variables et de constantes . . . 170
A.4 Le diagramme d'architecture . . . 171
A.4.1 Déclaration des instances statiques . . . 171
A.4.2 Définition des caractéristiques des binders . . . 172
A.5 Le diagramme de comportement . . . 173
A.5.1 Déclaration des triggers . . . 173
A.5.2 Déclaration des méthodes d'une classe . . . 173
A.5.3 Syntaxe des attributs globaux des sous-diagrammes . . . 175
A.5.4 Attributs des transitions simples . . . 176
A.5.5 Opérations autorisées sur les transitions de communication . . . 177
A.5.6 Garde des transitions . . . 181
A.5.7 Instanciation dynamique de composants . . . 181

B BNF du langage L*f*P sous sa forme textuelle **183**
B.1 Introduction . . . 183
B.2 Conventions de présentation . . . 183
B.3 Éléments de base du langage . . . 183
B.3.1 Casse des caractères . . . 183
B.3.2 Mots réservés et symboles utilisés par le langage **L*f*P** . . . 184
B.3.3 Identificateurs valides . . . 184
B.3.4 Chiffres et nombres . . . 185
B.4 Syntaxe des expressions . . . 185
B.5 Syntaxe de la partie déclarative . . . 186
B.5.1 Déclaration des types tableaux . . . 187
B.5.2 Déclaration des types énumérés . . . 187
B.5.3 Déclaration des intervalles de valeurs . . . 188
B.5.4 Déclaration des structures . . . 188
B.5.5 Déclaration des types port . . . 189
B.5.6 Déclaration de variables et de constantes . . . 189
B.5.7 Déclaration des composants . . . 189
B.5.8 Déclaration des méthodes des classes . . . 190
B.5.9 Déclaration des triggers . . . 192
B.5.10 Déclaration des binders . . . 192
B.5.11 Déclaration de la topologie . . . 193
B.6 Syntaxe des instructions du langage **L*f*P** . . . 194
B.6.1 Affectation . . . 194
B.6.2 Les blocs d'instruction . . . 194
B.6.3 Les boucles ‘*for*” . . . 195
B.6.4 Les boucles *“while”* . . . 195
B.6.5 Instruction *“break”* . . . 195
B.6.6 L'instruction d'alternative . . . 196
B.6.7 Instruction de saut . . . 196

B.6.8 Lecture d'un message 196
B.6.9 Attente d'activation de méthode 197
B.6.10 Multiplexage des opérations de réception de messages 197
B.6.11 Envoi d'un message de données 198
B.6.12 Appel d'une méthode d'un composant 198

Bibliographie **205**

Table des figures

2.1 Méthode de prototypage centrée sur la modélisation 19

3.1 Introduction de la méthodologie **L*f*P** 43
3.2 Méthodologie de développement associée au langage **L*f*P**. 45
3.3 Correspondance entre le MDA et la méthodologie **L*f*P** 47
3.4 Un exemple de diagramme d'architecture 49
3.5 Diagramme de comportement de la classe *Server* 50
3.6 Les trois catégories de types définis par **L*f*P** 52
3.7 Relations entre les types de données du langage **L*f*P** 53
3.8 Les types composants en **L*f*P** 54
3.9 Le type binder dans le système de typage lfp 66
3.10 exemple de binder synchrone dans un diagramme d'architecture 67
3.11 Différences sémantiques entre les multiplicité `all` et `1` 67
3.12 Déroulement d'une lecture sur un binder 69
3.13 Les trois types d'appels de méthode 74

4.1 Structuration du placement des composants 82
4.2 Structure d'une application générée à partir d'un modèle **L*f*P**. 87

5.1 Le processus de génération de code 89
5.2 structure logique de l'exécutif **L*f*P** 90
5.3 Association entre fonctionnalités de l'exécutif et classes d'implémentation 94
5.4 Diagramme de classe simplifié de l'exécutif **L*f*P** 94
5.5 Diagramme de classes de l'implémentation des références 99
5.6 Exemple de fichier de déploiement 102

6.1 Approche de transformation modèle vers modèle 106
6.2 Approche de transformation modèle vers texte 106
6.3 Le processus de génération de code 108
6.4 Structure du code d'une méthode 126
6.5 Traduction d'une instruction d'instanciation dynamique de composant 127

7.1 Diagramme de classe simplifié de l'étude de cas 132
7.2 Diagramme d'architecture du système de messagerie instantanée 133
7.3 Diagramme de comportement du média `msg_sender` 136
7.4 Diagramme de comportement du média `RPC` 136
7.5 Diagramme de comportement de la classe `Receiver` 137
7.6 Diagramme de comportement de la classe `Sender` 137
7.7 Diagramme de comportement de la classe `group` 139
7.8 Diagramme de comportement de la méthode `diffuse` 140

7.9 Diagramme de comportement de la méthode `register` . . . 140
7.10 Diagramme de comportement de la méthode `remove` . . . 141
7.11 Diagramme de comportement de la classe `server` . . . 142
7.12 Diagramme de comportement de la méthode `join` . . . 142
7.13 Diagramme de comportement de la méthode `quit_group` . . . 143
7.14 Fichier de déploiement de l'étude de cas . . . 144
7.15 Transition d'appel de fonction **L*f*P** et le code d'implémentation . . . 145
7.16 Instruction select et code généré correspondant . . . 147

8.1 Exemple d'automate de comportement de type *suite de transition* . . . 155
8.2 Deux implémentations du média RPC . . . 156

A.1 Lexème définissant un identificateur en **L*f*P**. . . . 166
A.2 Lexème définissant un nombre . . . 166
A.3 Syntaxe d'une expression valide en **L*f*P**. . . . 167
A.4 Règles de syntaxe pour les déclarations de types instanciés. . . . 169
A.5 déclaration de variables et de constantes . . . 170
A.6 Déclaration des composants du modèle (réservées au diagramme d'architecture) . 172
A.7 association des binders . . . 172
A.8 Un exemple de déclaration de binder . . . 173
A.9 Déclaration des triggers . . . 173
A.10 Déclarations des méthodes d'une classe . . . 174
A.11 syntaxe d'une suite de déclaration . . . 175
A.12 déclaration du corps d'un trigger . . . 175
A.13 Déclaration du corps d'une méthode . . . 176
A.14 Instructions disponibles sur les transitions **L*f*P**. . . . 176
A.15 Les trois types de transitions d'envoi / réception de méthode . . . 177
A.16 Opérations réalisables sur les ports d'une classes **L*f*P** . . . 178
A.17 Exemples d'opérations de communication depuis les classes **L*f*P** . . . 179
A.18 Opérations réalisables sur les ports d'un média **L*f*P** . . . 180
A.19 Exemples d'opérations de communication depuis les médias **L*f*P** . . . 180
A.20 Syntaxe des gardes des transitions . . . 181
A.21 paramètres d'instanciation d'une classe . . . 181

B.1 Liste des mots clés du langage **L*f*P** . . . 184
B.2 Symboles utilisés par le langage **L*f*P** . . . 184
B.3 Lexème définissant un identificateur en **L*f*P** . . . 184
B.4 Lexème définissant un entier . . . 185
B.5 Syntaxe des expressions valides en **L*f*P** . . . 186
B.6 BNF de la déclaration d'un type tableau et des intervalles de valeurs . . . 187
B.7 Syntaxe des déclarations de types énumérés en **L*f*P** . . . 187
B.8 BNF d'un type défini par intervalle de valeur . . . 188
B.9 BNF de la déclaration d'un type record . . . 188
B.10 BNF de la déclaration d'un type port . . . 189
B.11 BNF des déclarations de variables et constantes . . . 189
B.12 BNF de la déclaration d'un type composant . . . 190
B.13 BNF de la déclaration d'une méthode **L*f*P** . . . 191
B.14 BNF de la déclaration d'un trigger . . . 192
B.15 BNF de la déclaration des binders . . . 193
B.16 BNF de la déclaration de la topologie du modèle . . . 194

B.17 BNF de l'instruction d'affectation 194
B.18 Déclaration d'un bloc d'instructions 194
B.19 BNF d'une boucle *"for"* 195
B.20 BNF des boucles *"while"* 195
B.21 BNF de l'instruction break 195
B.22 BNF de l'instruction d'alternative 196
B.23 BNF de l'instruction goto et des labels 196
B.24 BNF de l'instruction de lecture de message 197
B.25 BNF d'une instruction d'attente d'activation de méthode 197
B.26 BNF de l'instruction accept 197
B.27 BNF de l'instruction d'envoi d'un message de donnée 198
B.28 BNF d'un appel de méthode en lfp 198

Chapitre 1

Introduction Générale

Un grand nombre d'applications informatiques fonctionnent de manière réparties et coopérative sur un ensemble de postes dédiés à une tâche commune. Ces applications sont de plus en plus fréquemment hétérogènes, étant déployées à la fois sur des serveurs de grande capacité et sur des terminaux légers de type PDA. Outre les besoins fonctionnels, elles doivent également prendre en compte des éléments *non-fonctionnels* tels que des impératifs de sécurité, ou la gestion de topologies particulières.

Dans ce contexte, nous nous intéressons au développement et au déploiement d'applications réparties fiables, c'est à dire ayant un comportement déterministe.

1.1 Problématique

Les aspects *contrôle* sont prépondérants dans les applications réparties car il faut assurer la collaboration entre les entités du système. C'est à ce niveau que doivent être prises en compte les difficultés liées à la répartition et l'asynchronisme des communications.

Les applications réparties fonctionnent sur un ensemble de machines connectées via un réseau. Elles s'exécutent souvent sur des sites faiblement couplés (par exemple, géographiquement très éloignés) ou encore reliés par des liens de communication non fiables (par exemple, des liaisons hertziennes). Ces applications s'exécutent dans un environnement asynchrone, caractérisé par l'absence d'horloge globale (et donc l'impossibilité de synchroniser l'exécution des composants de manière temporelle). De plus, les temps de communication sont très variables, même s'ils restent en général bornés.

Ces caractéristiques introduisent de l'indéterminisme lors de l'exécution et engendrent l'explosion combinatoire des exécutions possibles qui rend inopérante les techniques de tests «traditionnelles». En effet, il est difficile d'isoler des séquences d'exécution d'un tel système ou même de reproduire une séquence d'exécution particulière.

Nous identifions ainsi un premier problème : *pour garantir la cohérence des exécutions, il faut donc s'appuyer sur des protocoles d'interaction entre les composants de l'application répartie. La conception de ces protocoles est un problème particulièrement difficile qui doit être traité très tôt dans le cycle du développement.*

Une manière de résoudre le problème est d'adopter une démarche de développement *incrémentale*. On identifie au plus tôt l'architecture du système que l'on raffine tout au long du projet. Mais, dans un contexte de compétition de plus en plus exacerbé, ce type de démarche coûte extrêmement cher et prend du temps. On a ainsi un processus itératif dont la démarche s'apparente au RAD-Cycle [45].

Nous identifions ainsi un second problème : *la phase de développement doit être supportée par des outils qui permettent de diminuer le nombre d'itérations dans le processus (mieux prévoir*

l'impact des choix d'un protocole) mais aussi de diminuer le coût et la durée d'une itération (utilisation de générateurs de programmes, etc.).

De tels outils doivent nécessairement s'appuyer sur des notations capables de capturer les aspects dynamiques de ces systèmes. Si la dernière version d'UML [60] permet de mieux capturer ces aspects dynamiques, elle reste encore incomplète. L'utilisation de techniques formelles, par exemple basées sur le *model checking*, reste encore limitée.

Nous identifions ainsi un troisième problème : *face à la complexité des notations, des outils, et de la diversité des utilisations possibles, il faut disposer d'un processus type de développement*.

Ainsi, le développement d'applications réparties reste un problème difficile et coûteux, tout particulièrement dans un contexte de concurrence exacerbée entre acteurs du domaine. La «crise chronique du logiciel» identifiée dans [26] prend une importance toute particulière dans le cas des applications réparties.

1.2 Objectifs de ce mémoire

L'objectif général de la communauté est de rendre le développement d'applications réparties plus déterministe. Nous nous situons dans ce contexte.

Les principales caractéristiques de nos travaux sont les suivantes.

Une démarche par prototypage Notre démarche s'insère dans la mouvance actuelle qui privilégie le rôle prépondérant des modèles. Qu'elles soient nommés MDA (Model Driven Architecture [57]), MDE (Model Driven Engineering [44]) ou MDD (Model Driven Development [1]), ces approches préconisent toutes d'utiliser un modèle comme base du travail de d'analyse d'une part, de développement d'autre part. Dans ce contexte des applications réparties, ce type d'approche nous semble primordial car il permet d'analyser au plus tôt les protocoles élaborés entre les différents composants. De plus, couplé avec une assistance à la production de programmes, il facilite l'obtention au plus tôt de programmes exécutables permettant une validation dans un environnement d'exécution proche de l'architecture cible. D'une démarche incrémentale, on passe à un processus de développement par prototypage [47].

Dans ce contexte, les méthodes formelles permettent de raisonner de manière systématisée sur l'exécution du système et permettent de maîtriser la complexité induite par la répartition et le parallélisme. Cela nous semble primordial dans la phase de compréhension de la dynamique du système. De plus, elles permettent de caractériser progressivement le protocole d'interaction entre les composants du système. Cela nous semble un moyen intéressant pour aborder le premier problème que nous avons identifié.

Rendre ces techniques accessibles aux développeurs Si l'utilisation de méthodes formelles permet de mieux appréhender la dynamique d'une application répartie, elles posent cependant de nombreux problèmes dans le cadre d'une utilisation dans l'industrie. On cite en particulier les problèmes d'apprentissage de techniques complexes et de notations souvent peu intuitives du point de vue d'un ingénieur [53]. Ainsi, la difficulté de mise en oeuvre de ces méthodes freine leur déploiement dans l'industrie.

Il est donc nécessaire d'intégrer les méthodes formelles dans des standards industriels déjà largement diffusés telles qu'UML [59]. L'objectif de notre approche est de masquer leur complexité en utilisant des techniques automatisées basées sur un langage de haut niveau permettant de spécifier une solution opérationnelle. L'approche doit être intégrée dans un environnement de développement le plus standard possible.

Une fois le système élaboré et le protocole d'interaction entre les composants vérifié, il reste à l'implémenter. Pour assurer la meilleure relation entre la spécification de la solution ainsi obtenue,

il faut s'appuyer sur des techniques à base de génération de programmes. Ces éléments adressent le deuxième problème que nous avons identifié.
Pour cela, nous devons réfléchir aux relations entre :

- le langage de spécification de la solution opérationnelle utilisé par le développeur,
- le domaine d'application auquel ce langage est associé,
- les techniques formelles ainsi encapsulées (notations, techniques de vérification, etc.)
- les programmes produits à partir langage de spécification.

Pour être utilisables dans un contexte industriel, il faut outiller ces notations. Il faut un compilateur pour le langage de spécification de la solution, des outils de vérification (par exemple du model checking), des générateurs de programmes. De tels outils existent aujourd'hui mais :

- ils ne concernent que marginalement le domaine d'application visé (le développement d'applications répartis se réduit principalement à l'utilisation manuelle de middleware)
- le couplage entre ces différents outils reste limité.

Obtenir une démarche de développement dans un contexte industriel La plupart des développements industriels s'appuient sur des composants préexistants. Il faut donc intégrer ces composants dans la démarche orientée modèle en tenant compte du fait que leur origine est très diverse (par exemple, un COTS[1]). Cela pose un problème particulier si nous voulons maintenir une relation pertinente avec la phase de vérification formelle. Pour cela, il faut structurer les relations entre le modèle et son environnement d'exécution.

De plus, les problèmes de déploiement ne peuvent être ignorés car il faut pouvoir prendre en compte les variations des architectures cibles tout au long d'un projet. Pour cela, nous devons résoudre au niveau d'un générateur de programmes, la relation entre le langage de spécification de a solution et son environnement d'exécution. Cela revient à poser des hypothèses minimales de fonctionnement (par exemple offertes par un middleware).

Cela soulève le besoin d'une démarche méthodologique appropriée afin d'accompagner le développeur dans sa démarche de spécification, puis de développement.

1.3 Contexte du déroulement de cette thèse

Cette thèse s'est déroulée en partenariat entre la société SAGEM et l'université Pierre et Marie Curie (Paris VI) dans le cadre des travaux du projet RNTL[2] MORSE [89] (**M**éthodes et **O**utils pour la **R**éalisation et la vérification formelle de **S**ystèmes interopérables **E**mbarqués critiques).

Ainsi, nous nous sommes assignés un objectif supplémentaire : la mise en œuvre d'un prototype industriel d'environnement permettant de supporter la démarche. Nous nous sommes également efforcé de valider les points principaux de l'utilisation de nos techniques au moyens de cas d'études validé par les partenaires industriels du projet MORSE.

1.4 Organisation de ce mémoire

Ce mémoire est organisé selon trois parties principales :

- la modélisation et la méthodologie de développement ;
- la génération de code ;
- l'expérimentation et les retours sur expériences.

[1] Component Off the Shelf
[2] Réseau National des Technologies Logicielles [19]

1.4.1 Modélisation et développement

La première partie de ce mémoire est consacrée aux techniques de modélisation et de développement des applications réparties. Notre premier objectif est d'identifier les propriétés que doit avoir un langage de modélisation utilisable pour représenter une classe d'applications réparties aussi large que possible. Dans ce cadre, nous nous intéressons également à l'aspect méthodologique nécessaire à l'intégration du langage dans le cycle de vie d'un projet industriel.

Nous commençons donc par établir un état de l'art au chapitre 2. Celui-ci présente succinctement plusieurs approches existantes et les évalue en fonction de deux types de critères :

- l'adéquation entre le langage de modélisation et la classe d'application que nous visons ;
- l'adéquation entre les modèles produits et l'utilisation que nous souhaitons en faire dans le cadre de notre méthodologie.

En fonction des résultats de cette étude, nous proposons un méthodologie de développement d'applications réparties construite autour de la notation **L*f*P** [67](*Language for Prototyping*), et nous indiquons son positionnement dans l'approche de développement MDA.

Le chapitre 3 présente le langage **L*f*P** est la méthodologie d'utilisation permettant de l'insérer dans un contexte industriel. Ce langage est un prolongement des travaux exposés dans [67]. Ce nouveau langage propose les abstractions qui nous ont semblé pertinentes en fonction des résultats obtenus au chapitre précédent. Seule la sémantique du langage est présentée, sa syntaxe précise est définie par l'annexe A.

1.4.2 Exploitation des modèles et génération de code

Le deuxième volet de notre travail concerne l'exploitation des modèles exprimés dans le formalisme que nous venons de définir. Nos travaux ont principalement été orientés vers la production automatique du code d'implémentation de l'application et son déploiement dans un environnement cible connu, mais d'autres travaux effectués en parallèle sur la vérification des modèles **L*f*P** sont évoqués.

Le chapitre 4 présente les techniques mises en oeuvre pour déployer les spécifications **L*f*P** sur une architecture d'exécution répartie. Ce chapitre aborde les problèmes du placement des composants définis par la spécification, ainsi que ceux liés à la définition des exécutables requis pour les implémenter. Ce chapitre définit la notion d'exécutif nécessaire au déploiement des spécifications **L*f*P**.

Le chapitre 5 présente cet exécutif ; son rôle est de fournir un environnement stable pour le déploiement des applications produites à partir de **L*f*P**. Il offre au générateur de code une interface stable indépendante de la plate-forme cible. Le chapitre 5 présente l'architecture de l'exécutif ainsi qu'une implémentation prototype réalisée an java dans le cadre de nos travaux de thèse.

Le chapitre 6 présente la génération de code pour **L*f*P**. Nous avons retenu une approche basée sur la transformation de modèle. Ceci nous amène à présenter le processus de génération de code sous forme de *règles de transformation* définissant le code correspondant à chaque construction **L*f*P** en fonction de la plate-forme cible choisie. Nous proposons un ensemble de *"paterns"* génériques utilisables pour un ensemble aussi vaste que possible de langages cibles, puis nous donnons une instanciation de ces *"patterns"* pour le langage java et l'exécutif décrit au chapitre 5.

1.4.3 Expérimentation et perspectives

Le troisième volet de notre travail est consacré aux expérimentations réalisées à l'aide du langage **L*f*P**, ainsi qu'à l'analyse des premiers résultats.

Le chapitre 7 présente une des premières études de cas réalisées dans la cadre du projet MORSE. Cet exemple permet de présenter le langage **L*f*P** à l'aide d'un cas concret mais suffi-

samment simple pour être inséré dans ce mémoire. Il permet d'illustrer l'utilisation du générateur de code pour obtenir une application complète, et capable d'interagir avec un utilisateur. L'objectif est d'obtenir un premier retour d'expérience de l'utilisation du langage **L*f*P** et des techniques de génération de code associées. L'application choisie est une "messagerie instantanée", elle permet de réaliser simplement un grand nombre de tests tel que l'utilisation de bibliothèques de composants externes, et d'étudier la qualité du code généré.

Le chapitre 8 analyse et commente les résultats fournis par les travaux réalisés autour de **L*f*P** dans le cadre de cette thèse et dans le cadre du projet MORSE. L'objectif est de fournir des perspectives d'évolution pour le langage et les outils qui lui sont associés. Ce chapitre présente donc les premiers résultats obtenus dans le domaine de la vérification formelle du langage **L*f*P**, ainsi que les travaux restant à effectuer dans ce domaine. Les qualités du langage sont également étudiées, et nous proposons quelques évolutions compatibles avec la sémantique actuelle pour simplifier l'écriture des modèles et augmenter le pouvoir d'expression du langage. Enfin, nous proposons une voie d'amélioration des règles de génération de code basée sur l'analyse de la structure des automates.

Le chapitre 9 termine ce mémoire en rappelant nos principales contributions au domaine du développement d'applications réparties. Ce chapitre présente également les résultats attendus des travaux en cours de réalisation autour de **L*f*P**. Il envisage également l'avenir en présentant les voies de recherche restant à explorer pour améliorer l'approche défendue dans ce mémoire.

Chapitre 2

La modélisation des systèmes répartis

2.1 Introduction

Notre approche du développement se situe dans la lignée des méthodes dites par *prototypage* [47]. L'objectif est de construire aussi rapidement que possible une implémentation fonctionnelle d'une partie de l'application, et de l'enrichir jusqu'à obtenir l'application finale. Afin de d'obtenir rapidement ces prototypes successifs, nous proposons de centrer le développement sur le modèle de l'application. Le code de chaque prototype est produit par un outil de génération automatique de code à partir du modèle. On parle alors de développement *orienté modèle* [65] (de l'anglais *Model Based Development*).

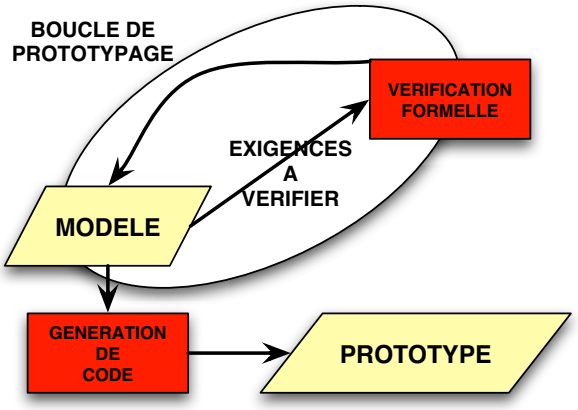


FIG. 2.1 – Méthode de prototypage centrée sur la modélisation

La principale difficulté de conception des applications réparties provient de l'indéterminisme introduit par l'asynchronisme des communications. Nos travaux portent principalement sur les moyens de maîtriser ces aspects à l'aide d'une méthodologie et d'un langage de modélisation adapté. Nos travaux sont donc centrés sur la modélisation de l'aspect contrôle de l'application. Il est défini par l'ensemble des mécanismes (protocoles) requis pour faire communiquer les composants applicatifs.

L'objectif des approches *orientées modèles* est d'élever le niveau d'abstraction du développement en passant du niveau *code* au niveau *modèle*. Comme l'illustre la figure 2.1, la boucle principale de prototypage est centrée sur un modèle fonctionnel de l'application. La modélisation devient donc l'activité centrale du développement d'une application, le choix du formalisme et de

la méthodologie utilisées sont donc critiques pour la qualité de l'application finale.

Pour être efficace, ce type d'approche doit être basée sur un modèle adapté à :

- la vérification rapide et simple des propriétés attendues de l'application (exigences) ;
- la production automatique du code de l'application correspondant au modèle.

La vérification de propriétés fonctionnelles est essentielle dans cette démarche. Toutes les exigences fonctionnelles exprimables sur le modèle peuvent alors être vérifiées sans passer par un processus de test particulièrement coûteux dans le cas des applications réparties. Les possibilités de vérification formelle du modèle et le formalisme d'expression des propriétés vérifiables sont donc des critères dérivés permettant d'évaluer les possibilités d'exploitation des modèles écrits dans un formalisme donné.

La génération du code d'implémentation du modèle permet la réalisation rapide de prototypes fonctionnels de l'application. L'objectif est de permettre de générer à faible coût une partie de l'application à partir de son modèle et de déployer le code produit dans l'environnement cible. Le processus de génération de code doit donc assurer que les propriétés vérifiées sur le modèle sont également vérifiées par le code généré.

Cette propriété est vraie si le générateur peut implémenter exactement la sémantique du langage de modélisation qui doit donc disposer d'une sémantique exécutable. Notre objectif est d'insérer le code produit à partir du modèle dans la version de déploiement de l'application. La sémantique du langage de modélisation doit donc être restreinte à des opérations pouvant être implémentées avec un niveau de performance acceptable pour le domaine d'application visé. Dans le cas général, le niveau de performance du code produit doit être équivalent à celui produit par un être humain.

L'approche de développement *orientée modèle* couramment évoquée est le MDA [57] dont l'objectif est également de franchir le niveau d'abstraction supplémentaire nécessaire pour recentrer le développement d'une application sur le développement de son modèle. Celui-ci est ensuite raffiné par *transformation de modèle* pour obtenir finalement le prototype de l'application. L'approche MDA est basée sur UML et à ce titre ne bénéficie que d'un niveau modéré de formalisation et d'automatisation [10].

Les bénéfices de l'utilisation des approches formelles dans le cadre du développement d'applications critiques ont été largement discutés. Les principaux travaux dans ce sens proviennent des domaines de l'informatique embarquée et certifiée où cette problématique est largement présente [70][71].

Certaines normes récentes telles que les *critères communs pour l'évaluation de la sécurité des Technologies de l'information* [12] rendent nécessaire l'utilisation des méthodes formelles pour obtenir leur plus haut niveau de certification. Ainsi la partie 3 des critères communs définit le niveau d'assurance de l'évaluation 7 (EAL-7) sous le nom de «*conception vérifiée à l'aide de méthodes formelles et testée*».

Bien que toutes les normes de certification ne demandent pas explicitement l'utilisation de méthodes formelles, elles définissent comment les résultats fournis par ces méthodes peuvent être intégrés dans le processus de certification. C'est par exemple le cas de la DO-178B [69] qui envisage l'utilisation des méthodes formelles et définit la manière dont les résultats fournis peuvent être pris en compte pour l'obtention de la certification.

L'objectif de ce chapitre est de définir parmi les approches existantes celles qui semblent le plus appropriées à la définition d'une méthodologie basée sur la modélisation. La première tâche de cette étude est de définir les critères d'évaluation des méthodes présentées, ce qui est l'objet de la section 2.2. Les sections suivantes évalueront un panel des solutions existantes pour la modélisation des applications réparties :

- la section 2.3 étudie UML qui est le standard de fait de la modélisation dans le monde industriel ;

– la section 2.4 présente le langage SDL [31], basé sur les automates communiquants par files de messages et couramment utilisé dans les télécommunications ;
– la section 2.5 s'intéresse à COOPN un langage de modélisation issu des réseaux de Petri et des notations algébriques ;
– la section 2.6 s'intéresse aux langages basés sur les processus communiquants ;
– la section 2.7 s'intéresse aux langages de description d'architecture à travers le langage AADL qui est représentatif des travaux récents dans ce domaine ;
– la section 2.8 introduit la notation **L*f*P** pour la modélisation des applications réparties.

La section 2.9 propose une synthèse des approches étudiées en fonction des critères que nous avons défini à la section 2.2. Enfin, la section 2.10 conclura ce chapitre.

2.2 Critères d'évaluation des approches présentées

Cette section présente les critères d'évaluation des approches de développement d'applications réparties. Ces critères sont les suivants :
– critères méthodologiques ;
– Intégration avec des composants existants ;
– Description de l'architecture de l'application.

2.2.1 Critères méthodologiques

Les approches que nous étudions dans ce chapitre sont caractérisées par l'utilisation d'un langage de modélisation. Quelques soient ses qualités, il est nécessaire de guider l'utilisateur dans sa démarche de conception de l'application. Cet aspect est traité en définissant une méthodologie de construction et d'exploitation du modèle de l'application. Cette méthodologie doit donc fournir guide d'utilisation du langage pour toutes les étapes du cycle de développement de l'application.

Dans le cadre de notre étude, nous retenons les critères méthodologiques suivants :
– intégration avec la notation UML ;
– définition des propriétés du modèle ;
– liaison avec les méthodes formelles ;
– liaison avec la génération de code.

Un de nos objectifs est de permettre le déploiement de méthodes formelles dans un environnement industriel, or la notation UML est devenue le standard incontournable du développement logiciel. Elle est couramment utilisée dans le monde industriel, une méthodologie de développement correctement intégrée dans un processus basé sur UML est donc plus facilement intégrée aux méthodes de développement des entreprises.

La définition des propriétés du modèle permet d'exprimer des contraintes que le modèle doit respecter. L'objectif est de spécifier les comportements corrects du modèle, et donc de permettre la détection des comportement déviants.

Ce point est à mettre en relation avec la liaison vers les méthodes formelles qui permettent de vérifier les propriétés énoncées sur le modèle. Nous considérons uniquement les approches *automatiques* pour la vérification des modèles. Ces techniques basées sur la simulation exhaustive du système (ou *model checking*) car leur déploiement reste plus facile que les approches basées sur les prouveurs de théorème.

Enfin, nous étudieront les possibilités de génération de code, c'est à dire la possibilité d'implémenter et de déployer automatiquement le modèle. Nous nous intéresserons aux approches permettant de produire du code destiné à être intégré dans l'application finale.

2.2.2 Intégration de composants existants

Le développement d'une application nécessite souvent l'utilisation de composants préexistants (*legacy components*). Ceux-ci peuvent provenir de différentes sources : composants provenant d'une version précédente de l'application, composants standards achetés *sur étagère* (COTS pour *Components Off the Shelf*), composants d'une autre application avec laquelle la nouvelle application doit communiquer, etc...

L'approche de développement proposée doit pouvoir intégrer ces composants à son processus de développement. Il faut donc disposer d'un moyen de

- spécifier l'interface des composants réutilisés ;
- spécifier les interactions entre le modèle et ces composants.

Ce critère doit être évalué en fonction des capacités de vérification formelle associées à la méthodologie. L'objectif est d'évaluer quelle influence les composants réutilisé peuvent avoir sur les résultats fournis par la vérification formelle.

2.2.3 Description de l'architecture de l'application

Les langages de description d'architecture (ou ADL pour *Architecture Description Language*) permettent de définir l'architecture d'une application. Ils mettent l'accent sur la structure logique de l'application : [54] définit qu'un ADL doit modéliser explicitement les *composants* de l'application, leurs *connecteurs* et leur *configuration*. Ils permettent d'avoir une approche formelle de la description de l'architecture d'un système [4].

Dans le cas d'une application répartie, la description architecturale est particulièrement importante. En définissant clairement les canaux d'interaction entre les composants, elle permet de maîtriser une partie de la complexité induite par la répartition. De plus, ces informations sont exploitées pour définir le déploiement de l'application sur l'architecture cible. En fonction des spécificités du langage utilisé, la description architecturale peut permettre d'aller jusqu'à la spécification directe du déploiement de l'application sur l'environnement cible.

2.3 L'incontournable UML

UML (*Unified Modeling Language*) est actuellement le langage de modélisation le plus utilisé dans l'industrie. Cette notation se présente comme "l'ensemble des meilleures méthodes d'ingénierie qui se sont montrées efficaces dans le domaine de la modélisation de grands systèmes"[1] [59]. UML est en fait une notation graphique adaptée pour représenter les systèmes logiciels selon une approche orientée objet issue de plusieurs notations (OMT, OOSE ET BOOCH).

2.3.1 Présentation de la notation

UML établit la séparation entre les aspects statiques et les aspects dynamiques du système modélisé. Pas moins de neuf diagrammes sont utilisés pour représenter ces aspects, mais cette richesse d'expression est également un des points faible d'UML : il devient difficile de maintenir la cohérence entre tous les diagrammes constituant le modèle. De plus, la sémantique même du langage semble souffrir d'incohérences [81]. Les versions plus récentes de la norme se sont attaquées à ces problèmes de formalisation, mais sans parvenir à fournir une spécification formelle du langage, notamment pour les aspects dynamiques.

[1] The UML represents a collection of the best engineering practices that have proven successful in the modeling of large and complex systems

Les diagrammes UML les plus utilisés pour la description statique des applications sont les diagrammes de classes, et les diagrammes d'objet. Ils permettent de définir de manière relativement fine les composants d'une application. Les principaux diagrammes dynamiques définis en UML sont les *statecharts*, les diagrammes de séquence, les diagrammes de collaboration et les diagrammes d'activités. Les statecharts sont issus des travaux de Harel [28] et ont été adaptés pour être cohérents avec une approche orientée objets. Les diagrammes de séquences proviennent des MSC (*Message Sequence Charts*) [33].

UML est au centre du MDA [57] (*Model Driven Achitecture)*. Ce dernier définit un modèle de développement basé sur la *transformation de modèle*. Cette méthodologie de développement propose de gagner un niveau d'abstraction par rapport aux approches classiques en permettant de manipuler les modèles de l'application pour obtenir par raffinements successifs un modèle implémentable. Néanmoins, cette approche sera surtout supportée par UML2.0, et sera basée sur un langage de transformation de modèle dont la spécification finale n'est pas encore connue. Il est donc encore un peu tôt pour juger des résultats de cette approche.

Pour l'instant, le principal intérêt d'UML dans le cadre d'un développement basé sur les modèles est de permettre la définition de profils. Ils permettent de spécialiser UML pour une utilisation particulière, par exemple, le profil SDL [32] définit la manière dont le langage SDL peut être intégré dans une approche UML. Cette union des deux langages permet de disposer du niveau de formalisme du langage SDL dans un contexte de développement objet.

2.3.2 Critères méthodologiques

Expression des propriétés

La notation UML définit un langage d'expression de propriété : OCL [85][86] (pour *Object Constraint Language*). Celui-ci permet d'exprimer des invariants sur le système modélisé. Les contraintes OCL font partie intégrantes de la spécification UML, elles permettent de préciser l'interprétation des diagrammes en précisant des contraintes qui doivent être respectées si le comportement de l'application est correct. L'évaluation d'une contrainte OCL ne modifie donc pas l'état du système modélisé.

On distingue trois types de contraintes OCL :

- les pré-conditions : conditions qui doivent être vérifiées avant l'exécution de la méthode à laquelle elles sont rattachées ;
- les post-conditions qui doivent être vérifiées après l'exécution de la méthode à laquelle elles sont rattachées ;
- les invariants qui doivent être vérifiés pour tous les états du système.

La principale limitation de ce langage provient de l'impossibilité de définir des chaînes causales d'évènements. En terme de vérification, les invariants OCL permettent de vérifier des propriétés de sûreté (absence d'occurrence d'évènements redoutés). De ce point de vue, le pouvoir d'expression d'OCL est beaucoup plus limité que celui de logiques temporelles causales telles que LTL ou CTL [11].

Liaison avec la vérification formelle

UML est un standard industriel, il est donc largement outillé. Mais la plupart des outils ne permettent que la saisie de modèle UML et des vérifications syntaxiques ou structurelles. L'outillage d'UML souffre en fait du manque de formalisme de la notation. Il est très pratique de l'utiliser pour une modélisation compréhensible et ordonnée d'un système et de l'organisation de son code source. Cette représentation peut servir à des revues de projets et autres analyses "manuelles" du modèle, mais elle n'est pas adaptée pour une exploitation automatique. En revanche de nombreux

outils permettent de connecter UML à d'autres formalismes par l'intermédiaire des profils. C'est cette approche qui a par exemple été retenue par Telelogic pour l'intégration avec le langage SDL.

Des travaux sur la vérification de spécifications UML ont été entrepris. Ils ne couvrent généralement qu'une petite partie du langage et font souvent de nombreuses hypothèses supplémentaires sur la structure des diagrammes utilisés. Un exemple de ce type d'approche peut être trouvés dans [46] dans le domaine des systèmes temps réels. Dans ce cas, les statecharts UML sont traduits en automates compatibles avec le modèle checker UPAAL [43]. Des travaux du même type ont été effectués avec d'autres formalismes comme les sync-charts [6], LOTOS [49] ou encore promela [90]. Ce type d'approche est une forme de *compilation* d'une partie de la spécification UML pour en obtenir une représentation dans un formalisme dont la sémantique est exécutable et qui servira de base à la vérification.

Génération de code

La génération de code souffre également du manque de formalisme de la notation UML. La plupart des ateliers proposent un module de génération de code, mais le code généré est limité aux attributs des classes et aux prototypes des méthodes. De plus, il n'existe pas à notre connaissance de générateur pour les applications réparties.

2.3.3 Description de l'architecture de l'application

Les interactions entre les composants peuvent être spécifiées de manière statique (liens dans un diagramme d'architecture) ou dynamique (messages ou appels de méthodes spécifiés dans un statechart). Mais il n'existe dans la norme actuelle aucune vue du système qui les définisse précisément et de manière indépendante du comportement. La version actuelle d'UML ne peut donc pas être considérée comme un ADL [5].

Cette faiblesse est corrigée par l'introduction en UML 2.0 des notions d'interfaces des composants. Le diagramme d'architecture (*component diagram*) permet ainsi de définir les notions de *port* et d'*interface* des composants de l'application. Néanmoins, la description de l'architecture d'une application en UML reste un problème ouvert [61].

2.3.4 Synthèse sur UML

UML reste un très bon langage de structuration pour les modèles d'une application ; c'est le standard *de-facto* des langages de modélisation. Néanmoins ses faiblesses sémantiques ne permettent pas de l'utiliser comme langage de modélisation formel pour décrire précisément l'architecture et le comportement d'une application répartie. Dans le contexte des applications réparties, les principaux obstacles sont l'impossibilité de vérifier la cohérence entre plusieurs diagrammes d'une spécification, l'absence d'un véritable diagramme de description d'architecture et les faiblesses sémantiques de la description comportementale. Malheureusement, la prochaine version de la norme (2.0) ne règle que partiellement ces problèmes, il reste notamment de nombreux points mal définis au niveau des interactions entre les différents diagrammes d'une spécification [10].

En revanche, la principale force du langage est sa capacité d'adaptation via le mécanisme des *profils*. Il est alors possible de spécialiser une partie de la sémantique du langage et de l'adapter au domaine considéré. Si cette spécialisation est faite avec le niveau de formalisation nécessaire, on peut alors envisager d'utiliser UML comme une notation de structuration des spécifications formelles et utiliser des approches de traduction vers des formalismes exécutables.

Enfin, l'absence de description architecturale précise en UML est un frein au développement d'application réparties. En effet, les ADLs sont particulièrement utiles pour maîtriser une partie de la complexité induite par la répartition. Ils permettent de représenter les interactions complexes

inhérentes aux communications entre les composants distants ainsi que les aspects liés au déploiement.

Le langage UML n'est donc pas suffisant pour décrire précisément une application répartie complexe. En revanche, il peut être utilisé comme langage de spécification de haut niveau. Le couplage avec une méthode formelle peut être réalisé par traduction et enrichissement du modèle UML de l'application.

2.4 Le langage SDL

2.4.1 Présentation du langage

Le langage SDL (Specification and Description Language) a été introduit en 1980 par la section standardisation de l'ITU (International Telecommunication Union) [31]. La sémantique formelle du langage est définie dans [34] pour la partie statique, et [35] pour la partie dynamique. La partie statique du langage est basée sur une décomposition architecturale hiérarchique des composants du système. La sémantique comportementale est basée sur le paradigme des processus communiquants reliés par des files de messages. Néanmoins, cette description du langage dite *formelle* n'est pas démontrée, et aucune preuve de cohérence n'a été effectuée.

Le langage SDL est soumis à un cycle de révision de quatre ans. La dernière version du langage baptisée SDL 2000 a été adoptée fin 1999. Depuis sa création, de nombreux ajouts ont été apportés à SDL, comme par exemple l'orientation objet, la gestion des appels de procédures distantes et la gestion de variables partagées. Le résultat est un langage de description très riche et facilement intégrable dans les démarches de développement actuelles.

Structuration d'une spécification SDL

Les applications décrites en SDL sont définies sous la forme d'un système (*system*). Le système est décomposé en *agents* qui définissent les composants de l'application, ils peuvent être des *blocks* pour la description de la structure statique ou des *process* pour la description comportementale.

L'interface d'un composant SDL est précisément définie

- types de messages acceptées ;
- variables partagées ;
- interactions avec les composants extérieurs à la spécification ;
- méthodes *exportées* (pouvant être appelées depuis l'extérieur du composant) ;
- types de messages envoyés et méthodes distantes appelées.

Le comportement de chacun des composants de l'application est décrit sous la forme d'un automate hiérarchique communiquant. On dispose d'opérations de haut niveau (appels de procédure distantes, accès direct aux variables partagées, etc...). Les communications se font exclusivement de manière asynchrone par des files de messages. Les opérations de haut niveau sont traduites dans ce paradigme par des règles de réécritures formellement définies dans la description du langage [35]. SDL permet donc de modéliser des systèmes dits *réactifs*, c'est à dire basés sur des composants actifs réagissant à des stimuli extérieurs (signaux SDL).

Enfin, d'un point de vue pratique, le langage existe en deux déclinaisons : une syntaxe dite graphique et une syntaxe textuelle. Ces deux syntaxes sont équivalentes et partagent la même grammaire abstraite.

2.4.2 Eléments Méthodologiques

Lien avec la notation UML

L'orientation objet du langage permet d'intégrer SDL dans les environnements de développement classiques de type UML. Cette caractéristique est très séduisante dans un environnement industriel car UML y est largement implanté. En revanche, SDL reste un profil spécifique, et même si la dernière grammaire graphique de SDL est très proche d'UML, il subsiste de nombreuses différences sémantiques.

L'intégration de SDL avec UML a été considérée dans la définition du nouveau standard : un profil UML normalisé pour l'intégration de SDL [32] est enregistré à l'OMG. Les diagrammes SDL sont intégrés dans la notation UML et les types de données présents dans les diagrammes SDL peuvent être définis en UML à l'aide des diagrammes de classe.

Expression et vérification des propriétés du modèle

Les outils industriels basés sur SDL [88] offrent des possibilité de simulation, et de vérification par *simulation exhaustive* du modèle. La faiblesse de ces outils provient de l'expression des propriétés : il n'est possible de vérifier qu'un ensemble restreint de propriétés portant sur les états (comme la recherche de *deadlock*).

Une autre forme de vérification plus limitée est proposée : la couverture des chemins, elle permet de vérifier que tous les chemins d'exécution d'une spécification SDL ont été parcourus, au moins une fois. Cette vérification est moins contraignante que le calcul de l'ensemble des états, mais ne permet pas de prouver formellement une propriété comportementale sur l'ensemble des exécutions possibles de du système modélisé.

La génération de code

Il est possible de produire du code réparti de qualité industrielle à partir d'une spécification SDL. Le code peut être intégré sur un Système d'exploitation (temps réel, ou classique) qui sera chargé du séquencement des tâches générées. Chaque tâche générée implémente le comportement d'une instance de composant SDL. Le code généré peut être intégré dans une application existante. La seule contrainte est que les composants SDL communiquent avec le système extérieur en utilisant la même interface que celle qu'ils utilisent pour communiquer entre eux. Cette contrainte peut nécessiter un travail d'interfaçage des composants existants s'ils n'ont pas été initialement conçus pour être intégrés cet environnement.

De nombreuses plates-formes adaptées au domaine des télécommunications sont supportées telles que l'UMTS [22] ou bluetooth ; ce qui confirme la vocation de SDL à être utilisé principalement pour ce type d'applications.

Génération de tests fonctionnels

Des outils de génération de tests permettent de produire des tests fonctionnels à partir d'une spécification SDL. Ces outils sont basés sur l'intégration de SDL et des Message Sequence Chart [33]. Les cas de tests sont générés en TTCN (Testing and Test Control Notation) [21]. Ce langage a été spécialement étudié pour décrire des scénarios de tests fonctionnels. Cette méthode est couramment acceptée, et semble donner de bons résultats [64]. Elle est couramment utilisée par des outils industriels tels que Autolink [77] développé en collaboration par Telelogic et l'université de Lübeck en Allemagne.

2.4.3 Utilisation de composants existants

Chaque spécification SDL est implicitement englobée dans une entité appelée *system*. Celle-ci est un agent SDL complet disposant donc d'une interface. C'est cette interface qui définit les interactions du système modélisé avec le monde extérieur.

Il est donc possible pour des composants SDL d'exporter des éléments de leur interface, et de communiquer avec la plate-forme cible ou des composants externe par une interface définie en SDL (signaux, variables partagées, appels de méthodes).

Il est possible de spécifier le niveau de partage :

- les éléments de type *controlled* sont modifiables ou activables uniquement par les composants de la spécification SDL ;
- les éléments de type *monitored* sont modifiables ou activables par l'environnement, leur valeur peut donc changer de manière complètement asynchrone et imprévisible pendant l'exécution de la spécification ;
- les éléments de type *shared* sont partagés en lecture et en écriture par la spécification et par l'environnement.

Ce schéma de communication permet d'assurer que les interactions avec les composants développés par d'autres méthodes sont prises en compte lors de la vérification de la spécification. La sémantique de ces interactions est définie avec la même précision que le langage SDL lui même, et le comportement de la spécification SDL reste calculable. Cette solution implique que les que les types des variables transmises par les composants soient être entièrement définis à l'intérieur de la spécification SDL. Il est donc impossible d'utiliser les composants définis dans une spécification SDL pour transférer des messages dont le contenu n'est pas exprimable dans ce langage.

2.4.4 Description de l'architecture de l'application

SDL a été conçu dès l'origine comme un langage de modélisation d'applications réparties. La description statique du modèle est très proche de ce qui peut être obtenu avec un ADL : on y retrouve les notions de composants, de liens de liens et de ports de communication. La sémantique de ces éléments est adaptée au déploiement d'applications réparties. SDL dispose donc des capacités de description d'architecture attendue d'un langage de modélisation.

2.4.5 Conclusion sur SDL

SDL est parfaitement adapté au développement d'applications réparties dont il couvre aussi bien les aspects architecturaux que les aspects comportementaux. C'est un standard industriel couramment utilisé dans le domaine des télécommunications. Néanmoins, compte tenu de sa sémantique, SDL est uniquement apte à représenter des systèmes réactifs.

Les outils associés au langage permettent de l'utiliser depuis les premières phases du développement grâce à sa bonne intégration avec UML, l'implémentation par génération de code, et le test de l'application générée par les mécanismes de génération automatique des scénarios de test. SDL couvre donc de nombreuses phases du développement et de la validation d'une application répartie.

Enfin, les interactions avec l'environnement du système modélisé ont la même sémantique que les interactions entre les composants SDL, ce qui est très pratique dans le cas où l'environnement peut interagir de manière réactive, mais reste pénalisant dans le cas contraire. L'intégration de composants externes devant échanger des données qui ne peuvent pas être modélisées en SDL seront très difficile.

Compte tenu des plateforme supportées par les outils et de son orientation initiale, SDL a principalement été utilisé dans le domaine des télécommunications pour lequel il est particulièrement

adapté. Son utilisation dans d'autres domaines est parfaitement envisageable grâce à ses capacités d'ADL et son intégration avec le paradigme objet.

2.5 Le langage CO-OPN

Le langage CO-OPN [8][7] est un langage de modélisation orienté objet basé sur les spécifications algébriques et les réseaux de Petri. Il est destiné au développement de spécifications d'applications réparties.

2.5.1 Présentation du langage

Par rapport aux réseaux de Petri de plus bas niveau, CO-OPN fourni une représentation structurée du système en insistant sur la notion d'encapsulation des données. Le langage permet de définir explicitement les notions de types de données, de classes et d'instances de ces classes (objets). Les spécifications CO-OPN sont donc constituées de deux types de modules :

- les structures de données définies sous forme de spécifications algébriques ;
- et les classes définies par un réseau de Petri algébrique spécifiant les objets actifs du système.

Tous les modules d'une spécification CO-OPN ont la même structure globale : une interface pour la déclaration des éléments exportés, et un corps contenant leur implémentation.

Les classes et objets

Une *classe* est une entité indépendante qui exporte une interface définie par un ensemble de services accessibles depuis l'extérieur. En CO-OPN/2, un objet est un réseau de Petri algébrique [68] dont les places sont les états internes ; ce sont des multi-ensembles de valeurs algébriques. Les transitions paramétrées définissent les *méthodes* de l'objet définissant son interface avec le reste du système modélisé. Les autres transitions correspondent à des évènements internes de l'objet et ne sont pas visibles de l'extérieur. Les instances de classes (ou *objets*) peuvent être créées dynamiquement.

Les objets CO-OPN sont actifs, les différentes instances évoluent donc en parallèle. De plus, chaque instance peut définir des activités concurrentes. Le langage utilise une sémantique de parallélisme "vrai", par rapport à l'expression du parallélisme par entrelacement. Les communications entre les objets sont synchronisées. Elles sont réalisées par le franchissement d'une transition paramétrée (ce qui peut être considéré comme un appel de service sur cet objet). Le langage définit trois opérateurs principaux de synchronisation (simultanéité, séquence et l'alternative non déterministe).

Héritage et sous-typage

Le langage CO-OPN définit les notions de sous-typage et d'héritage en les différenciant fortement. La notion de sous-typage implémentée par CO-OPN est basée sur la possibilité de substituer tout sous-type par son super-type sans changer le comportement global de la spécification. Ce principe est implémenté en CO-OPN par une relation de bissimulation entre le super-type et son sous-type dont la sémantique est restreinte à celle du super-type [52].

La relation d'héritage est beaucoup moins contraignante. Elle est surtout vue comme un moyen de ré-utiliser des parties d'une spécification déjà écrite et de l'adapter à de nouveaux besoins sans avoir à re-développer entièrement le comportement attendu.

2.5.2 Critères méthodologiques

Intégration avec UML et génération de tests fonctionnels

Le langage CO-OPN a été conçu de manière indépendante d'UML. Néanmoins, des travaux envisagent l'utilisation de CO-OPN comme un langage intermédiaire pour l'exploitation de spécifications UML [50] [51]. Ces travaux concernent pour le moment principalement la génération de tests fonctionnels pour spécification UML.

Ce travail passe par la génération d'une spécification CO-OPN à partir de la spécification UML de l'application. Si ces techniques ne sont pas encore utilisées dans le cadre des applications réparties, elles montrent la faisabilité de l'intégration de CO-OPN avec la notation UML. Néanmoins, CO-OPN n'est pas encore intégré à la notation UML et son utilisation nécessite donc l'apprentissage du langage et l'utilisation d'une méthodologie spécifique.

Expression et vérification de propriétés

Les spécifications CO-OPN sont vérifiées statiquement pour s'assurer qu'elles sont correctement définies et typées. Ces spécifications peuvent ensuite être interprétées et simulées pour étudier leur comportement. En revanche, il n'existe pas d'outil automatisé d'analyse du comportement d'une spécification CO-OPN. A notre connaissance, il n'existe donc pas d'outil permettant de vérifier une propriété sur le comportement d'une spécification CO-OPN.

Génération de code

La génération de code pour CO-OPN [14][13] produit du code concurrent exécutable sur une plate-forme centralisée. Le code produit peut être utilisé de deux manières :

– simuler le fonctionnement de la spécification pour la mise au point ;
– implémentation de la spécification dans l'environnement cible de l'application.

Le langage cible actuellement utilisé est Java. Les composants générés à partir des spécifications CO-OPN respectent l'interface définie par les JavaBeans [82] et peuvent donc être intégrés directement dans une application répartie utilisant cette plate-forme.

Il n'existe pas à notre connaissance de générateur de code réparti pour ce langage. La répartition doit être traitée par le développeur lors de l'intégration des composants générés dans l'application qui sera effectivement déployée.

2.5.3 Conclusion sur CO-OPN

CO-OPN est un langage basé sur les réseaux de Petri et les types de données algébriques. Il est orienté objet et basé sur l'encapsulation forte des données et des traitements qui leurs sont associés. Il propose de nombreuses constructions spécifiques à l'orientation objet (héritage) en faisant la distinction avec les fonctionnalités associées aux types de données (sous-typage strict).

Ses spécifications ont une sémantique formelle strictement définie et sont donc exploitables par des outils de simulation ; malgré cela, il n'existe pas d'outil de model-checking d'une spécification, ou de preuve de propriété. En revanche, un générateur de code configurable est disponible. Le code produit peut servir à la simulation de la spécification, ou à l'implémentation des composants en vue de leur intégration dans l'application finale.

2.6 Les langages basés sur l'algèbre des processus

Cette famille de langages est principalement représentée par le langage LOTOS [36]. Une implémentation de ce langage est produite et maintenue par le projet VASY [30] de l'INRIA.

2.6.1 Présentation du langage

L'argument principal avancé par ce langage est de disposer d'une base sémantique forte (basée sur l'algèbre des processus) permettant la représentation des applications réparties communiquant de manière asynchrone. Ce langage fut un des premiers langage de modélisation normalisé par l'ISO [36]. Depuis, il a évolué de manière parallèle en deux versions : E-lotos [24] qui a également été standardisé ISO [39], et LOTOS-NT qui est un sur-ensemble de E-LOTOS destiné à rendre le langage plus facile à utiliser. C'est ce dernier que nous présenteront brièvement car il dispose sensiblement du même pouvoir d'expression qu'E-LOTOS. Le manuel d'utilisation du compilateur LOTOS-NT [79] cite les différences suivantes entre ces deux langages :

- surcharge de noms de fonction : il est possible d'avoir plusieurs fonction portant le même nom si les types de leurs paramètres ou leur valeur de retour sont distincts ;
- les fonctions et processus peuvent avoir des paramètres en mode "in", "out" et "in out", ceci afin de pouvoir facilement relier les spécifications LOTOS-NT à des interfaces CORBA décrites en IDL [56] ;
- style de programmation impératif pour LOTOS-NT et fonctionnel pour E-LOTOS.

Structuration des spécifications LOTOS-NT

Les spécifications LOTOS-NT se présentent sous la forme d'un ensemble de fichiers sources. Il est possible d'utiliser un mécanisme d'inclusion de fichiers sources relativement proche de celui du `C` pour construire une spécification complète à partir de ces fichiers.

Le langage prévoit possibilité de structurer les spécifications sous forme de modules. La structure et l'utilisation des modules LOTOS-NT est très proche des structures définies par les langages de programmation structurés. Chaque module peut définir une partie publique spécifiant l'interface des éléments exportés ainsi que d'une partie privée pour les déclarations internes et l'implémentation de l'interface.

Un mécanisme de généricité permet de paramétrer chaque module en fonction de types de données définis lors de l'utilisation du module. Ce mécanisme est identique aux modules génériques du langage Ada [37].

Communication entre les processus

En LOTOS, l'unité de base définissant le composant d'un composant est le processus. Les communications entre processus sont réalisées par *rendez-vous* sur des *portes*[2] définies pour chaque processus.

Le mécanisme de rendez-vous peut être utilisé pour effectuer un point de choix entre plusieurs *portes* grace à l'opérateur "`[]`" qui définit un choix non-déterministe entre les portes. Il est possible d'associer une garde à chaque opération de rendez-vous ; ce mécanisme peut notamment être utilisé pour spécifier des informations temporelles sur le rendez-vous. Il est donc possible de définir des temps maximums d'attente (*time-out*) exprimées en temps logique. Les portes sont typées au niveau de leur déclaration par le type de données qu'elles transportent.

De plus, il est possible de définir des opérations de compositions entre les processus eux même. On dispose principalement de deux opérateurs pour exprimer le parallélisme et les synchronisation (ou l'absence de synchronisation) entre plusieurs processus :

- l'opérateur de *synchronisation* définit une exécution parallèle de deux processus pouvant se synchroniser sur un ensemble de portes ;

[2] de l'anglais *gates*

– l'opérateur d'*entrelacement*[3] qui permet d'exprimer la composition parallèle de plusieurs processus ne communiquant pas directement entre eux.

2.6.2 Critères Méthodologiques

Expression et vérification des propriétés du modèle

Le projet VASY a produit un ensemble d'outils de compilation et d'analyse des spécifications LOTOS et LOTOS-NT : CADP [40] [25] et TRAIAN [79]. Les propriétés à vérifier sont exprimées sous forme de logique temporelle (CTL). La vérification est réalisée par simulation exhaustive (*model checking*).

Les outils développés autour de LOTOS sont couramment utilisés par des universités ou centres de recherche. Ils ont permis de réaliser plusieurs études de cas telles que la validation de plusieurs couches du protocole `fire-wire` [80] [78] qui a permis de détecter des ambiguïtés dans sa spécification.

Génération de code

Les générateurs de code utilisant le langage LOTOS produisent principalement du code de simulation, et de vérification. Si le modèle est approprié, il est possible d'utiliser le code généré pour la vérification formelle dans un environnement d'exécution indépendant, et donc de l'utiliser comme implémentation de la spécification, néanmoins, cette opération n'est pas l'objectif premier. De plus elle est difficile à réaliser dans le cas d'applications réparties dans la mesure où aucune information de déploiement n'est fournie par le langage.

Il n'existe pas à notre connaissance d'environnement destiné à supporter l'exécution du code généré à partir d'une spécification LOTOS(-NT) dans un environnement réparti. Des recherches avaient été entreprises dans cette voie dans les années 1990 [9], mais elles ne semblent pas être poursuivies.

2.6.3 Utilisation de composants existants

Le langage LOTOS-NT intègre la notion de *composant externe* : les types de données et les modules peuvent avoir une implémentation externe en C.

Au niveau des types de données, la spécification LOTOS du type doit être donnée, en lui ajoutant le pragma `!external`. Ce dernier spécifie que l'implémentation du type et des opérateurs qui lui sont associés est fournie par un module externe. L'opérateur de comparaison du langage peut également être surchargé pour un type externe, le nom de la fonction implémentant effectivement la comparaison est spécifié à l'aide du pragma `!comparedby`.

Le langage LOTOS permet également d'introduire des composants externes au niveau des modules définis dans la spécification. Dans ce cas, seule l'interface du module doit être définie. Cette fonctionnalité permet d'intégrer des composants complets dans la spécification.

2.6.4 Description de l'architecture

Le langage LOTOS n'est pas un langage de description d'architecture. Il n'existe pas de vue spécifiquement dédiée à la description des interactions entre les éléments de l'application. La notion de composant n'est pas définie explicitement, et la définition des ports d'interaction entre les

[3] de l'anglais *interleaving operator*

processus de l'application reste fortement liée à l'aspect comportemental. De plus, aucune information de déploiement ou de configuration des composants n'est intégrable dans les spécifications comportementales.

L'usage principal de LOTOS reste donc la définition comportementale de la spécification. L'architecture logicielle à mettre en œuvre pour l'implémentation finale de l'application doit être définie indépendamment ; elle ne peut être déduite directement de la spécification LOTOS.

2.6.5 Conclusion sur les langages de la famille LOTOS

Le langage LOTOS fournit une base formelle solide pour la vérification formelle. Les aspects liés à la prise en compte et à la réutilisation de code externe sont très présents et parfaitement intégrés au langage. Il est possible de relier des modules de code existant au sein d'une spécification LOTOS. Enfin, le langage est relativement riche et permet une bonne structuration des spécifications.

La structure même du langage LOTOS (notamment le mode de communication entre processus) semble plus le destiner à la spécification de protocoles de bas niveau. L'étude de cas présentée dans [80] montrent qu'il est possible de modéliser en LOTOS des protocoles industriels réels, et de vérifier la correction de la spécification. Néanmoins, cette spécification ne joue aucun rôle dans la mise en œuvre de son implémentation. Cela signifie que le modèle reste un moyen de vérification et n'intervient pas directement dans le processus de production de l'application. Dans le cas d'une application logicielle, le langage LOTOS reste pour l'instant cantonné aux phases amont du cycle de développement.

LOTOS ne semble pas adapté à la spécification d'applications réparties de plus haut niveau. Les communications entre processus sont de relativement bas niveau (Rendez-vous) ; il n'intègre aucune description architecturale ou information sur le déploiement du modèle. Le langage permet donc de décrire précisément le parallélisme d'exécution mais reste relativement inadapté pour la description d'applications réparties à large échelle.

2.7 Les langages de description d'architecture

Pour cette étude, nous avons choisi d'étudier AADL (pour *Architecture Analysis & Design Language* [76]. Ce langage est issu des travaux réalisés sur Meta-H [75] qui a été utilisé avec succès pour de nombreuses études de cas industrielles [63]. Dans le cadre de notre étude, AADL est particulièrement intéressant car il est orienté vers le développement d'applications pour l'avionique modulaire et illustre donc bien la réponse à un besoin exprimé pour le développement d'applications réparties soumises à des contraintes de fiabilité fortes.

2.7.1 Présentation du langage AADL

AADL ne vise que la description des aspects architecturaux de l'application. Le comportement des entités n'est pas spécifié directement en utilisant ce langage. Les spécifications sont structurées sous formes de paquetages (*packages*) définissant un espace de nommage pour les entités qui y sont définies. Ils rassemblent un ensemble de composants, leur implémentation, leurs canaux de communication, ainsi que les librairies annexes qui leur sont associées.

Le langage AADL définit deux sortes de composants : les composants logiciels (*software components*) , et les composants de plate-forme d'exécution (*execution platform components*).

Composants logiciels

AADL distingue les types de composants logiciels suivants :

– données (*data*) : définition de l'interface d'un composant implémenté ;
– *thread* : processus léger ;
– groupe de thread (*thread group*) : ensemble de thread d'un même processus ;
– processus (*process*) : définition d'un espace d'adressage isolé.

Les composants AADL de type *data* représentent un type de donnée qui sera implémenté dans le système final. La structure du code source correspondant est précisée dans la partie implémentation du composant.

Les processus légers (*threads*) représentent une unité d'exécution séquentielle. Les groupes de threads rassemblent plusieurs threads partageant un ensemble de caractéristiques d'exécution commune.

Enfin les composants de type *processus* représentent des processus systèmes. Un processus définit un espace d'adressage virtuel protégé partagé par toutes les threads qu'il crée. Au moins une thread doit être déclarée à l'intérieur du processus pour que celui-ci soit exécutable. L'ensemble des fichiers sources reliés à un processus doivent permettre la construction d'une image exécutable autonome.

Composants de la plate-forme d'exécution

Les plate-formes d'exécution sont définies à l'aide des types de composants suivants :
– processeur (*processor*) : représente une unité d'exécution ;
– mémoire (*memory*) : une zone de stockage de donnée ;
– bus (*bus*) : représentent les moyens de communication existant entre les différents processeurs ;
– périphérique (*device*) : une entité permettant de communiquer avec l'extérieur du système.

Un composant de type *processeur* définit une abstraction du matériel et du logiciel nécessaires à l'exécution d'un ensemble de processus. Ils peuvent contenir des mémoires, ou y accéder par l'intermédiaire de bus. Chaque processeur doit au moins définir une zone de mémoire locale, ou un accès à un bus de communication.

Un composant de type *mémoire* définit une zone de stockage volatile ou non. Les données contenues sont accessibles aux threads des processus qui s'exécutent sur le processeur auquel la mémoire est rattaché. Le contenu d'une mémoire peut être fournie à tout processeur qui y est relié par un bus.

Un composant de type *bus* permet d'échanger des flots contrôle et de données entre les processeurs, les mémoires et les périphériques. Ils permettent de relier des composants logiciels instanciés sur des processeur différents.

Les composants de type *périphérique* représentent les moyens de communication entre le système et l'extérieur. Ils peuvent nécessiter l'utilisation de logiciels de traitement dédiés (drivers) et peuvent eux même être des systèmes complexes : un périphérique peut lui même être composé de plusieurs processeurs exécutant un logiciel complet. L'implémentation des périphériques n'est pas modélisée dans la spécification AADL.

Fonctionnalités et interfaçage des composants

L'interface des composants est spécifiée à l'aide de fonctionnalités (de l'anglais *features*). On distingue les types fonctionnalité suivants :
– les ports (*port features*) qui permettent les communications avec les bus ;
– et les sous-programmes (*subprogram features*) et leurs paramètres qui définissent l'interface du composant ;
– les accès aux sous-composants (*subcomponent access*) qui permettent d'exporter un sous-composant.

Les ports d'un composant définissent les points de communication par lesquels des données (entrantes et sortantes) sont susceptibles de circuler. Le sens de circulation des données est explicitement spécifié par la déclaration de chaque port. Ils définissent l'interface de communication du composant et permettent l'activation des sous-programmes par les autres composants de la spécification ainsi que l'appel de sous-programmes sur d'autres composants.

Les sous-programmes définissent l'interface fonctionnelle du composant et peuvent être appelés depuis l'extérieur. Parmis les sous-programmes, on distingue les sous programmes serveurs (*server subprograms*) qui disposent de leur propre thread d'exécution et qui peuvent être activés par un appel de procédure distant.

Enfin, les accès aux sous-comopsants permettent de définir des communications par accès partagés. Cette déclaration permet de rendre publique l'interface d'un composant *data* ou *bus* défini à l'intérieur d'un composant de plus haut niveau.

librairies annexes

Les librairies annexes sont un moyen de spécialisation et d'adaptation du langage en fonction des besoins d'un domaine d'application ou d'un projet particulier. Elles permettent d'insérer des constructions de langages définis de manière externe au standard dans une spécification AADL. Par exemple, la définition d'une annexe OCL permet de définir des contraintes OCL sur les composants d'une spécification AADL.

Expression du comportement dynamique

AADL est exclusivement un langage d'architecture, il est donc focalisé sur la description statique des liens entre les composants et le déploiement de la spécification.

La description du comportement des composants reste donc limitée à la spécifications de propriétés statiques standardisées telles que le nombre maximum de threads que peut contenir un processus.

En revanche, le langage définit les moyens de liaison entre les spécifications AADL et l'expression du comportement des différents composants, soit par l'intrmédiaire d'annexes spécifiques à un projet (mécanisme d'extension du langage), soit par l'intermédiaire des composants de type *data* qui permettent de relier des composants AADL à une implémentation dans un langage externe.

2.7.2 Critères Méthodologiques

Intégration avec la notation UML

L'intégration d'AADL dans une approche UML est possible grace à l'existence d'un profile UML dédié [20]. Néanmoins, AADL reste un langage dédié à un domaine d'application pour lequel l'intégration avec UML n'est pas une priorité.

Expression et vérification de propriétés

Le langage AADL permet de spécifier un ensemble de propriétés (*properties*) sur les types de composants définis pour modéliser l'application. Le langage AADL définit un ensemble de propriétés standardisées qui ne peuvent être modifiées, ainsi qu'un ensemble de propriétés spécifiques à chaque projet que le modélisateur peut définir en fonction des besoins de l'application. Néanmoins, l s'agit d'attributs susceptibles d'intervenir sur le fonctionnement du système, et pas de propriétés comportementales vérifiables sur la spécification.

Si des contraintes supplémentaires ou des assertions doivent être spécifiées, il est possible d'utiliser les librairies annexes pour les définir.

2.7.3 Intégration de composants existants

AADL est conçu dès l'origine pour permettre l'intégration de composants d'origines diverses. Ces composants sont intégrés par les composants de type `data` qui permettent une représentation précise de l'interface de chaque composant. Les composants intégrés peuvent être réalisés dans le formalisme le plus approprié à leur description, ce qui permet de les intégrer sous forme de code source, mais également sous forme de modèles formels (comoposants SCADE ou Simulink pour l'avionique). Le langage AADL impose de définir précisément l'interface de chaque composant externe ainsi que son déploiement sur l'architecture d'exécution.

De plus, AADL permet d'intégrer des systèmes existants complets par le mécanisme des périphériques (*devices*). Ces dernies sont utilisés pour représenter des systèmes arbitrairement complexes représentés par leur interface. Ce mécanisme permet de gérer les problèmes d'interaction entre des systèmes complexes.

2.7.4 Description de l'architecture

AADL est dédié à la description de l'architecture d'un système ; il est orienté vers la description de systèmes répartis. Une spécification AADL fourni trois informations complémentaires sur le système modélisé :

- la description des composants logiciels de l'application et leurs interactions ;
- la description de l'architecture matérielle servant de support d'exécution ;
- la description du déploiement des composants logiciels sur ce support d'exécution.

La description des composants logiciels suit une logique d'ADL : elle s'attache principalement à la description des interfaces des composants ainsi qu'à la manière dont ils sont reliés. La description des composants matériels suit le même principe adapté à la description des composants d'exécution (processeurs), des mémoires qui leur sont associés et des bus de communication qui les relient. Enfin le déploiment permet de définir le positionnement de chaque composant sur cette architecture d'exécution.

2.7.5 Conclusion sur AADL

AADL vise clairement un marché très précis : le développement d'applications modulaires dans un cadre embarqué. Le haut niveau de criticité des applications visées, ainsi que leur environnement particulièrement contraint justifient l'utilisation d'une notation spécifique. Les aspects méthodologiques associés seront particulièrement importants dans la mesure où la certification des applications produites est une problématique importante dans le domaine d'application visé par le langage.

L'objectif d'un langage de description d'architecture est avant tout d'exprimer les relations entre les composants de l'application. Ces interactions deviennent rapidement un point dur du développement dans le cadre d'applications réparties ou modulaires car elles sont construites par assemblage de composants dont les interfaces doivent être compatibles. C'est pour cette raison que le développement de l'avionique modulaire a conduit à la définition d'un langage de description d'architecture particulièrement précis sur la définition de l'interface entre les composants de l'application, mais également sur la description de l'environnement cible et de la manière dont les composants sont déployés.

La norme AADL évoque explicitement des points méthodologiques comme la génération automatique de code. L'objectif du code produit est double : relier entre eux les composants de

la spécification en fonction de leur interface et déployer ces composants sur un environnement d'exécution cible. AADL constitue une approche orientée avant tout sur le déploiement et la spécification de l'architecture de l'application afin de permettre une expression claire des liens de communications entre ses composants. Seules les interfaces des différents modules sont spécifiées. Les aspects comportementaux ou liés au contenu des données échangés sont décorélés de la spécification AADL et traités dans un autre formalisme.

Cette ouverture vers d'autres formalismes de spécification est une fonctionnalité très intéressante. Ainsi AADL propose un mécanisme d'extension du langage par un système d'annexes, et la possibilité de relier entre eux des composants écrits dans d'autres langages. Ainsi chaque composant peut être développé dans le formalisme le plus adapté, et intégré à l'application en décrivant son interface en AADL.

2.8 La notation L*f*P

L*f*P (**L**anguage *f*or **P**rototyping) est une notation graphique de modélisation d'applications réparties basée sur les automates communiquants. Elle a été développée par Dan Regep [67]. L'objectif de la notation **L*f*P** est de fournir un moyen de modéliation des interactions existant entre les comopsants d'une application répartie. Il s'intéresse donc à la partie contrôle de cette application, sans préjuger des opérations réalisées par les composants de la partie métier.

2.8.1 Présentation de la notation

La notation **L*f*P** définit plusieurs vues du système à modéliser ; les principales sont la vue architecturale qui permet de définir la structure de l'application modélisée, et la vue fonctionnelle qui permet de définir le comportement des composants précédemment définis.

Vue architecturale du système

La vue architecturale permet de décrire statiquement la structure du système en utilisant les trois types de composants suivants :

- les classes qui définissent les composants d'interaction avec la partie traitement de données de l'application ;
- les médias qui définissent les protocoles d'interactions entre les classes du modèle ;
- les binders qui définissent point d'interaction entre les classes et les médias.

Les *classes* définissent les composants du système modélisé. Elles définissent des acteurs dont l'interface est spécifiée dans la vue comportementale. Les *médias* implémentent les protocoles d'interactions entre les classes du modèle. Ils assurent le transport et le routage des messages échangés entre les classes. Les *binders* sont les ports d'interaction entre les composants de la spécification. Ils représentent les canaux d'échanges de messages entre les classes et les médias.

La complexité des protocoles d'interaction que l'on peut rencontrer lors du développement d'une application répartie est prise en compte au niveau des médias dont le comportement est définit de manière explicite dans la spécification. En intercalant les médias entre les classes du modèle, **L*f*P** permet donc de définir des liens de communication arbitrairement complexes entre les composants de l'application. Cette approche est basée sur la séparation entre les aspects liés au protocole d'utilisation d'un composant (la classe), et les aspects liés au routage des messages vers leur(s) destinataire(s) qui sont traités par les médias.

Muni de ces abstractions, **L*f*P** dispose donc du pouvoir d'expression d'un ADL dont la structure est prévue pour la représentation des applications réparties construite autour de protocoles d'interaction complexes.

Vue fonctionnelle du système

Le comportement des composants **L*f*P** est défini par un automate séquentiel dont l'interface est précisément définie. Afin de faciliter l'écriture des spécifications, les automates ont une structure hiérarchique basée sur le raffinement des transitions. La notation dispose des instructions de structuration du flot d'exécution classiques dans les langages de programmation, ainsi que d'instruction spécifiques pour le traitement des messages de communications.

La structure des automates utilisés pour spécifier le comportement d'un composant diffère selon sa nature. Les automates associés aux classes **L*f*P** permettent de définir explicitement une interface d'accès au composant. Cette interface est définie par l'ensemble des méthodes activables déclarées par la classe, et par l'ensemble des ports de communication qui lui sont associés. L'automate de comportement de la classe définit les méthodes activables au cours de l'exécution, ainsi que les traitements associés (corps de la méthode).

L'interface d'un média est uniquement constituée de l'ensemble des ports de communications auxquels il doit être connecté. L'automate de comportement défini la manière dont les messages sont routés entre les ports de l'interface.

2.8.2 Critères méthodologiques

Intégration avec la notation UML

Un des objectifs de conception de la notation **L*f*P** est de la rendre compatible avec UML. Il existe donc une correspondance simple entre les principaux éléments structurels d'une spécification UML, et la notation **L*f*P**. Dans le cadre de nos travaux, la notation définie par [67] doit être remaniée. Lors de la définition de la nouvelle version de **L*f*P** réalisée au chapitre 3, nous devons donc prendre en compte la nécessité de définir un profil UML adapté à une utilisation industrielle. La définition de ce profile est en cours dans le cadre du projet MORSE [89].

Expression et vérification de propriété

Les outils d'expression et de vérification de propriété ne sont pas encore définis pour la notation **L*f*P**. L'objectif de la notation est de permettre la vérification de formules de logique temporelle LTL ou CTL en utilisant des techniques de *model-checking*.

2.8.3 Description de l'architecture

L*f*P définit une vue dédiée à la description architecturale de l'application. Celle-ci est destinée à la représentation de l'architecture d'applications répartie et insiste sur la séparation entre les aspects liés au comportement des composants et ceux liés au comportement des liens de communication.

Cette représentation est particulièrement adaptée à la description d'application réparties mettant en œuvre des protocoles d'interaction complexes. De plus, la séparation entre les aspects liés aux liens de communication et le comportement des composants du modèle permet de simplifier la modélisation.

2.8.4 Intégration de composants extérieurs

La notation **L*f*P** permet d'intégrer des composants existants dans la spécification. Pour cela, elle définit des types *opaques* définis par leur interface. Chaque composant **L*f*P** peut faire appel aux fonctions définies par l'interface d'un type opaque. La spécification **L*f*P** est donc proactive par

rapport aux composants externes : c'est l'exécution des composants **L*f*P** qui définit le protocole d'interaction avec les composants externes.

Ce schéma d'interaction a été choisi car c'est celui qui a le moins d'influence sur la vérification formelle et évite l'explosion combinatoire. Pour cela, il faut garantir qu'un composant externe ne modifie pas l'état d'une variable de la spécification.

2.8.5 Conclusion sur L*f*P

L*f*P est une notation construite pour supporter le développement par prototypage d'une application répartie. L'intégration de l'environnement extérieur n'est pas encore précisément défini dans le langage même si cette possibilité est évoquée dans [67].

Cette notation est encore en cours de développement, néanmoins elle fournit une base très intéressante pour la recherche sur les applications réparties. Un de ses points forts réside dans la possibilité de définir simplement des protocoles d'interaction très complexes entre les composants en séparant les aspects liés au routage des messages de ceux liés au protocole d'interaction des classes. Cette approche permet de représenter simplement des protocoles d'interaction très compliqués.

Malgré ses aspects prometteurs, **L*f*P** n'est pas défini de manière suffisamment précise dans [67] ; de plus, de nombreux éléments de la sémantique du langage sont relativement compliqués et peuvent être simplifiés. Nénmoins, certains aspects de **L*f*P** tels que les *médias* sont très novateurs et doivent permettre d'améliorer la modélisation des applications réparties. En conséquence, il est intéresant de développer la notation **L*f*P** pour en faire un langage de modélisation à part entière et l'intégrer dans une démarche de développement basée sur le prototypage par génération de code, et la vérification formelle.

2.9 Synthèse des approches proposées

Cette section présente une synthèse des points spécifiques des approches que nous venons d'étudier. Notre objectif était de fournir un panorama des techniques et langages permettant de développer des applications réparties, et de définir les aspects du cycle de développement couverts par chacune de ces approches en fonction de besoins identifiés.

Nos principaux critères d'évaluation étaient les suivants :

- intégration dans une méthodologie standard (UML) ;
- possibilité d'expression de propriétés sur le modèle ;
- possibilité de vérification formelle des propriétés exprimées ;
- possibilité de génération de code.

Le tableau suivant récapitule les critères remplis par les différentes approches que nous avons étudiées.

Approche présentée	Synthèse	
UML	Méthodologie associée	Avec sa méthodologie associée (le RUP) c'est le plus utilisé dans le monde industriel. C'est une approche générique adaptée pour la réalisation de modèles de haut niveau.
	Expression de propriétés	OCL permet d'exprimer des invariants sur le système, il fait partie du standard de définition de la notation.
	Vérification formelle	Pas de vérification formelle directe des spécifications UML. Il existe des approches de vérification fonctionnant par traduction d'un sous-ensemble des diagrammes UML vers une notation formelle.

	Génération de code	La génération de code est souvent limitée aux squelettes des classes du modèle. Pour aller plus loin, il est nécessaire de définir des profils spécifiques à un domaine particulier.
	Autres points	Notation standard dans l'industrie et facilement adaptable par l'utilisation de profils spécifiques à un domaine.
LOTOS	Méthodologie associée	C'est un langage de modélisation indépendant de toute méthodologie pré-établie spécialement conçu pour la modélisation de protocoles.
	Expression de propriétés	Formules de logique temporelles (liées aux outils d'analyse)
	Vérification formelle	Vérification par *model checking* de formules CTL.
	Génération de code	Il n'y a pas à notre connaissance d'outil de génération de code dédié à la génération de prototypes.
	Autres points	Langage Normalisé par l'ISO
SDL	Méthodologie associée	Fortement intégré à la notation UML par l'intermédiaire d'un profil normalisé. SDL est fortement orienté vers une utilisation industrielle dans le domaine des télécommunications.
	Expression de propriétés	
	Vérification formelle	Calcul des étas accessibles, ou simulation.
	Génération de code	Génération de code réparti (asynchrone) embarqué, déployable sur des plate-formes temps-réel.
	Autres points	orienté "télécommunications", et surtout utilisé dans ce domaine
CO-OPN	Méthodologie associée	Début d'intégration à UML
	Expression de propriétés	Pas d'expression de propriétés.
	Vérification formelle	Il n'existe pas d'outil de vérification formelle de propriétés. Seul un simulateur guidé par l'utilsateur est disponible
	Génération de code	Génération de code concurent, mais centralisé.
	Autres points	Etudes en cours sur la génération de testsfonctionnels.
AADL	Méthodologie associée	Développement d'un profil UML dédié.
	Expression de propriétés	Ouverture du langage vers des formalismes d'expression de contraintes, expression de propriétés standards sur les composants
	Vérification formelle	Langage encore en cours de développeemnt, pas d'outil de vérification connnu
	Génération de code	Génération de code envisagée pour effectuer le lien entre les implémentations des composants définis dans la spécification et faire le lien avec l'environnement d'exécution
	Autres points	Langage de description d'architecture très précis incluant les aspects de déploiement.
L*f*P	Méthodologie associée	Intégration à UML envisagée par l'utilisation de profils
	Expression de propriétés	expression de propriétés de logique temporelle et d'invariants envisagée
	Vérification formelle	Sémantique définie pour permettre le *model checking* de la spécification.
	Génération de code	A réaliser, c'est le propos des travaux présentés dans ce mémoire.
	Autres points	

2.9.1 synthèse sur les aspects *langages*

Il existe un grand nombre de langages dédiés à la spécification d'applications réparties. Néanmoins, nous pouvons dégager deux tendances. La première consiste à utiliser une spécification purement comportementale, décorélée des aspects architecturaux du développement. La seconde consiste à utiliser des langages basés sur l'architecture de l'application. Cette dernière relègue les aspects comportementaux à la spécification du comportement des composants définissant l'architecture du système.

Cette seconde approche nous parait la plus intéressante car elle permet de couvrir le plus grand nombre d'étapes du cycle de développement de l'application. De plus, elle reste compatible avec une gestion de projet basée sur UML qui permet de traiter efficacement les aspects moins critiques du développement. De ce point de vue, nous pouvons donc citer SDL et AADL comme étant des approches de modélisation très efficaces. Néanmoins, ces deux langages ont des domaines d'application dédiés relativement restreints (respectivement les systèmes de télécommunication et l'avionique modulaire embarquée) bien qu'ils puissent être utilisés dans d'autres contextes.

Les approches de type LOTOS ont une base formelle très établie et sont plus génériques, mais ils ne permettent pas une structuration architecturale de l'application. Néanmoins leur base très formelles et très efficaces lors des vérifications de propriétés, en revanche, elles se montrent beaucoup moins expressives pour la modélisation, et conduisent souvent à des modèles très abstraits sans lien réel avec l'implémentation. A l'inverse, CO-OPN est un langage basé sur des réseaux de Petri de très haut niveau qui combine un fort pouvoir d'expression avec une sémantique formelle stricte. Néanmoins, les preuves de propriétés sont très difficiles à établir, et il ne propose pas d'outil de model checking. En revanche, il s'oriente vers des techniques de tests permettant de valider une implémentation. De plus, les modèles CO-OPN sont suffisamment représentatifs de l'application à produire pour permettre l'implémentation d'outils de génération automatique de code.

Les concepts développés par la notation **L*f*P** sont intéressants car ils permettent d'envisager la représentation d'applications réparties mettant en œuvre des protocoles de communications arbitrairement complexes. La notation couvre les aspects architecturaux du système en définissant les composants à implémenter, et leurs relation. Le comportement des composants est défini à l'aide d'une notation basée sur les automates communiquants. Les spécifications **L*f*P** contiennent donc toutes les informations nécessaires pour être utilisées directement dans le cycle de vie de l'application car elles représentent le système tel qu'il sera effectivement implémenté.

Afin de permettre l'exploitation des modèles, la notation **L*f*P** doit être précisée pour définir la sémantique opérationnelle des éléments architecturaux et des instructions. Il sera alors possible

- de simuler les modèles, ce qui ouvre la voie vers les techniques de model-checking ;
- de générer automatiquement du code exécutable dans l'environnement cible de l'application.

2.9.2 Synthèse sur les aspects méthodologiques

Il ressort de cette étude que l'utilité d'un langage dans le développement d'une application dépend autant du cadre méthodologique dans lequel il est utilisé que des qualités du langage lui même.

Ainsi, il existe de nombreux langages de modélisation aux qualités indéniables tels que LOTOS, ou Promela, mais ceux ci sont finalement peu utilisés car les résultats issus de l'activité de modélisation sont difficiles à prendre en compte dans le cycle de développement de l'application. Ces langages sont donc plus souvent utilisés pour la vérification de spécifications de manière indépendante d'une implémentation [80].

L'autre approche consiste à intégrer au maximum l'activité de modélisation dans le cycle de développement de l'application. Cette approche est défendue par des langages tels que SDL qui

visent à servir de base à la vérification du modèle, ainsi qu'à la génération de code.

De plus, les abstractions proposées par le langage **L*f*P** peuvent être représentées dans une notation telle qu'UML, il est alors possible d'envisager d'utiliser le mécanisme des profils pour produire le modèle **L*f*P** à partir d'une notation de plus haut niveau.

2.10 Conclusion

Ce chapitre a étudié les langages de modélisations dédiés au développement des applications réparties, ainsi que les langages de modélisation qui leur sont associés. Au delà des formalismes soutenant les langages, on distingue deux grandes familles de modèles :

- les modèles utilisés pour la vérification abstraite des spécifications ;
- les modèles utilisés directement dans le cycle de production de l'application.

La première famille de langage est utilisée principalement pour la vérification formelle des spécifications vis à vis des exigences exprimées. La seconde famille de langage permet en plus de produire des données qui seront utilisées directement pour l'implémentation (génération de code, de cas de tests, etc.). Nous souhaitons utiliser un langage de cette seconde famille.

Dans ce cadre, nous avons porté notre attention sur **L*f*P** qui permet de spécifier les aspects architecturaux et comportementaux de l'application en utilisant deux types de diagrammes dédiés, tout en maintenant la cohérence entre les deux représentations. Les concepts mis en avant par cette notation permettent :

- de spécifier des protocoles de communication variés et complexes ;
- d'intégrer des composants logiciels existants avec un minimum de contrainte ;
- de déployer les applications produites sur un large ensemble de plate-formes.

La notation a été conçue pour être produite à partir d'un ensemble de diagrammes UML munis d'un profil (défini dans le cadre du projet MORSE) permettant de masquer la complexité de la notation sous-jacente au modélisateur. Il est donc possible de l'utiliser comme langage *pivot* pour le traitement de la partie contrôle de l'application. Deux opérations principales seront réalisées sur le modèle **L*f*P** :

- la vérification formelle de propriétés comportementales issues des exigences fonctionnelles ;
- la génération de code exécutable dans l'environnement cible implémentant le modèle et respectant les propriétés prouvées à l'étape précédente.

Pour utiliser la notation **L*f*P** dans le cadre de nos travaux, il est nécessaire de la transformer en un langage de modélisation complet supportant une sémantique exécutable. En effet, la définition de la notation réalisée dans [67] reste incomplète. De plus, certains points de la notation initiale, notamment au niveau des relations entre les binders et les composants sont inutilement compliqués. Enfin, il faut définir une méthodologie d'utilisation du langage pour la représentation des applications réparties. Ce travail de définition du nouveau langage **L*f*P** et de la méthodologie de modélisation associée est réalisé au chapitre3.

Chapitre 3

La méthodologie et le langage L*f*P

3.1 Introduction

La notation **L*f*P** introduite par Dan Regep a été présentée à la section 2.8. Elle est basée sur des concepts permettant la modélisation de systèmes répartis. Néanmoins, la mise en œuvre de ces concepts n'est pas définie de manière suffisamment précise pour pouvoir exploiter automatiquement les spécifications **L*f*P**. Cette notation doit donc être re-définie afin d'en faire un langage de modélisation adapté à la description formelle de systèmes répartis.

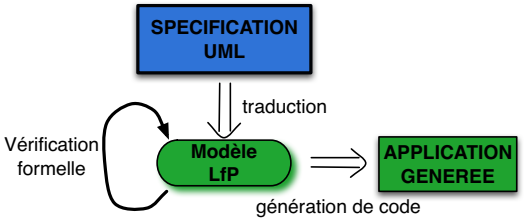


FIG. 3.1 – Introduction de la méthodologie **L*f*P**

Il est spécifiquement conçu pour le développement d'applications réparties asynchrones. La difficulté majeure de cette classe de système provient de l'absence d'horloges globale. Cette propriété entraîne un entrelacement des évènements propagés entre les composants de l'application, ce qui rend le développement des protocoles d'interaction particulièrement difficile. Pour cette raison, le langage **L*f*P** est orienté vers la représentation de la partie contrôle de l'application qui prend en charge les difficultés techniques spécifiques à l'asynchronisme.

L'objectif de ce chapitre est de définir le langage **L*f*P** ainsi que la méthodologie qui lui est associée. Cette dernière est présentée à la section 3.2. La section 3.3 présente le langage et ses concepts principaux ; la section 3.4 présente le système de typage du langage ; la section 3.5 présente les éléments du langage destinés à la description de l'architecture du système ; enfin la section 3.6. Enfin, la section 3.7 conclura ce chapitre de définition du langage.

3.2 Conception d'une méthodologie de développement pour L*f*P

Nous proposons une démarche de conception et de développement pour les applications réparties orientée modèle. Le développement «orienté modèle» [65] est centré sur un modèle de l'application destiné à plusieurs usages tels que la vérification formelle et la génération de code. Pour que ce type d'approche soit applicable dans un environnement industriel, il est nécessaire

d'utiliser le plus possible des techniques automatisées à chaque étape de la méthodologie pour cacher la complexité des techniques sous-jacentes [73].

Notre démarche est basée sur une approche par prototypage évolutif[47]. Ces dernières sont caractérisées par l'obtention rapide et à faible coût d'un prototype exécutable de l'application, destiné à être complété et amélioré au cours du développement.

L'approche que nous proposons repose sur trois techniques complémentaires :

1. des techniques de modélisation pour obtenir une description formelle de l'application à réaliser ;
2. des techniques de vérification de modèle pour valider les exigences fonctionnelles attendues ;
3. un générateur de code pour produire rapidement et à faible coût les prototypes exécutables.

Le modèle de l'application, développé en **L*f*P**, assure le lien entre la vérification formelle et la génération de code, pour cela, on le qualifiera de *modèle pivot*[67].

L'étape de modélisation repose sur l'analyse de la structure des applications réparties. La spécificité de ce type de système est de présenter une séparation forte entre deux aspects :

– l'aspect *contrôle* qui définit les interactions entre les composants de l'application et traite les contraintes liées à la répartition ;
– l'aspect *métier* qui correspond aux traitements effectués par les composants de l'application.

Définition 1 *L'**aspect contrôle** d'une application répartie définit l'ensemble des composants implémentant les protocoles de communication entre les composants de traitement de données (ou partie métier de l'application).*

Définition 2 *L'**aspect métier** d'une application répartie définit l'ensemble des composants destinés au traitement des données métier.*

3.2.1 Modélisation de l'application

La figure 3.2 présente la méthodologie basée sur le langage **L*f*P**. La première étape est le développement d'une spécification de haut niveau écrite en UML. Cette spécification doit mettre l'accent sur la séparation entre la partie *contrôle* de l'application, et la partie *métier*. Les interactions entre ces deux aspects doivent être précisément identifiés, et modélisés.

L'intégration entre le langage **L*f*P** et la notation UML est réalisée par la définition d'un profile UML spécifique pour le développement d'applications réparties. Sa définition a été réalisée en collaboration avec la société Aonix[87] dans le cadre du projet MORSE[89] L'objectif de ce profile est d'obtenir une modélisation précise de la structure de l'application, et de fournir une spécialisation de la sémantique d'UML permettant sa traduction automatisée en **L*f*P** par des techniques de transformation de modèle. Dans cette optique, le méta-modèle MOF[38] du langage **L*f*P** a été défini [55].

Les composants constituant la partie métier de l'application n'interviennent pas directement dans le contrôle de l'application, ils seront appelés par la suite composants externes car ils ne sont pas directement traités par notre méthodologie. En effet, nous souhaitons nous focaliser sur la partie contrôle de l'application qui englobe les difficultés spécifiques au développement des applications réparties. Les interactions entre les composants de contrôle et les composants externes seront réalisées par des appels externes, c'est à dire des appels aux méthodes des composants externes depuis les composants de contrôle. Cette solution permet de couvrir la majorité des interactions entre la partie contrôle de l'application et la partie traitement de données.

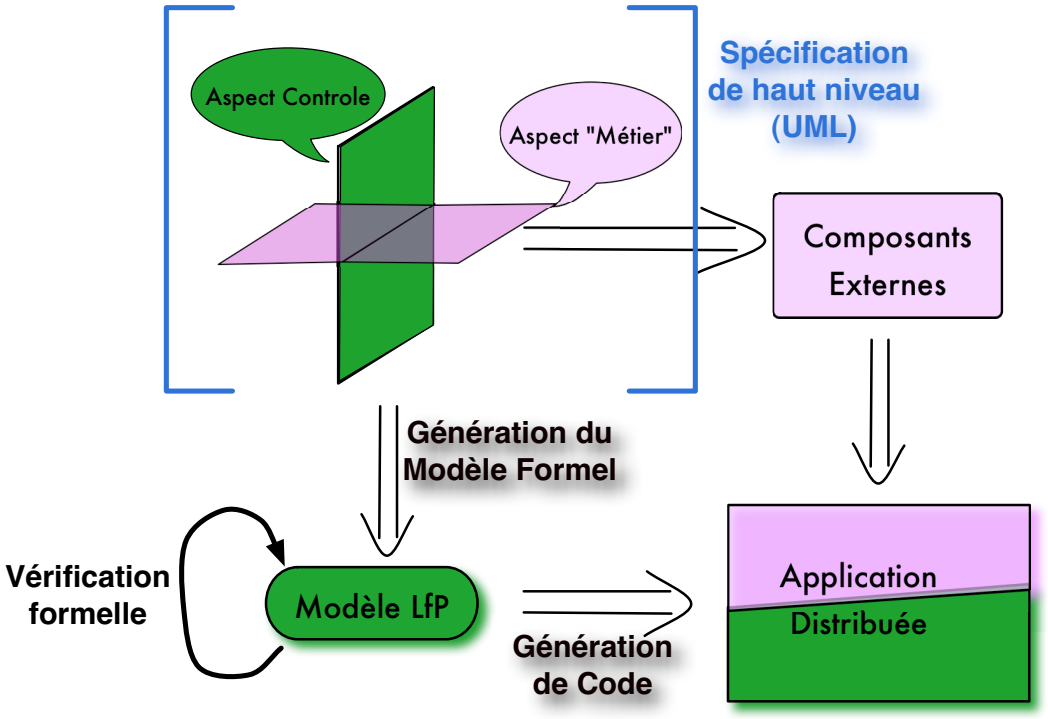


FIG. 3.2 – Méthodologie de développement associée au langage **L*f*P**.

L'interface entre la partie contrôle et la partie métier est donc définie dans la spécification UML par l'interface des composants externes. Cette interface doit être entièrement définie à l'aide de procédure ou de fonctions pouvant être appelées depuis la partie contrôle, chacun de ces appels définit un point d'interaction entre les deux aspects de l'application. Enfin, les composants externes ne doivent pas modifier l'état de la partie contrôle de l'application.

Définition 3 *Un **composant externe** est un composant*
- *appartenant à l'aspect métier de l'application ;*
- *qui ne modifie pas l'état de la partie contrôle.*

*Seule l'interface de ce composant avec la partie contrôle doit être modélisée ; elle est définie par un ensemble de fonctions et procédures que la spécification **LfP** peut appeler sur une instance de ce composant.*

Les composants externes peuvent provenir de plusieurs origines :
- composants provenant d'une version antérieure de l'application ;
- composants sur étagère fournis par un sous-traitant ;
- nouveaux composants développés pour l'application, mais à partir d'une autre méthode de développement.

L'interface des composants externes est spécifiée par un ensemble de méthodes (procédures ou fonctions) que l'aspect contrôle peut appeler.

3.2.2 Génération du modèle formel

L'étape de génération du modèle formel définie sur la figure 3.2 aboutit à la construction d'un modèle **L*f*P**. Celui-ci est construit par traduction des composants de contrôle de la spécification UML de l'application. Le modèle **L*f*P** ainsi obtenu définit donc la spécification de l'aspect contrôle de l'application.

Chaque modèle **L*f*P** définit de manière non-ambiguë les informations suivantes :

- comportement des composants de la partie contrôle de l'application ;
- interface des composants métiers ;
- interaction entre les aspects contrôle et métier sous forme d'appels aux méthodes d'interface des composants métiers.

La spécification ainsi obtenue servira de modèle pivot pour les deux étapes suivantes de la méthodologie : la vérification formelle et la génération automatique de code. Le langage **L*f*P** définit donc une sémantique sous-jacente pour les diagrammes dynamiques UML définissant les interactions entre les composants ; cette approche est très similaire à [16] en utilisant une autre base base formelle.

L'étape de traduction de la spécification UML vers le modèle **L*f*P** est fondamentale pour la méthodologie. Elle permet d'isoler la partie contrôle de l'application et de la représenter dans un formalisme disposant d'une sémantique exécutable adaptée à la représentation des applications réparties. Cette traduction permet également de vérifier la cohérence entre les informations fournies par les différents diagrammes UML utilisés. La spécification **L*f*P** définit donc un modèle de la partie contrôle de l'application exploitable directement par les outils de vérification et de génération de code. Le lien avec la partie métier est maintenu en utilisant les instructions **L*f*P** dédiées au traitement des composants externes.

3.2.3 Vérification formelle

La vérification formelle (étape 2 sur la figure 3.2) permet d'analyser le modèle **L*f*P** de l'application et de rechercher les comportements déviants par rapport aux propriétés attendues de la partie contrôle de l'application. Les techniques utilisées sont de type «vérification de modèle» (*model checking*) également appelées «simulation exhaustive». Ce travail permet de mettre au point les protocoles d'interaction entre les composants et de s'assurer qu'ils respectent leur contrat d'exécution.

3.2.4 Génération automatique de code

Lorsque le modèle **L*f*P** correspond aux exigences exprimées, le générateur de code produit automatiquement une implémentation de la spécification (étape 3 sur la figure 3.2) ; celle-ci est exécutable directement dans l'environnement cible de l'application. Le code généré correspond à la partie contrôle de l'application et intègre une couche d'interface avec la partie métier. Le générateur de code permet donc l'intégration des aspects métiers et contrôle pour obtenir une application complète. L'application finale est constituée de l'assemblage des composants de contrôle générés et des composants externes.

La génération de code réparti pour un modèle **L*f*P** est traitée dans les chapitres 4, 5 et 6. Ils définissent respectivement le déploiement du modèle **L*f*P** sur une architecture d'exécution, l'ensemble des bibliothèques nécessaires à l'exécution du code généré, et les règles de transformation permettant de produire le code.

L'intégration à la notation UML nous semble très importante car il s'agit d'un standard de fait dans le monde industriel. En revanche, l'utilisation des méthodes formelles reste relativement peu développé en raison d'un manque de personnel formé[53]. En intégrant notre approche à un langage déjà très utilisé, et en rendant la partie «formelle» aussi transparente que possible pour l'utilisateur, nous pensons pouvoir promouvoir rapidement l'utilisation de ce type de techniques dans l'industrie. Cette transparence sera assurée par l'extraction automatique de la spécification **L*f*P** à partir de la spécification UML. De même, les propriétés à vérifier sur le modèle pourront à terme être écrites dans le langage OCL (pour *Object Constraint Language*) qui fait partie du standard UML.

3.2.5 La méthodologie L*f*P dans le cadre du MDA

MDA est l'approche basée sur les modèles la plus couramment évoquée. Notre méthodologie propose un cadre d'application de cette approche destiné au développement d'applications réparties. La figure 3.3 met en parallèle l'approche MDA et les éléments de la méthodologie **L*f*P**.

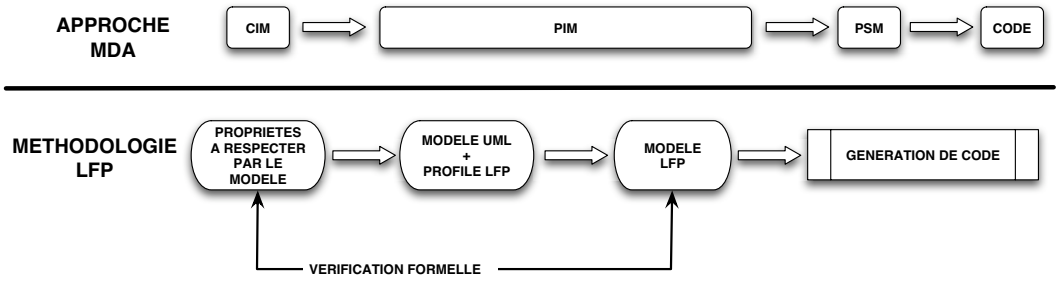


FIG. 3.3 – Correspondance entre le MDA et la méthodologie **L*f*P**

La première étape de MDA est la définition d'un modèle des exigences (ou CIM pour *Computation Independant Model*) pour que devra respecter le système. L'objectif de ce modèle est de permettre d'exprimer les besoins métiers de manière indépendante de la solution technique proposée. Dans le cadre de la méthodologie **L*f*P**, une partie de ces exigences peut être traduites en propriétés comportementales que l'application réalisée devra vérifier.

La seconde étape est la construction d'un modèle de l'application indépendant de la plate-forme cible (ou PIM pour *Platform Independant Model*). Dans le cadre de la méthodologie **L*f*P**, cette définition est en fait réalisée en deux étapes : une première version est réalisée en UML. Comme nous l'avons vu précédemment, ce modèle UML est traduit en **L*f*P** par transformation du modèle. Ce nouveau modèle définit un PIM partiel (il n'implémente que la partie contrôle) de l'application.

En effet, l'approche MDA ne fait que *recommander* [58] l'utilisation d'UML pour l'expression de ces modèles, et permet donc l'utilisation d'autres langages dans le cas où cela s'avère nécessaire. De plus, un modèle **L*f*P** est indépendant de la plate-forme sur laquelle l'application correspondante sera effectivement déployée. Un modèle **L*f*P** définit donc bien un PIM au sens MDA. La transformation du modèle UML vers le modèle **L*f*P** introduit donc une étape intermédiaire de raffinement lors de la définition du PIM de l'application.

Un des avantages de la méthodologie **L*f*P** est de permettre la vérification de la cohérence entre le PIM de l'application et une partie de son CIM. Ce mécanisme est possible par la traduction des exigences applicables à la partie contrôle dans des propriétés exprimées en logique temporelles. Celles-ci peuvent être vérifiées sur le modèle obtenu en utilisant des techniques de *model-checking*.

Dans le cadre du MDA, la vérification formelle peut être utilisée pour vérifier la conformité du PIM avec le modèle des exigences (cette vérification est explicitement demandée par l'approche MDA). L'objectif de la vérification formelle est en effet de vérifier des propriétés issues du modèle des exigences. Dans le cadre de la méthodologie **L*f*P**, on vérifiera principalement les exigences fonctionnelles concernant la partie contrôle de l'application.

L'étape suivante de l'approche MDA est la construction d'un PSM de l'application. Cette étape est implicite dans la méthodologie **L*f*P**. Elle est réalisée par le processus de génération de code qui prend en compte les spécificités de la plate-forme cible. Le PSM de l'application est donc construit implicitement par le générateur de code en fonction des règles de génération spécifiques à la plate-forme cible ; puis le code correspondant est généré. Le PSM correspondant au modèle **L*f*P** n'est donc jamais manipulé directement par les utilisateurs. Il permet de traiter le code de contrôle de l'application. La traçabilité entre le PIM **L*f*P**, le PSM et le code généré est assurée par

le générateur de code.

La méthodologie **L*f*P** est donc intégrable dans un environnement de développement basé sur l'approche MDA dont elle permet de remplir deux des objectifs principaux :

- la productivité des modèles : les modèles produits dans le cadre de notre approche ont un rôle actif dans le développement de l'application (vérification des exigences et génération de code) ;
- la pérénité des modèles : le PIM produit par l'utilisateur est appelé à être réutilisé lors des développements futurs, et pourra être implémenté sur toute plate-forme pour laquelle un générateur de code a été défini.

3.3 Présentation du langage

Le langage **L*f*P** doit être capable d'exprimer deux vues complémentaires sur le système :

- une vue architecturale définissant la topologie des connexions entre les composants de l'application ;
- une vue comportementale définissant le comportement des composants du système.

Le langage **L*f*P** définit donc deux types de diagrammes : la vue architecturale est définie par le *diagramme d'architecture* qui définit les composants de la spécification ainsi que les liens de communication qui les relient ; les *diagrammes de comportement* définissent le comportement de chacun des composants du diagramme d'architecture.

Les spécifications **L*f*P** ont donc la forme d'un ensemble de diagrammes hiérarchiques. Le diagramme de plus haut niveau est le diagramme d'architecture sur lequel tous les types de composants définis par le système modélisé doivent apparaître. La description de chacun de ces composant sera définie à l'aide d'un ensemble de diagramme de comportement dont la sémantique est basée sur les automates communiquants par des files de messages. Ces diagrammes de comportement ont eux même une structure hiérarchique : chaque transition peut être exprimée sous la forme d'un sous-automate.

3.3.1 Le diagramme d'architecture

Le diagramme d'architecture définit la vue statique du modèle. La sémantique détaillée des éléments du diagramme d'architecture est présentée à la section 3.5.

Définition 4 *On appelle **diagramme d'architecture** la description statique du modèle **L*f*P**, il définit :*

- *les composants participant à l'application ;*
- *les liens de communication entre ces composants ;*
- *les déclarations globales du modèle.*

Le diagramme d'architecture est donc composé de deux parties distinctes : un graphe orienté spécifiant les composants de la partie contrôle de l'application, et un ensemble de déclarations utilisées par ces composants.

Définition de l'architecture du modèle

On distingue deux types de composants : les *classes* et les *médias*. Les *binders* assurent la transmission des messages entre les composants du modèle. Ces trois types d'entités **L*f*P** sont visibles sur la figure 3.4.

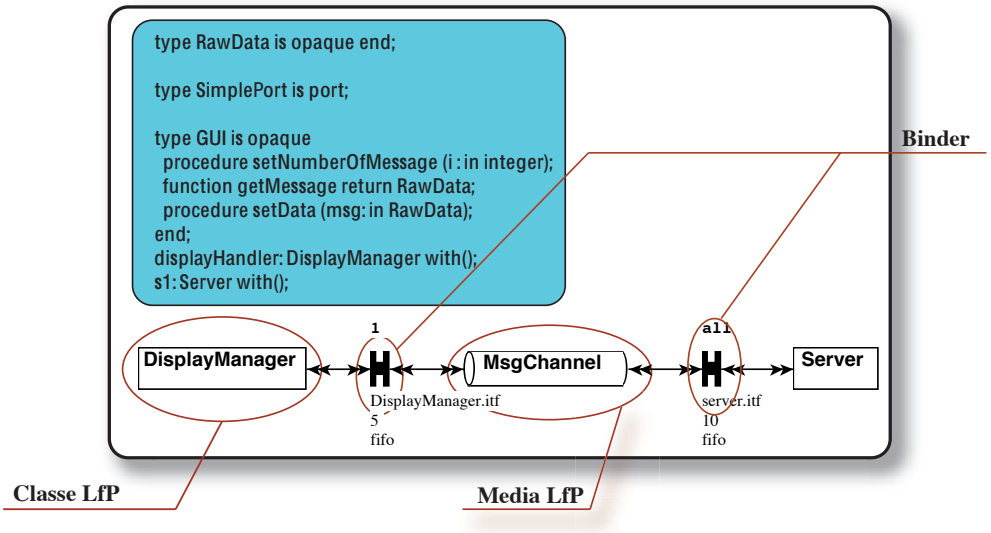


FIG. 3.4 – Un exemple de diagramme d'architecture

Définition 5 *Les **classes LfP** sont les composants applicatifs du modèle, elles assurent les liaisons entre les composants externes. Les classes sont déclarées sur le diagramme d'architecture, leur comportement est spécifié par un diagramme de comportement.*

Définition 6 *Les **médias** sont les composants réalisant les protocoles de communication entre les classes **LfP** définies pour le modèle ; ils assurent la transmission des messages entre les classes du modèle. Les médias sont déclarés sur le diagramme d'architecture, le routage des messages est spécifié par un diagramme de comportement.*

Définition 7 *Un **composant LfP** désigne indifféremment une classe ou un média du modèle.*

Sur l'exemple, on distingue deux classes et un média

- *DisplayManager* distribue des calculs sur une grappe de serveurs et récolte les résultats pour les afficher ;
- *Server* Chaque instance de cette classe définit un serveur de calcul utilisé par *DisplayManager* ;
- *MsgChannel* est le média utilisé pour définir les échanges de messages entre les deux classes précédentes.

Les médias assurent les communications entre les classes du système, ils sont donc systématiquement insérés entre les classes présentes sur le diagramme d'architecture.

Les classes et des médias sont reliés entre eux à l'aide de files de messages appelées *binder*. Sur la figure 3.4, ils sont de part et d'autre du média *MsgChannel*. Les binders définissent les files d'échange de messages utilisés pour les communications entre les diagrammes de comportement des composants.

Définition 8 *Un **binder** est une file de messages permettant la communication entre deux composants **LfP**.*

Enfin, les liens entre les composants et les binders définissent les chemins possibles pour les messages échangés entre les composants. Sur l'exemple de la figure 3.4, tous les éléments du modèle peuvent communiquer entre eux.

Déclarations du diagramme d'architecture

Sur la figure 3.4, les déclarations associées au diagramme d'architecture sont mises en évidence sur fond coloré. La portée des déclarations du diagramme d'architecture couvre tout le modèle, il s'agit d'éléments partagés par tous les composants de l'application.

On distingue trois sortes de déclarations sur les diagrammes d'architecture :
- les déclarations d'instances statiques de composant ;
- les déclarations de types ;
- les déclarations de constantes.

Les instances statiques de composants définissent l'état initial de l'application. Cet état est défini par la liste des composants instanciés lors du démarrage de l'application.

3.3.2 Les diagrammes de comportement des composants

Le comportement des composants du système est défini à l'aide de diagrammes de comportement qui définissent des automates communiquants par files de messages. Ces files sont implémentées par les binders définis sur le diagramme d'architecture. A titre d'exemple, le diagramme de comportement de la classe *Server* déclarée sur le diagramme d'architecture de la figure 3.4 est présenté sur la figure 3.5. Cette classe attend un message contenant des données à traiter, traite les données correspondantes, puis les traite à l'aide d'un composant externe, et retourne la réponse au client. Si nécessaire, un média est instancié dynamiquement pour assurer le routage du message.

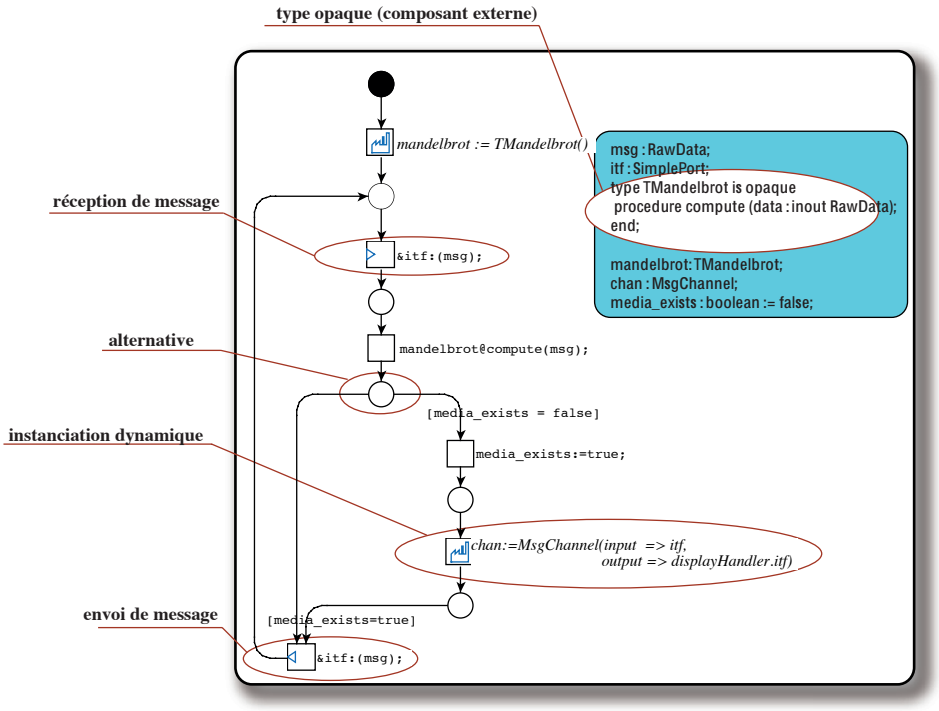


FIG. 3.5 – Diagramme de comportement de la classe *Server*

Définition 9 *Un **diagramme de comportement** d'un composant exprime le comportement d'un composant sous la forme d'un automate communiquant par files de message avec les automates des autres composants.*

Les diagrammes de comportement fournissent une interface graphique pour spécifier l'automate de comportement du composant. Afin de conserver des diagrammes de taille raisonnable, ils peuvent être définis de manière hiérarchique : il est possible d'introduire des sous-diagrammes exprimant une partie du comportement du composant.

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 3.6.

L'exécution du diagramme de comportement d'un composant **L*f*P** est séquentielle.

Eléments de base des diagrammes de comportement

Les principaux éléments des diagrammes de comportement sont :

- les *états* qui correspondent à des points de choix entre un ou plusieurs comportement possibles (alternatives) ;
- les *instructions* effectuées par le composant ;
- les *transitions* qui permettent de regrouper un ensemble d'instructions à exécuter par le composant.

les diagrammes de comportement définissent les instructions exécutées par le composant. La majorité des instructions peut être spécifié sous forme de texte associé à une transition (carré sur le diagramme de comportement). Il s'agit des instructions de structuration classiques des langages de programmation (boucles *for* et *while*, alternatives etc...).

Parmi les instructions disponibles, on peut distinguer les instructions de communication entre composants qui revêtent un caractère particulièrement important dans le contexte des applications réparties. Elles sont représentées par un carré avec un triangle indiquant la direction de la communication. On en distingue deux types :

- les instructions d'envoi / réception de message ;
- les instructions d'appel de méthode entre classes.

Les transitions textuelles permettent de spécifier une suite d'instruction directement sous une forme textuelle. Chaque transition définit un bloc d'instructions, il n'est pas possible de définir une variable locale à une transition. Une transition peut également être définie explicitement sous la forme d'un automate, elle est alors considérée comme un sous-diagramme.

Les états du diagramme définissent les états explicites de l'automate de comportement. On distingue trois types d'états :

- les états *initiaux* ;
- les états *intermédiaires* ;
- les états *finaux*.

Chaque diagramme ou sous-diagramme a exactement un état initial, au moins un état final et un nombre quelconque d'états intermédiare. L'état initial définit l'état dans lequel se trouve l'automate lors du début de son exécution. Chaque état final détermine la fin de l'exécution d'un automate ; un état final ne peut donc mener vers aucune transition. Les états intermédiaires définissent les états explicites de l'automate en cours d'exécution.

Les arcs des diagrammes de comportement sont orientés, ils représentent une instruction de saut. Chaque transition a exactement un arc sortant qui indique l'état dans lequel se trouve le composant après exécution de la transition. Un état initial ou intermédiaire a un ou plusieurs arcs sortant définissant la prochaine transition à exécuter. Les états terminaux n'ont pas d'arc sortant. Un état initial peut avoir zéro, un ou plusieurs arc entrants (retour du composant dans l'état initial). Un état intermédiaire ou final ou une transition a un ou plusieurs arc entrant.

Structuration hiérarchique des automates

Les éléments de structuration hiérarchique des diagrammes de comportement sont les suivants :

- les *sous-diagrammes* qui permettent de décomposer le contenu d'une transition sous la forme d'un diagramme de comportement ;
- 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, il peut être assimilé à une méthode privée ;
- Pour les classes, des *méthodes* qui correspondent à des points d'activation (elles sont exécutées sur réception du message d'activation correspondant).

Les *sous-diagrammes* permettent de définir la structure hiérarchique d'un automate. Les sous-diagrammes permettent de définir des variables locales, ainsi qu'une série d'instruction sous la forme d'un automate **L*f*P**. Cet automate est lui même susceptible de contenir des sous-diagrammes définissant ainsi plusieurs niveaux de hiérarchie.

Définition 10 *Un **sous-diagramme** est un diagramme **LfP** permettant de définir sous la forme d'un diagramme de comportement le contenu d'une transition.*

Les *sous-diagrammes* **L*f*P** sont nommés (implicitement ou explicitement par l'utilisateur), ils peuvent donc être la cible d'une instruction de saut (arc d'un diagramme de comportement).

Les *triggers* sont des automates nommés faisant partie du diagramme de comportement d'un composant. Ils définissent un ensemble de traitements appelables en plusieurs points du diagramme de comportement d'un composant.

Les *méthodes* sont des automates nommés faisant partie du diagramme de comportement d'une classe. Elles correspondent à des services exportés par les classes et peuvent donc être appelées par les autres classes du modèle.

3.3.3 Les types de données du langage L*f*P

Le système de typage du langage **L*f*P** est schématisé sur la figure 3.6 ; il est inspiré des langages de programmation usuels mais possède ses propres spécificités.

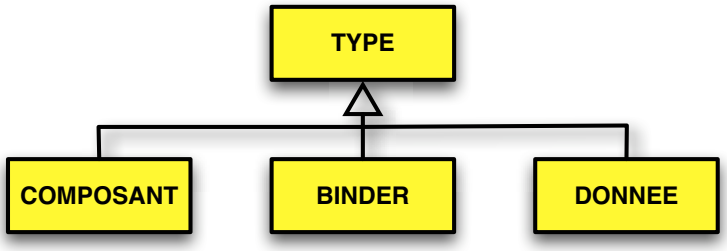


FIG. 3.6 – Les trois catégories de types définis par **L*f*P**

Le langage **L*f*P** distingue principalement trois catégories de types :

- les types composants qui rassemblent les définitions des classes et des médias du modèle ;
- le type *binder* qui représente les files de messages utilisées pour la communication entre les composants du modèle ;
- les types de données qui permettent de définir les structures de données manipulées par l'application.

Le système de typage complet du langage est présenté à la section 3.4.

3.4 Sémantique des types du langage et de leurs opérateurs

Cette section présente l'ensemble des types qu'il est possible de manipuler en **L*f*P**, ainsi que la sémantique des opérateurs qui leurs sont associés. Dans un premier temps, elle propose une présentation générale des types de données fournis par le langage **L*f*P**. Dans un deuxième temps, chaque sorte de type est détaillé dans une sous-section dédiée.

3.4.1 Présentation des types L*f*P

Nous avons vu sur la figure 3.6 que le langage **L*f*P** définit trois sortes de types : les types de données, les types composants, et les binders.

Présentation des types de données

Les relations entre les types de données sont illustrées sur la figure 3.7 qui complète la branche de droite de la figure 3.6.

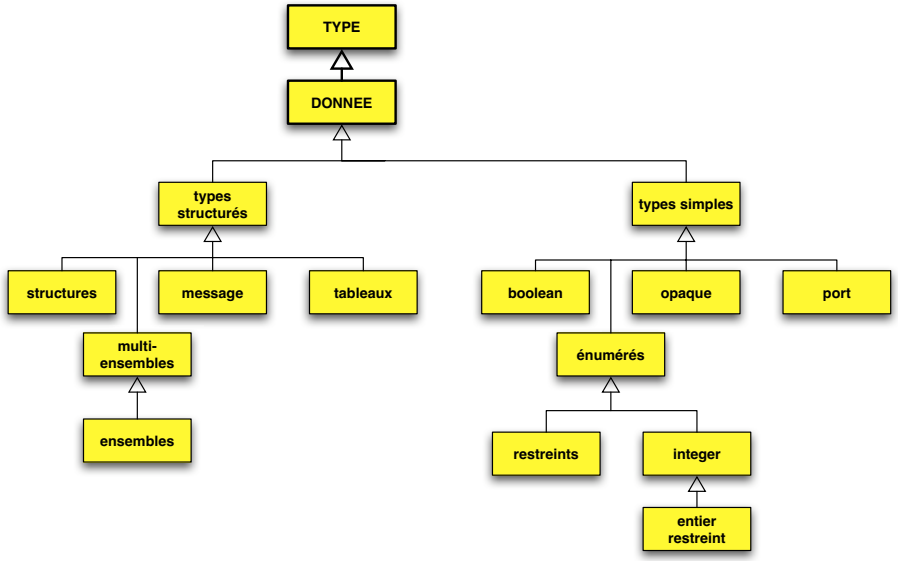


FIG. 3.7 – Relations entre les types de données du langage **L*f*P**

On distingue deux catégories de types : les types simples et les types structurés. Les types simples sont tous les types qui ne contiennent qu'une seule valeur ; les types structurés sont les types permettant de regrouper plusieurs valeurs d'un même type ou de types distincts.

Les types de données simples disponibles en **L*f*P** sont les suivants :

- **les types opaques** qui permettent de représenter les composants externes ;
- **le types `message`** qui est utilisé par les médias du système pour représenter les messages échangés par les classes **L*f*P** ;
- **le type `boolean`** qui fournit une représentation des valeurs booléennes ;
- **le type `integer`** qui fournit une représentation des valeurs entières ;
- **les types énumérés** dont la sémantique est proche des types énumérés des langages de programmation classiques ;
- **les types restreints** qui permettent de définir un sous-type d'un type énuméré de réduire la plage de définition des entiers.

Le langage **L*f*P** permet de définir des types énumérés ou des sous-types circulaires. Cela signifie que le successeur et le prédécesseur de toutes les valeurs du type est toujours défini. Ces types sont très proches des classes de couleurs circulaires des réseaux de Petri bien formés définis dans[15].
Les types structurés disponibles en **L*f*P** sont les suivants :

- tableaux (array) dont la sémantique est proche de celle des langages de programmation usuels ;
- structures ou articles (record) dont la sémantique est proche des structures des langages de programmation usuels ;
- les ensembles (set) définissent un regroupement de valeurs différentes du même types ;
- multi-ensembles (bag) définissent un regroupement quelconque de valeurs du même type (ce type est comparable à l'interface *collection* de la bibliothèque java standard).

Les types composants

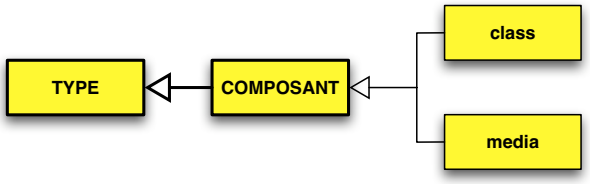


FIG. 3.8 – Les types composants en **L*f*P**

La figure 3.8 précise la branche de gauche de la figure 3.6 ; elle présente donc les types de composants disponibles en **L*f*P**. On en distingue deux types : les classes et les médias. Ils définissent les entités actives du modèle, leur exécution détermine le comportement de l'application.

Le type binder

Le type binder est un type natif du langage **L*f*P**, il correspond à la branche centrale de la figure 3.6. Ce type est uniquement disponible sur le diagramme d'architecture. Les composants du système manipulent ces entités via les types ports qui leur sont associés.

3.4.2 Les types de données

Les types opaques

Les types opaques permettent de représenter les composants externes du modèle. Ces composants représentent les parties du modèle qui ne sont pas modélisés en **L*f*P**. Les types opaques ont un fonctionnement très proche de celui des types privés définis dans le langage Ada[37] :

- l'utilisateur ne peut pas accéder à leur structure interne ;
- Ils sont uniquement définis par leur interface (liste de méthodes invoquables).

Contrairement aux composants **L*f*P**, les types opaques sont passés par valeur : lorsqu'un composant externe est utilisé en paramètre d'une méthode, la totalité du contenu du composant est dupliquée.

Les opérations définies sur les types opaques sont les suivantes :

Instanciation Le langage **L*f*P** fait le minimum d'hypothèse possible sur les types opaques. On les traite donc comme des variables locales contenant des références vers les instances de

composant réalisant leurs fonctionnalités. L'instanciation est réalisée par une instruction d'instanciation qui ne permet pas d'initialiser les attributs du composant externe.

L'instanciation dynamique d'un type opaque revient donc à créer une instance du type de donnée correspondant. Dans le cas d'un langage objet tel que java, cela donne lieu à l'exécution d'un *constructeur par défaut* qu'il est possible de définir, mais les possibilités de langages tels que le `C` sont beaucoup plus limitées. Ainsi, bien que l'interface des types opaques soit constante dans une spécification **L*f*P** quel que soit le langage cible, la manière de les implémenter varie grandement d'un langage à un autre.

Si un composant externe doit être initialisé en fonction d'un ensemble de paramètres externes à ce composant, il est nécessaire d'implémenter cette initialisation par une méthode de l'interface, et d'insérer explicitement un appel à cette fonction dans la spécification **L*f*P** avant toute autre opération sur ces composants.

Instruction d'affectation Les composants externes peuvent avoir une structure relativement compliquée. Par défaut, le langage **L*f*P** ne définit qu'une affectation simple pour les types opaques : la référence du composant est recopiée dans la nouvelle variable, mais le composant n'est pas dupliqué. Cette sémantique a été choisie car elle est compatible avec tous les langages de programmation, y compris ceux qui ne permettent pas de redéfinir les opérateurs d'affectation.

Dans le cas où une autre politique est nécessaire, il existe deux solutions :

1. implémenter explicitement une fonction externe dans l'interface du type opaque qui retourne une copie du composant ;
2. si le langage le permet : surcharger l'opérateur d'affectation du type de donnée implémentant le type opaque.

Cette solution nous permet de conserver la sémantique des spécifications **L*f*P** inchangée quel que soit le langage cible. Les spécificités liées à la plate-forme cible sont donc contenues dans l'implémentation des composants externes.

Appel d'une méthode externe L'appel d'une méthode externe correspond à l'appel de la fonction du même nom définit dans le type opaque.

Dans le cas où la fonction retourne une valeur, cette dernière doit être stockée dans une variable de même type dans la spécification **L*f*P**. Dans le cas où la valeur de retour est d'un type opaque, on utilise l'instruction d'affectation pour les types opaques.

Afin de rester compatible avec la sémantique du langage **L*f*P**, les méthodes des composants externes doivent respecter les propriétés suivantes :

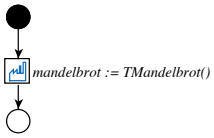
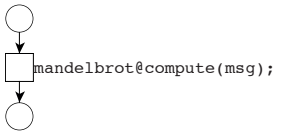
– temps d'exécution borné ;
– ne pas modifier l'état d'un composant **L*f*P** ;
– ne pas modifier l'état du protocole d'interaction entre les composants **L*f*P**.

Destruction d'une instance Les variables de types opaques sont considérées comme des variables locales du bloc où elles sont déclarées. Elles ne sont donc accessibles que depuis leur bloc de déclaration et les sous-blocs qui y sont inclus. Le langage **L*f*P** n'implémente pas de ramasse miette (*garbage collector*) pour les composants externes, ces derniers doivent être détruits explicitement par un appel à cet opérateur.

Le langage **L*f*P** ne peut pas gérer les problèmes liés aux références orphelines (référence vers une instance précédemment détruite), ou aux oublis d'initialisation des références (pointeur nul).

Exemple 1 *Opérations sur les composants externes extraites de la figure 3.5 (page 50) :*

`type TMandelbrot is opaque`
` procedure compute`
` (data : inout RawData) ;`
`end ;`
	Déclaration d'un composant externe `TMandelbrot` *disposant d'une méthode* `compute` *dont le paramètre :*
	– *est de type* `RawData` *;*
	– *peut être modifié par l'appel de méthode* `compute`*(mode inout).*

	*instanciation du composant externe.*
	*Appel de la méthode* `compute` *sur le composant externe précédemment instancié.*

Les types énumérés

Les types énumérés définissent une énumération de valeurs. Les valeurs des types énumérés sont implicitement ordonnées dans l'ordre de leur déclaration. Un type énuméré peut être circulaire, ce qui signifie que le successeur de la dernière valeur du type est la première valeur de la définition ; de même, le prédécesseur de la première valeur du type est la dernière valeur définie.

Les opérateurs suivants sont définis sur 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 ; cette opérateur est réalisée par l'attribut succ appliqué à une valeur du type.

Prédécesseur : retourne l'élément précédant du type. Si le type est circulaire, le prédécesseur du premier élément déclaré est le dernier élément déclaré ; cet opérateur est réalisé par l'attribut pred appliqué à une valeur du type.

Premier élément : retourne le premier élément déclaré du type énuméré ; cette opération est réalisée par l'attribut first appliqué au type.

Dernier élément : retourne le dernier élément déclaré du type énuméré ; cette opération est réalisée par l'attribut last appliqué au type.

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 :
– $v_1 \leq v_2 \Leftrightarrow (v_1 < v_2 \text{ or } v_1 = v_2)$
– $v_1 > v_2 \Leftrightarrow v_2 < v_1$
– $v_1 \geq v_2 \Leftrightarrow (v_2 < v_1 \text{ or } v_2 = v_1)$

Exemple 2 *La déclaration suivante définit un type énuméré circulaire :*

type jour_semaine is circular range (lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche) ;

Les égalités suivantes sont alors vérifiées :

$$jour\_semaine'succ(lundi) \quad = \quad mardi$$

$$\begin{aligned} jour\_semaine'pred(lundi) &= dimanche \\ jour\_semaine'succ(dimanche) &= lundi \\ lundi &< mardi \end{aligned} \tag{3.1}$$

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. Le type integer est un type énuméré non circulaire : si un résultat entier ne fait pas partie de l'intervalle choisi pour l'implémentation, une erreur doit être levée. En plus des opérateurs définis sur les types énumérés, le type integer définit les opérateurs arithmétiques standards.

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 définit par : $\sum_{i=1}^{b} a$

quotient : le quotient q de a par b est défini par : $q \times b + r = a$ où $q \times b$ désigne le produit de q par b.

opérateurs de comparaison : les relations d'ordre et de comparaison classiques sont définies sur les types entiers ($<, \leq, >, \geq, \neq, =$).

Les types restreints

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.

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 restreints :

Successeur : retourne le successeur de la valeur passée en argument dans l'intervalle de définition du type restreint ; cet opérateur est réalisé par l'attribut succ.

Prédécesseur : retourne le prédécesseur de la valeur passée en argument dans l'intervalle de définition du type restreint ; cet opérateur est réalisé par l'accès à l'attribut pred.

Premier élément : la définition de cet opérateur dépend du type parent :

- dans le cas d'un type défini par restriction sur les entiers, cet opérateur retourne la borne inférieure du type restreint ;
- dans le cas d'un type défini par restriction sur un type énuméré, cet opérateur retourne la première valeur définie pour le type restreint.

Cet opérateur est réalisé par l'attribut first.

Dernier élément : la définition de cet opérateur dépend du type parent :
- dans le cas d'un type défini par restriction sur les entiers, cet opérateur retourne la borne supérieur du type restreint ;
- dans le cas d'un type défini par restriction sur un type énuméré, cet opérateur retourne la dernière valeur définie pour le type restreint.

Addition : (uniquement pour les types définis par restriction sur les entiers) la somme de deux éléments a et b d'un type restreint retourne le $b^i eme$ successeur de a.

Soustraction : (uniquement pour les types définis par restriction sur les entiers) la différence de deux éléments a à b retourne le $b^i eme$ 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 $a = q \times b + r$.

Affectation : L'affectation affecte à une variable d'un type restreint, une valeur d'un type restreint compatible et réalise la conversion 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 ancêtre commun dans l'arbre de sous-typage.

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 par l'opérateur d'affectation. Cette opération est susceptible 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.

Exemple 3 *Soit le type définit à l'exemple 2 :*

type jour_semaine is circular range (lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche) ;

Les types suivants sont des restrictions valides du type jour_semaine :

type week_end is enum (samedi, dimanche) ;

type jour_ouvre is circular enum (lundi, mardi, maercredi, jeudi, vendredi) ;

On a alors les relations suivantes :

$$week\_end'succ(samedi) \quad = \quad dimanche$$

$$week\_end'succ(dimanche) \rightarrow \texttt{ERREUR} : le\ type\ week\_end\ n'est\ pas\ circulaire$$

$$jour\_ouvre'succ(vendredi) = lundi$$
$$jour\_ouvre'pred(lundi) = vendredi$$

$$lundi < samedi\ (Les\ valeurs\ sont\ de\ type\ jour\_semaine)$$

Le type `boolean`

Le type boolean permet de représenter les valeurs booléennes `true` et `false`. Les opérateurs applicables sur ce type sont les suivants :

opérateurs de comparaison : l'opérateur d'égalité ($=$) est définit pour les booléens, il retourne *true* si deux expressions booléennes sont égales. Par commodité, on définit également son contraire ($/=$) qui retourne *true* si deux expressions booléennes ont des valeurs différentes.

opérateurs booléens : on définit les opérations suivantes classiques (*and*, *or*, *not*) sur les booléens.

affectation : cet opérateur affecte à une variable de type `boolean` la valeur d'une expression booléenne.

Les types port

Les types ports sont des types référence vers les binders reliant les composants du modèle. Ils définissent

- les types des ports des composants **L*f*P** ;
- le type des discriminants des messages envoyés vers ce port.

Un type port peut être référencé avant la fin de sa définition. Cela signifie que le discriminant d'un type port peut contenir un élément du type en cours de définition.

On définit les opérations suivantes sur les ports des composants :

Envoi de message : cette opération réalise un envoi de messages sur l'instance de binder référencée par le port.

Lecture d'un message : cette opération réalise une lecture de message sur l'instance de binder référencée par le port.

Affectation : cette opération correspond à une recopie de référence, elle ne donne pas lieu à l'instanciation d'un nouveau binder.

3.4.3 Les types structurés

Le type `message`

Le type `message` est utilisé pour représenter les opérations effectuées sur les messages de l'application. Seuls les médias peuvent déclarer des variables de type message, pour les classes ce type est implicite et ne peut être utilisé directement.

Un message est constitué de deux parties :

- la partie **données** correspond au contenu du message qui doit être transmis au destinataire ;
- la partie **discriminant** correspond aux informations de routage destinées au média.

Les opérations définies sur les messages sont les suivantes :

affectation : L'affectation provoque la recopie du contenu du message et de son discriminant.

instanciation : un nouveau message est implicitement instancié à chaque fois qu'une classe utilise une des opérations d'envoi de message définies à la section 3.6, l'instanciation crée un nouveau message par recopie du contenu et du discriminant spécifiés en paramètre de l'opération.

lecture du discriminant : la lecture du discriminant d'un message ne peut être faite que dans un média, lors des opérations de lecture des messages définies à la section 3.6, elle permet d'accéder au contenu du discriminant du message.

lecture des données : la lecture du contenu du message ne peut être réalisée que dans une classe. Elle retourne les valeurs transmises au destinataire du message. Cette opération est effectuée lors des opérations de lecture des messages définies pour les classes à la section 3.6.

Les types tableaux

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

Le nombre de dimensions d'un tableau, ainsi que l'intervalle d'indice valide pour chaque dimension est fixé lors de la définition du type.

On définit les opérations suivantes sur les types tableaux :

Affectation : on définit l'affectation sur les tableaux par recopie de la totalité du contenu du tableau à droite de l'affectation vers le tableau à gauche de l'affectation. Les deux tableaux doivent être de même type.

Accès à un élément : permet d'accéder en lecture ou en écriture à un élément contenu dans le tableau en fonction de sa position (son indice).

Initialisation : permet d'affecter une valeur identique à tous les éléments du tableau, cet opérateur est surtout utilisé pour l'initialisation du tableau.

Les types articles

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 en lecture ou en écriture à chacun des champs du record par son nom.

Les types multi-ensembles

Les types multi-ensemble (bag) permettent de stocker une suite de valeur de même type. Un multi-ensemble peut contenir plusieurs occurrences d'une même valeur : la cardinalité des éléments du multi-ensemble n'est n'est pas bornée)

Opérations disponibles sur les multi-ensembles :

Ajout d'un élément : ajoute une instance d'un élément au multi-ensemble.

Retirer un élément : retire une occurrence d'un élément du multi-ensemble et retourne cet élément.

Cardinalité d'un élément du multi-ensemble : retourne le nombre d'occurrence d'un élément donné dans le multi-ensemble.

Cardinalité du multi-ensemble : retourne le nombre total d'occurrences d'élément présents 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 élément : 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 : $\{a_1, a_2\}$.

Inclusion stricte : retourne un booléen égal à true si le multi-ensemble à gauche de l'opération est inclus 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 éléments de a_2 sont supérieurs ou égaux aux cardinaux des éléments de a_1 et que a_1 et a_2 sont différents(i.e. : au moins un cardinal d'un élément de a_1 est strictement inférieur à celui de a_2 ou bien a_2 contient des éléments qui ne sont pas contenues dans a_1).

Inclusion : retourne un booléen égal à true si le multi-ensemble à gauche de l'opérateur est inclus dans le multi-ensemble à droite de l'opérateur, ou s'ils sont égaux.

Egalité : retourne un booleen égal à true si les deux paramètres sont des multi-ensembles et qu'ils contiennent les même éléments.

Exemple 4 *La déclaration suivante définit un multi-ensemble de jour_semaine :*

type multi is bag of jour_semaine ;

Soient a et b deux variable de type multi

$$\begin{aligned} := & \quad \{lundi, lundi, mardi, mardi, jeudi, samedi, dimanche\} \\ := & \quad \{samedi, dimanche\} \end{aligned}$$

On a alors les relations suivantes :

$$\begin{aligned} a+b &= \{lundi, lundi, mardi, mardi, jeudi, samedi, samedi, \\ &\quad dimanche, dimanche\} \\ a-b &= \{samedi, dimanche\} \\ b &< a \end{aligned}$$

Les types ensembles

Les types ensemble (set) 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. Un type ensemble peut être vu comme un type multi-ensemble dont la cardinalité des éléments a une borne supérieure égale à un.

Opérations disponibles sur les ensembles :

Ajout d'un élément : ajoute un élément à l'ensemble. Si la l'élément n'est pas déjà contenue par l'ensemble il est ajouté, sinon l'ensemble n'est pas modifié.

Retrait d'un élément d'un ensemble : retire un élément d'un ensemble et retourne cet élément. L'élément à retourner est choisi 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).

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.

Egalité : retourne un booleen égal à true si les deux paramètres de l'opérateur sont deux ensembles contenant les même valeurs.

Exemple 5 *La déclaration définit un ensemble de jours :*

type ens is set of jour_semaine ;

Soient a et b deux variables de type ens telles que :

$$\begin{aligned} a &= \{lundi, mardi, jeudi, samedi, dimanche\} \\ b &= \{samedi, dimanche\} \end{aligned}$$

On a alors les relations suivantes :

$$\begin{aligned} a+b &= \{lundi, mardi, jeudi, samedi, dimanche\} \\ a-b &= \{samedi, dimanche\} \\ b &< a \end{aligned}$$

3.4.4 Les types composants et leurs références

Les types composants désignent les composants du modèle. On définit deux sortes de composants : les classes et les médias. Les classes définissent les composants de l'application qui correspondent à l'aspect contrôle. Les médias désignent les composants qui permettent de relier ces classes entre elles et assurent la transmission des messages.

structure des composants L*f*P

Chaque composant **L*f*P** est doté d'une **partie statique** qui contient l'ensemble des déclarations locales, et d'une **partie dynamique** qui définit le comportement du composant pendant son exécution.

Définition 11 *La partie statique d'un composant peut contenir :*
- *des déclarations de types ;*
- *des déclarations de variables et de constantes ;*
- *des déclarations de trigger ;*
- *uniquement pour les classes : des déclarations de méthodes.*

L'ensemble des déclarations locales est ajouté à celles présentes sur le diagramme d'architecture pour définir le *contexte d'exécution* du composant. Les déclarations de types sont *locales* au composant : un type défini à l'intérieur d'un composant n'est visible et utilisable qu'à l'intérieur de ce composant.

Les déclarations de variables et de constantes forment les attributs du composant. La visibilité des attributs est définie en fonction de leur type :

– si le type de l'attribut est local au composant, l'attribut est *privé* et ne peut être accédé depuis l'extérieur du composant ;
– si le type de l'attribut est défini dans le diagramme d'architecture du modèle, l'attribut est *publique*, il peut donc être initialisé lors de l'instanciation du composant ;
– un attribut peut être lu par un autre composant pendant l'exécution du modèle si le type de l'attribut est un type *port* défini dans le diagramme d'architecture et que l'attribut apparait dans l'attribut *liaison* d'un binder du modèle.

Définition 12 *La partie dynamique d'un composant est constituée :*
– *du diagramme principal ;*
– *des triggers ;*
– *pour une classe de ses méthodes.*

Le diagramme principal définit l'automate principal du composant, c'est à dire les actions qu'il doit réaliser au cours de son exécution une fois qu'il a été instancié. Les triggers sont des sous-fonctions définies dans le corps du composant. Ils sont utilisés pour rassembler les traitements qui sont utilisés en plusieurs points de l'exécution du composant. On peut les considérer comme des méthodes privées d'un langage de programmation object classique.

Les déclarations de méthodes sont réservées aux classes et permettent de définir les services accessibles par appel de méthode. Les instructions contenues dans une méthode ne peuvent être exécutées que lorsque le composant reçoit un message d'activation correspondant à cette méthode.

Sémantique d'exécution des composants

Les composants **L*f*P** sont actifs : ils commencent leur exécution dès leur instanciation. Le diagramme principal commence son exécution lors de l'instanciation du composant. La structure de cet automate est hiérarchique : ses transitions peuvent elles mêmes être exprimées sous forme d'un *sous-automate*. Afin de simplifier l'écriture du composant, les triggers permettent de définir des sous-automates nommés pouvant être appelés en plusieurs endroits du composant comme une fonction

L'automate principal d'une classe peut exporter un ou plusieurs services (représentés par des méthodes **L*f*P**) qui définissent des sous-automates activables par les autres classes du modèle. Une méthode **L*f*P** ne peut être exécutée que si elle est invoquée par une autre instance de classe.

Dans le cas d'un média, seul l'automate principal est présent, en effet, ces composants ne sont pas destinés à fournir des services, mais à assurer le lien entre les services offerts par les classes.

Chaque élément permettant de regrouper plusieurs instructions **L*f*P** au sein de l'automate définit un *bloc nommé*.

Les éléments suivants du langage **L*f*P** définissent un bloc nommé :
– l'automate principal d'un composant ;
– les triggers d'un composant ;
– les méthodes d'une classes ;
– les transitions qui définissent un automate limité à une suite d'instructions structurées (instructions textuelles) ;
– les sous-diagrammes qui correspondent à des sous-automates complets et qui définissent un niveau de hiérarchie ;
– les états qui fournissent une syntaxe graphique pour un ensemble d'instructions dépendant du contexte et des transitions de sortie (cf. l'annexe A).

Les trois derniers éléments sont nommés implicitement ou explicitement par le modélisateur. Les méthodes et les triggers doivent être nommés explicitement. Les méthodes d'un composant

sont activées de manière externe par un appel de méthode. Les autres éléments sont activés de manière interne au cours de l'exécution du composant.

Les variables locales d'un automate sont visibles en tout point de l'automate et de ses sous-automates, ce qui donne les règles de visibilité suivantes :

- les variables définies dans le diagramme d'architecture sont visibles dans tout le modèle ;
- les variables définies dans l'automate principal d'un composant, sont visibles dans tout le composant ;
- les variables définies dans un trigger sont visibles dans l'automate du trigger et dans tous les sous-diagrammes qui y sont définis ;
- les variables définies dans une méthode sont visibles dans l'automate de la méthode et dans tous les sous-diagrammes qui y sont définis ;
- les variables définies dans un sous-diagramme sont visibles dans tout le sous-diagramme et dans les dans tous les sous-diagrammes qui y sont définis.

Les instructions disponibles pour la définition des automates des composants sont présentées à la section 3.6.

Sémantique des triggers

L'appel d'un trigger correspond exactement à l'appel d'une méthode privée dans un langage de programmation classique. Les triggers sont des automates exécutables uniquement depuis le composant où ils sont définis.

Le contexte d'exécution d'un trigger est défini par l'union :

- du contexte d'exécution du composant ;
- des déclarations locales du trigger.

Propriétés des triggers :

- les triggers doivent être déclarés dans les déclarations du diagramme principal d'un composant ;
- on ne peut pas définir de trigger local à un sous diagramme ;
- un trigger peut être appelé 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. 3.4.4), il peut en effet être appelé dans tout le diagramme de comportement de la classe, y compris dans le corps des méthodes.

La définition du corps d'un trigger est faite par une transition hiérarchique contenant le diagramme de comportement du trigger. Cette définition peut apparaître 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 appeler un trigger, on utilise une transition hiérarchique qui ne contient pas de sous-diagramme et qui porte le nom du trigger.

Sémantique des méthodes

Les méthodes sont définies dans les classes pour spécifier le traitement à effectuer lors de la réception d'un message d'activation donné. Le corps d'une méthode **L*f*P** peut contenir toutes les instructions disponibles dans l'automate principal du composant, sauf une instruction *select* portant sur des méthodes : une classe qui est déjà en train d'exécuter une méthode ne peut se mettre en attente d'une autre activation de méthode.

Dans une méthode, on ne peut donc pas insérer :

- des déclarations de méthodes ;

– des déclarations de triggers ;
– des instructions d'attente d'activation de méthode.

Les déclarations des triggers et des méthodes doivent être faites dans la partie déclarative de l'automate principal de la classe, elles ne peuvent donc être faites dans l'automate 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 une instruction d'activation de méthode dans le corps d'une méthode.

Chaque méthode doit être associée à la file de messages utilisée pour recevoir ses messages d'activation. Ce *binder d'activation* est spécifié sur une transition de réception de message spécifique qui doit être la première transition de la méthode.

Définition 13 *Le **binder d'activation** d'une méthode est la file de message (binder) dans laquelle les messages d'activations de la méthode doivent être envoyés.*

Le contexte d'exécution d'une méthode est constitué :
– du contexte d'exécution de la classe dans laquelle elle est définie ;
– l'ensemble des déclarations locales de la méthode.

Le nom, le type et le mode de passage des paramètres sont définis lors de la déclaration de la méthode. On distingue trois modes de passage pour les paramètres d'une méthode :
– `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 / restore : 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 :

Les fonctions sont des méthodes retournant une valeur : tous les paramètres d'une fonction doivent être en mode `in`. Les fonctions sont toujours synchrones : les appels de fonction sont bloquants tant que la valeur de retour n'a pas été reçue.

Les procédures synchrones sont soit des 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 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 appelant continue son exécution dès que le message d'activation a été écrit dans le binder d'envoi.

Toutes les méthodes synchrones (procédures ou fonction) doivent contenir une instruction `return` qui termine l'exécution de la méthode. Cette instruction construit le message de retour à partir des éléments suivants :
– le binder sur lequel est envoyé le message de retour, par défaut il s'agit du binder d'activation ;
– un discriminant pour le message de retour, par défaut, le discriminant du message d'activation est utilisé ;
– une valeur de retour :

– spécifiée par l'utilisateur dans le cas d'une fonction,
– constituée des valeurs des paramètres en mode *inout* et *out* dans le cas d'une procédure,
– vide dans le cas d'une procédure synchrone sans paramètre en mode `out` ou `inout`.

Références vers les composants L*f*P

Un type référence est associé implicitement à toute classe ou tout média défini dans le modèle. Une référence **L*f*P** vers un composant contient la désignation complète du composant, et permet d'accéder à l'instance correspondante indépendamment du déploiement et de la répartition.

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 invalide (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 3.6 est le seul à créer de nouvelles instances de composant.

On définit les opérations suivantes sur les références de composants :

Affectation : permet de dupliquer une référence vers un composant.

Déréférenciation : permet d'accéder aux ports du composant référencé.

L'*affectation* d'une référence permet de dupliquer la référence d'un composant ; le composant référencé n'est pas dupliqué.

La *déréférenciation* permet de fournir un accès en lecture aux ports d'un composant ; les autres attributs ne sont pas accessibles depuis l'extérieur du composant. L'accès au ports est un accès en lecture seule : l'opération de *déréférenciation* permet de connaître les binders référencés par les ports d'un composant **L*f*P**.

3.4.5 Sémantique des binders

Les binders sont les buffers assurant la transmission des messages entre les classes et les médias. La figure 3.9 complète la figure 3.6 pour les binders.

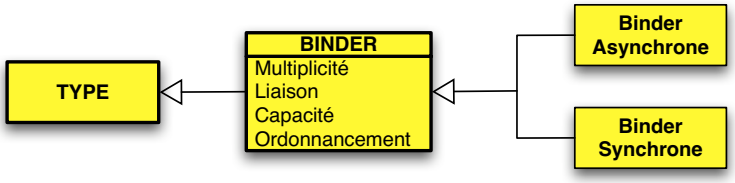


FIG. 3.9 – Le type binder dans le système de typage lfp

Les binders sont définis par les attributs suivants :
– *capacité* définit la taille du message en nombre de messages ;
– *ordonnancement* définit la politique d'ordonnancement des messages ;
– *liaison* définit le port et la classe auxquels le binder est associé ;
– *multiplicité* définit si la file de message est partagée entre plusieurs instances de la classe à laquelle il est relié par l'attribut *liaison* ;

On distingue deux types de binders : les binders synchrones et les binders asynchrones.
Les opérations réalisables sur les binders sont les suivantes :
– ajout de message ;
– lecture de message.

La figure 3.10 présente un binder synchrone reliant une classe `C1` à un média `M1` et illustre les attributs des binders. La suite de cette section présente les attributs statiques puis les opérations dynamiques associées aux binders.

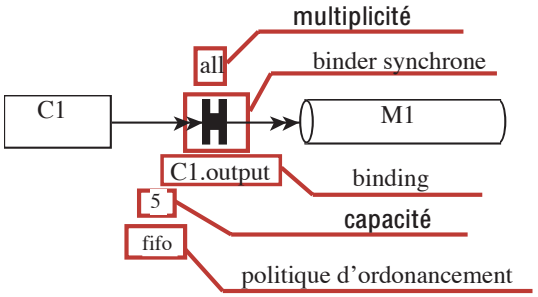


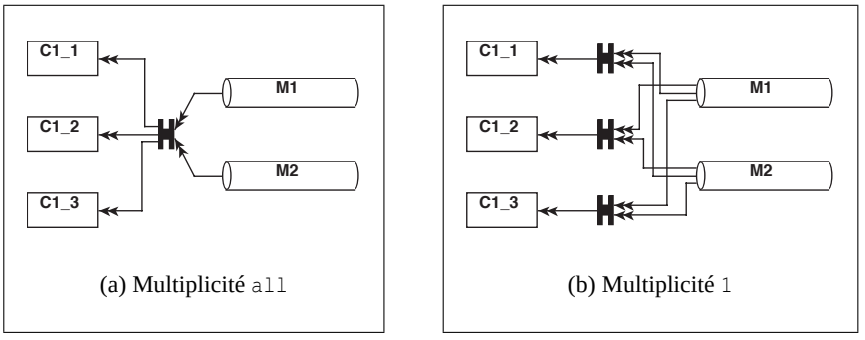
FIG. 3.10 – exemple de binder synchrone dans un diagramme d'architecture

Multiplicité des binders

La multiplicité du binder (attribut *multiplicity*) définit le nombre d'instances auxquelles le binder est connecté. Il y a deux valeurs possibles :

- `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 de chaque 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. Un binder de type `all` définit implicitement une variable globale 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 3.11 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. 3.11 – Différences sémantiques entre les multiplicité `all` et `1`

Dans le cas d'un binder de multiplicité `all` (figure 3.11(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 y accéder.

Dans le cas d'un binder de multiplicité `1` (figure 3.11(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.

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 3.10, le binder relie le port "output" de la classe `C1` au port "input" du média `M1`. Tout binder doit pouvoir contenir au moins un message.

Ordonnancement des messages

La politique d'ordonnancement du binder est spécifiée par l'attribut *ordonnancement* qui définit l'ordre dans lequel les messages peuvent être transmis. Deux politiques sont considérées par défaut :

- `fifo` : les messages sont fournis par le binder dans l'ordre de leur arrivée ;
- `bag` : les messages sont fournis par le binder dans un ordre quelconque ;

Sémantique de l'attribut liaison

L'attribut liaison relie un binder au port d'une classe définie sur le diagramme d'architecture, et définit un port de cette classe. Lors de la création de chaque instance d'une classe ses ports sont automatiquement initialisés à partir des informations présentes dans les attributs *multiplicité* des binders qui leur sont associés par leur attribut *liaison*.

Définition 14 *Un **port d'une classe** est un attribut de la classe relié à un binder sur le diagramme d'architecture. Les ports des classes sont des constantes.*

Les ports sont initialisés lors de l'instanciation de la classe et ne peuvent être modifiés. Cela signifie que si des variables de type "*port*" sont définies dans la classe et qu'elles n'apparaissent pas dans l'attribut *liaison* elles doivent être initialisées explicitement par l'instruction d'instanciation dynamique du composant pour désigner une instance de binder valide.

L'attribut *multiplicité* du binder détermine de quelle manière le port est initialisé :

- si la multiplicité est `1`, une nouvelle instance de binder est créée, et le port correspondant est initialisé pour référencer cette nouvelle instance ;
- 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 pour référencer cette instance.

Sur l'exemple de la figure 3.10, le port `output` de `C1` est relié à un binder de multiplicité `all`. Cela signifie que l'instance du binder doit être créée lors de l'initialisation de l'application. De plus, lors de la création d'instances de classes `C1`, l'attribut `output` doit être initialisé pour référencer cette instance de binder.

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 (messages 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.

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 : 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 : 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.

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 3.12) :

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

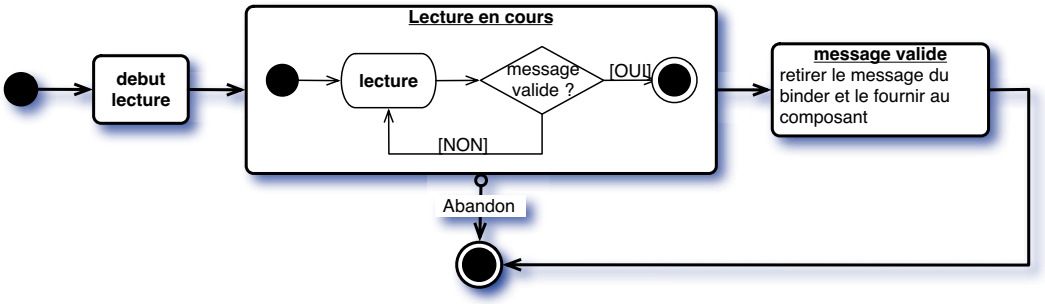


FIG. 3.12 – Déroulement d'une lecture sur un binder

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 3.6). Le message est alors retiré du binder, et son emplacement est à nouveau disponible.

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. L'exécution d'un composant qui demande un message étant suspendue jusqu'à réception d'un message valide, un composant ne peut abandonner une lecture sur un binder que si un message valide lui a été transmis par un autre binder.

3.5 Sémantique des éléments statiques du langage

Cette section couvre la sémantique du diagramme d'architecture du langage **L*f*P**. Ce diagramme est la base de toute spécification **L*f*P**, et définit la topologie de l'application ainsi que les moyens de communication entre les composants.

Le diagramme d'architecture est composé de deux parties complémentaires :
- un graphe orienté définissant la topologie de l'application ;
- une partie déclarative pour les éléments partagés du modèle.

3.5.1 Définition de la topologie de l'application

La topologie de l'application est définie par un graphe orienté dont les noeuds sont les composants et les binders de l'application. Les arcs définissent les possibilités de communication entre les composants.

Structure du graphe

La structure des applications **L*f*P** peut être contrainte par le langage : celui-ci impose une définition stricte du protocole d'interaction entre les classes du modèle sous forme de médias ; ce sont eux qui assurent le transport des messages entre les binders.

Le principe est donc d'intercaler un média entre toutes les classes du modèle. Le point de communication entre les classes et les médias est assurée par les binders. Un chemin valide entre deux classes $C1$ et $C2$ est donc défini par la séquence suivante :

$$C1 \rightarrow Binder \rightarrow media \rightarrow Binder \rightarrow C2$$

.

De plus, l'existence de ce chemin est une condition nécessaire pour que $C1$ puisse envoyer un message à $C2$; symétriquement l'existence du chemin inverse est requise pour que si $C2$ puisse envoyer un message vers $C1$.

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 transfert 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 transfert 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 3.10, 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 messages vers le binder. De même, le média ne peut que lire des données dans ce dernier.

3.5.2 Partie déclarative du diagramme d'architecture

La partie déclarative du diagramme d'architecture permet de définir :

- des types de données ;
- des constantes ;
- des instances statiques de composants.

L'ensemble des déclarations réalisées dans le diagramme d'architecture sont visibles dans tout le modèle ; elles définissent le contexte d'exécution commun de tous les composants du modèle.

Il n'est pas possible de définir des variables dans le diagramme d'architecture. Cette contrainte a été imposée pour assurer que la communication entre les composants du modèle passent systématiquement par un média.

Les instances statiques définissent les composants qui doivent être instanciés lors du démarrage de l'application. Il est possible de spécifier les valeurs initiales des attributs des composants créés. les instances définies statiquement sur le diagramme d'architecture sont toutes créées avant que les composants commencent leur exécution.

3.6 Sémantique comportementale du langage L*f*P

Cette section présente la sémantique dynamique du langage **L*f*P**, c'est à dire les instructions définissant le comportement des composants **L*f*P** lors de leur exécution.

Dans cette section, on s'attachera à présenter l'ensemble des instructions disponibles pour le langage **L*f*P**. Les opérations réalisées par chaque instructions seront explicitées, et leur syntaxe sera évoquée. La définition précise de la BNF du langage ainsi que la syntaxe graphique des automates est donnée par l'annexe A. Les instructions présentées dans cette section sont portées par les transitions des diagrammes de comportement.

3.6.1 Instructions internes

Une instruction est interne à un composant si :

- son résultat ne dépend que de l'état courant du composant ;
- ses actions ne modifient que l'état du composant dans lequel elles sont exécutées.

Le résultat d'une séquence d'instructions internes est donc indépendant de l'évolution du reste du système et de l'état des ports du composant. Une séquence d'instructions internes peut donc être considérée comme atomique du point de vue du modèle.

Les instructions internes définies dans le langage **L*f*P** sont les suivantes :

- les instructions de contrôle du flot d'exécution ;
- tous les opérateurs sur les types du langage (sauf l'accès au port d'un composant) lorsqu'ils portent sur des variables locales ;
- Tous les opérateurs des types de données sauf la déréférenciation (lecture de la valeur d'un port d'un composant).

3.6.2 Instructions de contrôle du flot d'exécution

Les instructions de structuration du code sont les suivantes :

- la structure conditionnelle ;
- les structures de boucle (boucle `for` et boucle `while` ;
- l'instruction de saut (*goto*).

Ces trois structures ont la sémantique classique des langages de programmation, et leur syntaxe est donnée à l'annexe A. La seule particularité provient du fait que la boucle for définit localement sa variable de boucle (comme c'est le cas par exemple en Ada).

Instruction de saut

L'instruction de saut est proche d'une instruction *goto* d'un langage de programmation assortie d'un ensemble de contraintes. Une instruction de saut peut être insérée n'importe où dans l'automate d'un composant. Sa cible correspond à l'emplacement où *saute* le flot d'exécution du composant. Cette instruction est représentée sur le diagramme de comportement par un arc orienté dirigé vers la destination du saut.

En **L*f*P**, les sauts doivent respecter les contraintes suivantes :

- la cible d'un saut est toujours un bloc nommé ;
- la cible d'un saut doit être dans le même bloc nommé que l'instruction elle même.

L'instruction de saut est représentée sur les diagrammes de comportement par un arc entre un état et une transition. Les contraintes ci-dessus signifient qu'il ne peut pas exister de saut direct entre une transition d'un automate vers un autre automate du composant (par exemple vers une transition d'un de ses sous-automates).

Lorsqu'un état est suivi de plusieurs instructions de saut vers des transitions différentes, il définit une alternative. La prochaine transition à exécuter est définie en fonction de l'évaluation de sa garde : la transition exécutée est la première dont la garde est évaluée à *true*.

On distingue deux manières de tester les gardes des transitions de sortie :

- si aucune priorité n'est affecté aux arcs, les gardes des transitions de sortie de l'état sont testées dans un ordre quelconque ;
- si des priorités sont spécifiées sur les instructions de sortie, les gardes sont testées dans l'ordre spécifié par les priorités.

L'alternative

L'alternative en **L*f*P** est très proche de celle utilisée dans le langage C : elle ne permet que deux branches et ne propose pas de construction de type *elsif* comme le propose le langage ada. La condition d'une alternative **L*f*P** est une expression à valeur booléenne (une valeur entière n'est pas acceptée).

Les instructions de boucles

L*f*P des boucles *for* et *while* très proches de des constructions du langage ada. Les boucles *for* permettent de parcourir un intervalle de valeurs de type énuméré ou entier.

La boucle *while* est définie par une condition de sortie à valeur strictement booléenne (les condition de sortie à valeur entière utilisées par le langage C ne sont pas acceptées).

3.6.3 Instanciation dynamique

L'instanciation dynamique correspond à la création d'une nouvelle instance d'un composant pendant l'exécution du modèle, cette opération est réalisable en tout point du diagramme de comportement. Instancier un composant signifie que les éléments nécessaires à son exécution sont également instanciés :

- dans le cas d'un média, il s'agit principalement de son support d'exécution ;
- dans le cas d'une classe, il faut également initialiser tous les ports présents sur le diagramme d'architecture, et donc instancier l'ensemble des binders qui lui sont connectés.

Les règles d'instanciation des binders sont spécifiées à la section 3.4.5.

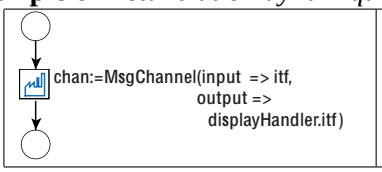
L'instruction d'instanciation dynamique permet de spécifier des valeurs initiales pour les attributs de la nouvelle instance de composant. Seuls les attributs dont le type est visible dans le

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.

La séquence logique d'opérations correspondant à une instanciation dynamique est donc la suivante :

1. création de l'instance du composant ;
2. initialisation de ses attributs avec les valeurs spécifiées dans leurs déclaration ;
3. initialisation des attributs spécifiés par l'instanciation dynamique ;
4. si le composant est une classe, initialisation de ses ports, soit en les reliant à un binder statique existant, soit en créant les nouvelles instances de binders nécessaires.

Exemple 6 *Instanciation dynamique d'un composant **LfP** :*



*Instanciation dynamique d'un composant **LfP** en initialisant ses attributs* `input` *et* `output`.

3.6.4 Instructions de communication des classes

Les instructions de communication sont toutes les instructions qui font interagir une instance de composant avec le reste du modèle. Ces instructions regroupent l'accès au port d'un composant, et les différentes formes d'envoi de messages.

Attente d'activation d'une méthode

Cette instruction correspond à l'attente d'un message d'activation sur le binder correspondant à la méthode activable. Elle réalise les opérations suivantes :

- lecture d'un nouveau message sur le binder ;
- si le message n'est pas un message d'activation, le message est refusé ;
- si le message est un message d'activation, vérifier que la méthode activée est bien la méthode attendue ;
- si la méthode activée n'est pas la méthode attendue, le message est refusé ;
- si la méthode activée est la méthode attendue, elle est exécutée et l'instruction se termine ;
- si un message est refusé, le message est laissé dans le binder, et l'opération se met en attente du prochain message.

Cette opération est donc bloquante dans les cas suivants :

- lorsque le binder d'activation d'une méthode est vide ;
- lorsque le binder d'activation d'une méthode est de type *fifo* et que le premier message n'est pas un message d'activation valide pour la méthode ;
- lorsqu'aucun des messages contenu dans un binder *bag* n'est un message d'activation valide pour la méthode.

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".

Un appel de procédure synchrone contient les éléments suivants :

– le port de la classe appelante par lequel le message est envoyé ;
– un discriminant explicite ;
– éventuellement le nom du type du composant appelé ;
– le nom de la méthode appelée ;
– les paramètres effectifs de l'appel.

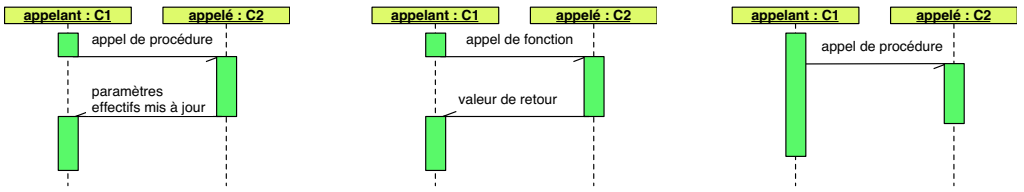
Le port de la classe appelante référence le binder dans lequel le message sera ajouté pour être ensuite routé par un média.

Le discriminant est un ensemble de paramètres passés au média pour assurer le routage du message. Les valeurs passées au discriminant sont recopiées dans le message envoyé. Les types de ces valeurs doivent être cohérents avec ceux définis dans le type du port vers lequel le message est envoyé.

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. 3.13 – 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 messages 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 jour 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 3.13(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.

Exemple 7 *Exemple d'appel de procédure synchrone :*

Appel de la procédure synchrone `handle_request`*. L'appel est envoyé sur le port* `itf`*, le discriminant contient la variable* `id` *et le paramètre est* `i`*.*

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.

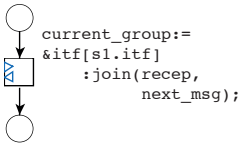
Le principe d'un appel de fonction est proche de celui d'un appel de procédure synchrone. Seule la manière dont est transmise la valeur de retour de la méthode : dans le cas d'un appel de fonction, la valeur de retour est écrite dans une variable indépendamment des paramètres effectifs. Les paramètres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

Un appel de fonction effectue les opérations suivantes :

- envoi de message dans la file de sortie ;
- réception du message de retour dans la file d'entrée ;
- 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.

Le composant suspend son exécution pendant l'attente du message de retour de la fonction. L'exécution reprend lorsque la valeur de retour est disponible.

Exemple 8 *L'exemple suivant (extrait de l'étude de cas du chapitre 7) montre un appel de fonction en **LfP** :*



Appel de la méthode join d'un composant. La valeur de retour de la fonction est stockée dans la variable `current_group`*. Le discriminant est constitué de la valeur* `s1.itf`*, enfin, les paramètres d'appel sont* `recep` *et* `next_message`*.*

Appel de procédure asynchrone

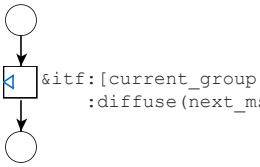
Un appel de procédure asynchrone contient les mêmes éléments qu'un appel de procédure synchrone. Ce mécanisme est illustré sur la figure 3.13(c). Cette opération ne peut se trouver que sur une transition d'envoi.

Les opérations suivantes sont réalisées lors de l'appel de procédure 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 3.13(c)). Aucun message de retour n'est lu.

Exemple 9 *L'exemple ci-dessous présente un appel de procédure asynchrone extrait de l'étude de cas du chapitre 7 :*



Appel de la procédure asynchrone `diffuse`*. Le port sur lequel l'appel est envoyé est le port* `itf` *du composant courant. Le discriminant est* `current_group.itf` *(le port itf du composant current group), et le paramètre d'appel est* `next_message`*.*

Instructions d'envoi et réception de messages de données

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). Par opposition aux messages d'activation des méthodes, ces messages sont appelés *messages de données*.

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 variables qui recevront les valeurs transportées par le message

La lecture des messages dans les classes consiste à affecter le contenu du message retourné par 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. Dans le cas d'un binder fifo, si le premier message retourné n'est pas correctement typé, le modèle est erroné. Dans le cas d'un binder bag, on continue la lecture jusqu'à trouver un message compatible.

L'opération de lecture est bloquante si le binder dans lequel le message est lu est synchrone ou 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, elle ne peut être portée que par une transition d'envoi. Cette opération nécessite les éléments suivants :

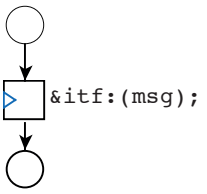
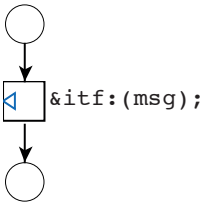
- le port qui référence le binder dans lequel le message doit être envoyé ;
- un discriminant optionnel permet de spécifier des paramètres de routage utilisés par le média ;
- un ensemble de valeurs correspondant au contenu du message, il n'y a pas de typage du contenu des messages lors de l'envoi.

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.

Exemple 10 *Exemples d'instructions d'envoi et réception de messages extraites de la figure 3.5 (page 50) :*

	*Lecture d'un message sur le port* `itf`*. Le contenu du message est stocké dans la variable* `msg`*.*
	*Écriture d'un message dans le port* `itf`*. Le corps du message est le contenu de la variable* `msg`*.*

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 contiennent pas de partie à transmettre, et sont donc simplement traités par les médias.

3.6.5 Instructions de communication des médias

Les médias ont une interface de communication basée sur l'envoi et la réception de messages. Elle est différente de celle des classes car l'objectif est ici de router les messages vers leurs destinataires, et non pas d'exploiter le contenu des messages. Pour cette raison, à l'intérieur d'un média seul le contenu du discriminant d'un message est accessible en lecture. Le contenu du message lui même n'est visible qu'à travers une variable de type message dont le contenu et la structure ne peuvent être altérés.

On distingue deux opérations de manipulation de messages dans les médias :

– la lecture d'un message dans un binder ;
– l'envoi d'un message dans un binder.

Dans un média, les opérations de lecture est associée à une *garde*, c'est à dire à une expression booléenne déterminant la validité du message reçu.

Une opération de lecture réalise les opérations suivantes :

– 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 et le message est retiré du binder,
  – 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.

3.6.6 Multiplexage d'opérations de lecture

Que ce soit dans les médias ou dans les classes, les opérations d'attentes de messages sont intéressantes si elles peuvent être multiplexées, c'est à dire s'il est possible d'attendre un message sur plusieurs ports à la fois.

Ce multiplexage est réalisée par l'instruction *select* qui permet de lancer simultanément plusieurs opérations de lecture sur des binders différents. Cette instruction se termine lorsqu'une des opérations de lecture s'est terminée (un message a été lu). Les autres opérations sont abandonnées, et les messages correspondants ne sont pas consommés.

Cette instruction permet d'attendre soit des messages de données soit des messages d'activation sur un ensemble de ports du composant. On ne peut pas mélanger ces deux types de messages au sein d'une même instruction *select*.

L'instruction *select* peut être utilisée dans les trois cas suivants :

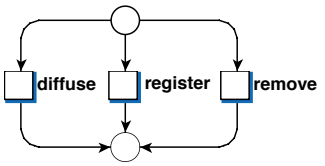
- dans une classe : attente d'un message d'activation d'une méthode ;
- dans un média : attente d'un message.

Attente d'activation de plusieurs méthodes

Dans le premier cas, l'instruction select permet de spécifier une liste de méthodes activables, et attend un message d'activation valide pour une de ces méthodes sur les binders qui leur sont associées. Cette instruction termine lorsqu'un message d'activation d'une méthode spécifiée dans la liste est reçu sur le binder qui lui est associé. Cette opération effectue les opérations suivantes :

1. lister les ports d'activation des méthodes considérées ;
2. attendre un message sur un de ces ports (opération de lecture sur les binders correspondants) ;
3. exécuter la méthode si le message reçu est un message d'activation et que le port de réception correspond au port d'activation de la méthode appelée ;
4. sinon, refuser le message et retourner au point 2.

Exemple 11 *Exemple d'attente d'activation de procédures (extrait de l'étude de cas du chapitre 7 :*



*Attente d'activation de méthode dans une classe **LfP** : Les trois méthodes `join`, `diffuse`, `quit` peuvent être activées. L'exécution d'une des méthode est déclenchée par l'arrivée du message d'activation correspondant.*

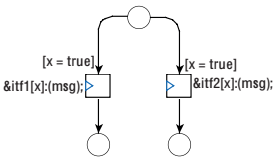
Multiplexage d'attente de messages dans les médias

Dans le cas de l'attente de message dans un média, l'instruction est terminée lorsqu'un message a été accepté, c'est à dire lorsque la garde de l'instruction de lecture a été évaluée à `true`. Cette opération effectue donc la série d'opérations suivantes :

1. lister les ports sur lesquels un message peut être reçu ;
2. attendre un message sur l'un de ces ports (accès en lecture sur les binders correspondants) ;
3. lorsqu'un message est reçu, lecture du contenu de son *discriminant* ;
4. évaluer la garde de l'instruction de lecture ;

5. si la garde s'évalue à true, le message est accepté : le contenu du message est stocké dans les variables du média, l'instruction est terminée ;
6. si la garde est fausse, le message est refusé, et le composant retourne au point 2.

Exemple 12 *Exemple de multiplexage de lecture de message dans le cas d'un média :*



Lectures multiplexée dans un média :
- *La branche de gauche attend un message sur le port* `itf1`*. Le message lu sera effectivement consommé si la valeur contenue dans le discriminant est égale à* `true`*.*
- *La branche de gauche attend un message sur le port* `itf2`*. Le message lu sera effectivement consommé si la valeur du discriminant est égal à* `true`*.*

Gestion de la priorité

Il est possible de spécifier les priorité des opérations incluses dans les instructions *select*. Ces priorités permettent de définir dans quel ordre sont testées les opérations spécifiées. Ainsi, si deux messages peuvent être consommés, c'est l'opération la plus prioritaire qui sera effectuée.

Dans le cas où deux opérations de même priorité peuvent être effectuées, le choix de l'opération à exécuter est laissé à l'implémentation et aucune hypothèse (notamment l'équité) n'est requise par le langage.

3.7 Conclusion

Ce chapitre a présenté le langage **L*f*P** qui sert de langage pivot à la méthodologie présentée à la section 2.8. Ce langage a été défini en spécialisant et en précisant les concepts définis dans [67] afin d'en faire un langage de modélisation efficace. Le langage **L*f*P** a été conçu pour permettre l'implémentation de cette méthodologie,

L*f*P est basé sur deux vues complémentaires du système : la vue architecturale définit les relations statiques entre les composants. La vue comportementale définit le comportement dynamique de chacun des composants de la spécification. Le comportement est défini à l'aide d'automates communiquant par files de messages. Les aspects les plus importants du langage sont ceux liés aux communications entre les composants. On distingue deux types d'interactions : l'envoi de message et l'appel de méthode distante. Dans les deux cas, le routage des messages est assuré par les médias : ce sont les composants dédiés qui assurent le respect des protocoles de communication définis entre les classes du modèle.

Le concept de type opaque a été ajouté pour relier la spécification **L*f*P** aux composants dont le comportement n'est pas spécifié dans ce langage. Ces composants représentent la partie métier de l'application et sont représentés dans la spécification **L*f*P** sous forme de types opaques. Ces types sont définis par leur interface composée par un ensemble de méthodes que les composants **L*f*P** peuvent utiliser. Ces instructions d'appel sont réalisées de manière à ne pas restreindre la portée des résultats fournis par la vérification formelle.

Ces mécanismes permettent de modéliser de façon fiable les interactions entre les parties contrôle et métier de l'application. Ils permettent au processus de génération de code de générer les instructions nécessaires aux interactions avec les composants représentés par les types opaques.

Ces fonctionnalités nous permettent de décrire précisément la partie contrôle d'une application réparties. De plus, **L*f*P** et la méthodologie associée fournissent une base solide pour la modélisa-

tion et la vérification d'applications réparties. Dans ce contexte, il nous reste à traiter l'implémentation des spécifications ainsi vérifiées. La suite de ce mémoire traitera de la génération automatique de code réparti pour implémenter les modèles **L*f*P**. Les chapitres 4, 5 et 6 traitent respectivement des aspects liés au déploiement de la spécification sur une architecture cible, de l'exécutif nécessaire au fonctionnement du code généré et des règles de transformation de **L*f*P** vers un langage de programmation.

Chapitre 4

Déploiement

4.1 Introduction

L'objectif de nos travaux est de produire du code exécutable à partir d'une spécification **L*f*P**. La première étape de notre travail consiste donc à définir le déploiement des éléments du langage **L*f*P** sur la plate-forme d'exécution de l'application à produire. L'objectif du déploiement est d'adapter une application à une architecture cible pour permettre son exécution.

Le déploiement d'une application comporte principalement trois aspects :

- partitionnement de l'application ;
- placement de ces exécutables sur les noeuds du réseau ;
- placement des composants créés pendant l'exécution de l'application.

Le partitionnement de l'application définit son découpage en un ensemble exécutables autonomes permettant d'implémenter la spécification. Ce point couvre la définition des exécutables à produire pour implémenter la spécification. Ceux-ci doivent regrouper les composants définis par la spécification dans des unités d'exécution autonomes. Les exécutables produits par le générateur de code doivent ensuite être placés sur les noeuds devant participer à l'application. Enfin, Le troisième point concerne le placement des composants créés pendant l'exécution de l'application. Le placement des composants **L*f*P** est réalisé par l'utilisateur en fonction de critères tels que la topologie du réseau considéré, ou l'optimisation des performances globales de l'application.

La construction des exécutables de l'application, ainsi que leur placement sur l'architecture d'exécution définissent la partie statique du déploiement. En effet, les décisions relatives à ces deux points sont exclusivement prises en compte avant l'exécution de l'application. En revanche, le placement des composants créés pendant l'exécution de l'application correspond à la partie dynamique du déploiement. Le placement des nouvelles instances peut être réalisé en fonction de critères statiques, mais également en fonction de critères dynamiques dépendant du comportement de l'application. La politique de déploiement est donc susceptible d'influencer non seulement la structuration initiale de l'application, mais également l'évolution de cette structuration pendant toute l'exécution.

La sémantique du langage **L*f*P** est indépendante du déploiement, le processus de génération de code doit donc garantir que le comportement de l'application restera le même quelque soit les options de déploiement retenues par l'utilisateur. Elles sont en revanche utilisées par le générateur de code pour produire les informations nécessaires à l'initialisation de l'application et influencent la composition des exécutables qu'il produit.

La section 4.2 présente la première étape de la définition du déploiement d'une spécification **L*f*P**. Elle montre comment les composants d'une spécification doivent être regroupés pour définir un ensemble d'exécutables qui implémentent l'application spécifiée sur un environnement

d'exécution cible. La section 4.3 présente la structure de ces exécutables et l'initialisation de leur exécution. La section 4.4 définit les informations permettant de les placer sur un environnement cible constitué d'un ensemble de noeuds d'exécution. La section 4.6 définit la structure d'une application construite à partir d'une spécification **L*f*P** et introduit les interactions entre le code généré et son environnement. Cette section rassemble les informations des sections précédentes pour proposer la structure générique permettant d'implémenter une application **L*f*P** en intégrant les informations de déploiement.

4.2 Architecture de placement

Afin de faciliter la définition du placement des composants, on adopte l'architecture d'application présentée sur la figure 4.1. Les *composants applicatifs* désignent les composants de l'application générée. Chaque composants applicatif inclus donc un ensemble de composants **L*f*P** ainsi que les externes (composants de la partie métier de l'application) qui leur sont associés.

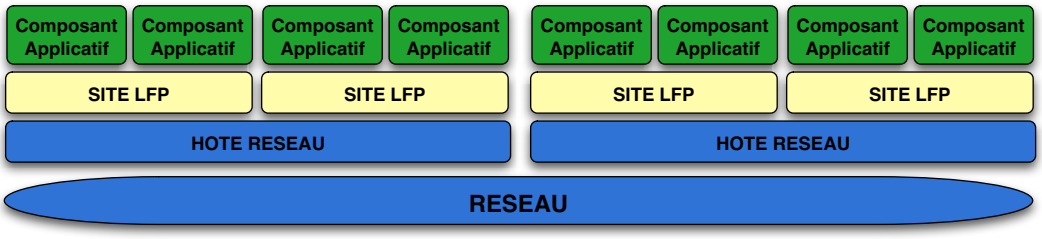


FIG. 4.1 – Structuration du placement des composants

Un site **L*f*P** rassemble un ensemble de composants déployés sur un même hôte réseau. Chaque site **L*f*P** définit un processus autonome exécutable sur la plate-forme cible. Chaque hôte réseau peut héberger un ou plusieurs sites **L*f*P**. De cette manière, il est possible de déployer plusieurs composants sur un même hôte en les isolant dans des exécutables distincts. Cette approche facilite également le redéploiement de l'application.

Les hôtes participant à l'application sont reliés par un réseau de communication assurant le transport des messages de l'application. Il est donc nécessaire de pouvoir implémenter la sémantique des opérations de communication définies par le langage **L*f*P** sur cette plate-forme.

Définition 15 *Un **site LfP** est défini par*
- *une instance de l'exécutif associé au générateur de code ;*
- *l'ensemble des instances de composants **LfP** gérés par cette instance de l'exécutif*
- *l'ensemble des instances de composants externes gérés par les composants **LfP** du site.*

*Un site **LfP** doit être déployé sur un seul hôte réseau. Plusieurs sites **LfP** peuvent être déployés sur le même hôte réseau.*

L'unité de déploiement est définie par les sites **L*f*P** : tous les composants définis dans un site **L*f*P** doivent être déployés sur le même hôte réseau. En revanche, l'unité de parallélisme reste le composant **L*f*P** : chaque exécutable qui implémente un site **L*f*P** est décomposé en autant de thread qu'il y a de composants instanciés.

Le déploiement réparti d'une spécification **L*f*P** nécessite de considérer trois niveaux de communication entre les composants :
- communication entre composants du même site **L*f*P** ;
- communication entre composants de sites **L*f*P** distincts sur le même noeud du réseau ;
- communication entre composants instanciés sur des noeuds distincts.

Dans le premier cas, les deux composants disposent d'un espace d'adressage commun qui leur permet de communiquer directement. Dans le deuxième cas, les deux composants sont instanciés dans des espaces d'adressages différents, mais ils peuvent communiquer sans passer par une interface réseau, ce qui évite les opérations d'encodage et de décodage nécessaires pour le transport sur le réseau. Enfin, dans le dernier cas, il faut prévoir l'encodage des données à transmettre, et l'envoi des messages correspondants au site distant par l'exécutif. L'implémentation optimisée de ces trois politiques est très différente, elle dépend du niveau de communication requis, et chacune de ces implémentation dépend fortement de la plate-forme de déploiement requise. Enfin, intégrer l'implémentation de ces politiques par le générateur de code implique d'intégrer des aspects liés au déploiement dans le code généré. L'application produite ne peut donc être redéployée sans repasser par le processus de génération de code.

Notre objectif est de rendre le processus de génération de code aussi indépendant que possible du déploiement. Cela nécessite donc que l'interface de communication entre les composants soit indépendante du niveau de communication à prendre en compte. Pour cela, nous introduisons une bibliothèque de communication capable de transmettre de manière optimisée des messages entre deux composants de l'application quelque soit le niveau de communication requis. Celle-ci masque les problèmes liés aux choix de déploiement des composants et assure la pérennité du code généré lors des changements de politique de déploiement.

Cette approche nous a amené à construire un *exécutif* (ou *runtime*) associé au générateur de code. Celui-ci rassemble l'ensemble des services utilisés de manière récurrente par le générateur de code pour implémenter la spécification **L*f*P**. Son étude complète est l'objet du chapitre 5.

4.3 Partitionnement de l'application

La sémantique du langage **L*f*P** est indépendante du déploiement de l'application, ce qui signifie que les applications produites doivent avoir le même comportement quel que soit le déploiement effectivement réalisé. Afin de permettre d'optimiser le déploiement en fonction des caractéristiques attendues de l'application, il est donc nécessaire de disposer d'un processus de déploiement permettant de placer chaque composant sur un noeud quelconque de l'application sans changer le comportement global du système.

Les informations de déploiement d'un modèle doivent donc spécifier l'ensemble des exécutables nécessaires pour implémenter l'application, et pour chaque exécutable l'ensemble des composants **L*f*P** requis. Chaque exécutable ainsi construit définit une *partition* de l'application.

4.3.1 Structuration des exécutables

On distingue deux manières de définir un exécutable :
- liste d'instances statiques ;
- liste de composants.

La première méthode permet de définir des exécutables dont l'exécution est destinée à implémenter une ou plusieurs instances statiques définies sur le diagramme d'architecture de la spécification. Ces exécutables doivent être démarrés lors de l'initialisation de l'application, ce qui déclenche l'exécution des instances statiques définies sur le diagramme d'architecture.

Définition 16 *On appelle **instances statiques** d'un modèle l'ensemble :*
- *des instances statiques de composants (définies sur le diagramme d'architecture) ;*
- *des binders de multiplicité **all**.*

Ces composants doivent être instanciés lors de l'initialisation de l'application.

Les informations de déploiement doivent donc au minimum définir le placement de l'ensemble des instances de composants (classes et médias) ainsi que des binders `all` définis sur le diagrammes d'architecture. Elles sont utilisées pour construire les références distribuées qui seront utilisées pour la localisation de ces composants. Les références sont générées par le processus de déploiement à partir des informations de placement fournies par l'utilisateur. Elles sont accessibles à tous les composants du système et peuvent être utilisées lors de leur initialisation. Ce comportement est cohérent avec la sémantique **L*f*P** car les instances statiques de composants **L*f*P** et les binders `all` sont des variables globales de la spécification.

La deuxième méthode permet de définir un exécutable correspondant à un ensemble de composants instanciés par un évènement extérieur au système. Dans ce cas, il est nécessaire de définir quels types de composants cet exécutable peut implémenter. Cette approche est particulièrement utile par exemple pour définir la partie cliente d'un système client serveur L'exemple du chapitre 7 utilise abondamment cette fonctionnalité.

Les composants **L*f*P** étant actifs, ils s'exécutent de manière parallèle. Chaque exécutable généré à partir d'un site **L*f*P** est donc constitué d'un ensemble de threads. Chaque composant dispose de son thread ce qui permet une exécution simultanée de tous les composants déployés dans l'exécutable.

Pour cela, on dispose de deux politiques possibles :

- définir un ordonnanceur au niveau de l'exécutif **L*f*P** ;
- associer un thread système à chaque composant **L*f*P**.

La première solution permet de mieux contrôler le parallélisme d'exécution et permet de mieux contrôler le nombre de threads créés dans le système à un instant donné. La deuxième solution est plus simple à implémenter, mais offre moins de contrôle sur le parallélisme d'exécution et la consommation de ressources systèmes.

Le comportement global du système reste le même quelque soit la solution choisie car les aspects liés au parallélisme d'exécution et les synchronisations entre les composants sont effectuées en utilisant les opérations définies dans le langage **L*f*P**. Le choix de la politique utilisé sera surtout effectué en fonction de la plate-forme cible et des contraintes particulières inhérentes aux applications que doivent supporter l'exécutif et le générateur de code.

4.3.2 Spécialisation des exécutables

Afin de leur conserver une taille raisonnable, il est nécessaire de produire des exécutables qui ne contiennent que les composants strictement nécessaires à leur fonctionnement.

Chaque exécutable doit inclure :

- l'ensemble des types de données utilisés pour construire les composants statiques qui y sont déployés ;
- l'ensemble des types de données manipulés par ces composants.

Le premier est déduit simplement des informations de déploiement qui spécifient l'ensemble des instances statiques déployées sur chaque site **L*f*P**. Ensuite, le générateur de code doit utiliser ces informations pour définir l'ensemble des types de données manipulés par ces composants et les inclure dans l'exécutable.

Ce type de politique a un impact sur la spécification du déploiement : les modifications de déploiement effectuées en dehors du processus de génération de code ne doivent pas modifier la répartition des composants à l'intérieur des sites **L*f*P** de l'application.

4.3.3 Démarrage des exécutables

La première opération que doit réaliser un exécutable généré pour une application **L*f*P** est d'assurer le démarrage de l'application. Le comportement de chaque exécutable est donc défini par l'ensemble des instances statiques déployées sur le site **L*f*P** qu'il implémente.

On distingue deux sortes d'instances statiques :
- les composants **L*f*P** (classes ou médias) définis sur le diagramme d'architecture ;
- les binders de multiplicité `all` (binders statiques).

Chaque exécutable doit donc réaliser les opérations suivantes :
- initialisation des références partagées ;
- instanciation des composants statiques locaux ;
- démarrage des services de l'exécutif ;
- lancement de l'exécution des composants **L*f*P**.

Les références partagées sont les références des instances statiques : ensemble des binders `all` et des instances statiques de composants **L*f*P**. Ces références seront résolues dynamiquement par le *service de localisation* de l'exécutif qui permet d'accéder à un composant à partir de sa référence. Ces références sont construites à partir des informations de démarrages produites par le processus de déploiement. Leur construction est indépendante de l'instanciation des composants référencés.

La deuxième étape est la construction de l'ensemble des composants statiques déployés sur le site **L*f*P** local. L'ordre d'instanciation est défini par l'utilisateur. Une fois instanciés, les composants sont initialisés en fonction des informations fournies par la spécification.

La troisième étape est l'enregistrement ces composants dans les structures interne du service de localisation de l'exécutif. Cette opération permet la résolution des références qui leur sont associées. L'exécution des composants **L*f*P** reste bloquée jusqu'au démarrage complet du site. A la fin de cette opération, l'exécutif peut accepter les requêtes des autre sites de l'application.

Enfin, l'exécution des composants peut démarrer, le site est alors complètement fonctionnel. Il peut répondre aux sollicitations des autres sites de l'application. La définition de l'ordre de démarrage des sites est laissée à la charge de l'utilisateur. L'application est considérée comme démarrée lorsque l'ensemble des instances statiques définies sur le diagramme d'architecture ont commencé leur exécution.

4.4 Placement des exécutables

Le placement des exécutables définit la manière dont ils sont répartis sur les noeuds du réseau utilisé pour déployer l'application. Selon que l'on souhaite optimiser les performances ou la facilité de redéploiement de l'application, les informations de démarrage générées à partir des informations de déploiement peuvent soit être intégrées directement dans le code généré, soit être lues sous forme d'un fichier de configuration au démarrage de l'application. Il s'agit de la partie de l'application générée la plus dépendante de l'implémentation de l'exécutif et du générateur de code associé.

L'utilisation d'un fichier de configuration lu lors du démarrage de l'application permet de redéployer facilement l'application : le placement des processus peut être modifié de manière indépendante de la génération de code. En revanche, la structure de chaque processus ne peut être modifiée. L'autre solution consiste à inclure ces informations directement dans le code généré. Cette solution optimise le temps de démarrage de l'application. En revanche, toute modification du déploiement de l'application nécessite de re-générer le code de l'application.

Ainsi la solution à retenir correspond à un équilibre entre la nécessité d'envisager le redéploiement de l'application, et la recherche de performance. Dans le cas d'une orientation entièrement

dédiée à la facilité de déploiement, on peut envisager de construire des exécutables capables d'instancier n'importe quel composant de l'application et configurés dynamiquement à l'aide d'un fichier de démarrage spécifiant les instances à créer pour chaque exécutable. Si l'optimisation du temps d'exécution est le critère prioritaire, on défini des exécutables correspondant à une partition de l'application, et intégrant l'ensemble des informations de démarrage nécessaires.

Dans certains cas, le placement des exécutables ne peut pas être connu à l'avance et dépend d'éléments extérieurs à la spécification. C'est souvent le cas des applications réparties à large échelle, ou des applications client / serveur auxquelles les clients peuvent se connecter de manière asynchrone depuis n'importe quel hôte réseau. Un exemple de ce type de spécification est fourni par l'étude de cas du chapitre 7 qui présente un système simplifié de messagerie instantanée. Aucune information de placement n'est associé à ce type d'exécutables. Aucune instance statique ne peut être déployée dans un exécutable sans information de placement car la référence de ces instances dépend du placement de l'exécutable qui la contient et doit être connue dès le démarrage de l'application. Ces exécutables sont lancés par les utilisateurs de l'application et de manière indépendante de l'exécution des autres partitions.

4.5 Déploiement des instances dynamiques

Cette section décrit le déploiement des instances créées dynamiquement pendant l'exécution de l'application. La sémantique **L*f*P** étant indépendante du déploiement, quelque soit l'exécutable auquel cet élément est ajouté, le comportement global du système reste inchangé.

Cette propriété doit être assurée par l'exécutif associé au générateur de code. Pour cela, celui-ci doit offrir un service d'instanciation dynamique capable de créer et d'initialiser tout composant **L*f*P**. Le site devant porter la nouvelle instance n'est pas spécifié par la sémantique **L*f*P**. Le code correspondant au composant à instancier doit être intégré à l'exécutable pour que l'instanciation soit réalisable. La politique de placement des instances dynamiques doit donc être définie en fonction de la manière dont les exécutables de l'application sont structurés.

La solution la plus simple pour le placement des instances créées dynamiquement est de les intégrer au site **L*f*P** qui crée la nouvelle instance. Cette politique est relativement simple à implémenter et respecte la sémantique du langage, mais d'autres politiques sont possibles pour prendre en compte des propriétés telles que la charge des différents noeuds de l'application, ou l'optimisation du placement des composants.

Afin de séparer le choix du site d'instanciation des aspects liés à la génération de code, la création des nouvelles instances est réalisée par l'exécutif en fonction des paramètres fournis par le code généré. Les informations à préciser sont le type de composant à instancier et les valeurs des attributs à initialiser. Cette solution permet de séparer la politique de placement des nouvelles instances du processus de génération de code. Dès lors, le placement des nouvelles instances peut être confié à l'exécutif **L*f*P** en fonction de paramètres définis statiquement dans les informations de déploiement ou dynamiquement en fonction de l'état de l'environnement d'exécution lors de la création de l'instance.

4.6 Structure d'une application L*f*P

La figure 4.2 présente la structure d'une application répartie générée à partir d'une spécification **L*f*P**. La partie centrale de l'application correspond aux composants **L*f*P** dont le code est directement généré à partir de la spécification. Ils implémentent le comportement des classes et médias définis dans le modèle en s'appuyant sur un ensemble de fonctions de bas niveau regroupées dans une bibliothèque d'exécution constituant l'exécutif (ou *runtime*) **L*f*P**. La traduction

des classes et médias **L*f*P** est présentée en détail par le chapitre 6 qui présente les *patterns* de génération pour les opérateurs définis sur les types de données du langage **L*f*P**.

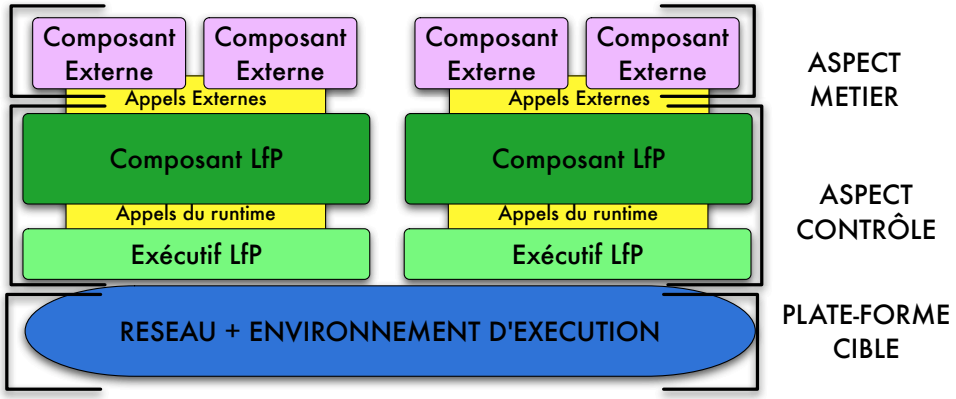


FIG. 4.2 – Structure d'une application générée à partir d'un modèle **L*f*P**.

Définition 17 *Le **runtime** est l'ensemble des fonctions et composants de bas niveau utilisées par le générateur de code pour implémenter la sémantique **LfP** dans un environnement donné.*

Le runtime **L*f*P** rassemble donc l'ensemble des opérations de bas niveau pré-programmées partagées par un grand nombre d'opérateurs du langage **L*f*P**. Il assure notamment la localisation des instances de composants, l'encapsulation des accès au réseau pour rendre la spécification indépendante du déploiement, ainsi que plusieurs services de plus bas niveau.

Le runtime fournit également un ensemble de composants réutilisés par le générateur de code pour implémenter les types natifs du langage **L*f*P** ainsi que les opérations qui leur sont associés. L'implémentation de ces types peut varier grandement en fonction de l'environnement d'exécution réel. Le runtime protège donc le code généré (et le générateur) des évolutions de l'environnement d'exécution.

Chaque instance de runtime définit un site de l'application, et donc une unité de déploiement. Un site **L*f*P** doit donc être entièrement déployé sur un seul noeud du réseau physique, et il est possible de déployer plusieurs sites **L*f*P** sur un même noeud. D'après ce qui précède, la figure 4.2 représente une application composée de deux sites qui peuvent être déployés soit sur le même noeud, soit sur deux machines distinctes.

Pour chaque site, une partie du runtime est partagée entre les composants instanciés sur un même site, par exemple la gestion des accès distants ou certains services de bas niveau comme l'instanciation dynamique. Chaque composant dispose également d'une partie spécifique du runtime qui gère l'état du composant et des files de messages (*binders*) qui lui sont associés. La structure et l'implémentation du runtime sont présentés au chapitre 5.

Les interactions entre les composants **L*f*P** et la partie de traitement de donnée de l'application (partie métier) sont réalisées par l'intermédiaire des *appels externes* (appels des méthodes des composants externes implémentant la partie métier de l'application). L'ensemble formé par un composant **L*f*P**, et l'ensemble des composants externes auxquels il est relié forment un composant de l'application. L'implémentation de ces interaction sera précisée à la section 6.4.2 qui présente l'implémentation des opérateurs des types opaques.

4.7 Conclusion

Ce chapitre a présenté la manière dont les spécifications **L*f*P** sont déployées sur une plate-forme cible. Il a précisé les informations nécessaires au déploiement d'une spécification **L*f*P** : placement des composants et structuration des exécutables de l'exécution.

La sémantique du langage **L*f*P** étant indépendante du déploiement, elle ne définit pas de contraintes sur le placement des composants. En conséquence, la politique de placement peut être laissée à l'utilisateur en fonction des besoins de son application ou de l'environnement dans lequel elle s'exécute. Les composants définis par la spécification **L*f*P** sont regroupés en exécutables qui définissent une unité de répartition et sont ensuite placés sur les différents noeuds du réseau.

Les informations nécessaire à l'initialisation de chaque exécutable sont déduites des informations de déploiement fournies par l'utilisateur. Celles-ci doivent définir la liste des exécutables à créer, la liste des composants contenus dans chaque exécutable ainsi que son placement sur l'architecture cible.

Le déploiement d'une spécification **L*f*P** dans un environnement réparti nécessite que les opérations de communication définies par le langage soient implémentées pour des communications locales ou distantes. Cette tâche a été confiée à un exécutif qui assure l'indépendance du code généré (et donc des règles de génération de code) vis à vis du déploiement et de la répartition en proposant une interface de communication unifiée et indépendante de la localisation des composants.

De manière plus générale, l'utilisation d'un exécutif permet de fournir une interface stable pour les services requis de manière récurrente par le code généré. Le rôle principal de l'exécutif est de fournir un environnement stable au code généré quelque soit la plate-forme sous-jacente. L'exécutif associé à la génération de code pour **L*f*P** est détaillé au chapitre 5.

Chapitre 5

L'exécutif L*f*P

5.1 Introduction

L'objectif de ce chapitre est de définir l'exécutif nécessaire au code réparti généré à partir d'une spécification **L*f*P**. Le processus de génération de code prend en entrée deux types d'informations complémentaires : la spécification **L*f*P** qui définit les aspects comportementaux à implémenter, et les informations de déploiement qui spécifient comment les composants générés sont répartis sur l'architecture d'exécution. La génération de code produit les informations de démarrage de l'application et les composants qui implémentent le comportement des composants **L*f*P** (cf. figure5.1). L'exécutif **L*f*P** est définit par l'ensemble des services requis pour l'exécution du code généré.

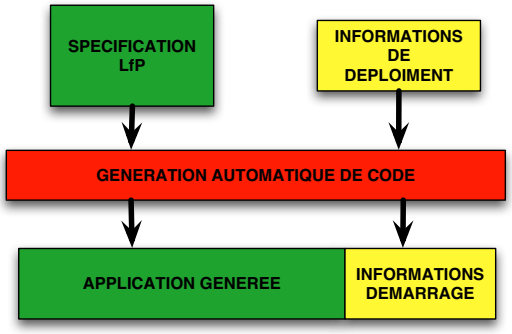


FIG. 5.1 – Le processus de génération de code

L'exécutif fournit donc une plate-forme virtuelle permettant l'exécution du code généré en définissant une interface identique quelque soit la plate-forme cible sous-jacente. Il doit donc éviter que le code généré ait recours aux interfaces spécifiques d'une plate-forme cible.

Les principaux services rendus par l'exécutif sont :

- le transport des messages de l'application ;
- la création et la localisation des instances des composants **L*f*P** ;
- l'implémentation des binders et des types de base du langage **L*f*P**.

Ces trois services couvrent l'ensemble des aspects du langage dont l'implémentation dépend de la plate-forme cible. L'interface de communication permet d'implémenter la sémantique des instructions de communication du langage. Pour assurer le transport des messages, l'exécutif doit pouvoir localiser chaque instance de composant ou de binder en fonction de sa référence. Les structures de données nécessaires sont construites lors de l'instanciation du composant. Enfin, les types de données de base du langage sont implémentés par l'exécutif sous forme d'une bibliothèque de composants utilisés directement par le code généré.

L'objectif de ce chapitre est d'identifier et de présenter l'ensemble des composants nécessaires à la mise en place d'un exécutif remplissant ces fonctionnalités. Une implémentation prototype utilisant le langage Java est proposée. La section 5.2 présentera les fonctionnalités remplies par l'exécutif et propose une structuration générique. La section 5.3 présente une implémentation de cette architecture utilisant le langage java ; cette implémentation servira de plate-forme de déploiement pour le prototype de générateur de code développé dans le cadre de cette thèse. La section 5.4 présente le format des informations de démarrage utilisées par cette implémentation pour spécifier le déploiement de la spécification **L*f*P**. La section 5.5 présente les perspectives de développement de l'exécutif **L*f*P**, et son intégration avec des plate-formes existantes. Enfin, la section 5.6 clôture ce chapitre de présentation de l'exécutif **L*f*P**.

5.2 Fonctionnalités de l'exécutif L*f*P

L'objectif de l'exécutif **L*f*P** est de fournir une interface de bas niveau contenant l'ensemble des structures et fonctionnalités permettant d'implémenter la sémantique du langage. On y trouve donc l'ensemble des primitives couramment utilisées par le code généré, ainsi que l'ensemble des bibliothèques de données requises pour assurer l'implémentation d'une spécification **L*f*P**.

L'exécutif doit rendre les services suivants :

- la gestion des accès au réseau pour la communication entre les sites de l'application ;
- la résolution des références réparties vers les composants **L*f*P** et les binders ;
- le transfert des messages de l'application.
- l'implémentation des binders ;
- la gestion des ressources allouées aux composants (*threads* et mémoire) ;

L'ensemble de ces services permet de s'abstraire de la plate-forme cible et de fournir un environnement stable quel que soit l'environnement d'exécution réel.

5.2.1 Structure de l'exécutif

La structure de l'exécutif est illustrée sur la figure 5.2.

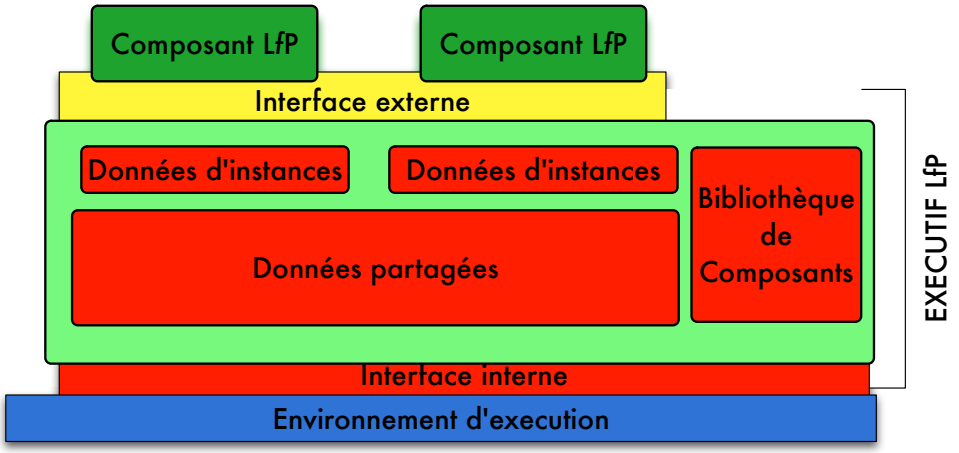


FIG. 5.2 – structure logique de l'exécutif **L*f*P**

L'interface externe définit la liste des fonctions que l'exécutif met à disposition des composants **L*f*P** (code généré). Elle les isole de la plate-forme de déploiement en proposant une interface

dont la sémantique et la structure reste fixe. C'est une condition nécessaire à la portabilité du code généré. Le travail de portage de **L*f*P** vers une nouvelle plate-forme est celui au portage de l'exécutif.

La structure de cette partie de l'exécutif est fortement dépendante du code généré pour les composants et du langage cible considéré. Pour cela, elle sera définie plus précisément au fur et à mesure que les traductions des opérateurs **L*f*P** seront fournies.

L'interface interne définit l'ensemble des primitives permettant d'assurer la communication entre les sites de l'application, donc entre deux instances de l'exécutif. Elle doit pouvoir traiter les requêtes suivantes :

- ajout / lecture d'un message dans un binder local ;
- refus / acceptation d'un message ;
- demande de référence d'un port d'une classe ;

Ces requêtes sont reçues par l'exécutif local au composant cible. Néanmoins, les informations envoyées ou retournées doivent parfois transiter par le réseau pour être traitées. Ces informations sont souvent partiellement ou entièrement constituées de types de données structurés générés à partir des types de données provenant de la spécification **L*f*P**. Il faut donc prévoir les fonctions permettant d'encoder et de décoder ces données pour les transporter sur le réseau.

Les données d'instances sont les données spécifiques à chaque instance de composant. Elles sont constituées des informations relatives aux transactions en cours entre le composant et les binders reliés à ses ports.

Les données partagées définissent les données locales au site de l'application qui permettent aux composants d'interagir. Il s'agit des structures de résolution des références **L*f*P** et les services associés : instanciation dynamique et destruction des instances de composants.

La structure de résolution des références peut être implémentée sous la forme d'une table de composant. Lorsqu'un composant est créé sur le site local, il doit être inséré dans cette table qui conserve les informations nécessaires à la gestion de ses ressources, notamment la liste des binders qui lui sont associés. C'est cette structure qui sera utilisée lors de la destruction du composant, et lors des requêtes sur les valeurs de ses ports.

L'environnement d'exécution représente la plate-forme cible d'exécution de l'exécutif et de l'application générée. Il s'agit du système d'exploitation et d'un éventuel *middleware* utilisé pour implémenter l'exécutif.

La bibliothèque de composants désigne un ensemble de composants facilitant l'implémentation des spécifications **L*f*P**. Ces composants sont utilisés aussi bien par le générateur de code pour générer les éléments des composants dépendant de la plateforme que par les structures internes de l'exécutif.

Cette bibliothèque regroupe :

- les références vers les composants ;
- les binders et leurs références ;
- les messages échangés par l'application.

5.2.2 Structure de la bibliothèque de composants

Les composants de cette bibliothèque sont utilisés pour implémenter les types **L*f*P** et par le code généré.

Les références vers les composants L*f*P

Les choix de conception de **L*f*P** imposent les contraintes suivantes sur les références de composants :

- transparence à la répartition : elles peuvent désigner des composants locaux ainsi que des composants distincts ;
- indépendance vis à vis de leur localisation : elles peuvent migrer d'un site vers un autre ;
- déréférenciation : les références vers les classes du modèle permettent d'accéder aux ports de l'instance référencée.

Ces contraintes interdisent l'utilisation des mécanismes classiques de pointeurs proposés par les langages de programmation. L'exécutif doit donc fournir un ensemble de composants génériques pour l'implémentation des références **L*f*P**.

L'implémentation des références **L*f*P** varie en fonction de l'environnement cible, mais on y trouve toujours au moins deux informations : la localisation de l'instance qui détermine sur quel site est localisée le composant et la référence locale qui désigne l'instance sur son site local.

L'implémentation des références de classes **L*f*P** doit permettre d'implémenter l'opérateur de déréférenciation (cf. 3.4.4). L'implémentation de cette fonctionnalité dépend du langage et de la plate-forme cible ; l'exécutif est donc ici particulièrement important pour fournir une interface stable aux composants **L*f*P**. Si l'exécutif fait le choix de définir un type de référence par composant du modèle, chaque port de la classe référencée peut être introduite comme un attribut de la structure de données correspondante. Si l'exécutif utilise un seul type pour représenter les références des composants **L*f*P**, une fonction associée doit permettre de retrouver le port correspondant.

Les binders et leurs références

Les binders permettent de définir les files de messages reliant les composants. Ces composants sont interfacés directement avec les composants **L*f*P** ; leur interface doit être parfaitement stable. De plus, le comportement et l'interface des binders reste identique pour toutes les spécifications **L*f*P**. En conséquence, ces composants sont implémentés directement dans l'exécutif et leur implémentation doit prendre en compte les spécificités de la plate-forme de déploiement.

L'interface des binders est définie par :

- une fonction d'ajout de message ;
- une fonction de lecture de message ;
- une fonction d'acceptation de message ;
- une fonction de refus de message ;
- une fonction d'abandon de lecture.

La fonction d'ajout de message permet d'insérer un message dans le binder si ce dernier n'est pas déjà plein. Son comportement doit implémenter la sémantique vue à la section 3.4.5 pour l'ajout de message dans un binder.

Les autres fonctions permettent d'implémenter la sémantique des lectures de messages dans les binders. Elle est implémentée par une transaction qui commence par la demande de lecture d'un message et se termine soit par un abandon de lecture, soit par l'acceptation du message fourni par le binder[1].

La lecture de message retourne le prochain message fourni par ce binder, sans le retirer du binder ;

Le refus de message permet au lecteur de refuser un message dans les cas suivants :

- la garde associée à l'opération de lecture n'est pas vérifiée ;

[1] Ce comportement est illustré par la figure 3.12 p. 69 qui détaille le fonctionnement d'une opération de lecture sur un binder

– la méthode activée par le message n'est pas activable dans l'état courant.
Le message refusé reste dans le binder ; si un autre message est disponible, il peut être proposé.

L'abandon de lecture permet au lecteur de signifier au binder qu'il abandonne l'opération de lecture sans lire de message (cette opération est appelée si le composant a reçu un message valide fourni par un autre binder).

L'acceptation de message signale une fin de l'opération de lecture et le retrait du message du binder.

Les références vers les binders sont représentées en **L*f*P** par le type `port`. La sémantique de ce type imposent aux références vers les binders de nombreux points communs avec les références vers les composants :
– indépendance à la répartition ;
– indépendance vis à vis de leur localisation ;
– interface identique pour toutes les spécifications **L*f*P**.

Pour ces raisons, on implémente les références vers les binders dans l'exécutif **L*f*P**. Le mécanisme de résolution des références des binders étant le même que celui des références des composants, elles partagent les mêmes structures de données de l'exécutif.

Le type message

Ce type permet de traiter les messages échangés par les composants **L*f*P**. Il est inutile de définir un type pour chaque sorte de message transporté par l'application. De cette manière, le type utilisé pour implémenter les messages de l'application est unique et réutilisable pour toutes les spécifications. Nous l'avons donc intégré à l'exécutif **L*f*P**.

Le type message est implémenté par un composant définissant les informations suivantes :
– le tableau *discriminant* sert à transporter les valeurs du discriminant ;
– le tableau *paramètres* sert à transporter le contenu du message ;
– un champ *type* définit le type du message (soit un message de données, soit un message d'activation de méthode, soit un message contenant la valeur de retour) ;
– un champ *méthode* contient le nom de la méthode activée si le message est un message d'activation.

Chacun de ces champs doit contenir ces valeurs sous une forme pouvant être envoyée sur le réseau. Chacune des valeurs de la partie donnée comme de la partie discriminant sera donc ajoutée sous forme encodée pour le transport sur le réseau, c'est à dire sous la forme d'un tableau d'octets. La fonction de dé-sérialisation sera utilisée lors du décodage du message.

Ces informations sont suffisantes pour déterminer si le message est valide ou non dans un contexte donné (attente de message ou activation de méthode). Bien que ce soit possible, il n'est pas nécessaire d'utiliser un type dédié pour chaque message de l'application. Utiliser un type unique disposant d'une interface fixe facilite l'implémentation du générateur de code.

5.3 Implémentation de l'exécutif

Cette section présente l'interface et l'implémentation de l'exécutif réalisée pour le prototype du générateur de code. Elle est découpée en deux parties : la première présente la structure interne de l'exécutif réalisé, la seconde présente l'interface de l'exécutif avec le code généré. Cette section couvre les aspects liés à l'implémentation de l'exécutif. Une documentation plus complète de ce prototype ainsi que le code source correspondant sont disponibles sur [89].

5.3.1 Architecture du prototype de l'exécutif

Nous allons maintenant étudier les structures de données utilisées pour implémenter la partie interne de l'exécutif en Java. Pour cela, nous présentons l'interface des classes de l'exécutif et les mécanismes d'implémentation utilisés.

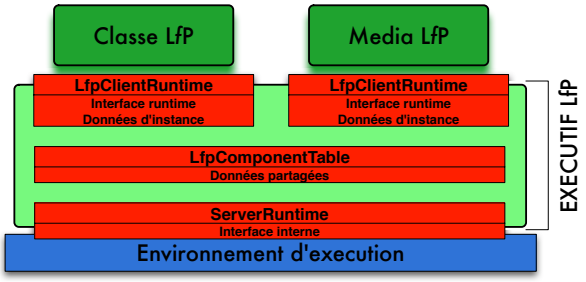


FIG. 5.3 – Association entre fonctionnalités de l'exécutif et classes d'implémentation

La figure 5.3 reprend la description de l'exécutif de la figure 5.2 (page 90) en précisant les principales classes Java utilisée pour l'implémentation de chaque fonctionnalité. La figure représente un site d'une application **L*f*P**, celui-ci est constitué d'une instance de l'exécutif et un ensemble d'instances de composants **L*f*P**.

Au niveau de chaque site, on distingue une instance de l'interface interne (`ServerRuntime`), et une instance des données partagées entre les composants instanciés sur un site **L*f*P** (`LfPComponentTable`). Pour chaque composant instancié sur le site on distingue une instance de `LfPClientRuntime` qui assure l'interface entre le composant et l'exécutif et contient la majorité des données d'instances.

Afin de simplifier la lecture de la figure 5.3, la bibliothèque de composants n'est pas représentée, et seules les classes de haut niveau sont présentées.

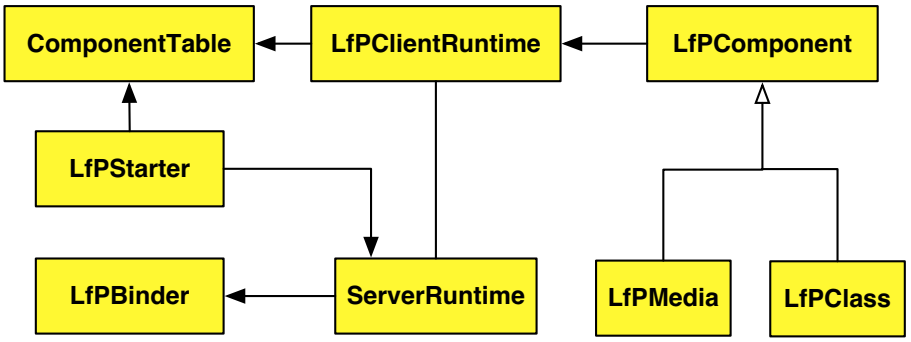


FIG. 5.4 – Diagramme de classe simplifié de l'exécutif **L*f*P**

La figure 5.4 complète la figure 5.3 en présentant le diagramme de classe simplifié de l'exécutif **L*f*P**.

Les données partagées vues dans la présentation générique de l'exécutif à la section 5.2 sont implémentées par la classe `LfPComponantTable`. Cette classe contient la table des instances de composants du noeud local ainsi que l'interface d'accès à cette structure de données. C'est donc cette classe qui s'occupe du service de localisation des objets et permet de retourner leur référence Java locale. Ce composant peut être implémenté très simplement à l'aide d'une table de hachage qui retourne la référence d'un objet en fonction d'une clé définie lors de l'instanciation du composant.

Les classes `LfPClientRuntime`, `LfPComponent`, `LfPMedia` et `LfPClass` correspondent aux données d'instances et à l'interface externe des de l'exécutif (cf. figure 5.2). Les classes `LfPMedia` et `LfPClass` contiennent les attributs nécessaires respectivement à l'implémentation des classes et des médias :

- *self* contient la référence **L*f*P** de l'instance courante du composant ;
- *runtime* de type `LfPClientRuntime` contient la référence vers le reste des données d'instances.

Toute classe qui implémente une classe **L*f*P** (*resp.* un média **L*f*P**) hérite de la classe `LfPClass` (*resp.* `LfPMedia`).

La classe `LfPClientRuntime` rassemble la grande majorité des primitives fournies par l'exécutif. Elle peut être considérée comme l'implémentation de l'interface externe de l'exécutif telle qu'elle a été définie à la section 5.2.1. L'implémentation de ces services est basée sur les données partagées contenues dans `ServerRuntime`. Les services fournis par `LfPClientRuntime` sont donc les suivants :

- *échanges de messages*, ce qui inclut notamment la gestion des opérations à réaliser sur les binders pour les lectures de messages.
- *instanciation dynamique* de nouveaux composants.

Les opérations en cours avec les binders sont toujours réalisées par l'intermédiaire de canaux de communication de type `Socket`. Cette classe doit donc conserver un tableau des références des binders ainsi que l'état de l'opération en cours. Cette exigence est réalisée en utilisant un tableau de Sockets correspondant à l'ensemble des opérations en cours. Enfin, l'interface interne est principalement traitée par la classe `ServerRuntime` qui implémente la réception des requêtes internes à l'exécutif et les transmet aux composants appropriés pour le traitement. Chaque composant chargé du traitement renverra directement la valeur de retour à l'émetteur de la requête.

Les requêtes transmises entre les instances de l'exécutif (donc entre les sites de l'application) sont implémentées par la classe `LfPRequest` (non représentée sur le diagramme). Elle définit le type de la requête, ses paramètres et identifie l'émetteur. Les paramètres sont transmis séparément de la requête, sur le même canal de communication.

L'initialisation de l'exécutif est assurée par la classe `LfPStarter` qui réalise les opérations suivantes lors du démarrage de l'application :

- initialisation des structures internes de l'exécutif (table des composants et mécanisme de réception des requêtes) ;
- lecture des informations de déploiement ;
- création des instances statiques locales et initialisation de leurs références ;
- initialisation des références statiques distantes ;
- démarrage de l'exécution des instances statiques.

Ces opérations sont rendues possibles par la lecture des informations de déploiement spécifiées dans un fichier de configuration passé en paramètre lors du démarrage de l'application. Ce fichier dont la structure est présentée à la section 5.4 contient la localisation de toutes les instances statiques du modèle.

Il est possible de rendre le déploiement partiellement dynamique, par exemple dans le cas d'un système client / serveur où le site d'instanciation des clients n'est pas connu à l'avance. Dans ce cas, `LfPStarter` accepte un argument supplémentaire qui définit un ensemble de composants à créer lors de l'initialisation du noeud local. Cette fonctionnalité permet de créer des exécutables dont le placement n'a pas été défini dans le fichier de configuration car ils ne contiennent pas d'instance statique.

5.3.2 Implémentation de la bibliothèque de composants de l'exécutif

L'exécutif repose également sur un ensemble de composants chargés d'implémenter directement une partie de la sémantique du langage **L*f*P** : binders et messages sont implémentés par des classes Java incluses dans l'exécutif et instanciées en fonction du comportement du code généré à partir de la spécification.

Les binders

Les binders **L*f*P** sont implémentés dans l'exécutif par la classe `LfPBinder`. Un binder est une file de messages bidirectionnelle avec une sémantique particulière associée aux opérations d'accès en lecture et en écriture.

L'implémentation repose donc fortement sur la classe Java `LinkedList` fournie par les librairies standards du langage, et qui implémente une liste chaînée. Les binders sont implémentés comme des composants passifs (ils n'ont pas de thread associée). Chaque appel d'une opération sur le binder donne lieu à la création d'une *requête* qui peut être bloquante ou non.

Les requêtes potentiellement bloquantes sont :
- les ajouts de messages sur les binders synchrones ;
- les lectures de message.

Les requêtes non bloquantes sont :
- acceptation de message ;
- refus de message ;
- abandon de l'opération.

Le comportement de ces composants est paramétrable en fonction d'un nombre réduit de paramètres connus lors de leur instanciation :
- leur capacité ;
- leur politique d'ordonnancement ;
- leur mode d'interaction.

La *capacité* d'un binder définit le nombre de messages qu'il peut contenir. La *politique d'ordonnancement* définit le prochain message qui peut être retourné lors d'une demande de lecture de message. Enfin le *mode d'interaction* définit le comportement de la fonction d'ajout de message dans le cas où la file est pleine : soit le message est abandonné, soit la requête est bloquante jusqu'à ce qu'un emplacement se libère.

L'implémentation d'un binder peut donc être réalisée en Java en utilisant une instance de la classe `LinkedList` pour chaque direction du binder (média vers classe et classe vers média). La file à utiliser est choisie en fonction de la fonction appelée, et du composant qui l'a émise.

Plusieurs requêtes peuvent être envoyées simultanément sur chaque file de messages, il est donc nécessaire de protéger les accès à ces données. Pour cela, on utilise les mécanismes Java traditionnels en incluant les opérations devant être réalisées de manière atomique dans des blocs *synchronize* qui fournissent un accès en exclusion mutuelle sur les composants cibles. Cette instruction sera utilisée pour toutes les séquences d'instructions manipulant le contenu d'une des listes chaînées.

Il est impossible d'utiliser un thread par requête en cours sur les binders d'un site car ce nombre serait potentiellement beaucoup trop élevé. Le nombre de requêtes simultanées n'étant théoriquement pas limité, la solution du *pool* de threads de taille bornée n'est pas applicable ici car elle risquerait d'introduire des interblocages. C'est par exemple le cas si toutes les requêtes en cours attendent qu'un message soit consommé par une requête mise en attente par manque de thread disponible. Néanmoins, une solution ne limitant pas le nombre maximum de thread n'est pas envisageable.

Chaque direction du binder doit donc gérer explicitement une liste de requêtes en attente. Lorsqu'une requête modifie l'état de la file de messages, le binder tente d'effectuer toutes les requêtes en attente jusqu'à ce que son état n'évolue plus.

De cette manière, les requêtes sur les binders peuvent être implémentées à l'aide d'un *pool* de thread dont la taille ne dépend pas du nombre de binders instanciés sur le noeud. Si aucun parallélisme effectif n'est attendu de l'exécutif, ce *pool* peut être réduit à une seule thread.

Les méthodes d'accès au contenu du binder ne sont pas synchronisées (pas de modificateur `synchronized` dans leur déclaration). La cohérence de la structure du binder est assurée par les structures de données manipulées en interne. Ces méthodes peuvent donc être appelées simultanément par plusieurs threads.

L'interface de la classe `LfPBinder` est la suivante :

public void addMessage(LfPMessage message, ObjectOutputStream output, Socket channel)

Cette méthode permet d'ajouter le message contenu dans le paramètre `message` au binder. La file dans laquelle le message doit être ajouté est choisie en fonction du type de la référence de l'expéditeur. Cette référence est contenue dans le message. Le paramètre `output` contient la référence du canal de communication dans lequel l'acquittement de l'opération doit être envoyé. Le paramètre `channel` est nécessaire car cette fonction a la charge de fermer proprement la socket de communication une fois l'opération effectuée.

L'appel de cette méthode n'est jamais bloquant, si l'opération ne peut être effectuée, la requête est stockée dans la file de requête interne au binder, et l'acquittement n'est pas envoyé, ce qui bloque le composant appelant conformément à la sémantique **L*f*P**.

Les requêtes mises en attentes sont réexécutées dès que l'état de la file de messages cible change.

public void readMessage(Socket channel, ObjectInputStream input, ObjectOutputStream output, LfPComponentReference requester)

Cette fonction permet de lire un message dans un binder. Le canal de communication est passé par les paramètres `channel`, `input` et `output` qui permettent la communication avec le composant qui a émit la requête de lecture et dont la référence est fournie par le paramètre `requester`.

Le paramètre `output` permet d'envoyer les messages au composant, `input` permet de récupérer sa réponse pour déterminer si le message a été accepté ou refusé. Enfin, le type exact de `requester` (`LfPMediaReference` ou `LfPClassReference`) permet de déterminer la file dans laquelle le message doit être lu.

public LfPBinderReference getReference()

Cette méthode retourne la référence **L*f*P** complète de l'instance du binder.

Les messages de l'application

Les messages échangés par l'application sont toujours représentés par une instance de la classe `LfPMessage`. Cette classe contient les informations suivantes :

- le type du message (message d'activation ou message de donnée) ;
- le discriminant associé au message sous forme d'un tableau de valeurs ;
- le nom de la méthode activée s'il s'agit d'un message d'activation ;
- les valeurs transportées par le message :
  - dans le cas d'un message d'activation il s'agit des paramètres de la méthode,
  - dans le cas d'un message de données, il s'agit des données véhiculées par le message,

– la valeur de retour dans le cas d'un message de retour d'un appel de fonction ;
– la référence du dernier binder dans lequel le message a été envoyé.

La référence du dernier binder a avoir stocké le message est utilisée par les classes lors de la réception d'un message d'activation. Elle permet de vérifier que le binder qui a fourni le message correspond au port d'activation de la méthode. Cette valeur est mise à jour par l'exécutif lors de chaque insertion du message dans un binder. La classe `LfPMessage` est bien entendu taguée `Serializable` ce qui permet de l'envoyer directement sur une socket Java sans avoir à s'occuper du codage de son contenu.

L'interface de la classe `LfPMessage` est donc principalement une interface de type *get and set* sur ses attributs :

`public Object getDiscriminant(int index)`

Cette méthode permet d'accéder aux valeurs du discriminant du message. le paramètre `index` définit la position de la valeur recherchée dans le discriminant.

`public Object getParameter(int index)`

Cette méthode est le pendant de `getDiscriminant` pour la partie *donnée* du message. Elle permet donc d'accéder au contenu d'un message de données, ou aux paramètres d'un message d'activation de méthode.

`public void setDiscriminant(Object value, int index)`

Cette méthode permet de spécifier le discriminant d'un message. la valeur à insérer dans le discriminant est passée par le paramètre `value`, et sa position par le paramètre `index`.

`public void setParameter(Object value, int index)`

Cette méthode est le pendant de `setDiscriminantValue` pour la partie *données* du message.

`public void setMethodName(String newName)`

Cette méthode permet de spécifier que le message est un message d'activation de méthode. Le nom de la méthode à activer est fourni par le paramètre `newName`.

`public void setMessageSender(LfPReference sender)`

Insère la référence **L*f*P** du composant qui envoie le message. Cette information sera utilisée par l'opération d'insertion du message dans le binder pour définir la file à laquelle il doit être ajouté.

`public void setReturnValue(Object value)`

Cette méthode permet de spécifier que le message est un message de retour, ainsi que la valeur retournée.

Les références L*f*P

Une référence **L*f*P** désigne soit un composant (classe ou média), soit un binder (port). La référence identifie l'élément pointé de manière unique pour toute l'application et doit permettre aux autres composants de l'application d'accéder à l'élément référencé. Une référence **L*f*P** est donc définie par deux éléments : la définition du noeud sur lequel l'instance a été créée ; la référence de l'instance dans la table des instances de composants. La référence d'un noeud sera définie par ne nom d'hôte complet du noeud tel que retourné par la commande Unix *hostname*, et le numéro de port sur lequel le serveur de requêtes est en attente. De cette manière, il est possible déployer plusieurs sites **L*f*P** sur la même machine ce qui s'est révélé particulièrement pratique lors des phases de test de l'exécutif.

Comme illustré par la figure 5.5, les références **L*f*P** sont implémentées par plusieurs classes en fonction de l'objet désigné par la référence.

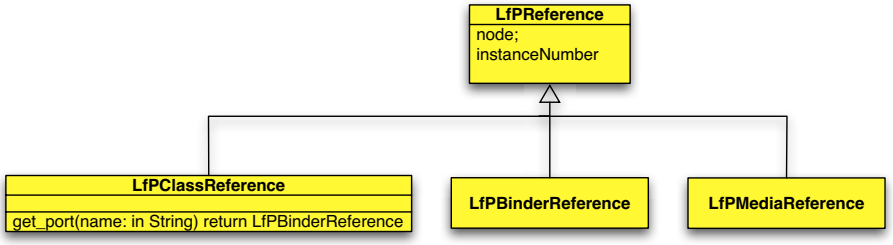


FIG. 5.5 – Diagramme de classes de l'implémentation des références

La classe `LfPReference` est abstraite et permet de définir un parent commun entre toutes les références des composants.

La classe `LfPClasseReference` implémente les références vers les classes **L*f*P**. Cette classe introduit une fonction de dé-référenciation qui retourne la référence du binder correspondant à un port dont le nom est passé en paramètre.

La classe `LfPMediaReference` implémente les références vers les médias.

La classe `LfPBinderReference` implémente les références vers les binders. Elle implémente directement les types *port* définis dans le langage **L*f*P**. Cette implémentation de l'exécutif **L*f*P** n'implémente pas un typage fort des références car les contraintes de typage ont été vérifiées lors de la génération du code.

Toutes les classes de l'exécutif représentant des références **L*f*P** implémentent l'interface suivante :

`public Socket connectToHost()`

Cette méthode retourne un canal de communication vers l'hôte **L*f*P** qui gère l'instance désignée par la référence.

`public boolean isEqual(LfPReference other)`

Cette méthode retourne un booléen égal à true si la référence passée en paramètre désigne la même instance de composant que la référence sur laquelle la méthode est invoquée.

`public boolean isLocal()`

Cette méthode retourne un booléen égal à true si l'instance référencée est sur l'hôte **L*f*P** local (même adresse IP et même numéro de port).

La classe `LfPComponantReference` implémente une méthode supplémentaire :

`public getBinder(String portName)`

Cette méthode permet d'accéder à un port d'un composant. Le nom du port est passé en paramètre dans une chaîne de caractères.

5.3.3 Interface externe de l'exécutif

L'interface externe de l'exécutif définit l'ensemble des fonctionnalités que celui-ci offre aux composants générés. Elle est assurée par la classe `LfPClientRuntime`. Celle-ci rend principalement deux sortes de services : le traitement des messages et l'instanciation de composants **L*f*P**. La suite de cette section fournit une description des principales fonctions.

Le traitement des messages

Le traitement des messages est assuré par les fonctions suivantes :

```
public LfPMessage getMessage (LfPBinderReference[] targetBinder )
```

Cette fonction implémente la lecture de message sur une liste de binder (potentiellement réduite à un seul binder). La liste doit être ordonnée par ordre de priorité croissante. Si aucune lecture n'est déjà en cours, une nouvelle lecture est démarrée. Si une opération de lecture a déjà été démarrée par ce composant, le message précédemment retourné est refusé et le prochain message disponible est retourné.

Cette méthode est bloquante lorsqu'aucun message n'est disponible.

```
public void commit (LfPMessage message)
```

Cette fonction permet d'accepter un message. Elle met fin à la transaction en cours. Le paramètre doit référencer le message accepté par le composant.

Cette méthode n'est jamais bloquante.

L'instanciation des composants L*f*P

L'implémentation des composants statiques est réalisée par les fonctions suivantes :

```
public LfPComponentReference newStaticComponant(String instanceName, String
typeName, String [] attributeList , Object [] initValuesList )
```

Cette fonction implémente l'instanciation d'un composant statique.

Les paramètres de cette méthode sont les suivants :
- le nom de la variable globale qui définit cette instance ;
- le nom du type de composant à instancier ;
- un tableau qui contient la liste des noms des attributs initialisés lors de l'instanciation ;
- un tableau qui contient la liste des valeurs des attributs du tableau précédent.

La valeur de retour est une référence **L*f*P** vers le composant statique.

Si la référence est locale, le composant est effectivement instancié et inséré dans le système de gestion des références.

Si le composant est déployé sur un noeud distant, seule sa référence est créée. Les informations nécessaires à l'instanciation et à la construction de la référence sont définies dans les informations de déploiement.

Cette fonction est appelée dans le code généré pour l'initialisation de chaque noeud du modèle.

```
public LfPBinderReference newStaticBinder( String instanceName, int capacity ,
String policy
```

Cette fonction implémente l'instanciation d'un binder statique.

Les paramètres de cette fonction sont les suivants :
- le nom de l'instance de binder statique ;
- la capacité en nombre de messages du binder ;
- sa politique d'ordonnancement sous forme d'une chaîne de caractère[2].

La valeur de retour est la référence **L*f*P** vers le binder statique dont le nom est défini par le premier paramètre.

Si la référence est locale, cette fonction crée l'instance de binder et l'insère dans le système de gestion des références. Sinon elle ne construit que la référence. Les informations nécessaires pour l'instanciation du binder et la création de la référence sont fournies par les informations de déploiement.

[2] Actuellement, cette fonction ne supporte que la création de binder `fifo`, c'est donc la seule valeur valide pour ce paramètre

Cette fonction est appelée dans le code généré pour l'initialisation de chaque noeud du modèle.

```
static LfPComponantReference newInstance (String className, String []
attributeNames , Object [] initialValues )
```

Les paramètres sont les suivants :

– le nom de la classe qui implémente le composant à instancier ;
– la liste des noms des attributs à initialiser ;
– la liste des valeurs initiales correspondantes.

Cette fonction crée dynamiquement une nouvelle instance de composant et l'insère dans le système de résolution de références. Elle implémente l'opération d'instanciation dynamique du langage **L*f*P**.

Si le composant est une classe **L*f*P**, ses ports sont initialisés, et les instances de binder correspondantes sont créées.

La valeur de retour de la fonction est une référence **L*f*P** vers la nouvelle instance.

5.4 Format des informations de démarrage

L'initialisation de l'application est paramétrée par les informations de déploiement. Elles définissent les éléments instanciés *statiquement* lors du démarrage de l'application. Ces informations sont lues par l'exécutif lors de la phase d'initialisation de chaque exécutable participant à l'application.

5.4.1 Informations nécessaire à la définition d'une instance statique

Une instance statique est définie par :

– le noeud sur lequel elle doit être instanciée ;
– sa clé dans la table des composants.

Le noeud qui porte l'instance est définit par un nom d'hôte ou une adresse IP et un numéro de port. La clé dans la table des composants est définie statiquement, les valeurs sont quelconques, mais deux composants d'un même noeud ne peuvent pas partager la même clé. Par convention, on utilise des valeurs négatives pour les clés des composants statiques afin de les différencier aisément des composants instanciés dynamiquement dont les clés seront strictement positives.

L'implémentation prototype ne permet pas de générer automatiquement ces informations qui doivent être spécifiées manuellement par l'utilisateur pour chaque déploiement de son modèle.

5.4.2 Structure du fichier de déploiement

La structure d'un fichier de déploiement est très simple, toutes les fonctionnalités envisagées au chapitre 4 n'ont pu être implémentées dans le prototype de l'exécutif. Le noeud *instances* sert de racine au fichier de déploiement. Les noeuds fils sont de type *instance* et définissent chacun une instance statique du modèle à l'aide des attributs suivants :

name : le nom de l'instance statique entièrement en majuscules.

Ce nom est celui définit dans le diagramme d'architecture pour les composants. Dans le cas d'un binder statique il s'agit du nom du port auquel le binder est associé.

Ce paramètre est passé en paramètre lors des appels aux méthodes `createStaticComponent` et `createStaticBinder` de l'exécutif ; il sert de clé pour déterminer la localisation de l'instance.

hostName : le nom DNS de la machine sur laquelle est créée l'instance ;

port : le numéro de port utilisé par le noeud sur l'hôte réseau ;

instanceNumber : la clé de l'instance dans la table des composants.

Chaque noeud de type *instance* permet donc de spécifier le placement d'une instance statique.

Les sites **L*f*P** sont définis par un couple *hostname* / *port* unique. Chaque site **L*f*P** sera exécuté dans une machine virtuelle distincte dans laquelle est créée un thread par composant.

Enfin, le numéro d'instance fournit une clé unique sur le site **L*f*P** permettant au service de nommage de retrouver la référence du composant à l'aide de la référence **L*f*P**.

Dans le cadre de ce prototype, la structure des exécutables n'est pas spécifiée, toutes les classes de l'application sont incluses dans l'exécutable. Le langage Java utilise un mécanisme de liaison dynamique, seules les classes requises par chaque processus sont donc chargées en mémoire.

5.4.3 Exemple de fichier de déploiement

```
<!DOCTYPE staticInstancesList >
<staticInstances>
<instance name="ITF" hostName="framekit dev.lip6.fr" port="12345" instanceNumber=" 3" />
<instance name="S1" hostName="framekit dev.lip6.fr" port="12345" instanceNumber=" 1"/>
<instance name="S2" hostName="glaucos dev.lip6.fr" port="12346" instanceNumber=" 1" />
<instance name="C1" hostName="lucifer.lip6.fr" port="12347" instanceNumber=" 1" />
<instance name="C2" hostName="themis.lip6.fr" port="12347" instanceNumber=" 2" />
<instance name="C3" hostName="scylla.lip6.fr" port="12347" instanceNumber=" 3" />
<instance name="C4" hostName="athena.lip6.fr" port="12347" instanceNumber=" 4" />
</staticInstances>
```

FIG. 5.6 – Exemple de fichier de déploiement

La figure 5.6 montre un exemple de fichier de déploiement. Le système est initialement composé de 7 instances de composants et / ou de binders statiques. Le type et les caractéristiques de l'élément à instancier sont retrouvés dynamiquement en fonction du nom du composant (premier paramètre de chaque noeud `instance`).

Cette figure présente six sites **L*f*P** instanciés sur six machines différentes. Le premier site est instancié sur la machine `framekit-dev.lip6.fr`. Ce site contient les deux instances statiques nommées *ITF* et *S1*. Ces deux composants seront instanciés dans cet ordre et leur exécution commencera lorsque le site sera entièrement démarré.

Les autres sites **L*f*P** contiennent chacun une instance statique de composant et sont instanciés sur une machine distincte. Les autres composants nécessaires à l'exécution des sites seront chargés dynamiquement par la machine virtuelle Java.

5.5 Perspectives de développement de l'exécutif

Cette section présente les perspectives de développement de l'exécutif **L*f*P** sur des plate-formes variées. Dans un premier temps, nous évoquons les perspectives d'implémentation de l'exécutif dans des environnements contraints en terme de ressources de mémoire et de calcul, puis nous présentons une évolution possible de l'exécutif **L*f*P** vers les applications réparties dans des environnements industriels standards.

5.5.1 Perspective dans les environnements contraints

Les éléments de la plate-forme java que nous avons utilisé sont les sockets et l'interface fournie par le langage pour la sérialisation des objets (leur encodage pour envoi sur le réseau). Seule

l'interface de sérialisation est spécifique au langage java. Cette interface évite d'implémenter directement les foncions d'encodage et de décodage des données pour leur envoi sur le réseau. Ceci n'est pas un obstacle car la sérialisation d'objets est bien maîtrisée par des protocoles tels que SOAP [84] ou IIOP [56].

Cette implémentation montre donc qu'il est possible de porter l'exécutif **L*f*P** sur une grande variété de plate-formes. Le projet MORSE doit en produire une version adaptée à la plate-forme drone de la société SAGEM et démontrer la portabilité de cette interface.

5.5.2 Liaison avec des environnements standards

D'autres perspectives sont également envisageables dans le domaine des systèmes d'information. Dans ce nouveau contexte, il est nécessaire d'intégrer **L*f*P** avec les intergiciels les plus utilisés tels que CORBA, COM/DCOM. la première approche consiste à utiliser un protocole standard tel que IIOP ou SOAP. Cependant, cela amène à construire plusieurs versions de l'exécutif utilisant chacune une plate-forme spécifique et incompatibles entre elles. Ce problème d'incompatibilité entre les différents intergiciels est appelé *paradoxe des intergiciels*.

La solution à ce problème est proposé par les intergiciels dits *schizophrènes* [62]. Ceux-ci se caractérisent par la capacité à séparer l'application du protocole de communication effectivement utilisé pour transporter les données sur le réseau. PolyOrb [66] montre qu'il est possible d'implémenter un intergiciel schizophrène de manière efficace.

L'aspect le plus intéressant de cet intergiciel est la séparation entre :

- la personnalité *applicative* qui définit l'interface avec les composants de l'application ;
- et la personnalité *protocolaire* qui définit le protocole de transport des données sur le réseau.

La personnalité applicative définit l'intergiciel utilisé pour définir l'interface des composants déployés sur un site de l'application. L'intérêt de PolyORB est de permettre de relier des composants ayant des personnalités applicatives différentes.

La personnalité protocolaire définit le protocole de transport de données de manière indépendante de la personnalité applicative. De cette manière, il est possible d'utiliser le protocole SOAP pour faire communiquer des composants CORBA.

Enfin, des composants utilisant des protocoles de communication différents peuvent communiquer grâce à des passerelles de traduction des messages. Cette fonctionnalité améliore la réutilisation d'application déjà développées en utilisant d'autres intergiciels.

L'exécutif **L*f*P** peut être envisagé comme une personnalité *applicative* de PolyORB ; les applications **L*f*P** pourront alors être reliées à toute les applications utilisant un protocole de communication implémenté par une personnalité protocolaire de PolyORB. Cette solution offre de bonnes perspectives pour l'utilisation de **L*f*P** pour le développement d'applications réparties industrielles, et leur intégration dans des environnements existants.

5.6 Conclusion

Ce chapitre a montré la structure d'un exécutif permettant l'implémentation d'un générateur de code pour le langage **L*f*P**. La première étape a présenté les services requis par le générateur de code. Il fournit un environnement d'exécution stable définissant :

- une abstraction des services que la plate-forme cible doit fournir au code généré ;
- une interface constante entre le code généré et son environnement d'exécution.

L'interface de l'exécutif avec les composants **L*f*P** est utilisée par les composants générés pour assurer :

- la création dynamique d'instances de composants ;
- les envois et réceptions de messages.

Enfin, certains types **L*f*P** de base comme les binders sont directement implémentés dans l'exécutif. Ils font partie de la bibliothèque de composants fournis pour faciliter l'implémentation des composants générés.

La section 5.3 a montré comment cet exécutif peut être implémenté dans le langage Java. Cette plate-forme a été choisie pour réaliser le prototype du générateur de code car elle fournit un ensemble de librairies réduisant la charge de développement sans faire d'hypothèses contraignantes sur l'environnement d'exécution. De plus, en utilisant une API d'assez bas niveau, il est possible d'isoler les éléments nécessaires sur la plate-forme pour l'implémentation de l'exécutif.

Cette implémentation montre que l'exécutif **L*f*P** peut être porté vers des plate-formes diverses offrant ainsi autant de possibilité de déploiement pour le code généré. De plus celui-ci garanti la portabilité du code généré.

L'interface utilisée par les composants **L*f*P** peut être implémentée sur une grande variété de plates-formes. L'implémentation actuelle utilise les fonctionnalités du langage Java, mais d'autres environnements de déploiement tels que Poly-ORB sont envisageables. Ce dernier fournit la structure nécessaire pour faire interagir des composants définis en **L*f*P** avec plusieurs environnements standards. Cette perspective est particulièrement intéressante pour l'intégration dans des environnements standards d'applications produites à partir de **L*f*P**. En effet, en créant une personnalité applicative pour l'exécutif **L*f*P**, il est possible de bénéficier de l'ensemble des personnalités protocolaires déjà développées.

L'environnement cible de la génération de code est maintenant défini par l'interface de l'exécutif. L'étape suivante consiste à implémenter les règles de transformation permettant de produire du code exploitant cette interface pour implémenter les modèles **L*f*P**.

Chapitre 6

Règles de transformation pour la génération de code

6.1 Introduction

L'objectif de ce chapitre est la présentation des règles de traduction des spécifications **L*f*P** vers un langage de programmation . Cette présentation est effectuée en deux étapes :

- présentation des règles de transformation indépendantes du langage cible ;
- instanciation des règles de transformation pour un environnement cible basé sur java.

Les *patterns* de code généré sont regroupés selon les mêmes règles que celles utilisées au chapitre 3. Ils sont donc présentés en commençant par l'implémentation du système de typage du langage, puis les règles d'implémentation du diagramme d'architecture, pour terminer sur les instructions du langage.

Les paterns présentés dans ce chapitre sont définis en fonction de l'exécutif défini au chapitre 5. Le code produit utilise l'interface définie par cet exécutif.

Les techniques de génération de code que nous avons utilisé sont inspirées des techniques de transformation de modèle actuellement en cours de développement à l'OMG. Elles présentent l'avantage d'établir une frontière concrète entre les aspects syntaxiques et structurels des formalismes. La structure d'un formalisme est alors représentée par son méta-modèle. Cette approche est particulièrement intéressante dans le cadre de la méthodologie **L*f*P** qui est principalement basée sur l'exploitation du modèle de l'application à réaliser.

La section 6.2 présente succinctement les techniques actuellement en cours de développement dans le cadre de la transformation de modèle. La section 6.3 présente le lien entre la transformation de modèle et la génération de code ; elle fait également le lien avec les techniques effectivement utilisées pour implémenter le générateur de code.

Les sections suivantes présentent les règles de transformation utilisées pour la génération de code à partir du langage **L*f*P**. Cette approche est faite en deux temps : les sections 6.4, 6.5 et 6.6 présentent respectivement les règles de transformation pour le système de typage du langage, les éléments du diagramme d'architecture, et les éléments dynamiques du langages. Ces règles visent avant tout à être applicable pour le plus grand nombre possible de langage cible et ne prennent donc pas en compte les spécificités d'un langage cible particulier. Dans de nombreux cas, plusieurs propositions sont faites en fonction des familles les plus courantes de langages de programmation.

La section 6.7 propose une instanciation des règles de transformation de **L*f*P** pour le langage java en utilisant l'exécutif présenté au chapitre 5. Ces règles ont été implémentées par le prototype du générateur de code utilisé pour traiter l'étude de cas du chapitre 7.

6.2 La transformation de modèles

Dans le cadre de la méthodologie **L*f*P**, la transformation de modèle intervient lors de deux étapes fondamentales :

- la construction du modèle **L*f*P** à partir du modèle UML ;
- la génération du code de l'application à partir du modèle **L*f*P**.

Cette section présente un panorama des techniques de transformations de modèles applicables dans ce contexte. Une attention particulière est portée à leur application dans le cadre de la génération de code.

6.2.1 Catégories de transformations de modèles

On distingue deux catégories de transformations de modèles [18] :

- l'approche *modèle vers modèle* ;
- l'approche *modèle vers texte*.

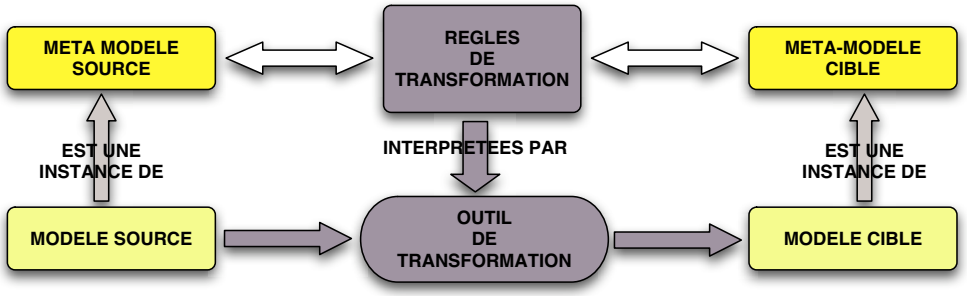


FIG. 6.1 – Approche de transformation modèle vers modèle

La figure 6.1 présente les méthodes de transformation de type *modèle vers modèle*. Chaque modèle est exprimé sous la forme d'une instance de son méta-modèle à l'aide d'une structure de donnée appropriée. Les règles de transformation sont exprimées en fonction des entités et structures définies par ces derniers. Elles sont interprétées pour chaque modèle source par l'outil de transformation de modèle. Cette approche est caractérisée par le fait que le modèle source et le modèle produit sont exprimés de manière structurée en fonction de leur méta-modèle respectif. Le passage d'un modèle de type PIM à un PSM est l'exemple le plus classique de ce type de transformation.

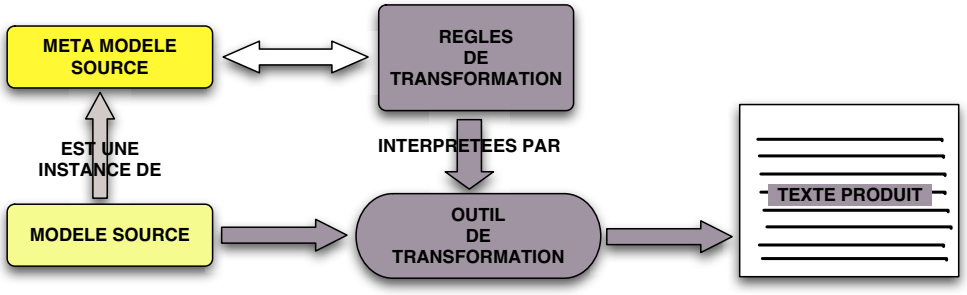


FIG. 6.2 – Approche de transformation modèle vers texte

La figure 6.2 illustre l'utilisation de transformations de type modèle vers texte. Le modèle source est décrit en fonction de son méta-modèle, mais le modèle résultat est produit de manière

non structurée sous forme de texte (code source, fichier XML, etc...). Les règles de transformation sont donc écrites en fonction du méta-modèle du modèle source, et de la syntaxe des fichiers produits. Le méta-modèle cible n'intervient plus directement dans le processus de transformation. Les modèles produits sont structurés en fonction de la syntaxe concrête du formalisme cible. Ce type de transformation est donc limitée à des sauts sémantiques relativement faibles entre le modèle source et le modèle cible pour que les transformations restent maintenables.

On distingue deux familles d'approches pour implémenter des transformations de modèle :
- la manipulation directe du modèle en utilisant une API.
- les approches utilisant un langage dédié.

La manipulation directe signifie que les règles de transformations sont directement écrites dans un langage de programmation *standard* (C++, Java, etc...), en utilisant une API dédiée permettant d'accéder au contenu du modèle telle que JMI [41]. L'autre solution permet l'utilisation d'un langage dédié à l'écriture des règles de transformation. La définition d'un tel langage est actuellement l'objet du RFP[1] de l'OMG [27].

6.2.2 Langages de transformation de modèles

Nous distingueront trois formes de langages pour la transformation de modèles :
- les approches *déclaratives* basées la définition de relations entre les entités des méta-modèles ;
- les approches déclaratives basées sur la manipulation des entités des méta-modèles ;
- les approches hybrides qui mêlent les approches déclaratives et impératives.

Les approches déclaratives

L'approche déclarative est principalement défendue par des travaux comme [3]. Elle consiste à définir les transformations par une série de relations entre les entités du méta-modèle source et celles du méta-modèle cible. Ces relations sont exprimées sous forme de prédicats. L'exécution de ce type de spécification s'apparente à de la programmation logique de type *prolog*.

Les approches impératives

Ces approches sont basées sur l'expression des actions nécessaires à l'implémentation des transformations. Ce type d'approche est défendue par le projet *triskell* de l'IRISA. Le langage MTL (*Model Transformation Language*) lié à l'environnement UMLAUT NG [29] reprend une sémantique et une syntaxe inspirées des langages de programmation classiques. Les opérations à effectuer sur les modèles sont définies explicitement, ainsi que l'ordre d'application des règles.

Les approches hybrides

Ces approches sont caractérisées par l'utilisation des deux approches précédemment évoquées. Elle est notamment défendue par ATL (pour *ATLAS Transformation Language*) [42] et TRL [2].

Dans le cas d'ATL, le style de programmation "recommandé" est déclaratif, mais le langage combine une partie déclarative et une partie impérative. TRL distingue explicitement l'utilisation de ces deux techniques. Le style déclaratif est utilisé pour spécifier le résultat de la transformation, et le style impératif est utilisé pour spécifier l'implémentation.

L'approche hybride de TRL a rassemblé plusieurs candidats à la réponse pour le RFP de l'OMG sur les langages de transformation de modèle, pour fournir une réponse commune. Le

[1] *Request For Proposal*

langage fait explicitement la distinction entre la partie déclarative pour la spécification de la transformation, et la partie impérative pour son implémentation.

6.3 Transformation de modèle et génération de code pour L*f*P

Cette section présente le lien entre les techniques de génération de transformation de modèle présentées à la section 6.2.1 et les techniques utilisées pour la génération de code. Dans un premier temps, nous présentons les étapes nécessaires à la génération de code et leurs similitudes avec les processus de transformation de modèle. Dans un deuxième temps, nous présentons succinctement les structures de données utilisées pour l'implémentation du générateur de code.

6.3.1 Processus de génération de code

Le fonctionnement du générateur de code **L*f*P** est illustré sur la figure 6.3. Celui-ci fonctionne en deux étapes distinctes :

- une étape sémantique ;
- une étape syntaxique.

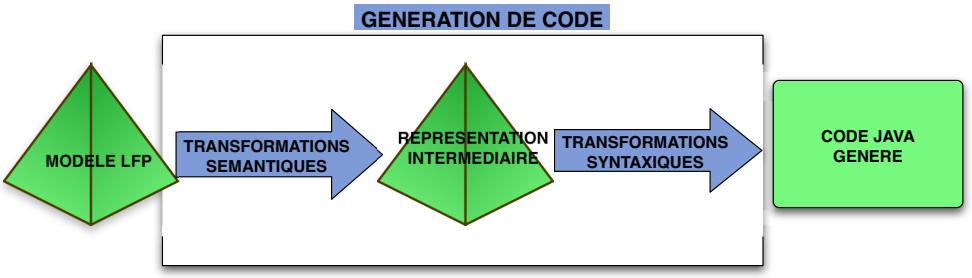


FIG. 6.3 – Le processus de génération de code

La première passe est une transformation de type *modèle vers modèle*. Elle permet d'effectuer le saut entre la sémantique du langage **L*f*P**, et la sémantique de la plate-forme cible. Cette passe permet de construire une représentation du modèle utilisant les caractéristiques de la plate-forme cible. Celle-ci est définie en fonction des caractéristiques du exécutif **L*f*P**. Elle définit une abstraction de l'environnement d'exécution cible définie par :

- un modèle de typage ;
- un ensemble d'instruction ;
- un modèle de répartition.

L'étape de transformation sémantique doit effectuer la correspondance entre les structures de données et les instructions fournies par le langage **L*f*P** d'une part, et leur implémentation en fonction des caractéristiques de la plate-forme cible d'autre part. Le modèle est donc produit sous une forme structurée et est exprimé en fonction d'un jeu d'instruction et de types de données implémentables dans l'environnement cible.

Pour réaliser cette tâche, le générateur de code doit également effectuer toutes les vérifications possibles permettant de s'assurer de la validité de la spécification **L*f*P** prise en entrée. Le générateur de code se comporte comme un compilateur qui doit effectuer des vérifications syntaxiques et sémantiques sur la spécification. Les principales vérifications sémantiques à effectuer sont classiques dans le domaine de la compilation telles que les vérifications de typage. Par ailleurs, cette opération est nécessaire pour que les règles de transformation puissent utiliser les instructions et opérateurs appropriés pour implémenter la spécification **L*f*P** sur la plate-forme cible.

Les règles de transformation doivent donc analyser la structure de la spécification source pour identifier les constructions à traduire dans le formalisme cible. Pour cela, les règles de transformation se basent sur une analyse de la représentation structurée du modèle.

La deuxième étape de la génération de code est une transformation de type modèle vers texte. Celle-ci interprète le modèle produit à la première étape en fonction des fonctionnalités du langage cible (syntaxe et constructions disponibles). Elle permet d'associer une syntaxe concrète aux opérateurs définis par la représentation abstraite.

Cette approche en deux temps permet de distinguer les aspects liés au fonctionnement de la plate-forme cible de ceux liés au langage d'implémentation choisi. Elle permet d'envisager la portabilité du générateur de code vers de nouveaux langages cibles. Cette opération nécessitera l'implémentation d'un exécutif fournissant une interface équivalente pour le nouveau langage, et la ré-écriture des règles de transformation syntaxiques. En revanche, les règles de transformation sémantique ne devraient pas être impactées. C'est un point très important car il s'agit de la partie du générateur de code la plus difficile à mettre au point.

6.3.2 Structuration des représentations des modèles

La représentation de la spécification L*f*P

La structure de donnée permettant de représenter le modèle **L*f*P** reprend le même type de structuration que le langage lui même. Elle est constituée d'un ensemble de graphes orientés reliés par des relations de hiérarchie. Elle définit donc un graphe dont les noeuds sont les diagrammes de la spécification **L*f*P** (représentés par des graphes reprenant la structuration des automates) et dont les arcs sont les relations de hiérarchies à respecter entre les diagrammes.

- chaque noeud du graphe correspond à un élément graphique de la spécification **L*f*P** ;
- chaque arc du graphe correspond à un arc de la spécification **L*f*P**.

Chaque attribut des éléments graphiques est représenté sous la forme d'un arbre de syntaxe abstraite attaché au noeud correspondant dans le graphe.

Pour la représentation du diagramme d'architecture, on distingue donc les types de noeuds suivants :

- classe ;
- média ;
- binder.

Les déclarations globales du diagramme de classe sont stockées dans un attribut dédié de la structure graphique, sous la forme d'un arbre de syntaxe abstraite.

Chaque noeud représentant un composant (classe ou média) est relié par un lien de hiérarchie au graphe représentant l'automate de comportement correspondant. Lorsqu'une classe ou un média apparaît en plusieurs exemplaires sur le diagramme d'architecture, tous les noeuds correspondant référencent le même sous-graphe représentant le diagramme de comportement du composant.

Pour les diagrammes de comportement, on définit les types de noeuds suivants :

- état ;
- transition ;
- transition hiérarchiques.

Les états correspondent aux états de l'automate de comportement. Les transitions peuvent être hiérarchiques ou non. Les transitions non hiérarchiques contiennent un ensemble d'instructions représentées sous la forme d'un arbre de syntaxe abstraite.

Les transitions hiérarchiques peuvent représenter des triggers, des méthodes ou des sous-diagrammes. La distinction est faite en fonction des déclarations associées au diagramme de comportement : si une transition hiérarchique porte le nom d'une méthode (resp. d'un trigger), elle

contient le corps de la méthode déclarée (resp. du trigger déclaré). Si une méthode ou un trigger apparaît plusieurs fois dans le diagramme de comportement, toutes les instances référencent le même sous-graphe représentant son contenu.

Chaque diagramme de comportement dispose d'un attribut définissant ses déclarations locales. Elles sont stockées dans un arbre de syntaxe abstraite qui lui est associé.

Les arcs du graphe correspondent aux arcs de l'automate **L*f*P** reliant les états et les transitions.

Structuration de la représentation intermédiaire

La représentation intermédiaire est une structure de donnée correspondant à un arbre de syntaxe abstraite pour la plate-forme définie par l'exécutif Chaque classe à produire est représentée sous la forme d'un arbre dont les noeuds représentent des appels aux fonctions de l'exécutif ou une instruction de base implémentable dans le langage cible.

La structure de l'arbre de syntaxe abstraite est prévue que toutes les informations nécessaires à la génération du code correspondant à un noeud de l'arbre soient disponibles localement dans ce noeud. Pour utiliser cette structure, l'utilisateur doit définir une fonction d'écriture pour chaque type de noeud défini dans l'arbre. Le code peut donc être généré en un seul parcours sur la structure.

La suite de ce chapitre est consacrée aux règles de transformation du langage **L*f*P** vers un langage de programmation. La présentation des transformations est réalisée par la définition des règles de transformation à appliquer à chacun des éléments du langage vers un langage de programmation cible.

6.4 Règles de transformation pour les types L*f*P

L'objectif de cette section est de définir les règles de transformation requises pour implémenter le système de typage **L*f*P** dans un langage de programmation. Pour chaque type défini en **L*f*P**, elle propose un pattern de structure de donnée associée. Cette structure de donnée doit être implémentable dans une classe de langage cible aussi large que possible. Pour chaque type, elle précise également l'implémentation des opérateurs qui lui sont associés.

6.4.1 Le type `boolean`

Ce type est fourni en standard par presque tous les langages de programmation ; il peut alors être utilisé sans modification particulière. Si le type booléen n'est pas défini de manière native dans le langage cible, il peut être construit à partir du type représentant les entiers.

6.4.2 Les types opaques

Les types opaques permettent de représenter les composants externes de la spécification. Au moment de la génération de code, ils sont utilisés pour relier le code de contrôle de l'application avec le code de traitement de données.

Les opérations suivantes doivent être implémentées sur les types opaques :

- l'instanciation ;
- l'appel d'une méthode (appel externe) ;
- l'affectation ;
- la destruction d'instance ;
- sérialisation

Instanciation d'un type opaque

Cette instruction est implémentée par une instanciation dynamique du type implémentant le composant externe. Il s'agit de l'instruction standard du langage cible, aucun exécutif n'est requis dans ce cas. Dans certains langages objets, cette opération provoque l'appel d'un constructeur par défaut, ou l'initialisation des variables définies dans l'implémentation du composant.

Implémentation des appels externes

Les appels aux méthodes du composant sont réalisés de manière très simple : en appelant la méthode de même nom sur l'instance désignée par la variable de type opaque. La sémantique des appels externes est proche de celle des langages objets dans la mesure où un appel externe est toujours lié à une instance d'un type opaque. Dans le cas où le langage cible n'est pas orienté objet, le premier paramètre de chaque fonction de l'interface d'un type opaque doit être la référence de l'instance sur laquelle l'opération est effectuée.

Le langage **L*f*P** étant insensible à la casse, on propose d'utiliser la convention suivante pour l'implémentation des composants externes : lorsque le langage cible est sensible à la casse, tous les noms des fonctions pouvant faire l'objet d'un appel externe doivent être en majuscules.

Opération d'affectation

L'affectation entre types opaques est définie dans la sémantique **L*f*P** pour être implémentée par une simple affectation standard du langage dans le code généré. Là encore, aucune fonctionnalité n'est requise de la part de l'exécutif.

Destruction d'une instance d'un type opaque

La destruction d'une instance d'un type opaque peut être également réalisée en utilisant l'instruction de libération mémoire du langage cible. Aucune vérification ne peut être faite sur la gestion de la mémoire des composants externes. En particulier dans le cas de types opaques contenant des références vers d'autres composants n'est pas traité au niveau de **L*f*P**.

Si le composant est écrit dans un langage objet fournissant des *destructeurs*, il est recommandé d'utiliser cette fonctionnalité pour libérer l'espace utilisé par le composant. Si cette solution n'est pas applicable, il faut prévoir une méthode de destruction dans l'interface du type opaque et de l'appeler explicitement dans la spécification **L*f*P** avant de détruire les instances de ce type.

Sérialisation d'un type opaque

La fonction de sérialisation d'un type opaque doit être implémentée par tous les composants opaques susceptibles d'être migré via le réseau, c'est à dire pour tout type opaque inclus dans un message, que ce soit dans la partie donnée ou dans la partie discriminant.

Cette fonction dépend grandement de la structure du type, et reste entièrement à la charge du programmeur du composant correspondant. De nombreux outils sont disponibles en fonction de la plate-forme choisie parmi lesquels on peut citer l'interface de sérialisation du langage Java ou encore le protocole SOAP.

6.4.3 Les types composants

Les types composants correspondent aux éléments structurants du modèle **L*f*P**. Nous les utilisons donc également pour structurer le code généré.

Dans le cas d'un langage objet, chaque composant **L*f*P** est implémenté par une classe. Chacun des attributs du type composant est implémenté sous la forme d'un attribut de la classe. Les types définis dans le composant sont définis sous forme de classes internes et sont donc invisibles de l'extérieur du composant comme spécifié dans la sémantique du langage **L*f*P**.

Si l'exécutif utilise des accès réflexifs directement sur la structure du composant, il est nécessaire que les attributs ciblés par ces appels soient déclarés *publics*.

Dans le cas d'un langage procédural, les composants seront implémentés sous formes de modules exportant une structure. Chaque attribut du composant est implémenté par un champs de la structure. Les types de données seront implémentés par des types internes au module. Le langage **L*f*P** interdit les accès directs aux attributs d'un composants. Seuls les ports des composants sont accessibles en lecture. Cela signifie qu'il est possible pour un composant **L*f*P** de connaître le binder référencé par les ports d'un autre composant.

La structure d'implémentation du type n'a pas besoin d'être exportée s'ils sont réalisées par des références insérées dans les structures internes de l'exécutif lors de l'instanciation (cf. 6.6.7).

Code généré pour une méthode d'un composant

Les méthodes des composants **L*f*P** peuvent être appelées en plusieurs endroits de l'automate du composant. Il est donc relativement naturel de les implémenter sous forme de fonction (méthodes dans le cas des langages orientés objets). Toutes les méthodes doivent être implémentées sous forme de fonction.

Le code correspondant au corps de la méthode est généré selon les mêmes règles que l'automate principal en fonction des opérations qu'il contient. La seule instruction spécifique au corps des méthodes est l'instruction return qui envoie le message de retour à l'appelant. Cette instruction doit construire le message de retour. C'est ce message qui est retourné par la fonction qui implémente la méthode. Il sera envoyé à l'appelant par le code généré pour l'instruction d'attente d'activation de méthode (cf. 6.6.6).

Code généré pour les triggers

Les triggers sont les méthodes privées des composants **L*f*P**. Il est possible de les générer sous forme de macros, ou de fonctions. Nous préférons utiliser la deuxième solution qui semble plus générale. Cela rejette la responsabilité de l'éventuel *inlining* du corps du trigger au compilateur du langage cible. Le corps des triggers est produit à partir des mêmes règles de génération de code que le reste de l'automate de comportement du composant.

Contrairement aux méthodes, les triggers sont activés localement. Un appel de trigger peut donc être implémenté par un appel de fonction. Les triggers **L*f*P** ne peuvent modifier que des attributs du composant. Dans le cas d'un langage objet, la fonction qui implémente un trigger modifiera les attributs de la classe par effet de bord. Dans le cas d'un langage procédural, la fonction prendra un seul paramètre : la structure correspondant à l'instance du composant qui sera modifiée par effet de bord.

Sérialisation d'un composant

Les composants ne sont pas mobiles, ils ne circulent donc jamais sur le réseau et n'ont pas besoin d'une fonction de sérialisation. Seules les références vers ces types doivent pouvoir être sérialisées. La sérialisation de ces références peut être assurée par l'exécutif.

La migration de composant n'est donc pas encore intégrée au langage **L*f*P**. L'intégration de cette fonctionnalité n'est pas prévue dans un futur proche bien qu'il s'agisse d'une évolution possible du langage.

6.4.4 Les types énumérés et leurs types restreints

Les types énumérés sont des constructions communes à beaucoup de langages de programmation, néanmoins **L*f*P** fournit un ensemble d'opérateurs sur les types énumérés qui ne sont pas nécessairement pris en compte dans tous les langages, surtout dans le cas des types énumérés circulaires. De plus, **L*f*P** permet de définir des restrictions de types (sous-types) qui ne sont pris en compte que par un nombre restreint de langages.

Nous proposons une implémentation pour des types énumérés circulaires ou non dans un langage cible ne définissant pas les types énumérés, il sera possible de la simplifier en fonction des propriétés du langage cible choisi. Les templates utilisés peuvent être intégrés à l'exécutif et fonctionnent pour les types énumérés, et les types définis par restriction sur des types énumérés.

Un type énuméré peut être représenté par un intervalle de valeurs entières. Celui-ci est défini par $[first..last]$ où first correspond à la première valeur valide du type, et last à la dernière. Dans le cas d'un type énuméré, first est égal à 1 et last est égal au nombre de valeur déclaré. Pour les types restreints, les valeurs de first et last sont les indices des bornes du nouveau type.

Exemple 13 *Par exemple, soit les définitions suivantes :*

```
type enum is range (a, b, c, d) ;
type restricted is range (b..c) of enum ;
```

L'intervalle de valeur valide pour le types `enum` *est* $[1..4]$, *et* $[2..3]$ *pour le type* `restricted`.

Cela nous amène à implémenter les opérateurs définis pour les types énumérés de la manière suivante :

`first` correspond simplement à la borne inférieure de l'intervalle de définition du type considéré ;

`last` correspond à la borne supérieure de l'intervalle de définition du type considéré ;

`pred` peut être implémenté de la manière suivante :

```
function PRED(x, first : int)
    inty := x − 1 ;
    Si y < first alors
        raise_exception Value_Out_Of_Bounds
    sinon
        return y ;
    fin Si
fin function
```

`succ` peut être implémenté de la manière suivante :

```
function SUCC(x, last : int)
    inty := x + 1 ;
    Si y > last alors
        raise_exception Value_Out_Of_Bounds
    sinon
        return y ;
    fin Si
fin function
```

Dans le cas de types circulaires, les templates pour `succ` et `pred` sont légèrement différents :

– `pred` peut être implémenté de la manière suivante :

```
function PRED(x, first, last : int)
    Si x = first alors
        return last
    sinon
        return x − 1 ;
    fin Si
fin function
```

– `succ` peut être implémenté de la manière suivante :

```
function SUCC(x, first, last : int)
    Si x = last alors
        return first ;
    sinon
        return x + 1 ;
    fin Si
fin function
```

Les valeurs passées à ces fonctions sont systématiquement dans l'intervalle de définition du type considéré. En effet, si le résultat d'une opération sur un type énuméré est détectée par l'opération elle même et provoque une erreur d'exécution.

Les opérateurs de relation définis sur les types énumérés ($>$, $<$, $<=$, $>=$ et $=$) sont implémentés en utilisant directement les opérateurs correspondants que le langage cible définit sur les entiers. Leur comportement est identique à celui des opérateurs définis par le langage **L*f*P** pour les types énumérés.

La seule limitation de l'approche que nous venons de présenter provient du fait que le type énuméré défini en **L*f*P** n'apparaît pas explicitement dans le code généré. L'absence de typage explicite rend rapidement ce dernier illisible et difficile à vérifier.

6.4.5 Le type integer et ses types restreints

Le type **L*f*P** integer peut être implémenté directement par un type entier du langage cible dont les bornes coïncident avec les bornes retenues par la vérification formelle pour le type `integer` du langage **L*f*P**.

L*f*P permet de définir des plages de valeurs restreintes pour ce type sous la forme de sous-types. Les bornes de ces types sont quelconques, mais la sémantique des opérateurs est héritée du type `integer`.

L'implémentation des opérateurs `first, last, pred, succ` est la même que celle vue précédemment pour les types définis par restriction pour des types énumérés. En revanche, les opérateurs arithmétiques nécessitent une implémentation particulière.

Types non circulaires définis par restriction

La fonction suivante permet d'implémenter le calcul de la somme de deux éléments d'un type défini par restriction sur les entiers :

```
function ADD(x, y, first, last : int)
    s := x + y ;
    Si  s > last or s < first alors
        raise_exception Value_Out_Of_Bounds ;
    sinon
        return s ;
    fin Si
fin function
```

Ce template peut être repris en modifiant l'opérateur arithmétique pour calculer la différence, le produit et le quotient de deux valeurs.

6.4.6 Les types structurés

Cette section présente des templates de génération pour les types structurés du langage **L*f*P**.

Les types tableau

Les tableaux sont des structures très classiques des langages de programmation. Le template de code correspondant à cette structure utilise repose sur les fonctionnalités standard d'un langage de programmation pour la manipulation des tableaux, sans avoir recours à une bibliothèque de fonction. Le générateur produit donc un type tableau de même structure que le type tableau **L*f*P** dans le langage cible, et implémente les opérateurs associés.

Le langage **L*f*P** fournit trois opérateurs sur les tableaux : l'affectation, l'accès à un élément et l'initialisation. L'accès à un élément (en écriture ou en lecture) est toujours proposée par le langage cible, cette opération peut donc être traduite directement. Néanmoins, de nombreux langages de programmation fixent l'indice de départ des tableaux à zéro ce qui n'est pas le cas en **L*f*P**. Il est donc nécessaire de prévoir un décalage pour l'indice du tableau.

Le cas de l'affectation est plus difficile, en effet, dans de nombreux langages (C, C++, java), les types tableaux sont en fait des pointeurs vers une zone de mémoire. L'opérateur d'affectation par défaut du langage ne recopie donc pas le contenu du tableau, mais sa référence. Dans ces langages, il est donc nécessaire de générer une fonction de recopie du tableau.

En **L*f*P**, les dimensions des types tableaux sont déterminées statiquement par la définition du type du tableau, il est donc toujours possible de définir un type tableau correspondant dans le langage cible sans passer par un mécanisme d'instanciation dynamique. Les dimensions du tableau peuvent être spécifiées à chaque déclaration de variable, ce qui évite le recours à l'instanciation dynamique et donc aux tableaux de pointeurs intermédiaires pour les tableaux à plusieurs dimensions.

Dans le cas d'un langage orienté objet, on pourra définir un type objet pour chaque type tableau, ce qui permet de relier la fonction d'initialisation au type, en utilisant par exemple un constructeur.

Les templates suivants correspondent au code généré pour un langage gérant les tableaux de la même manière que le langage `C`. Nous considérons donc des tableaux dont les dimensions sont fixées lors de leur déclaration. Ces templates fonctionnent pour des tableaux avec un nombre quelconque de dimensions.

– fonction d'initialisation :

```
procedure INITIALISE(tableau : in out type_tableau ; value : type_element)
    for i1 = 0 à dimension1.last do
        for i2 = 0 à dimension2.last do
            ...
                    ▷ Autant de boucles imbriquées que de dimensions au tableau
            tableau[i1][i2]... = value;
            ...
        fin for
    fin for
fin procedure
```

– fonction de duplication de tableau :

```
function DUPLIQUE(tableau : type_tableau )
    copie : type_tableau
    for i1 = 0 à dimension1.last do
        for i2 = 0 à dimension2.last do
            ...
                    ▷ Autant de boucles imbriquées que de dimensions au tableau
            copie[i1][i2]... = tableau[i1][i2]...;
            ...
        fin for
    fin for
    return copie
fin function
```

Les types structures (`record`)

Les types structures existent dans tous les langages sous différentes formes (`struct` en C, articles en Ada, etc...). Dans les langages objets, il est toujours possible de définir une classe pour implémenter une structure.

La traduction d'un type structure est relativement simple : elle consiste à reproduire une structure dans le langage cible (ou une classe dans un langage objet) dont les champs correspondent aux champs du type `record` **L*f*P**.

Les ensembles et les multi-ensembles

Les ensembles et multi-ensembles **L*f*P** sont présentés à la section 3.4.3. Ces structures sont typiquement absentes des langages de programmation classiques. Il est donc nécessaire de proposer une implémentation pour ces types. L'implémentation changera beaucoup en fonction du langage cible. L'objectif est de définir une structure de donnée de longueur variable, ainsi qu'une interface implémentant la sémantique des opérateurs **L*f*P**.

Si le langage cible comporte la notion de template (ou généricité) comme l'Ada, le C++, ou la future version de Java, il sera possible de se reposer sur un composant paramétrable en fonction du type d'élément contenu dans l'ensemble. Dans ce cas, les types ensembles peuvent être implémentés sous forme de composants, avec comme paramètre générique le type des éléments de l'ensemble.

Dans le cas d'un langage ne disposant pas de cette fonctionnalité, on dispose de deux solutions : soit on utilise un système de pointeurs non typés (le `void*` du C), soit on génère le composant en fonction du nom du type des éléments.

L'implémentation de ces types peut être réalisée simplement à l'aide par exemple de listes chaînées ou de composants *vecteurs* tels qu'on peut en trouver dans les bibliothèques de composants du langage Java.

6.5 Règles de génération des éléments statiques du langage

Le diagramme d'architecture a deux aspects du point de vue de la génération de code. La partie déclarative du diagramme permet de rassembler les déclarations de types et de constantes qui sont partagées par tous les composants de l'application. La partie architecturale définit la structure de l'application.

6.5.1 Partie déclarative du diagramme et déploiement de l'application

La manière de traiter la partie déclarative a été principalement vue à la section 6.4. Les types déclarés dans le diagramme d'architecture doivent donc être définis dans des unités globales qui peuvent être utilisées par les composants. La partie architecturale déclare les types composants qui sont utilisés dans le modèle, ainsi que leurs ports.

En plus des types et constantes, la partie déclarative définit les instances statiques des composants du modèle ; elles déterminent quels composants doivent être créés par l'exécutif lors de l'initialisation de l'application. Les opérations à effectuer sont les suivantes :

- déterminer les instances locales dans la liste des instances statiques ;
- instancier les composants statiques locaux ;
- initialiser l'ensemble des références vers les composants statiques.

Les informations de déploiement doivent donc spécifier sur quel site est instancié chaque composant statique. Les informations de déploiement sont donc reliées au diagramme d'architecture, mais sont traitées de manière externe : elles sont fournies dans un fichier de configuration indépendant de la spécification **L*f*P**, ce qui permet de définir plusieurs configurations de déploiement pour une spécification donnée.

Les informations de déploiement sont utilisées lors du démarrage de l'application pour initialiser les références vers les composants statiques définis sur le diagramme d'architecture. Celles-ci doivent contenir l'identifiant du site **L*f*P** dans lequel ce composant sera intégré et pour chaque site **L*f*P** l'hôte réseau sur lequel il sera déployé.

Le générateur de code peut vérifier la cohérence entre le fichier de déploiement fourni et les composants déclarés dans la spécification **L*f*P**. Les informations de déploiement sont ensuite traduites en informations de démarrage qui seront utilisées lors de l'initialisation de l'application.

On dispose alors de deux approches pour les informations de démarrage de l'application :

- les inclure statiquement dans le code source généré à partir de la spécification **L*f*P** ;
- les maintenir dans un fichier de configuration lu par l'application lors de son démarrage.

La première solution est très efficace en temps de démarrage de l'application, la seconde nécessite la lecture et l'interprétation du fichier de configuration. De plus, des modules spécifiques de lecture de ce fichier doivent être rajoutés dans l'exécutif. Cette approche n'est finalement pas très pénalisante dans la mesure où elle n'affecte que le démarrage de l'application et non son exécution normale, et elle permet de changer le placement des sites **L*f*P** en modifiant simplement le fichier de configuration.

6.5.2 Partie architecturale du diagramme

La partie architecturale sera utilisée principalement dans la première phase de la génération de code : l'analyse du modèle. Il n'y a pas de code qui soit directement généré à partir de ces

informations. En revanche, elle permet de déterminer la liste des composants de l'application, et de vérifier la validité des interactions entre composants rencontrées lors de l'analyse du modèle. Ces vérifications sont effectuées statiquement par analyse de la structure du diagramme d'architecture et des diagrammes de comportement des composants.

Lorsque le générateur de code identifie une opération d'appel de méthode dans une classe C_1, il peut vérifier qu'il existe au moins une classe C_2 reliée à C_1 par un média qui déclare une méthode activable par ce message.

Une méthode est activable par un appel si :

- elle porte le même nom que la méthode appelée ;
- les types de ses paramètres formels sont compatibles avec les valeurs contenues dans les paramètres effectifs de l'appel.

Le même type de mécanisme est mis en place pour les envois de messages de données. Le générateur cherche une classe :

- contenant une instruction de lecture acceptant un message de même type que le message émis ;
- reliée à la classe émettrice par un média.

6.6 Présentation des règles de génération des éléments dynamiques

Cette section présentera les règles de génération de code portant sur la partie comportement des composants **L*f*P**. Elle présentera donc les règles de transformation des diagrammes de comportement des spécifications **L*f*P** vers du code source. A chaque fois que nécessaire, elle précisera les services que doit fournir l'exécutif pour que le code résultant puisse être exécuté. Les règles sont listées en fonction du type d'instruction représenté.

6.6.1 Code généré pour une structure d'alternative

Le langage **L*f*P** définit deux types d'alternatives présentées à la section 3.6.2 :

- soit la syntaxe de type *if… then… else* utilisée sur les transitions ;
- soit la définition d'un état suivi de plusieurs transitions qui correspond à une alternative basée sur la structure graphique du langage.

La forme textuelle de l'alternative

Cette forme se retrouve directement dans tous les langages de programmation. De plus, l'alternative telle qu'elle est définie en **L*f*P** correspond à son expression la plus simple. Il est donc possible de traduire directement une opération *if… then… else* dans le langage cible, seule la syntaxe concrète change.

La forme graphique de l'alternative

Cette forme est plus une *macro* qu'une véritable alternative au sens des langages de programmation. Pour un état suivi de n ($n \neq 1$) transitions, le pseudo-code correspondant est :

Si alors $garde_1$
    goto label1 ;
fin Si
Si alors $garde_2$
    goto label2 ;
fin Si
...
Si alors $garde_n$
    goto label3 ;
fin Si

Dans le cas où il n'y a qu'une transition de sortie, la garde doit être vide, et il est alors possible de simplifier ce schéma en ne conservant qu'une instruction *goto* vers la transition de sortie.

6.6.2 Code généré pour les structures de boucles

Les boucles disponibles en **L*f*P** sont des sous-ensembles des boucles disponibles dans les langages de programmation classique. La transformation est principalement syntaxique et est donc fortement dépendante du langage cible.

Dans de nombreux cas, la difficulté de traduction provient de l'expression de sortie de boucle. Les conditions portant sur les entiers peuvent être reprises sans modification. En revanche, si l'expression de sortie de boucle porte sur un type énuméré ou un type circulaire défini dans la spécification **L*f*P**, l'expression doit être traduite en une expression équivalente dans le langage cible.

6.6.3 Code généré pour un envoi de message de données

Le code généré pour un envoi de message de données doit réaliser les opérations suivantes :

- instancier un nouveau message ;
- insérer l'ensemble des données du message ;
- insérer les éléments du discriminant ;
- envoyer le message vers le binder cible.

Le premier point correspond à l'instanciation du composant de l'exécutif destiné à contenir les messages. Les deux points suivants utilisent l'interface fournie par ce composant pour spécifier le contenu du message. Les données doivent être encodées pour être transportée sur le réseau. Enfin le dernier point est un appel à l'exécutif qui traite l'envoi du message dans le binder destinataire.

6.6.4 Code généré pour une réception de message de données

La réception d'un message de donnée est réalisée en deux étapes :

- lecture du message sur le binder ;
- dé-sérialisation du contenu du message.

La lecture est réalisée par l'appel correspondant à l'exécutif **L*f*P**. Cet appel est bloquant jusqu'à ce qu'un message soit disponible. La valeur de retour de cet appel est un message sous forme sérialisée.

Ensuite, chaque élément contenu dans le message doit être dé-sérialisé pour être inséré dans la variable cible. Le générateur de code détermine quelle fonction de dé-sérialisation appeler en fonction du type de la variable à laquelle la valeur est affectée.

6.6.5 Code généré pour un appel de méthode

Un appel de méthode correspond à l'envoi d'un message d'activation, et pour les méthodes synchrone, l'attente d'un message de retour.

L'envoi du message d'activation est réalisé de manière très similaire à l'envoi d'un message de donnée. Les différences proviennent du type du message, et du fait qu'il contient en plus un nom de méthode à activer.

Cette première partie du code est commune à tous les types d'appels de méthodes **L*f*P**. Le code généré diffère ensuite en fonction du type de méthode appelé.

Cas d'un appel de procédure asynchrone

Dans le cas de l'invocation d'une méthode asynchrone, il n'y a pas de message de retour, et le composant appelant peut continuer son exécution directement après l'envoi du message.

Cas d'un appel de fonction

Dans ce cas, la fonction retourne toujours une valeur unique qui doit être stockée dans une variable. Une fois le message d'activation envoyé sur le binder cible, le code généré appelle la fonction de lecture de message sur le même binder. Le premier message retourné par cet appel à l'exécutif doit être un message de retour contenant une seule valeur (la valeur de retour de la fonction).

Ce message doit être dé-sérialisé en utilisant la fonction associée au type retourné par la fonction avant d'être affectée à la variable prévue pour stocker la valeur de retour.

Cas d'un appel de procédure synchrone

Dans le cas d'un appel de procédure synchrone, le message de retour contient une valeur par paramètre en mode `out` ou `inout`. Chacune de ces valeurs doit être dé-sérialisée par la fonction associée au type du paramètre formel avant d'être affecté à la variable contenant le paramètre effectif.

6.6.6 Activation de méthode et lecture de message par les médias

Ces deux instructions sont spécifiques car elles peuvent toutes les deux refuser un message s'il ne correspond pas à une caractéristique donnée. Dans le cas du média, le discriminant du message doit respecter une garde. Dans le cas d'une activation de méthode, la méthode activée par le message doit être activable dans l'état courant de la classe.

Attente d'activation de méthode

Cette opération doit prendre en compte le fait que plusieurs méthodes peuvent être activables à un instant donné (instruction de type *select* définie à la section 3.6.6). Le code généré doit donc effectuer les opérations suivantes :

binder_list ← liste des binders d'activation des méthodes ;
msg ← lire_message (binder_list) ;
Boucle
  Si *msg* est un message d'activation **et** la que méthode activée par *msg* est dans la liste des méthodes activables **et** que *msg* a été lu sur le binder d'activation de la méthode activée **alors**
    Consommer le message et abandonner les autres opérations de lecture ;
    exécuter la méthode activée par *msg* avec les paramètres transmis par le message ;
    Sortir de la boucle (**break**) ;
  sinon
    refuser le message et rester en attente d'un nouveau message sur ce binder ;
  fin Si
fin Boucle

La liste des binders doit être ordonnée en fonction de la priorité qui est associée à chaque opération incluse dans l'opération *select*. Cela permet à l'appel à l'exécutif de les traiter dans l'ordre approprié.

Une fois la méthode exécutée, il faut envoyer son éventuelle valeur de retour :

▷ Traitement du message de retour
Si La méthode appelée est une procédure synchrone sans paramètre en mode `out` ou `inout` **alors**
  envoyer un message de retour vide sur le binder d'activation ;
sinon Si la méthode appelée est une procédure avec des paramètres en mode `out` ou `inout` **alors**
  envoyer le message de retour contenant les paramètres mis à jour sur le binder d'activation de la méthode ;
sinon Si la méthode appelée est une fonction **alors**
  envoyer le message contenant la valeur de retour sur le binder d'activation de la méthode ;
fin Si

Comme il est impossible de prévoir quelle sera la méthode effectivement appelée, le comportement à adopter est défini dynamiquement lors de l'exécution. Le code généré doit donc prendre en compte explicitement les trois cas possibles (procédure asynchrone, procédure synchrone, fonction).

Lectures de message(s) dans les médias

La lecture des messages dans un média est conditionnée à l'évaluation d'une garde : si la condition de garde est fausse, le message est refusé : il reste dans le binder jusqu'à ce qu'il soit consommé par une autre opération de lecture. Il s'agit d'un appel à la fonction de *refus de message* évoqué dans la description de la bibliothèque de l'exécutif à la section 5.2.2.

Le code généré pour une instruction de select dans un média doit donc réaliser les opérations suivantes :

binder_list ← liste des binders d'activation des méthodes ;
method_list ← liste des méthodes activables, ordonnée en fonction de la priorité (opération réalisée par une fonction de l'exécutif) ;
msg ← lire_message (binder_list) ;
Boucle
  Si la garde associée à l'opération de lecture s'évalue à vrai **alors**
    Consommer le message et abandonner les autres opérations de lecture ;
    Sortir de la boucle (**break**) ;
  sinon
    refuser le message et rester en attente d'un nouveau message sur ce binder ;
  fin Si
fin Boucle

Interface de l'exécutif associé

Les opérations de lecture multiplexées utilisent les fonctions suivantes de l'interface de l'exécutif.

- une opération de lecture d'un message ; sur une liste de binders ; en tenant compte de la priorité de chaque opération ;
- une opération permettant d'accepter un message et de finaliser l'ensemble des opérations de lecture en cours (*commit*) ;
- une opération permettant de refuser un message (*weak abort*) ;
- une opération permettant de savoir sur quel binder un message a été lu ;
- une opération permettant de déterminer la nature d'un message.

Les trois premiers points portent sur l'interface de l'exécutif. Ils définissent le cahier des charges des fonctions de traitement des messages. Les deux derniers service sont associés au type `message` qui fait partie de la bibliothèque de composants de l'exécutif et doit fournir une interface de manipulation dédiée à ces opérations.

L'opération de lecture d'un message sur une liste de binder est utilisée pour récupérer le prochain message reçu par le composant. Cette opération doit permettre de prendre en compte les priorités des opérations indiquées dans la spécification **L*f*P**.

La fonction de *commit* est appelée lorsqu'un message est accepté. Elle permet de signifier la fin de la transaction de lecture à tous les binders impliqués dans la transaction :

- signifier au binder qui contenait le message que celui-ci a été accepté (appel de la fonction d'acceptation de message) ;
- signifier la fin de la transaction de lecture aux autres binders (appel de la fonction d'abandon de transaction).

La fonction permettant de refuser un message est appelée lorsqu'un binder délivre un message qui n'est pas utilisable dans l'état courant du composant qui effectue la lecture. Elle appelle la fonction de refus de message sur le binder qui a émis le message, et ne met donc pas un terme à la transaction de lecture : le binder peut être à nouveau interrogé lorsqu'un nouveau message sera disponible

Les deux fonctions précédentes ne sont pas réalisées directement par le code généré car les instances des binders ne sont pas forcément locales. Il est alors nécessaire de traiter les interactions avec le réseau ce qui est un des rôles principaux de l'exécutif.

La fonction permettant de savoir sur quel binder le message a été lu fait partie de l'interface du type **message**. Au niveau d'une classe **L*f*P** elle est utilisée dans le cas d'une attente d'activation de méthode et permet de savoir si la méthode est activée via la bonne file de message. Dans le cas d'une lecture de messages dans un média, elle permet d'identifier le message effectivement

consommé.

L'opération permettant de déterminer la nature d'un message est également supportée par le type `message`. Elle permet de déterminer si un message est un message d'activation (message d'appel de méthode) ou un message de données (interface de *message passing*). Elle est requise par les composants **L*f*P** pour déterminer la validité d'un message lors de sa réception.

6.6.7 L'instanciation dynamique

En **L*f*P**, l'instanciation dynamique d'un composant crée une nouvelle instance de ce composant, lui associe les ressources nécessaires à son exécution et retourne une référence **L*f*P** sur le nouveau composant.

Les ressources associées à un composant **L*f*P** sont :

- les binders nécessaires pour initialiser ses ports ;
- le thread qui lui est associé.

L'instanciation peut être réalisée de plusieurs manières en fonction des capacités du langage cible. Dans le cas d'un langage disposant de fonctionnalités réflexives, cette opération peut être entièrement réalisée par un appel à l'exécutif. C'est lui qui gérera directement la création, l'initialisation et l'enregistrement des ressources liées à la nouvelle instance.

Dans le cas d'un langage ne disposant pas d'une interface réflexive, il est nécessaire d'effectuer une partie des opérations directement dans le code généré, les appels à l'exécutif se limitent à l'insertion des références vers le nouveau composant et ses ressources dans les structures internes à l'exécutif.

Une implémentation réflexive

Dans le cas où l'on dispose d'un langage réflexif, et que l'on souhaite utiliser cette fonctionnalité, on peut générer un code réalisant les opérations suivantes pour instancier un composant **L*f*P** :

new_instance ← variable qui désigne la nouvelle instance de composant ;
type_name ← chaîne de caractères contenant le nom de la classe du composant ;
liste_attributs ← tableau de chaînes de caractères contenant des noms des attributs à initialiser ;
liste_valeurs ← tableau des valeurs initiales des attributs à initialiser ;
new_instance := **create_instance**(*type_name*, *liste_attributs*, *liste_valeurs*) ;

Cette implémentation repose sur l'appel à **create_instance** qui doit être implémentée dans l'exécutif et réalise les opérations suivantes :

- instancier un objet de type *type_name* ;
- insérer le nouveau composant dans la structure interne de l'exécutif ;
- initialiser tous les attributs dont les noms et les valeurs initiales sont passés en paramètres ;
- créer les instances de binder correspondant aux ports de multiplicité **1** ;
- relier tous les ports de multiplicité **all** aux instances de binder correspondantes ;
- créer le fil d'exécution (*thread*) correspondant au nouveau composant et démarrer son exécution.

Chacune de ces opérations peut être réalisée par des accès utilisant les propriété de réflexivité du langage cible sur la structure du composant à instancier.

Une implémentation non réflexive

Si le langage cible n'est pas réflexif ou qu'on ne souhaite pas utiliser de propriété réflexive, il est nécessaire de générer le code ad-hoc effectuant les opérations effectué par l'appel à `new_instance` dans la solution réflexive.

Le code généré doit donc réaliser les opérations suivantes :
- la création de la nouvelle instance du composant ;
- insertion de la nouvelle instance dans les structures internes de l'exécutif ;
- une série d'affectation pour initialiser les attributs spécifiés dans l'instruction **L*f*P** d'instanciation dynamique ;
- une série d'affectation pour initialiser les ports de multiplicité **1** du composant avec des binders instanciés par un appel à l'exécutif ;
- une série d'affectation pour initialiser les ports de multiplicité **all** avec les instances statiques de binders ;
- la création du fil d'exécution du composant, et l'instruction pour lancer son exécution.

Les informations nécessaires sont fournies par l'instruction **L*f*P** d'instanciation dynamique :
- nom du type à instancier ;
- liste des attributs initialisés lors de l'instanciation ;
- liste des ports statiques du composant.

Les informations complémentaires sont fournies par le diagramme d'architecture :
- liste des ports du composant ;
- liste des ports statiques (de multiplicité **all**).

On obtient ainsi un code plus volumineux qu'en utilisant la solution réflexive, mais plus rapide car il évite un grand nombre d'accès réflexifs sur le contenu des composants. De plus les étapes de la création d'un nouveau composant apparaissent en clair dans le code.

Cette solution repose sur l'exécutif pour les services suivants :
- l'insertion d'un composant dans les structures de l'exécutif ;
- l'instanciation de nouveaux binders et leur enregistrement comme port du nouveau composant.

Le premier service permet de déclarer le composant pour qu'il soit reconnu comme une destination possible pour une requête. Le deuxième service permet d'associer au composant les ressources qu'il utilise dans l'exécutif afin de les libérer lorsqu'il terminera son exécution.

6.6.8 Terminaison de l'exécution d'un composant

Lorsqu'un composant termine son exécution, l'exécutif doit libérer les ressources qui lui sont allouées et considérer toute requête vers ce composant comme une erreur.

Les ressources utilisées par le composant sont définies dans les tables de composants de l'exécutif. Il suffit donc de signifier leur destruction par un appel à l'exécutif avec en paramètre la référence du composant à supprimer.

6.7 Le code Java généré

Cette section décrit les règles mises en place pour l'implémentation du générateur de code **L*f*P**. On s'intéresse principalement aux règles de génération des automates et de quelques types de données représentatifs afin de montrer comment les règles générales présentées au chapitre précédent peuvent être instanciées sur une plate-forme cible telle que Java.

6.7.1 Principes de génération des types de données

Java est un langage orienté objet, l'implémentation des types de données est donc adaptée à ce contexte. Dans la mesure du possible, nous générons toujours une classe java pour chaque type de données **L*f*P**. Si les opérateurs valides pour ce type ne sont pas implémentables directement par une instruction java, ils sont implémentés par des méthodes de la classe. Chaque variable **L*f*P** est traduite par une variable java dont le type est la classe générée pour son type **L*f*P**.

Les types tableau

La structure d'une classe générée pour un type tableau est la suivante :
- une liste d'attributs définit les valeurs des indices valides pour chaque dimension du tableau (les indices des tableaux **L*f*P** ne commencent pas nécessairement à 0) ;
- un attribut *values* définit un tableau dont le nombre de dimensions et la taille sont définies en fonction du point précédent ;
- une méthode sert à initialiser le tableau ;
- une méthode de recopie implémente la sémantique de l'affectation pour les tableaux.
- deux constructeurs permettent de construire un tableau vide et un tableau dont le contenu est initialisé à une valeur donnée.

Les accès aux cases du tableau sont réalisés en accédant à l'attribut *values* ; l'indice est calculé en fonction de l'indice de la première case dans la déclaration du type **L*f*P**.

Les types entiers

Le type **L*f*P** *integer* est implémenté par le type java `Integer`.

Les types énumérés

Les types énumérés sont des constructions très classiques des langages de programmation, mais ne sont pas définis en java. Il est donc nécessaire de les *émuler*. Pour cela, nous utilisons une classe dont la structure est la suivante :
- un attribut par valeur du type énuméré portant le nom de la valeur qu'il représente et don la valeur initiale correspond à sa position dans la déclaration du type en partant de 1 ;
- un attribut *value* définit la valeur d'une variable de ce type ;
- une méthode calcule le successeur de la valeur courante ;
- une méthode calcule le prédécesseur de la valeur courante ;
- un constructeur permet l'initialisation d'une variable ;

Les valeurs immédiates rencontrées dans le modèle sont implémentées en faisant référence explicitement à l'attribut de la classe. Le nom de la valeur immédiate apparaît donc clairement dans le code. Cette construction n'est pas prévue pour optimiser la vitesse d'exécution du code, mais sa lisibilité afin de simplifier le debugage.

Les types *record*

Les types records ont une traduction triviale en java : il suffit de déclarer une classe contenant un attribut par champ de la structure **L*f*P** à implémenter.

6.7.2 Code généré pour les composants L*f*P

Le code généré implémente les instructions **L*f*P** présentes dans le modèle. Nous allons maintenant voir comment ces instructions peuvent être traduites en java.

Structuration du code d'un composant

Chaque composant **L*f*P** est généré sous la forme d'une classe java qui dérive de la classe `LfPClass` lorsque le composant est une classe et de `LfPMedia` lorsque le composant est un média. Ces deux classes définissent l'environnement minium pour leur classe de composant **L*f*P**.

Les composants **L*f*P** étant actifs, les classes générées doivent implémenter l'interface `Runnable`. La fonction principale de la *thread* est `run`, elle ne prend aucun paramètre, ne retourne aucune valeur et ne doit pas lever d'exception. Cette méthode contient les instructions correspondant à l'automate principal du composant.

Les déclarations de l'automate principal du composant **L*f*P** sont faites directement à l'intérieur de la classe java générée. Dans le cas de types structurés, cela conduit à définir des classes internes ; les variables sont traduites sous formes d'attributs. L'implémentation de l'exécutif utilisant la réflexivité pour accéder à la structure interne des composants **L*f*P**, les attributs et classes internes sont donc publiques.

Les méthodes des classes **L*f*P** sont implémentées sous forme de méthodes dans les classes java générées. Elles seront appelées par les instructions générées pour traiter les attentes de messages d'activation. Enfin, les triggers sont également implémentés par des méthodes. Elles sont appelées partout où le trigger est appelé dans l'automate initial.

on a donc le tableau de correspondance suivant :

entité **L*f*P**	construction java
composant **L*f*P**	classe java
méthode **L*f*P**	méthode java
trigger	méthode java

L'instruction goto

Cette instruction ne peut pas être traduite directement en java qui ne propose aucune construction équivalente. La structure du code généré doit donc prévoir l'implémentation de cette instruction.

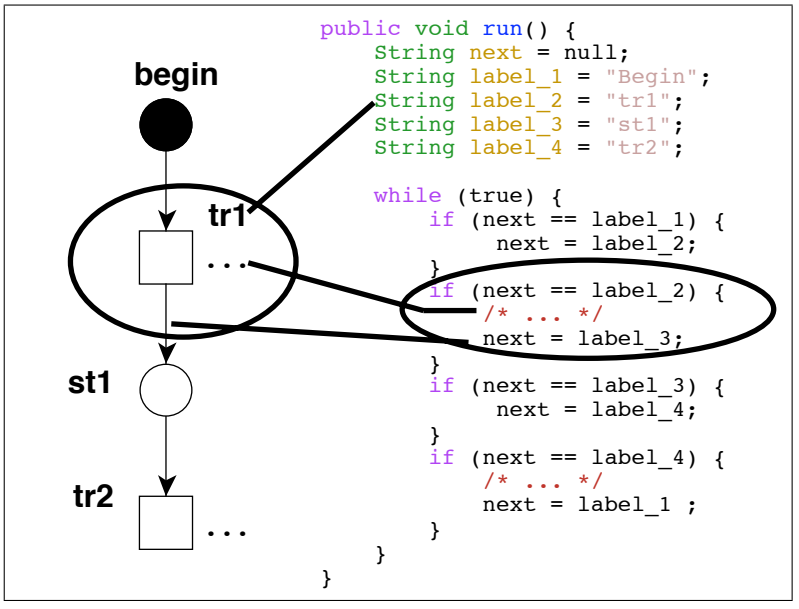


FIG. 6.4 – Structure du code d'une méthode

La structure du code d'un automate **L*f*P** est illustrée par la figure 6.4. Chaque automate est implémenté par une boucle *while* qui s'exécute tant que l'automate n'a pas atteint son état final. La variable prédéfinie *next* définit la prochaine transition (ou le prochain état) dont les instructions doivent être exécutées.

Chaque transition ou état nécessite donc les trois éléments suivants (mis en valeur sur la figure 6.4) :

- un label identifiant l'état ou la transition ;
- un bloc d'instructions exécuté lorsque la valeur de next *next* correspond au label associé ;
- l'affectation du label du prochain bloc à exécuter à *next*.

L'état final d'un automate de comportement marque la fin de son exécution. En terme de code java généré, cela signifie qu'il faut sortir de la boucle *while* qui l'implémente. On génère donc une instruction *break* pour les états finaux des automates de comportement.

Il est possible d'utiliser cette structure de manière hiérarchique en imbriquant les boucles while pour implémenter les sous-automates.

Les instructions de structuration

Les instructions de structuration sont les alternatives et les boucles. Ces instructions **L*f*P** sont très standard et peuvent être implémentées directement en java par la structure équivalente.

Instanciation dynamique

La solution retenue pour l'implémentation de l'instanciation dynamique de composants est fortement réflexive. Elle est basée sur un appel à l'exécutif qui sera chargé de construire la nouvelle instance, lancer son exécution et retourner sa référence **L*f*P**.

La figure 6.5 montre une instruction **L*f*P** d'instanciation dynamique d'un composant et le code correspondant. Les deux premières instructions permettent d'initialiser les paramètres de l'appel à l'exécutif qui réalisera l'instanciation :

- *var_temp1* est un tableau de chaînes de caractères contenant les noms des attributs à initialiser ;
- *var_temp2* est un tableau contenant les valeurs initiales de ces attributs.

Enfin, la l'appel à l'exécutif *newInstance* construit la nouvelle instance à partir de ces paramètres et du nom de la classe qui implémente le composant. Le typage des valeurs initiales des attributs est vérifié statiquement par le générateur de code, mais les erreurs dynamiques doivent être vérifiées par l'exécutif.

comp := composant (attribut_1 => 5,
attribut_2 => var_2)

```
/* Nom des attributs à initialiser */
String[] var_temp1 = { "attribut_1",
                       "attribut_2"};
/* valeur des attributs à initialiser */
Object[] var_temp2 = {5, var_2};

runtime.newInstance ("composant",
                     var_temp_1,
                     var_temp_2);
```

FIG. 6.5 – Traduction d'une instruction d'instanciation dynamique de composant

Terminaison d'un composant

La terminaison d'un composant est principalement laissée à la charge de l'exécutif. Avant de retourner, la fonction principale du composant utilise l'appel à la fonction `removeComponent` de l'exécutif qui retire le composant des structures internes et rend invalides les références vers ce composant.

Le ramasse-miette (*garbage collector*) de la machine virtuelle Java peut alors libérer l'espace mémoire occupé par cette instance. Pour que la mémoire soit correctement libérée, il faut que toutes les références au composant aient été supprimées (mises à *null* dans les structures de l'exécutif).

Envoi de messages

Nous avons réalisé cette opération directement par un appel à la fonction `sendMessage` définie par l'exécutif. Cet appel doit être réalisé après construction du message à envoyer. Le message est construit à partir de la classe `LfPMessage` présentée dans l'exécutif.

6.7.3 Initialisation de l'application

La partie déclaration du diagramme d'architecture regroupe toutes les constantes et instances statiques du modèle. Ces déclarations sont traduites dans la classe `GlobalDeclarations`. Une fonction `initialize` est générée pour initialiser chacune de ces constantes et instancier les composants statiques. C'est cette fonction qui est appelée par la classe `LfPStarter` de l'exécutif pour démarrer l'exécution du noeud local.

Le code généré pour cette méthode implémente la section d'action suivante :

- initialisation des variables globales ;
- création des binders statiques (binders *all*) par l'appel à `newStaticBinder` ;
- initialisation des références vers les composants statiques et création des instances locales par `newStaticComponent`.

Lorsqu'un site **L*f*P** démarre son exécution, il analyse le fichier de démarrage qui lui est fourni en paramètre. Celui-ci contient les caractéristiques des références des instances statiques des composants du modèle au format défini à la section 5.4. Ces informations définissent complètement les références des composants. La première étape consiste donc à initialiser toutes les variables globales contenant des références vers les instances statiques.

La deuxième étape consiste à instancier les composants correspondant. Seules les instances locales au site **L*f*P** en cours d'initialisation sont créées.

Une fois toutes les instances locales crées, l'exécution des composants peut démarrer et le site commence sa phase de fonctionnement normal.

6.8 Conclusion

Ce chapitre a montré comment la génération de code réparti pour le langage **L*f*P** a été réalisée. Les techniques utilisées sont empruntées aux travaux menés sur la transformation de modèle, en les adaptant au cas particulier de la génération de code.

Dans le cadre du `MDA`, le terme *génération de code* désigne le passage d'un PSM au code correspondant. Dans le cadre de la méthodologie **L*f*P**, ce terme désigne en plus la construction du PSM avant son implémentation. La plate-forme cible est définie par l'exécutif. La génération de code telle que nous la concevons inclus donc nécessairement une étape de transformation de modèle permettant d'implémenter la spécification **L*f*P** en fonction des caractéristiques de l'exécutif dont la stabilité garanti la pérénité des règles de transformation.

Ce PSM définit un modèle d'implémentation utilisé par la deuxième phase de la génération qui produit le code. Cette phase applique la syntaxe concrète du langage cible sur le modèle pour produire le code généré.

La séparation des considération liées à la plate-forme cible de celles liées au langage cible permet d'assurer la maintenabilité et la portabilité du générateur de code.

Les règles de transformation visent à l'implémentation des composants du langage **L*f*P** en fonction des caractéristiques du langage cible. On distingue principalement trois types de règles pour la génération :

– la vérification des aspects statiques ;
– la traduction du système de typage ;
– la traduction des instructions dynamiques du langage.

La vérification des aspects statiques permet de vérifier que les contraintes posées par les aspects architecturaux du langage sont correctement respectés dans la partie dynamique des composants. Ces vérifications sont essentielles pour garantir la cohérence entre le modèle produit et la spécification **L*f*P**.

Le deuxième axe concerne la traduction du système de typage du langage **L*f*P**. Celle-ci consiste à traduire les types de données définis dans la spécification **L*f*P**. Cet aspect inclus la traduction des opérateurs définis pour chacun de ces types. Plusieurs solutions sont envisagées en fonction du langage cible. Dans le cadre du langage Java, cette traduction donne lieu à la construction de classes dont les méthodes correspondent aux opérateurs définis sur le type de donnée.

Enfin, le troisième axe correspond à la traduction des instructions définies sur les automates de comportement des composants. Pour chaque instruction, un équivalent est fourni pour un langage de programmation. Ces traductions sont basées sur les primitives fournies par la plate-forme cible définie par l'exécutif. Celui-ci fournit une plate-forme stable qui assure la pérénité des règles de transformation.

Nous avons défini une implémentation de ces règles ciblant le langage Java. Le générateur de code et l'exécutif associé visent principalement les caractéristiques suivantes :

– séparation des aspects de déploiement et de génération de code ;
– utilisation de la réflexivité pour simplifier la structure du code généré ;
– forte utilisation des concepts orientés objets du langage java.

Les aspects liés au déploiement sont spécifiés dans un fichier de configuration parsé lors de l'initialisation de chaque site de l'application. Celui-ci peut être modifié de manière indépendante du générateur de code et ainsi permettre de re-configurer l'application sans modification du code généré.

Le langage cible permet d'utiliser des constructions réflexives pour manipuler la structure des composants. Afin de simplifier le code généré, ces constructions sont abondamment utilisées. L'objectif est de réduire la taille du code généré et d'en simplifier la structure en reportant une partie des opérations dans l'exécutif **L*f*P**. Ceci permet de simplifier les règles syntaxiques associées à certaines opérations. Par exemple l'instanciation dynamique de composant est implémentée en un seul appel lorsque la réflexivité est utilisée (cf. 6.6.7).

Enfin, le code généré est fortement orienté objet. Par exemple, chaque type **L*f*P** est en effet traduit sous la forme d'une classe Java, et chaque opérateur associé au type en **L*f*P** est implémenté sous la forme d'une méthode. L'objectif est ici de structurer au mieux le code produit pour améliorer sa lisibilité et faciliter la mise au point du générateur.

Ce prototype de générateur de code, relié à l'exécutif décrit au chapitre 5 représente la première implémentation fonctionnelle de la chaîne de génération de code pour **L*f*P**. Cette étape importante dans le cadre du projet MORSE permet de tester les options de déploiement et les stratégies de génération de code présentées dans ce mémoire. Notre objectif est maintenant de présenter un exemple concret d'utilisation de ce générateur de code sur un exemple de taille raisonnable.

Chapitre 7

Expérimentation : le projet MORSE

7.1 Introduction

Ce chapitre présente une étude de cas pour illustrer l'utilisation du langage **L*f*P** dans un cas concret. L'étude présentée est un système de messagerie instantanée de type *Internet Relay Chat*. Il s'agit donc d'une application capable de gérer plusieurs groupes de discussion en parallèle.

Cette application représente un bon compromis entre la taille du modèle qui doit rester compatible avec ce mémoire d'une part, et la couverture des fonctionnalités du langage **L*f*P** et du générateur de code d'autre part. Les études de cas plus importantes sont réalisées dans le cadre du projet MORSE qui prévoit l'étude d'exemples industriels beaucoup plus volumineux et complexes appliquées au domaine des drones SAGEM.

L'objectif de ce chapitre est donc d'appliquer sur l'exemple choisi l'ensemble des outils **L*f*P** disponibles pour la génération automatique de code. Ce chapitre commence donc par présenter l'exemple utilisé (section 7.2) puis présentera le modèle **L*f*P** correspondant. La présentation est faite en deux temps : le diagramme d'architecture du modèle (section 7.3) puis les diagrammes de comportement des composants (section 7.4). Enfin, un déploiement possible de cette application est étudié à la section 7.5.

La suite du chapitre est consacrée à l'étude du code produit par le générateur (section 7.6). Cette section présente le code généré pour les instructions les plus significatives du modèle en mettant l'accent sur l'implémentation des communications entre les composants.

7.2 Présentation de l'exemple

Le principe du système de messagerie modélisé ici est le suivant :

- un serveur attend les connexions de nouveaux clients ;
- un client peut se connecter sur un serveur soit pour se joindre à un groupe existant, soit pour créer son propre groupe de discussion ;
- lorsqu'il est connecté à un groupe, un client peut envoyer des messages à tous les autres membres du groupe ;
- lorsqu'il est connecté à un groupe, un client reçoit tous les messages envoyés par un membre du groupe ;
- un client peut quitter le groupe auquel il est connecté à tout instant.

Du côté serveur, l'application de messagerie instantanée gère un ensemble de groupes de discussion indépendants les uns des autres. Chaque serveur doit pouvoir gérer plusieurs groupes de discussion simultanément. Il gère les arrivées et départ des utilisateurs, ainsi que leurs changements de groupes. Chaque client de l'application doit être capable de lire gérer les interactions

avec l'utilisateur pour lui permettre de s'abonner à un groupe ou de le quitter, lire les messages qu'il souhaite envoyer, et afficher les messages qu'il reçoit.

Afin de conserver un modèle d'une complexité raisonnable pour cet exemple, nous faisons les **hypothèses simplificatrices** suivantes :

1. chaque client n'est connecté qu'à un seul groupe de discussion à un instant donné ;
2. les liaisons de communication sont fiables ;
3. les clients sont *fiables* et respectent toujours le protocole d'interaction prévu par le serveur.

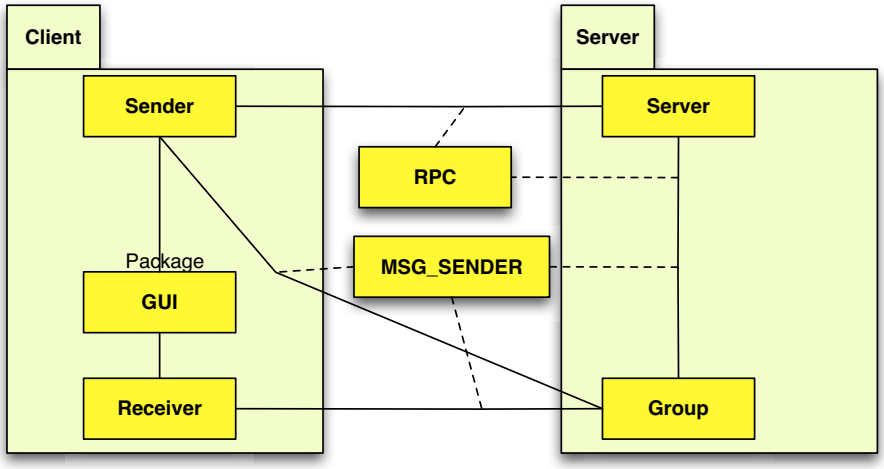


FIG. 7.1 – Diagramme de classe simplifié de l'étude de cas

La figure 7.1 présente le diagramme de classe UML simplifié du système. Un *client* du système de messagerie est constitué d'une instance des trois classes suivantes :

- `receiver` qui réceptionne les messages en provenance du service de diffusion ;
- `sender` qui envoie les messages à destination du service de diffusion ;
- `GUI` l'interface graphique du client qui gère les interactions avec l'utilisateur humain.

La partie serveur du système est assurée par deux classes : la classe `groupe` implémente le comportement d'un groupe de discussion. La classe `server` gère l'ensemble des groupes actifs sur le serveur.

La classe `server` assure la gestion des groupes :

- création et destruction de groupes ;
- client souhaitant rejoindre un groupe ;
- gestion de la liste des groupes actifs.

La classe `group` assure les tâches suivantes :

- enregistrement des nouveaux clients ;
- diffusion à tous les participant de chacun des messages soumis par un client ;
- gestion des départs des clients ;
- terminaison lorsque le groupe est vide.

Les communications entre les clients et le serveur sont assurées par la classe d'interaction `RPC` qui permet d'assurer une interaction de types *appel de procédure distante*. Tous les appels de méthodes passant par ce type d'interaction doivent fournir une valeur de retour.

Les communications entre le client et le groupe auquel il appartient sont assurées par la classe d'interaction `MSG_SENDER` qui permet une communication de type *envoi de message* entre deux classes.

Les interactions entre les groupes et le serveur sont assurés par deux classes d'interaction distinctes : `RPC` pour les appels de procédures synchrones, `MSG_SENDER` pour les appels de méthode sans valeur de retour (équivalent à un envoi de message).

7.3 Partie statique du modèle : le diagramme d'architecture

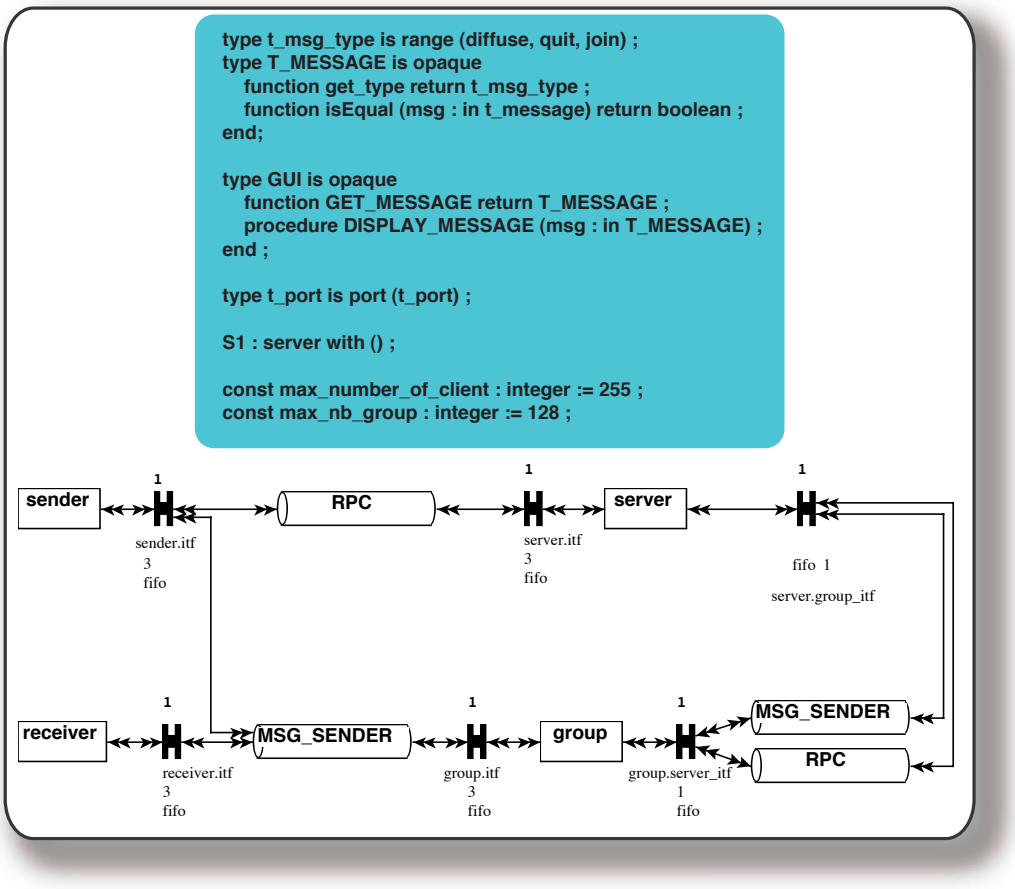


FIG. 7.2 – Diagramme d'architecture du système de messagerie instantanée

La figure 7.2 présente le diagramme d'architecture du système considéré. Il définit la vue statique du modèle.

7.3.1 Définition des composants

La partie graphique du diagramme d'architecture définit les composants du système. On y retrouve les composants vus sur le diagramme de classe. Les classes `sender`, `receiver`, `server`, et `group` sont implémentées sous forme de classes **L*f*P**. En revanche, les classes d'interaction `RPC`, et `MSG_SENDER` ont été traduites sous forme de médias. Ces composants définissent la partie contrôle de l'application, leur comportement sera donc modélisé dans la spécification **L*f*P**.

L'interface graphique du système est représenté en utilisant le type opaque `GUI`. Celui-ci définit une interface utilisée par les composants **L*f*P** pour interagir avec l'interface proposée aux utilisateurs humains du système.

Le diagramme d'architecture précise également le fonctionnement des liaisons entre les composants en détaillant les files de messages (représentées par les binders), ce qui permet de définir la topologie de l'application. Par exemple, la classe `server` utilise deux files de messages distinctes. Ainsi, pour ses interactions avec les clients, le serveur utilise un binder relié à un média de type `RPC` correspondant au port `itf` de la classe. Pour ses interactions avec les instances de la classe `group`, le serveur utilise le binder `group_itf`. Ce dernier peut être relié à la classe `group` par deux types de médias : `MSG_SENDER` pour les appels de procédures asynchrones (sans message de retour), et `RPC` pour les appels de méthodes nécessitant une valeur de retour (fonctions). Ces deux médias sont reliés au binder `server_itf` qui définit la file de messages utilisés par le groupe pour traiter les messages du serveur.

Les clients du système de messagerie (classes **L*f*P** `sender` et `receiver`) communiquent avec les groupes par l'intermédiaire d'un média de type `MSG_SENDER`. Chacune de ces classes définit sa file de message gérant ses interactions avec les autres composants du modèle. Tous les binders du modèle sont de multiplicité `1` ce qui signifie qu'une nouvelle instance de la file de messages est créée pour chaque instance de la classe définie dans l'attribut liaison.

7.3.2 Définition des types de données

La partie déclarative du diagramme d'architecture de la figure 7.2 définit les types et constantes partagés par les composants du modèle.

Le type `GUI`

Le type `GUI` est un composant externe de traitement de données ; son rôle est de traiter les interactions entre l'application et l'utilisateur, et de les traduire en évènements qui pourront être traités par la partie contrôle de l'application. Pour cette raison, elle est traduite dans la spécification **L*f*P** par un type opaque défini dans le diagramme d'architecture du modèle.

Le type `GUI` définit donc l'interface de la partie controle avec l'interface graphique de l'application. Il fournit deux primitives permettant de modéliser les interactions entre la partie controle de l'application et la partie de traitement de donnée :

- `get_message` retourne le prochain message que le client veut envoyer ;
- `display_message` provoque l'affichage d'un message reçu depuis le groupe de discussion courant.

Le type `t_msg_type`

le type énuméré `t_msg_type` définit la nature d'un message de l'application. Les variables de ce type peuvent prendre trois valeurs :

- `diffuse` qui correspond à un message à diffuser à tous les membres du groupe courant, Par convention, le contenu d'un message de type `diffuse` est le texte à diffuser à tous les abonnés du groupe.
- `quit` qui correspond à l'action de quitter le groupe de discussion auquel le client participe, le message est alors vide ;
- `join` qui correspond à l'action de créer un groupe de discussion ou de le rejoindre s'il existe déjà ; par convention, le contenu d'un message de type `join` est le nom du groupe à rejoindre ou à créer.

Le type `t_message`

le type opaque `t_message` définit l'interface de la partie controle avec un message envoyé par le client au système de messagerie. Le contenu du message n'est pas accessible à la partie controle. Selon le cas d'utilisation, il peut s'agir soit :

- du nom d'un groupe auquel le client veut se joindre ;
- du message que le client diffuse au groupe.

Dans les deux cas ces valeurs sont fournies par l'utilisateur sous la forme de chaînes de caractères de longueur variable. Elles ne peuvent donc pas être manipulées directement en **L*f*P**. Le type `t_message` fournit donc deux *appels externes* devant respecter le contrat suivant :

- `get_type` retourne le type du message sous forme d'une valeur de type `t_msg_type` ;
- `isEqual` retourne un booléen égal à `true` si le message passé en paramètre a le même contenu que le message sur lequel l'appel est réalisé.

Le type `t_port`

Le type `t_port` correspond à un type de port pour lequel les discriminants des messages doivent contenir exactement une valeur de type `t_port`. Il est utilisé pour définir les types des ports de toutes les classes du modèle.

Les variables globales du modèle

Les variables globales du système définissent les constantes partagées par tous les composants de l'application, ainsi que les instances statiques des composants.

La constante `max_number_of_client` définit le nombre maximum de personnes pouvant se connecter à un des groupes géré par le serveur.

La constante `max_number_of_group` définit le nombre maximum de groupe de discussion que peut héberger un serveur.

La variables `S1` définit l'instance de la classe `server` qui sera créée au démarrage du système de messagerie pour recevoir les messages des clients.

7.4 Partie dynamique du modèle : comportement des composants

Cette section présente les diagrammes de comportement des composants du système de messagerie instantanée. Ils définissent le comportement dynamique de l'application.

7.4.1 Modélisation des médias

Le modèle comporte deux médias : `RPC` et `msg_sender`.

Le média `msg_sender`

Ce média implémente l'envoi d'un message depuis un composant source vers un destinataire dont la référence est fournie dans le discriminant du message. Son diagramme de comportement est donné sur la figure 7.3.

Ce média déclare trois variables :

- `msg` de type message contiendra le message reçu par le média ;
- `input` de type `t_port` est le port reliant le média au composant émetteur du message ;
- `target` également de type `t_port` est le port reliant le média au destinataire du message.

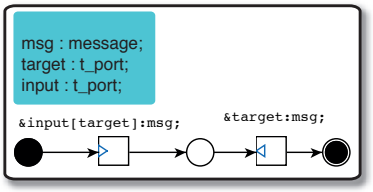


FIG. 7.3 – Diagramme de comportement du média `msg_sender`

Lors de la création d'une instance de `msg_sender`, l'utilisateur doit initialiser l'attribut `input`. Lorsque son exécution débute, le média se met en attente d'un message sur ce port. Comme le spécifie la définition du type `t_port`, le message doit contenir la référence d'un binder dans son discriminant. Cette valeur est affectée à l'attribut `target`. Le contenu du message est sauvegardé dans l'attribut `msg`. L'attribut `target` est ensuite utilisé comme port de sortie par l'instruction d'envoi de message.

Une fois que le message a été déposé dans le binder cible, le média termine son exécution. Une instance de `msg_sender` ne peut donc servir à envoyer qu'un seul message.

Le média RPC

Le diagramme de comportement de ce média est donné par la figure 7.4.

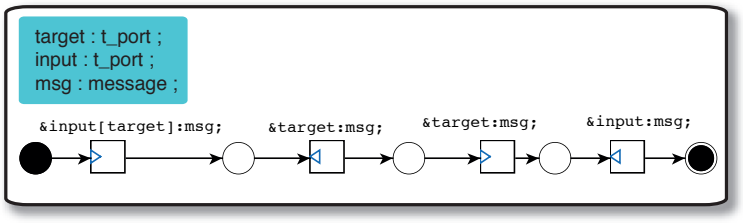


FIG. 7.4 – Diagramme de comportement du média `RPC`

Le média `RPC` déclare trois attributs de même nom et de même type que `msg_sender`. Le début de l'exécution de `RPC` est similaire : le média se met en attente sur le port `input` initialisé lors de l'instanciation, puis le dépose dans le binder spécifié dans le discriminant du message reçu.

Ensuite, `RPC` se met en attente sur le binder où il a déposé le premier message (troisième transition du média). Lorsqu'il reçoit un message , il le dépose dans le binder désigné par `input` (quatrième transition). Cette série d'opérations permet de transmettre le message d'activation d'une méthode et le message de retour correspondant.

Après avoir traité un appel de méthode, le média termine son exécution. Une nouvelle instance de `RPC` doit donc être créée avant chaque opération d'appel de méthode utilisant ce média.

7.4.2 La réception des messages du côté client

Ce rôle est assuré par la classe `receiver`. Son comportement est très simple et est présenté sur la figure 7.5. Cette classe attend un message sur son port `itf`. Ce message doit contenir une seule valeur de type `t_message`. Ce message est ensuite simplement passé à l'interface graphique pour être affiché pour lecture par l'utilisateur. Cette opération est réalisée par l'appel à la méthode externe `display_message` définie sur le type opaque `GUI`. L'instance de `receiver` retourne ensuite dans son état initial pour attendre le prochain message.

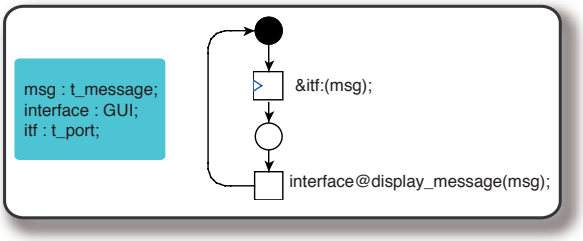


FIG. 7.5 – Diagramme de comportement de la classe `Receiver`

7.4.3 L'envoi des messages par le client

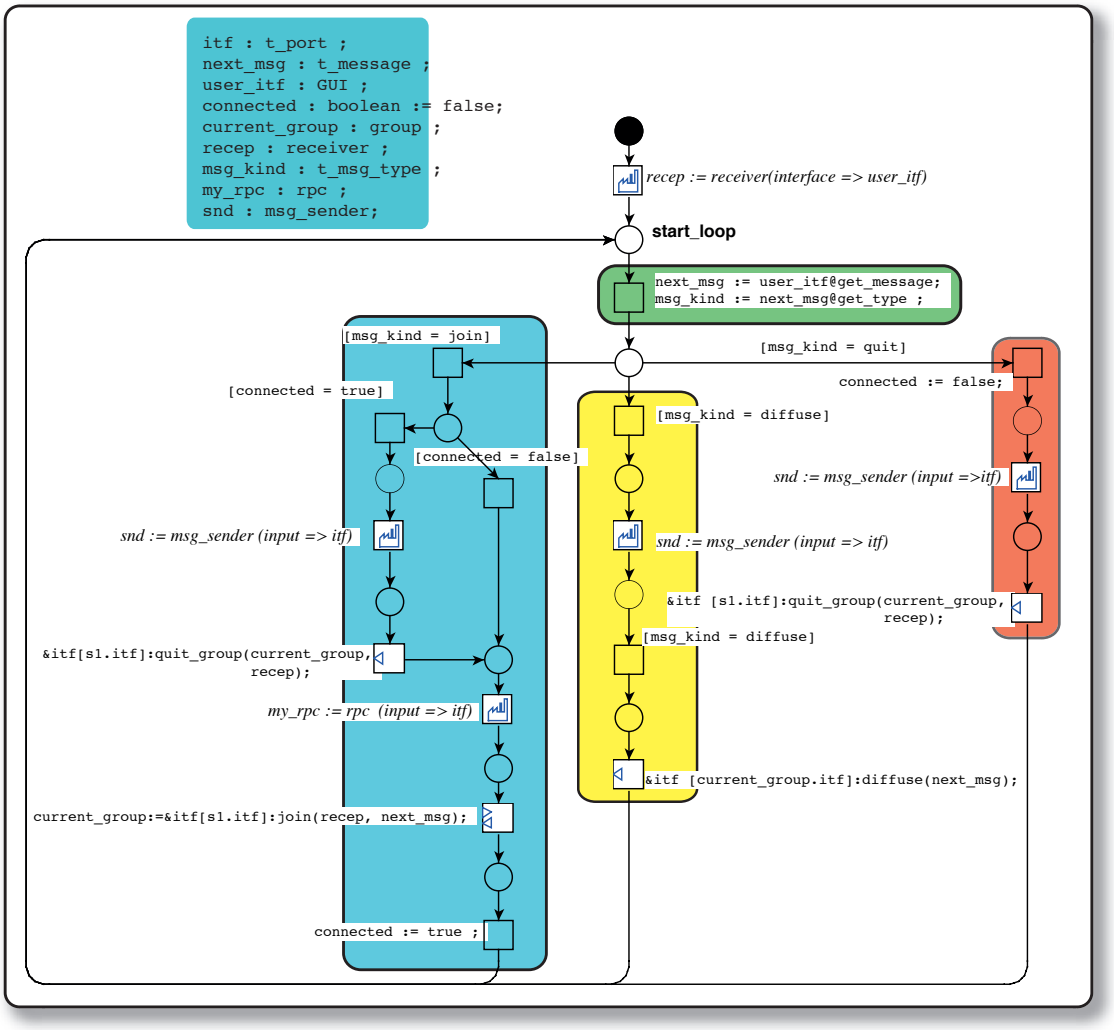


FIG. 7.6 – Diagramme de comportement de la classe `Sender`

L'envoi des messages par le client est assuré par la classe sender dont le diagramme de comportement est présenté sur la figure 7.6. La première action réalisée par la classe `sender` est de créer l'instance de `receiver` qui lui est associée pour le client. Le composant externe `GUI` est initialisé lors de sa déclaration. Ensuite, le composant entre dans une boucle correspondant à la lecture d'un message entré par l'utilisateur sur l'interface graphique et à son traitement en fonction

de son type.

La lecture du message, ainsi que l'identification de son type sont effectués par la transition de sortie de l'état `start_loop` (sur fond vert sur la figure). Le traitement correspond aux trois branches présentes sur fond coloré de la figure 7.6.

Traitement des messages de type `diffuse`

La branche centrale correspond au traitement des messages de type `diffuse`. Dans ce cas, le message est envoyé directement au groupe pour qu'il le diffuse à tous ses abonnés.
Cette branche réalise les opérations suivantes :

- la première transition sert à définir l'alternative menant dans cette branche : `msg_kind` doit contenir la valeur `diffuse` ;
- ensuite le média servant à l'envoi de message est instancié, avec comme binder d'entrée le binder `itf` de l'instance courante ;
- la transition d'envoi de message crée un message d'activation pour la méthode asynchrone `diffuse` avec comme paramètre le message à diffuser, et un discriminant de message contenant la référence du binder où le message doit être déposé.

Le média utilisé est de type `msg_sender` car la méthode `diffuse` est asynchrone et ne donnera pas lieu à un message de retour.

Traitement des messages de type `join`

La branche la plus à gauche, correspond au traitement d'un message de type `join`, ce qui signifie que l'utilisateur cherche à rejoindre un groupe dont le nom est spécifié dans le corps du message.

Cette branche réalise les opérations suivantes :

- si le client est déjà connecté à un groupe (la garde `[connected = true]` s'évalue à `true`), il commence par le quitter en appelant la méthode `quit_group` sur le serveur.
- dans tous les cas, le client appelle la méthode `join` de la classe `server` pour rejoindre le groupe dont le nom est contenu dans message.

La valeur de retour de la fonction `join` de la classe `server` contient la référence du groupe auquel l'utilisateur appartient après la connexion.

Avant chaque envoi de message, un média approprié est instancié : `MSG_SENDER` pour les appels de méthode asynchrone, et `RPC` pour les appels de méthode synchrones.

Traitement des messages de type `quit`

La branche la plus à droite correspond au traitement d'un message `quit`, ce qui signifie que le client souhaite quitter le groupe auquel il est actuellement connecté.

Cette branche effectue les opérations suivantes :

- instanciation d'un média de type `MSG_SENDER` ;
- appel de la méthode asynchrone `quit_group` de la classe `server` avec en paramètre la référence du groupe à quitter et la référence du client qui se désabonne.

Quelle que soit la branche suivie, le client saute ensuite à l'*état* `start_loop` qui définit le début de la boucle d'interaction avec l'interface graphique.

7.4.4 La gestion du groupe : la classe `group`

Cette classe permet aux abonnés d'un groupe de communiquer par diffusion : chaque message émit par un client est transmis par tous les abonnés du groupe. Les instances de cette classe sont

toujours créées par le serveur qui héberge le groupe, à la demande d'un utilisateur.

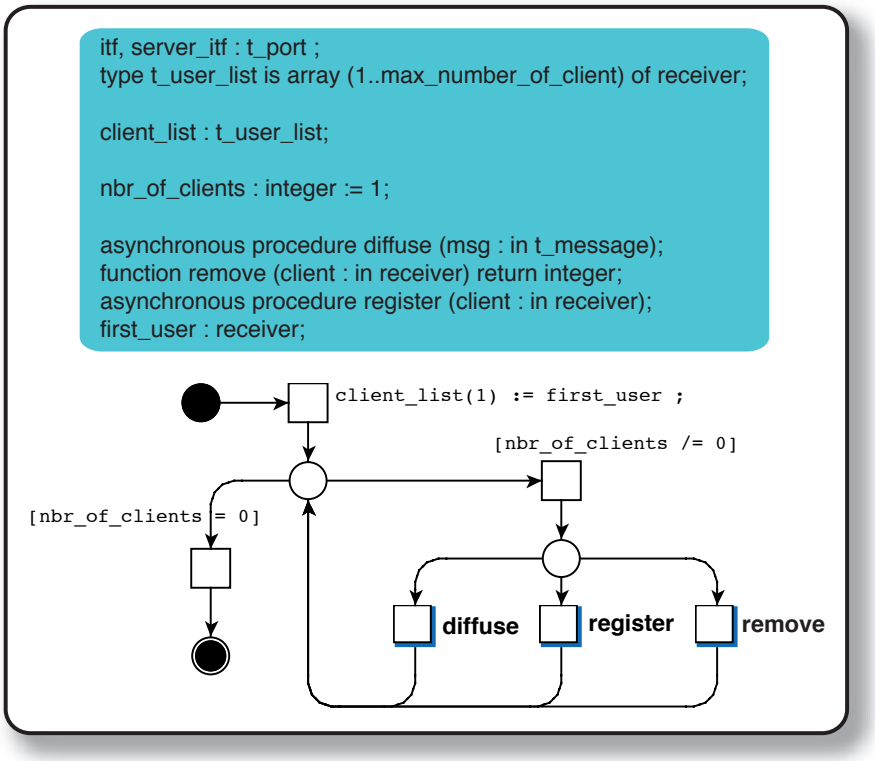


FIG. 7.7 – Diagramme de comportement de la classe `group`

Cette classe utilise un type tableau `t_user_list` pour conserver la liste des clients connectés. Cette liste est définie par la variable `client_list`. Enfin, `nbr_of_clients` définit le nombre de clients connectés à un instant donné sur le groupe. Cette valeur est initialisée à 1, car un groupe doit au moins avoir un client pour exister. La référence du premier client du groupe est conservée dans la variable `first_user`. Cette variable est initialisée par la classe `server` avec la référence du client qui provoque l'instanciation du nouveau groupe. Chaque client est identifié par la référence de l'instance de la classe `receiver` qui lui est associé.

Lors de son instanciation, le groupe commence par insérer le premier utilisateur dans la liste de ses abonnés. Le groupe arrive alors dans son état central : si le nombre d'abonnés est supérieur ou égal à 1, trois méthodes sont activables :

- `diffuse` permet de diffuser un message à l'ensemble du groupe ;
- `register` permet de rajouter un nouveau client dans la liste des abonnés ;
- `remove` retire du groupe le client passé en paramètre et retourne le nombre d'utilisateurs restant.

Si le nombre d'abonné est égal à zéro (0), le groupe n'a plus de raison d'être et l'instance termine son exécution. Elle sera retirée de la liste des groupes valides par le serveur.

La méthode `diffuse`

La méthode diffuse permet à un abonné de transmettre un message à tous les abonnés du groupe. Cette méthode dont le diagramme de comportement est présenté sur la figure 7.8 parcourt la liste des abonnés au groupe et leur transmet le message passé en paramètre.

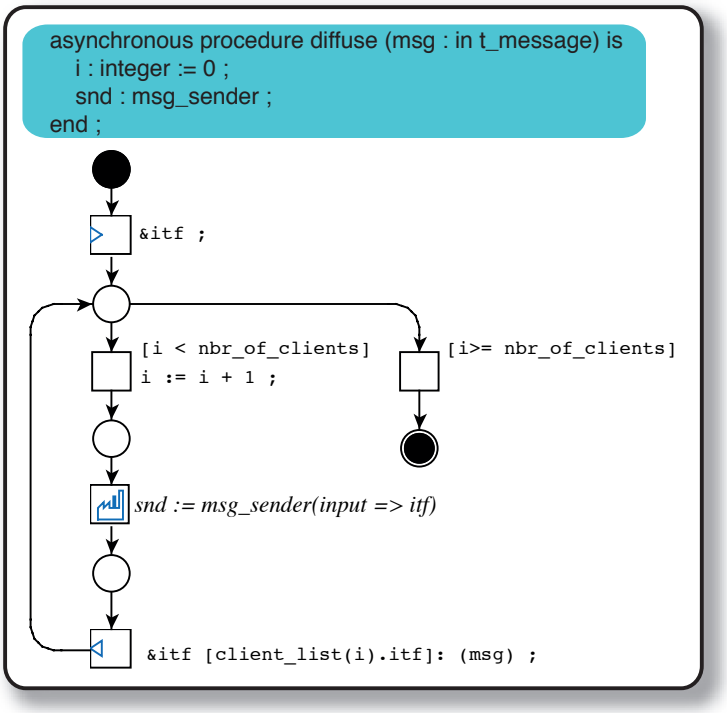


FIG. 7.8 – Diagramme de comportement de la méthode `diffuse`

Le parcours du tableau est effectué par la boucle gauche du diagramme. Le corps de la boucle incrémente le compteur d'indice, crée une instance du média `msg_sender` pour communiquer avec chaque abonné, et lui envoie le corps du message.

La méthode `register`

La méthode `register` présentée sur la figure 7.9 ajoute un abonné à un groupe existant. Le paramètre de cette procédure est l'utilisateur à ajouter au groupe. Il est ajouté dans le tableau des abonnés du groupe, et le nombre d'abonné est incrémenté.

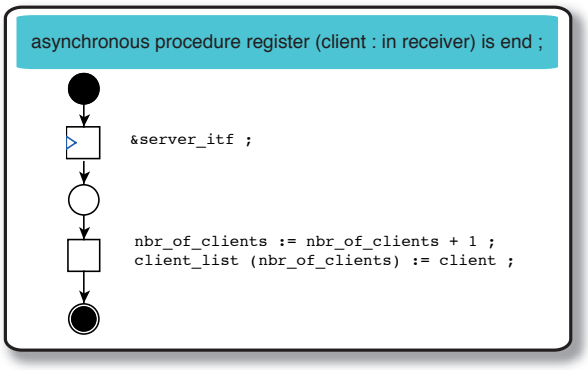


FIG. 7.9 – Diagramme de comportement de la méthode `register`

La méthode `remove`

La méthode `remove` permet au serveur de retirer un utilisateur du groupe. Le paramètre de la procédure est la référence de l'utilisateur à supprimer ; son diagramme de comportement est présenté sur la figure 7.10.

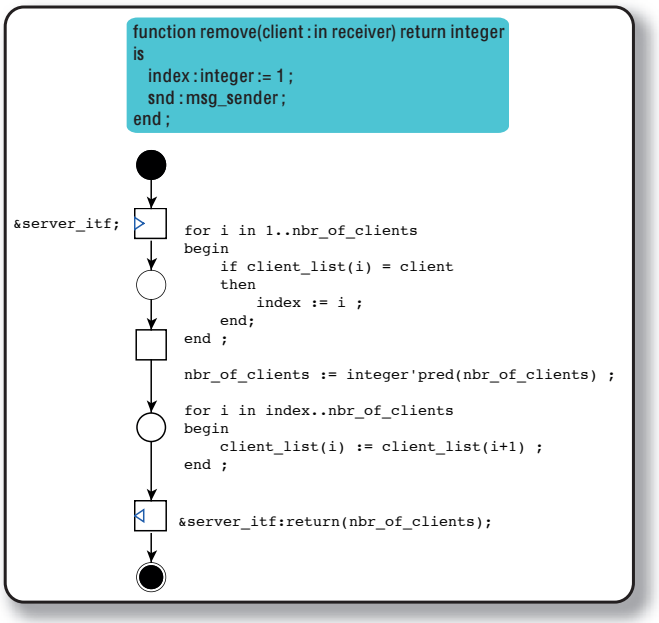


FIG. 7.10 – Diagramme de comportement de la méthode `remove`

La procédure `remove` commence par retirer l'utilisateur du tableau contenant la liste des abonnés et elle décrémente le compteur du nombre d'abonnés. Cette valeur est ensuite envoyée comme valeur de retour de la fonction.

Pour retirer l'utilisateur, le groupe commence par déterminer son indice dans le tableau contenant la liste des utilisateurs, puis à partir de cet indice, il décale toutes les valeurs d'une case vers la gauche.

7.4.5 Le serveur de gestion de groupes

L'objectif de la classe `server` est de générer les différents groupes de discussion dont il a la charge. Le serveur offre une interface basée sur deux méthodes :

- la fonction `join` qui permet de se joindre à un groupe existant ou d'en créer un nouveau, elle retourne la référence du groupe auquel l'utilisateur vient de s'abonner ;
- la procédure asynchrone `quit_group` qui permet à un utilisateur de se désabonner d'un groupe.

Le diagramme principal de cette classe est présenté sur la figure 7.11. Les instances de la classe `server` sont en permanence en attente d'activation d'une des deux méthodes vues précédemment.

La méthode `join`

La fonction `join` doit être appelée par un client désirant se joindre à un groupe où le créer ; son diagramme de comportement est donné par la figure 7.12. Sa valeur de retour est la référence du groupe auquel l'utilisateur vient de s'abonner.

Cette méthode a deux paramètres formels :

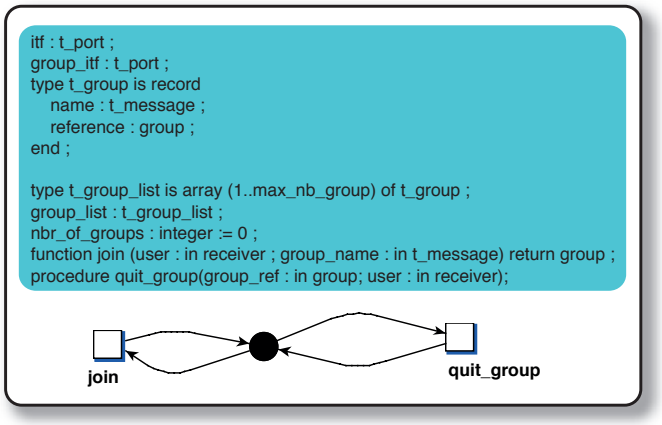


FIG. 7.11 – Diagramme de comportement de la classe `server`

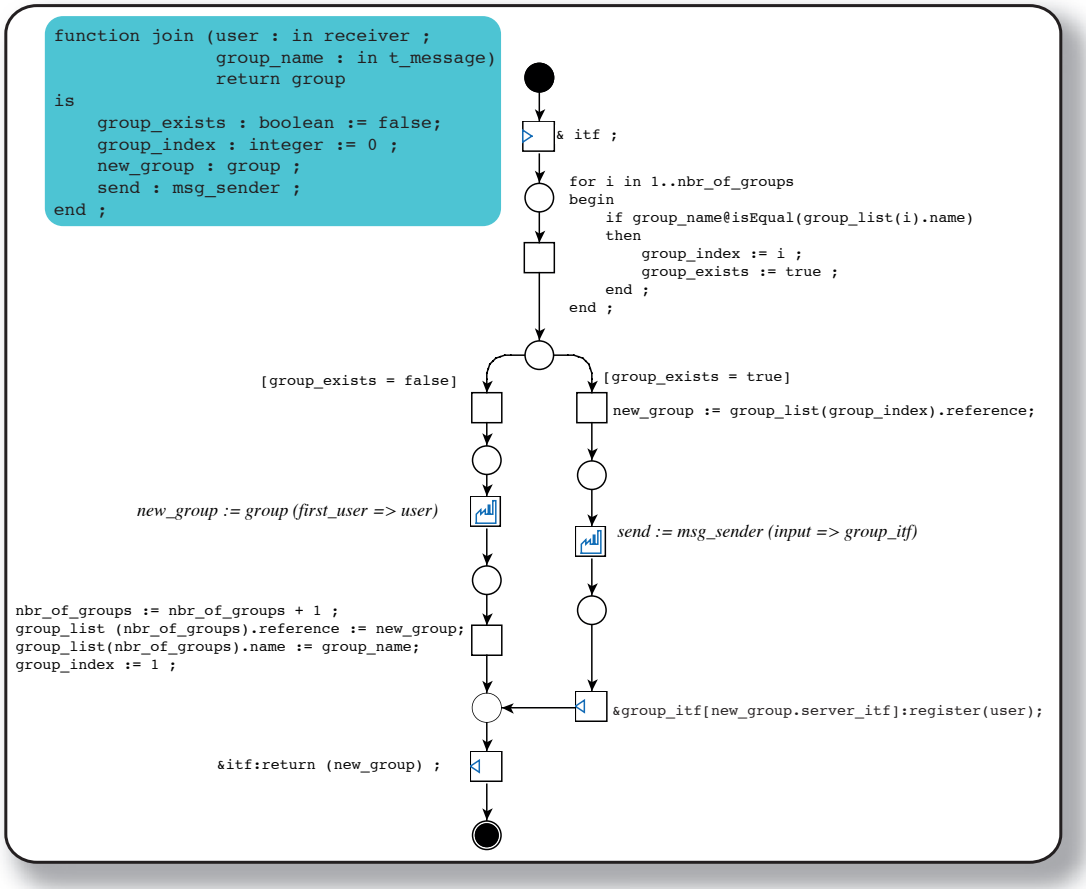


FIG. 7.12 – Diagramme de comportement de la méthode `join`

– `user` est la référence vers le client qui appelle la fonction ;
– `group_name` est le message qui contient le nom du groupe à créer ou à joindre.

Cette méthode commence par déterminer si le groupe auquel l'utilisateur veut s'abonner existe déjà. Cette opération est réalisée par une boucle for qui parcours le tableau contenant la liste des groupes existants. Si le nom d'un des groupes existants est égal au nom du groupe demandé, son

indice dans ce tableau est retenu, et le booléen `group_exist` reçoit la valeur true.

Si le groupe existe déjà (branche de droite), une instance de `msg_sender` est créée pour communiquer avec ce groupe. Puis la procédure asynchrone `register` est appelée pour abonner l'utilisateur au groupe.

Si le groupe n'existe pas encore (branche de gauche), un nouveau groupe est instancié. Lors de l'instanciation, on initialise l'attribut `first_user` avec la référence de l'utilisateur qui provoque la création du groupe. Puis le nouveau groupe est ajouté à la liste des groupes gérés par le serveur.

Enfin, la méthode retourne la référence du groupe auquel l'utilisateur a été abonné. Le message de retour est envoyé sur le port `itf` qui est utilisé pour toutes les communications avec les clients.

La méthode `quit_group`

Cette fonction est appelée par le client lorsqu'il souhaite quitter le groupe de discussion auquel il participe. Son diagramme de comportement est donné par la figure 7.13.

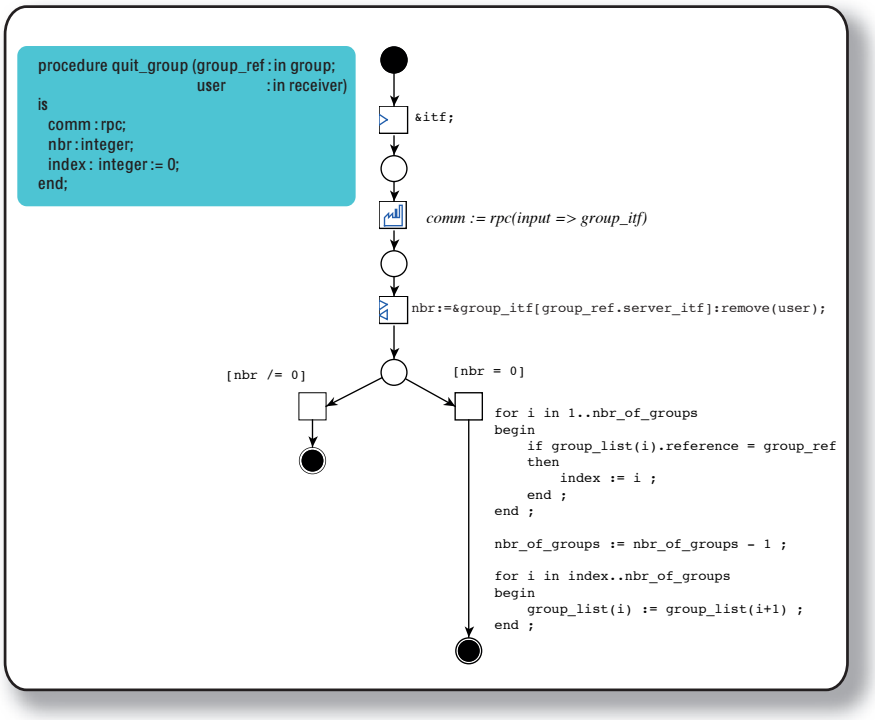


FIG. 7.13 – Diagramme de comportement de la méthode `quit_group`

La méthode `quit_group` prend deux paramètres

- `group_ref` : la référence du groupe que l'utilisateur veut quitter ;
- `user` : la référence de l'utilisateur qui se désabonne.

`quit_group` commence par appeler la méthode `remove` sur le groupe passé en paramètre. Cette méthode retire du groupe l'utilisateur qui lui est fourni en paramètre et retourne le nombre d'abonnés restant. Si ce nombre est supérieur à zéro, l'exécution de `quit_groupe` est terminée (branche de gauche). Si ce nombre est égal à zéro, il faut supprimer le groupe de la liste des groupes du serveur (branche de droite).

Pour supprimer le serveur de sa liste, le serveur commence par déterminer son indice dans cette liste (première boucle for), puis à partir de cet indice, il décale toutes les valeurs du tableau d'une case vers la gauche. Une fois ces opérations effectuées, la méthode termine son exécution.

7.5 Déploiement de l'exemple

L'exemple de la messagerie instantanée est d'un déploiement relativement simple. On doit distinguer deux types de sites :

- un site connu à l'avance sur lequel est déployé le serveur ;
- des sites clients instanciés à la demande des utilisateurs sur n'importe quelle machine connectée au réseau.

Le prototype du générateur de code ne permet pas de définir finement la structure de chaque exécutable. On définit donc un seul exécutable dont l'exécution est spécialisée en fonction des paramètres de démarrage. Dans le cas du langage Java, cette solution n'est pas excessivement pénalisante car le langage utilise un système de liaison dynamique : seules les classes requises seront chargées en mémoire.

7.5.1 Le site du serveur de groupes

Le site du serveur est défini par l'instance statique `S1` définie sur la figure 7.2. Le fichier de démarrage correspondant est présenté sur la figure 7.14, il contient la localisation du serveur et sera partagée par tous les sites participant à l'application.

```
<instances>
  <instance name="S1" hostName="glaucos.lip6.fr" port="12345" instanceNumber=" 1" />
</instances>
```

FIG. 7.14 – Fichier de déploiement de l'étude de cas

Pour les besoins de cet exemple, on considère un système constitué d'un seul serveur, le fichier de déploiement utilisé est donc réduit à sa plus simple expression : la seule instance statique à définir est le serveur auquel les clients doivent se connecter. Les groupes de discussion sont créés dynamiquement par le serveur, ils n'apparaissent donc pas dans le fichier de démarrage.

Lors du lancement de l'exécutable serveur, seul le fichier de démarrage est passé en paramètre. A l'aide des informations dont il dispose, l'exécutif peut déduire qu'il est exécuté sur le site **L*f*P** qui doit assurer le rôle de serveur.

7.5.2 Les sites clients

Les clients seront créés de manière asynchrone par les utilisateurs. Un exécutable destiné à servir de client doit être lancé avec deux paramètres :

- le fichier de configuration définissant les instances statiques du modèle ;
- le type de composant **L*f*P** à instancier pour démarrer le site.

Le fichier de démarrage permet d'initialiser les variables globales du modèle. Ici il s'agit de l'adresse du serveur à utiliser pour la diffusion des messages. Ces informations permettent donc d'initialiser `S1`.

Le deuxième paramètre est utilisé pour créer une instance d'un composant dont l'exécution définira le comportement de l'exécutable. Dans le cas de cet exemple, il s'agit de la classe **L*f*P** `Sender` qui définit le comportement d'un client. Les classes supplémentaires seront chargées par la machine virtuelle Java pendant l'exécution du programme. Le site sera connu du reste de l'application grâce aux références transmises par les clients lors de l'appel à la méthode `join` (référence de l'instance de la classe `receiver` qui identifie le client au niveau du serveur).

7.6 Présentation du code généré

Le prototype de générateur de code a permis de produire le code correspondant à la spécification **L*f*P** que nous venons de voir. Il consiste en 1135 lignes de java. Les composants externes de l'application ont été produits manuellement ; ils représentent 270 lignes de code pour obtenir une interface de *chat* minimaliste.

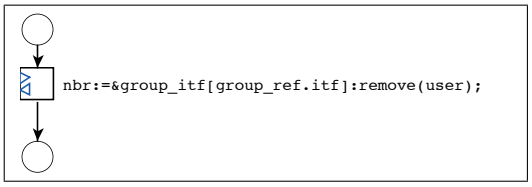
Le code généré pour le modèle de diffusion implémente le modèle et a rendu la spécification **L*f*P** exécutable dans un environnement réparti. Cette section présente quelques extraits significatifs de ce code afin d'illustrer par un exemple concret les règles présentées au chapitre 6.

7.6.1 Implémentation des appels de fonction

La figure 7.15 présente un extrait du code généré pour la classe serveur (7.15(b)), ainsi que la transition correspondante (7.15(a)). Elle est extraite de la méthode `quit_group` de la classe serveur dont le diagramme de comportement a été présenté sur la figure 7.13. Le code de la figure 7.15(b) implémente l'appel de la méthode `remove` de la classe `group`. Cette méthode est appelée par le serveur lorsqu'un client souhaite se retirer du groupe.

Les caractéristiques de cet appel sont les suivantes :

- Le message d'activation doit être déposé dans le port `group_itf` de la classe `serveur` ;
- la méthode appelée est une fonction (cf. figure 7.10) dont le prototype est le suivant :
- la valeur de retour est stockée dans la variable `nbr`.



(a) Un exemple de transition d'appel de fonction

```
if (next == label_74) {
    /* METHOD_CALL BEGINS */
    msg = new LfPMessage();
    msg.setMethodName("REMOVE");
    msg.setBinder(GROUP_ITF);
    msg.setDiscriminant(GROUP_REF.getBinder("ITF"), 1);
    msg.setParameter(USER, 1);
    runtime.sendMessage(msg);
    msg = runtime.getSimpleMessage (GROUP_ITF);
    NBR = (INTEGER) msg.getReturnValue();
    /* METHOD_CALL ENDS */
    next = label_70;
}
```

Construction du message d'appel

envoi du message

attente du message de retour

lecture et affectation de la valeur de retour

(b) Code correspondant à la transition

FIG. 7.15 – Transition d'appel de fonction **L*f*P** et le code d'implémentation

La figure 7.15(b) montre le code correspondant à cette transition. La variable `next` est un marqueur qui définit la prochaine transition à exécuter. Le test de la première ligne permet donc de s'assurer que la classe `server` est prête à exécuter l'appel de méthode.
Les lignes 3 à 7 construisent le message d'activation de la méthode appelée :

– la ligne 3 crée l'instance qui contiendra le message d'activation
– la ligne 4 insère dans le message le nom de la méthode à activer ;
– la ligne 5 insère le binder cible dans le message (binder dans lequel il sera déposé) ;
– la ligne 6 insère le premier (ici le seul) élément du discriminant. Ce dernier est déterminé par dé-référenciation de la variable `group_ref`. Cette opération est implémentée par l'appel à la méthode `get_binder` de la class `LfPReference`. Le paramètre de cette méthode est le nom du port dont la référence est recherchée (ici, `ITF`).
– enfin la ligne 7 insère le paramètre effectif de la fonction, c'est à dire le paramètre `user`.

La ligne 8 envoie le message au moyen d'un appel à la méthode `sendMessage` de l'exécutif associé à cette instance de serveur. La ligne 9 est un appel bloquant à la fonction de l'exécutif qui retourne le message de retour de la fonction. Une fois ce message reçu, la valeur de retour effective est extraite et affectée à la variable prévue à cet effet dans la spécification **L*f*P**.

Enfin, la ligne 12 termine l'exécution de la transition en affectant le label de l'état de sortie à la variable next.

7.6.2 Code généré pour une instruction select

La figure 7.16 présente le code produit pour l'état central de la classe `group` dont le diagramme de comportement était présenté sur la figure 7.7. La configuration de l'état représenté est rappelée sur la figure 7.16(a). Lorsque le groupe atteint cet état, il attend un message d'activation pour l'une des trois méthodes suivantes :

– `diffuse;`
– `remove;`
– `join;`

Le code correspondant à cette configuration est présenté sur la figure 7.16(b). Le code est inséré dans une alternative dont la condition est évaluée à true lorsque l'instruction peut être exécutée. Les lignes 2 et 3 du code généré instancient un tableau qui contient la liste des binders sur lesquels les méthodes sont en attente : les méthodes `register` et `remove` sont activées par le port `server_itf`, alors que la méthode diffuse est activée par le port `itf`. Ce tableau sera passé en paramètre lors de l'appel à l'exécutif chargé d'attendre le message d'activation.

La ligne 4 sauvegarde le label de l'instruction courante dans une valeur intermédiaire qui sera utilisée pour détecter la réception d'un message d'activation valide.

Le corps de l'instruction *select* est réalisé à l'intérieur de la boucle do { /∗...∗/} while (); . Cette boucle commence par attendre un message sur la liste de binder qui vient d'être construite grâce à la méthode `getMessage` de l'exécutif. Lors de son premier appel, celle-ci va initier les transaction avec les binders qui lui sont passés en paramètre, et retourner le premier message reçu.

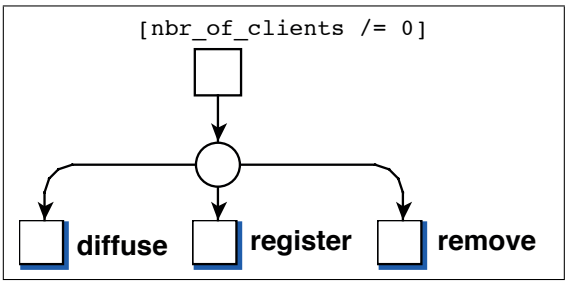
Ensuite, le contenu du message est testé : On teste si le nom de la méthode activée par le message est celui d'une méthode activable pour cette instruction[1], et que le binder sur lequel le message a été reçu est bien celui sur lequel la méthode activée était en attente.

Si le message n'est pas un message d'activation compatible avec une des méthodes activable, la valeur de `next` n'est pas changée, et une nouvelle itération de la boucle commence. Cette fois-ci, la méthode `getMessage` retourne le prochain message fourni par les binders.

Si le message est un message d'activation compatible avec une des méthodes en activables :

– le marqueur `next` reçoit le label de la transition qui exécute la méthode correspondante.
– les transactions en cours avec les ports de la classe sont terminées par l'appel de la méthode `commit` de l'exécutif, ce dernier :

[1] Si le message n'est pas un message d'activation, l'appel à `getMethodName()` retourne une chaîne vide qui peut être comparée au nom des méthodes activables



(a) Etat d'attente d'activation de la classe `group`

```
if (next == label_4) {
    LfPBinderReference[] prefix22 =
        new LfPBinderReference[] {SERVER_ITF, ITF, SERVER_ITF};
    String savedNext23 = next;
    do {
        msg = runtime.getMessage(prefix22);
        if ((msg.getMethodName().equals("REGISTER") ) &&
            (SERVER_ITF.isEqual(msg.getBinder()))) {
            next = label_1;
            runtime.commit (msg);
        }
        if ((msg.getMethodName().equals("DIFFUSE") ) &&
            (ITF.isEqual(msg.getBinder()))) {
            next = label_6;
            runtime.commit (msg);
        }
        if ((msg.getMethodName().equals("REMOVE") ) &&
            (SERVER_ITF.isEqual(msg.getBinder()))) {
            next = label_5;
            runtime.commit (msg);
        }
    } while ( savedNext23 == next );
}
```

(b) Code généré pour une instruction select

FIG. 7.16 – Instruction select et code généré correspondant

  - signifie au binder qui a fourni le message que ce dernier est accepté,
  - annules toutes les autres requêtes de lecture,
  - ré-initialise les structures de l'exécutif liées aux transactions ;
- le flot d'exécution sort de la boucle do { /*...*/} while (); car la valeur de `next` a été modifiée.

Le label associé à une méthode correspond à une transition de l'automate qui va effectuer les opérations suivantes :

- lecture des paramètres du message ;
- appel de la méthode correspondante ;
- envoi de la valeur de retour si applicable.

Une fois ces opérations effectuées, l'appel est clôturé du point de vue de la classe appelée. L'automate saute alors à l'état de sortie de la transition modélisant la méthode appelée.

7.7 Conclusion

Cette étude de cas a permis de tester le générateur de code sur un exemple de modèle de taille raisonnable pour être présenté dans ce mémoire. Il a permis de valider plusieurs points de la méthodologie vue au chapitre 3.

Le langage **L*f*P** s'est montré apte à la description de cet exemple, et a permis de le modéliser en peu de temps. De plus, le découpage entre la partie traitement des données et la partie contrôle s'est faite sans problème. L'interface entre ces deux aspects a pu être spécifiée sans problème bien qu'elle comprenne la gestion d'évènement déclenchés par les utilisateurs (envois de messages). Cela permet de valider l'approche par *types opaques* préconisée par **L*f*P**. Dans le cas de cet exemple, ils permettent de traiter :

– l'interface graphique nécessaire à un utilisateur humain ;
– le contenu des messages échangés par les utilisateurs.

A partir de la spécification **L*f*P**, le générateur de code a produit rapidement un programme exécutable. La mise au point de cette application de messagerie instantanée a été très peu coûteux en terme de développement. Seuls les composants externes ont du être développés à la main. Sur cet exemple, ils ne représente que 270 lignes de java. La partie *contrôle* de l'application représente quant à elle 1135 lignes de java produites par le générateur de code.

Cette messagerie instantanée a pu être entièrement produite en déchargeant le programmeur des aspects liés à la distribution. La totalité du code produit de manière "traditionnelle" par un codage manuel est centralisé et concerne l'interface graphique. Les aspects liés au réseau sont entièrement encapsulés par la description en **L*f*P** de la partie contrôle de l'application. Ainsi, les protocoles de communication mis en jeu par cet exemple ont entièrement été générés par le prototype de générateur de code à partir de la description **L*f*P** présentée dans ce chapitre.

La génération de code a donc permis d'accélérer le développement de l'application. De plus, une modification du protocole de communication peut être effectuée et validée sur la spécification **L*f*P** avant de générer le code correspondant, ce qui réduit le temps de développement requis pour les cycles de maintenance.

L'étude présentée dans ce chapitre, ainsi que des travaux menés dans le cadre du projet MORSE ont montré la validité de l'approche **L*f*P** pour le développement d'applications réparties. Le chapitre suivant propose un ensemble d'évolutions et de perspectives destinées à rendre l'utilisation du langage **L*f*P** plus efficace.

Chapitre 8

Perspectives et retours sur expériences

8.1 Introduction

Ce chapitre a pour objectif de montrer les résultats des expérimentations menées sur le langage **L*f*P** présenté dans ce mémoire, ainsi que sur l'implémentation et l'utilisation de la méthodologie présentée à la section 3.2.

Le projet MORSE doit proposer une chaîne d'outils permettant la mise en oeuvre de cette méthodologie, dans ce cadre, le LIP6 et le LABRI travaillent sur la vérification formelle de spécifications **L*f*P**. Les techniques utilisées sont basées sur une représentation symbolique de l'espace des états accessibles du système utilisant une représentation issue des diagrammes de décision : les DDD [17].

Parallèlement à ces travaux, les exemples de développement en **L*f*P** dont celui présenté au chapitre 7 ont permis de proposer des évolutions et améliorations du langage. On distingue deux axes principaux d'amélioration : la simplification du langage (simplification de la représentation des comportements les plus couramment rencontrés), et l'extension de son pouvoir d'expression (représentation de nouveaux comportements). Les extensions de **L*f*P** présentées dans ce chapitre sont donc principalement liées à la simplification du schéma de communication entre deux classes du modèle, et au traitement des messages par les composants. Avant l'intégration de ces extensions dans le langage, il est nécessaire de mesurer leur impact sur les étapes de la méthodologie **L*f*P**, notamment l'impact sur la vérification formelle.

Les études réalisées sur le code généré pour les exemples réalisés dans le cadre de cette thèse, mais également dans le cadre de MORSE ont fourni une première évaluation des stratégies de génération de code utilisées dans les prototypes d'outils. Elle a mené à a définition de pistes d'amélioration, principalement sur les stratégies de génération de la partie comportementale du modèle.

Ce chapitre est donc structuré de la manière suivante : la section 8.2 présente les travaux menés dans le cadre du projet MORSE sur la vérification de spécifications écrites en **L*f*P** ; la section 8.3 présente les possibilités d'évolution du langage **L*f*P** ; la section 8.4 présente les possibilités d'amélioration des stratégies de génération de code. Enfin la section 8.5 conclura ce chapitre en rappelant les principales évolutions envisagées pour le langage **L*f*P**, mais également pour les outils qui lui sont associés.

8.2 Vérification formelle de spécifications L*f*P

Les principaux travaux de recherche sur la vérification formelle du langage **L*f*P** sont effectués en collaboration entre le LaBRI et le LIP6 autour des techniques de *model checking* développées

par ces deux laboratoires. Les techniques utilisées sont basées sur un format d'arbre de décision : les DDD [17] qui fournissent une représentation compacte de l'espace d'état du système.

Un DDD est un arbre de décision pour lequel chaque variable peut prendre une valeur entière dans un domaine fini mais non connu à l'avance. Ils fournissent une représentation compacte de l'espace d'état grâce au partage des sous-arbres communs, et rapide à manipuler grâce au cache d'opération qui permet de n'effectuer qu'une fois les opérations à réaliser sur les sous arbres partagés.

Les opérations ensemblistes classiques (union, intersection, etc.) sont définies sur les DDD. Ils implémentent les opérations correspondantes sur les espaces d'états représentés. Pour le calcul de l'espace des états accessibles d'un modèle, les DDD's sont manipulés à l'aide d'homomorphismes appliqués sur tous les chemins de la structure, depuis la racine vers les noeuds terminaux. De manière schématique, chaque instruction de la spécification modélisées doit être implémentée sous la forme d'un homomorphisme sur un DDD représentant un ensemble d'états du système.

Cette approche permet donc de construire en un seul parcours de la structure de donnée l'ensemble des états successeurs de l'espace d'états représentés par le DDD sur lequel l'homomorphisme est appliqué. L'ensemble des états produits par tous les franchissements possibles d'une transition depuis un ensemble d'état de départ peut donc être calculé en un seul parcours de la structure.

Ce nouvel ensemble d'états successeurs peut ensuite être ajouté (opération d'union) à l'état calculé précédemment. L'ensemble des états accessibles du système a été atteint lorsque plus aucune transition ne produit de nouveaux états, c'est à dire lorsque l'union de l'ensemble d'états n'est plus modifié par l'union avec les états produits par le franchissement des transitions.

Deux étapes sont donc nécessaires pour réaliser la vérification de spécifications **L*f*P** à l'aide de DDD :

- choix d'une structure de représentation d'un état du modèle ;
- définition des homomorphismes permettant de calculer les états successeurs d'une transition.

La structure du codage d'un état est définie pour le langage à l'aide de templates pour chaque sorte de type défini pour le langage. Le template est instancié pour chaque déclaration de type réalisée par la spécification. Les homomorphismes utilisés pour manipuler la représentation ainsi obtenus doivent être définis pour chaque spécification en fonction des instructions portées par les transitions du système.

Les travaux de vérification formelle pour le langage **L*f*P** menés dans le cadre de MORSE sont détaillés dans [23] et [48]. Ils montrent qu'il est possible de mener à bien la vérification de propriétés pour des modèles **L*f*P**. Ces travaux permet d'envisager de disposer rapidement d'un outil de vérification formelle de spécification **L*f*P**. Actuellement il est possible de vérifier des propriétés d'accessibilité sur le modèle ; des travaux sont en cours pour étendre les propriétés vérifiables aux formules de logique LTL.

Par ailleurs, des travaux indépendants de MORSE envisagent l'utilisation d'autres techniques formelles pour la vérification de modèles **L*f*P**.

8.3 Évolutions du langage L*f*P

Cette section propose un ensemble d'évolutions possibles pour le langage **L*f*P**. Ce langage permet de modéliser une vaste classe de systèmes répartis. Néanmoins, reste possible d'améliorer le pouvoir d'expression du langage en intégrant quelques instructions supplémentaires, ou de simplifier l'écriture de certaines spécifications.

8.3.1 Liaisons entre les composants

Actuellement, le lien entre deux composants applicatifs est réalisé par une combinaison binder / média dont le pouvoir d'expression est extrêmement puissant pour représenter des schémas de communication très complexes. En revanche, dans le cas de communications simples (de type lien fifo par exemple), ce mécanisme est relativement lourd et induit des recopies de messages pouvant être évitées.

Dans les cas simples, ce mécanisme pourrait avantageusement être remplacé par une liaison directe entre les deux classes (canal de messages). Afin de ne pas modifier la sémantique actuelle du langage, il est nécessaire d'introduire un nouveau type de donnée appelé *channel* pour représenter les canaux directs entre les classes.

Un canal représente une file de messages bi-directionnelle et asymétrique. Il est en fait composé de deux files de messages : la première permet de transmettre les messages de données, la deuxième file est dédiée aux éventuels messages de retour des fonctions appelées ou les valeurs mises à jour des paramètres en mode `out` et `inout` des procédures synchrones.

Par défaut, un canal est associé à une classe (comme un binder), mais on distinguera également les canaux statiques partagés entre plusieurs classes.

Définition des canaux

Un canal est défini par quatre attributs :

- *liaison* qui définit le port de la classe auquel est attaché le canal ;
- *ordonnancement* avec deux valeurs possibles : `bag` et `fifo` définissant l'ordonnancement des messages dans le canal ;
- *fiabilité* de type booléen définissant si le lien est fiable (pas de perte de message) ou non ;
- *capacité* en nombre de message.

L'attribut *liaison* définit le composant auquel appartient le canal. Un composant est responsable de l'instanciation de tous les canaux qui lui appartiennent.

L'attribut *ordonnancement* définit la manière dont les messages sont traités dans le *channel*. Dans le cas d'un *channel* `fifo`, les messages sont délivrés dans l'ordre de leur arrivée dans le canal. Dans le cas d'un canal `bag`, les messages peuvent être lus dans n'importe quel ordre.

L'attribut *fiabilité* définit la qualité de service fournie par le *channel*. Dans le cas d'un canal non fiable, tout message envoyé est susceptible d'être perdu. Cette politique est simple, mais elle permet de modéliser ou d'englober simplement le comportement de nombreux types de liens de communication.

Un canal peut être utilisé pour communiquer avec le composant auquel il appartient. Tout composant relié avec ce canal par la description de l'architecture peut l'utiliser. Ce lien est apparenté à celui existant entre un média et le binder d'une classe ; il est indépendant de l'attribut *liaison*. Ainsi, un composant peut ne pas être relié à un canal lui appartenant, ce qui interdit les communications entre les différentes instances.

Définition des canaux statiques

Les canaux statiques sont des files de messages partagées par plusieurs instances de classes. Tout composant relié à un canal statique sur le diagramme d'architecture peut y lire ou écrire un message.

Un canal statique est défini par trois attributs : *ordonnancement*, *fiabilité* et *capacité* dont la définition est la même que pour les canaux.

Instructions d'accès aux canaux

Un canal de communication dispose d'une interface proche de celle des binders déjà définis dans le langage **L*f*P**. Néanmoins, la structure des messages et leur traitement est légèrement différent.

Un message envoyé via un canal contient systématiquement les informations suivantes :
- *émetteur* : référence de l'émetteur du message ;
- *destinataire* : désignation du destinataire du message ;
- *type* : type du message envoyé ;
- *données* : corps du message.

Le champs *émetteur* du message désigne de manière unique le composant qui a envoyé ce message dans le canal. Le champs *destinataire* du message désigne le ou les composants susceptibles de traiter ce message. Ce champs peut contenir :
- la référence explicite d'une instance de composant ;
- la définition d'un type de composant.

Dans le premier cas, seul le composant spécifier peut consommer le message, dans le deuxième cas, tout composant du type spécifier peut consommer le message. Si le champs *destinataire* est laissé vide, tout composant relié au canal peut consommer message.

Le champs *type* du message détermine la manière dont celui-ci peut être traité ; il peut prendre trois valeurs selon que le message est un message de données, un message d'activation de méthode, ou un message de retour. Enfin la partie *données* du message contient les données qui seront exploitées par le composant.

La sémantique des opérations d'envoi de message de données est la même que pour les opérations d'ajout de messages dans les binders synchrones (ces opérations sont bloquantes). La file utilisée est celle dédiée au type de message envoyé. Le champs *émetteur* d'un message d'activation de méthode sera utilisé automatiquement pour définir le champs *destinataire* du message de retour correspondant. Un message de retour est donc toujours adressé à l'instance de composant qui a émis le message d'activation correspondant.

Extension des types canaux

Il serait très intéressant de laisser à l'utilisateur la possibilité de surcharger les opérations d'accès aux canaux. Cette extension permet d'obtenir des files de messages susceptibles de définir leur comportement en fonction des messages reçus.

On peut définir au moins deux cas d'utilisations :
- définir des priorités (statiques ou dynamiques) entre les messages déposés dans la file ;
- dans le cas des canaux statiques, routage des messages dont les destinataires ne sont pas spécifiés statiquement.

Le pouvoir d'expression d'un canal dont les opérations natives ont été surchargées est presque identique à celui d'un média. De plus, elle fournit des fonctionnalités qui n'étaient pas disponibles en **L*f*P** : actuellement, il est impossible de réorganiser le contenu d'un binder, et les mécanismes de choix du prochain message sont relativement basiques (gardes dans les médias, ou nom de la prochaine méthode à activer dans les classes).

De plus, comme les opérations sont effectuées directement sur la file de message du composant, il est possible de fournir une implémentation très efficace de ce mécanisme. Par rapport à l'instanciation d'un média, on gagne :
- la lecture du message par le média (aspect transactionnel coûteux) ;
- l'instanciation d'un composant **L*f*P** ;

– la possibilité de travailler sur les messages déjà transmis mais pas encore pris en compte par le composant cible.

Dans le cadre d'un langage objet, il sera possible d'implémenter ces canaux par extension d'une classe existante implémentant le comportement de base, ce qui permet d'envisager une implémentation relativement aisée et très efficace des instructions spécifiées par le modélisateur.

8.3.2 Système de traitement des exceptions

Les exceptions sont un mécanisme de spécification très puissant permettant de spécifier simplement la gestion des erreurs, ou d'introduire des discontinuités dans l'exécution d'un composant. Elles peuvent être facilement rajoutées à la sémantique actuelle du langage **L*f*P** en utilisant ses mécanismes de structuration actuels. Un sous diagramme peut être dédié à la représentation de la clause *try* (bloc susceptible de lever une exception), il suffit alors de lui adjoindre un ensemble de blocs de sortie dédiés au traitement de chaque exception susceptible d'être levée.

Dans le cas d'une exception levée pendant un appel de méthode entre composants, on peut envisager un mécanisme de levée d'exception distante. Ce mécanisme nécessite que l'appel de méthode doit bloquant (procédures synchrones ou fonctions) pour que l'appelant soit en attente d'un message de retour qui sera alors utilisé pour transmettre l'exception.

Les exceptions sont supportées directement par un très grand nombre de langages de programmation, et ne poseront donc pas de problème crucial pour la génération de code.

8.3.3 Ajout de temporisation pour le traitement des messages

Les instructions d'envoi et de réception de messages **L*f*P** sont très complètes. Le principal manque du langage se situe dans la gestion des pertes de messages. Plus précisément, le langage **L*f*P** ne définit pas de moyens de réagir à l'absence de messages.

Un moyen efficace de parer à cette lacune est de fournir un mécanisme de *timer* sur les opérations de lectures de messages. Ces instructions resteraient alors bloquantes uniquement pendant le temps défini par le timer. Une fois le timer expiré, si aucun message n'a été lu, le composant reprend son exécutions.

Ce mécanisme de traitement des timers peut par exemple être intégré de au langage en utilisant des exceptions : lorsque le timer arrive à échéance et qu'aucun message n'a pu être lu, le timer lève une exception. Le bloc de traitement de l'exception définissant alors les traitements à effectuer en cas d'absence de messages.

Cette capacité à répondre à l'absence de message est très importante dans le cas de la modélisation d'applications fonctionnant avec des liens non-fiables (liaisons hertziennes par exemple, mais aussi réseaux *best effort*).

En terme de génération de code, elle ne pose pas de problème d'implémentation et de génération de code. En revanche, en terme de vérification, elle introduit une complexité supplémentaire en terme de nombre d'états du système. Lorsqu'une opération de lecture est lancée et qu'aucun message ne peut être lu, on doit considérer le cas où le timer parvient à échéance. Dans les versions précédentes du langage, le composant restait systématiquement bloqué en attente de lecture jusqu'à l'arrivée d'un hypothétique message. Néanmoins cette modification correspond à une forte augmentation du pouvoir d'expression du langage et est nécessaire pour la modélisation d'applications communiquant par des canaux non fiables.

8.3.4 Gestion des ports d'activation des méthodes

Actuellement, la gestion des ports d'activation des méthodes est très statique : la déclaration de la méthode définit l'unique port sur lequel la méthode sera activable. Ce mécanisme est relati-

vement statique et ne permet donc pas de prendre en compte l'état courant pour l'activation d'une méthode.

Une gestion plus dynamique des binders serait intéressante, et permettrait de limiter le nombre de binders. Dans ce cadre, on peut envisager l'évolution suivante :

- le port d'activation défini lors de la déclaration de la méthode devient le port d'activation par défaut et cette déclaration devient facultative ;
- si un port est spécifié lors de la lecture du message d'activation, le port par défaut est ignoré ;
- si aucun port n'est spécifié lors de la lecture du message, le port d'activation est utilisé ;

Dans le cas où le port d'activation par défaut n'est pas défini et qu'aucun port d'activation n'est spécifié lors de la lecture du message, il s'agit d'une erreur de syntaxe. Cette erreur peut être détectée statiquement lors de la compilation de la spécification **L*f*P**.

Ce mécanisme reste compatible avec la sémantique actuelle du langage et ne modifiera pas le comportement des spécifications existantes.

8.3.5 Syntaxe du langage L*f*P

La syntaxe actuelle du langage **L*f*P** est massivement graphique. Celle-ci présente l'avantage d'être très lisible et permet de suivre intuitivement le flot d'exécution de chaque composant. Néanmoins, elle est peu adaptée dans le cadre d'une spécification générée. En effet, pour qu'une spécification générée soit exploitable, la syntaxe graphique impose :

- la définition automatique du placement des entités graphiques ;
- la génération du format de représentation des graphiques.

Pour ces raisons, et dans le cadre du projet MORSE, il a été décidé de donner une syntaxe textuelle à **L*f*P**. Cette syntaxe servira de format d'échange entre les partenaires du projet MORSE. Elle est détaillée par l'annexe B. Le jeu d'instructions ainsi que la sémantique du langage sous-jacent restent les mêmes que ceux détaillés au chapitre 3. La syntaxe proposée intègre une partie des modifications du langage **L*f*P** que nous venons d'étudier, notamment :

- la gestion des ports par défaut ;
- les canaux (type *channel*).

Ces deux modifications nous semblent prioritaires car elles permettent de simplifier l'écriture des spécifications **L*f*P** les plus courantes et peuvent donc être applicables dans le cadre du projet MORSE.

8.4 Optimisation du code généré

Cette section est dédiée à l'amélioration du code généré pour les modèles **L*f*P**. L'implémentation actuelle du générateur de code vise à produire du code correct d'un point de vue fonctionnel, c'est à dire dont le comportement est identique au comportement de la spécification **L*f*P**. En revanche, les aspects liés à l'optimisation des performances n'ont pas été abordés lors de l'implémentation.

Lorsque les automates des diagrammes de comportement ont une structure spécifique, il est possible d'optimiser le code généré. L'exemple le plus simple est donné par la figure 8.1 qui rappelle le diagramme de comportement du média `RPC` de l'étude de cas. Ce dernier est une séquence de transitions exécutées les unes après les autres sans possibilité de branchement. Dans ce cas, l'implémentation actuelle du générateur de code va produire un ensemble d'instructions `goto` inutiles. De plus, la traduction de cette instruction en java étant particulièrement coûteuse, le code produit est peu efficace par rapport à un code écrit "à la main". Ce problème est beaucoup moins sensible si le langage cible propose une instruction `goto` native.

Il est cependant nécessaire de revoir les règles de transformations pour optimiser le code produit dans cette configuration qui se retrouve souvent dans les spécifications **L*f*P**. Elle est par exemple également présente dans le diagramme de comportement de la classe sender (figure 7.6 page 137) : les deux branches de gauche sont également des suites de transitions.

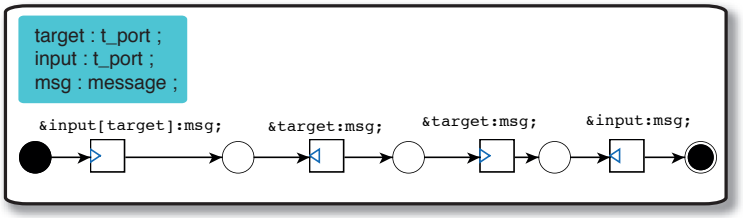


FIG. 8.1 – Exemple d'automate de comportement de type *suite de transition*

La figure 8.2 permet de se rendre compte des différences entre le code produit automatiquement par l'implémentation actuelle du générateur (8.2(a)) et le code optimisé (8.2(b)).

Le code optimisé peut être généré automatiquement à condition de détecter les séquences de transitions correspondantes et de les fusionner. Cette détection est basée sur les états de la spécification : on peut fusionner deux transitions reliées par un état qui n'a qu'un arc entrant et un arc sortant. Cette détection peut être implémentée au niveau du générateur de code et permet d'éviter les instructions `goto` inutiles.

Enfin, on peut noter que cette optimisation est surtout importante en java qui ne dispose pas d'une véritable instruction `goto`. L'amélioration en terme de performance sera beaucoup moins sensible dans un langage tel que `C++`. En revanche, le gain en terme de lisibilité restera le même. En effet, la structure du code optimisé est beaucoup plus simple et lisible que celle du code actuellement produit.

8.5 Conclusion

Le projet MORSE a fourni une première implémentation de la méthodologie développée autour du langage **L*f*P**. Les travaux actuels permettent d'envisager de disposer rapidement d'un outil permettant de vérifier des formules LTL sur un modèle **L*f*P**. Parallèlement, un outil de traduction de spécifications UML vers **L*f*P** est en cours de construction.

Dans le cadre de ce projet, plusieurs études témoins ont montré que le langage peut **L*f*P** être amélioré pour remplir plus efficacement son rôle. Deux axes principaux ont été définis dans le cadre du projet MORSE : la facilité d'utilisation, et le pouvoir d'expression de la partie comportementale. Dans ce chapitre, nous avons vu plusieurs extensions possibles pour ce langage. Elles concernent les points suivants :

- la liaison entre les classes du modèle ;
- la gestion des exceptions ;
- les timers sur les réceptions de messages.

Toutes ces évolutions sont compatibles avec la sémantique actuelle du langage **L*f*P**, ce qui est un gage de pérennité pour les modèles déjà développés. Le premier point est intégré dans le langage dans le cadre de MORSE. Il permet de gérer les connexions directes entre les classes du modèle lorsque les protocoles d'interaction correspondent à des schémas connus. L'objectif de cette modification du langage est de simplifier l'écriture des modèles en n'utilisant le mécanisme des médias que lorsqu'ils sont réellement nécessaires.

Avant d'intégrer chaque modification, il faut considérer principalement son impact sur la vérification formelle des spécifications. Dans le cas des liens entre les classes, l'impact est positif car

```
public class RPC extends LfPMedia implements Runnable {
    LfPBinderReference TARGET = null ;
    LfPBinderReference INPUT = null ;
    LfPMessage MSG = new LfPMessage() ;
    /* This class contains no method */
    /* This class contains no trigger */
    /*
     * main method of the class
     */
    public void run ()
    {
        try{
            LfPMessage msg = null ;
            String next ;
            String label_104 = "+RPC+ 10 10";
            String label_105 = "+RPC+ 9 9";
            String label_106 = "+RPC+ 8 8";
            String label_107 = "+RPC+ 7 7";
            String label_108 = "+RPC+ 6 6";
            String label_109 = "+RPC+ 5 5";
            String label_110 = "+RPC+ 4 4";
            String label_111 = "+RPC+ 3 3";
            String label_112 = "+RPC+ 2 2";
            /* sub_diagram for  */
            next  = label_112 ;
            while (true) {
                if (next == label_104) {
                    next = label_108;
                }
                if (next == label_105) {
                    next = label_109;
                }
                if (next == label_106) {
                    next = label_110;
                }
                if (next == label_107) {
                    break ;
                }
                if (next == label_108) {
                    MSG.setBinder(INPUT) ;
                    runtime.sendMessage(MSG) ;
                    next = label_107;
                }
                if (next == label_109) {
                    MSG = runtime.getMessage(TARGET) ;
                    runtime.commit (msg) ;
                    next = label_104;
                }
                if (next == label_110) {
                    MSG.setBinder(TARGET) ;
                    runtime.sendMessage(MSG) ;
                    next = label_105;
                }
                if (next == label_111) {
                    MSG = runtime.getMessage(INPUT) ;
                    TARGET = (LfPBinderReference)
                                MSG.getDiscriminant( 1);
                    runtime.commit (msg) ;
                    next = label_106;
                }
                if (next == label_112) {
                    next = label_111;
                }
            }
        }catch (Exception e) {
            e.printStackTrace() ;
            System.exit(1);
        }
        try {
            runtime.removeComponant(this.selfReference) ;
        }
        catch (LfPRuntimeError lrExcp) {
            lrExcp.printStackTrace() ;
        }
    }
}
```

(a) Code généré automatiquement pour le média RPC

```
public class RPC extends LfPMedia implements Runnable {
    LfPBinderReference TARGET = null ;
    LfPBinderReference INPUT = null ;
    LfPMessage MSG = new LfPMessage() ;
    /* This class contains no method */
    /* This class contains no trigger */
    /*
     * main method of the class
     */
    public void run ()
    {
        try{
            /* Lecture du message dans le binder INPUT */
            MSG = runtime.getMessage(INPUT) ;
            TARGET = (LfPBinderReference) MSG.getDiscriminant( 1);
            runtime.commit (msg) ;

            /* Envoi du message dans le binder TARGET */
            MSG.setBinder(TARGET) ;
            runtime.sendMessage(MSG) ;


            /* Lecture du message de retour */
            MSG = runtime.getMessage(TARGET) ;
            runtime.commit (msg) ;

            /* Envoi du message de retour dans le binder INPUT */
            MSG.setBinder(INPUT) ;
            runtime.sendMessage(MSG) ;

        }catch (Exception e) {
            e.printStackTrace() ;
            System.exit(1);
        }
        try {
            runtime.removeComponant(this.selfReference) ;
        }
        catch (LfPRuntimeError lrExcp) {
            lrExcp.printStackTrace() ;
        }
    }
}
```

(b) Code optimisé pour le média RPC

FIG. 8.2 – Deux implémentations du média RPC

il permet de réduire le nombre d'états de la spécification pour des schémas de communication les plus courants. En revanche, les autres ajouts du langage doivent être validés afin de déterminer si elles sont compatibles avec un processus de vérification formelle efficace.

Du point de vue de l'implémentation des modèles, il est nécessaire d'améliorer les règles de génération afin d'obtenir un code de meilleure qualité et de réduire le surcoût lié à la génération de code. Cette optimisation peut être réalisée en analysant de manière beaucoup plus fine la structure des automates **L*f*P**. Cette analyse permet de structurer le code produit, et d'éviter les instructions de saut inutiles. Cette optimisation est particulièrement importante pour les cibles telles que le langage java qui ne définissent pas d'instruction de saut (`goto`).

Enfin, on peut constater qu'une grande partie du surcoût de la génération de code provient de

manques d'optimisation du générateur, et pas de la structure du langage, ce qui permet la production d'un générateur de code plus efficace dans le cadre du projet MORSE. Celui-ci reprend les règles principales de générations définies dans ce mémoire, et les applique au langage `C++`, en améliorant également la prise en compte de la structuration de l'automate.

Chapitre 9

Conclusion Générale

Nous nous sommes intéressés dans ce travail au développement d'applications réparties à large échelle dans un environnement asynchrones. L'objectif principal de nos travaux est d'améliorer la fiabilité de ce type d'application en intégrant dans le cycle de développement une méthodologie de prototypage basée sur les méthodes formelles.

Notre approche est basée sur la modélisation. Le modèle de l'application est principalement utilisé de deux manières différentes : la vérification formelle et la génération de code. La première permet de vérifier que la solution choisie respecte les contraintes fonctionnelles définies dans la spécification. La seconde produit un prototype fonctionnel exécutable dans l'environnement cible de l'application. Pour réaliser ces modèles, nous proposons **L*f*P**, un langage de modélisation basé sur les automates communiquants par file de messages. Le langage est orienté vers la modélisation de l'aspect contrôle de l'application qui doit gérer la complexité induite par l'environnement réparti.

Nous visons l'intégration de notre approche de développement dans un standard industriel : UML. Pour cela, nous proposons une méthodologie de développement encadrant l'utilisation du langage **L*f*P** dans un profil UML adapté à la classe d'application visée. la méthodologie **L*f*P** est basée sur les techniques de prototypage. Elle est centrée sur trois opérations principales :

- l'obtention du modèle **L*f*P** de l'application à partir de la spécification UML ;
- la vérification automatisée des propriétés fonctionnelles attendues de l'application par *model-checking* ;
- la génération automatique du code de l'application et son déploiement sur l'architecture d'exécution cible.

Nos principales contributions à ces travaux sont :

- la définition du langage **L*f*P** à partir d'une notation existante mais insuffisamment formalisée ;
- la définition de la méthodologie de développement associée ;
- la définition des méthodes de déploiement et de génération automatique de code pour ce langage.

La sémantique et la syntaxe du langage **L*f*P** ont été définie de manière suffisamment précise pour permettre l'exploitation des spécifications à la fois pour la vérification formelle et pour la génération de code. La méthodologie **L*f*P** permet d'intégrer ces techniques dans standard reconnu de l'industrie, ce qui est une condition nécessaire pour leur assurer une diffusion significative. Enfin, nous avons défini les règles de transformation permettant de déployer et d'implémenter l'application à partir de son modèle réalisé en **L*f*P**.

Un prototype fonctionnel de l'outil de génération de code a été réalisé. Celui-ci a constitué une étape importante du projet MORSE. Avec la définition du langage **L*f*P**, l'implémentation

de l'outil de génération de code constitue notre principale contribution à l'implémentation de la méthodologie.

9.1 Apports méthodologiques

La définition d'une méthodologie de développement adresse le troisième problème identifié dans l'introduction générale : la nécessité de disposer d'un guide de développement permettant de maîtriser la complexité du développement d'un système réparti de grande taille.

Notre méthodologie centrée sur la modélisation s'intègre dans la mouvance MDA. Un des objectif de l'OMG avec cette approche est d'intégrer les modèles dans le processus de développement. Notre méthodologie fait des choix techniques autres que ceux recommandés par l'OMG qui prône l'utilisation d'UML à toutes les étapes du MDA. Néanmoins, notre approche de l'utilisation des modèles reste sensiblement la même.

Notre choix de **L*f*P** se justifie par les caractéristiques des applications que nous ciblons, ainsi que par la volonté d'intégrer des techniques de vérification formelles. Le principal attrait de cette approche est d'offrir un pont entre des techniques formelles et un environnement industriel standard.

Nous pensons qu'il est possible de favoriser le déploiement des approches formelles dans l'industrie à condition de masquer leur complexité derrière un langage connu et maîtrisé par les ingénieurs. Notre méthodologie doit apporter un guide d'utilisation de cet environnement pour permettre l'exploitation des spécifications obtenues par nos outils.

Notre méthodologie permet donc de combiner la puissance d'expression d'UML avec un niveau de formalisme suffisant pour appliquer des techniques de vérification formelles sur les spécifications. Bien que des progrès aient été réalisés ces dernières années [72][74], les techniques basées sur les prouveurs de théorèmes telles que [83] nécessitent encore l'intervention d'utilisateurs experts pour démontrer les propriétés attendues d'un système. Nous préférons donc l'utilisation de techniques de *model checking* qui permettent de vérifier automatiquement une propriété exprimée sur le modèle.

Les technologies utilisées pour implémenter la méthodologie **L*f*P** permettent d'adresser les deux points suivants énoncés dans l'introduction générale :

Protocoles de communications entre les composants de l'application. La méthodologie **L*f*P** sépare dès la spécification de haut niveau les aspects de contrôle de l'application des aspects métiers. Elle fournit un langage et des outils pour traiter spécifiquement cette partie contrôle tout en gardant un lien avec les composants nécessaires pour implémenter les aspects métiers pour assurer la cohérence des analyses.

La vérification formelle et la génération de code permettent de détecter très tôt dans le cycle de développement les erreurs de conception. Des mesures correctives peuvent alors être appliquées jusqu'à l'obtention d'un modèle correct. La génération de code permet également de s'assurer que l'application produite respecte bien l'ensemble des propriétés vérifiées sur le modèle.

9.2 Définition du langage L*f*P

L*f*P est un langage de modélisation dont l'objectif est la modélisation de la partie *contrôle* d'une application répartie. L'objectif du langage est de fournir un ensemble d'opérations permettant de modéliser de manière suffisamment précise le comportement de l'application pour permettre la génération automatique de code.

Cette contrainte est antinomique avec la vérification formelle qui nécessite un niveau d'abstraction aussi élevé que possible pour gagner en efficacité et lutter contre l'explosion combinatoire du nombre des états accessibles.

L*f*P est donc un compromis permettant d'assurer ces deux fonctions avec la même efficacité. Pour cela, les types de données manipulables par le langage ont été limités. Lorsqu'une opération ne peut pas être implémentée, elle doit être représentée par une méthode externe. Pour cela, **L*f*P** propose la notion de type opaque qui représente une donnée accessible uniquement à l'aide d'une interface et dont la valeur ne fait pas partie de la représentation de l'état du système. Néanmoins, les instructions définies par le langage sont suffisantes pour représenter les interactions entre les composants définissant la partie *contrôle* de l'application.

Afin de garantir la fiabilité de l'application produite, notre méthodologie utilise deux types de techniques complémentaires : la vérification formelle et la génération de code. La vérification formelle permet de vérifier que le modèle choisi respecte les contraintes fonctionnelles définies pour l'application. La génération de code permet :

- d'assurer la continuité dans le processus de développement de l'application ;
- de réduire les délais de production d'un prototype fonctionnel de l'application ;
- de systématiser le processus de déploiement de l'application.

Ces caractéristiques permettent d'améliorer le niveau de fiabilité de l'application tout en réduisant le temps de développement.

9.3 La génération de code

On distingue plusieurs objectifs distincts pour la génération de code :

- maquettage ;
- simulation du fonctionnement de l'application ;
- utilisation en environnement de production.

Les deux premières options sont surtout utiles en phase de mise au point pour s'assurer que les options de développement choisies sont appropriées. La troisième option est la plus intéressante : le code généré est alors directement intégré dans l'application finale. C'est cette option que nous avons développé dans ce mémoire. Notre objectif est en effet d'obtenir à faible coût un prototype exécutable de l'application en évitant la traditionnelle fracture entre la spécification de l'application et son implémentation.

Nous avons envisagé la génération de code comme une transformation de modèle. Celle-ci est réalisée en plusieurs étapes permettant de gérer efficacement le saut sémantique entre le code généré et le modèle de départ. Dans le cas de la génération de code pour **L*f*P**, seules deux étapes ont été nécessaires : une étape sémantique permet de construire un arbre de syntaxe abstraite du programme généré. Les instructions utilisées correspondent aux opérations de bases définies par la plate-forme cible et un sous ensemble des instructions des langages de programmation courants.

L'étape de traduction sémantique est de loin la plus difficile et la plus importante du processus de génération de code. Celle-ci produit donc une représentation structurée, dépendante de la spécification de la plate-forme cible mais indépendante du langage cible. En effet, le modèle produit par les règles de transformation sémantique dépend des caractéristiques de l'exécutif sur lequel le modèle sera déployé. En revanche, le langage utilisé pour l'implémentation n'est pas fixé. Les instructions produites peuvent être implémentées en utilisant la plupart des langages impératifs courants. Cette représentation définit donc un PSM au sens MDA.

La deuxième étape de la transformation de modèle est syntaxique, elle consiste à définir la syntaxe concrète des instructions. Chacune des instruction présentes dans le PSM est donc implémentée dans le langage cible. La plate-forme de déploiement est définie par l'exécutif associé au

générateur de code. Celui-ci fournit un environnement d'exécution constant nécessaire à l'exécution du code généré.

Un des avantages de cette approche est de distinguer les aspects liés au langage cible de ceux liés à la nature de la plate-forme de déploiement choisie. Cet aspect est très important car il permet de changer le langage cible sans devoir modifier les règles sémantiques. Celles-ci dépendent en effet principalement de la structure de l'exécutif sur lequel le code produit est déployé.

9.4 Perspectives

Le travail que nous avons mené est actuellement en phase d'industrialisation dans le cadre du projet MORSE. Dans ce contexte, un profil UML pour **L*f*P** est en cours de développement par la société Aonix, ainsi qu'un outil de traduction des spécifications UML vers le langage **L*f*P**. Parallèlement, le LaBRI travaille sur l'outil de vérification formelle permettant d'appliquer des techniques de model checking sur les spécifications ainsi obtenues. Afin de valider cette approche, une étude industrielle sera menée par SAGEM la plate-forme de drones commercialisée par cette société.

La prise en compte des contraintes temporelles . Les techniques actuelles permettent d'améliorer la maîtrise des aspects comportementaux des applications réparties. Elles permettent de s'assurer que l'application produit les résultats attendus de manière déterministe. La prochaine étape est de permettre de spécifier et de vérifier des contraintes sur les délais d'exécution de l'application. L'objectif est de traiter des problèmes liés à l'analyse de performance et au temps réel. La prise en compte des aspects temporels doit être intégrée à l'ensemble des outils de la méthodologie **L*f*P**.

Vers un déploiement plus flexible. Le langage **L*f*P** reste relativement limité en terme de déploiement notamment pour les systèmes dont la liste des participants n'est pas connue au moment de la mise en route de l'application. Dans ce cas, seule la structure des exécutables peut être définie. Il est nécessaire d'expérimenter d'autres politiques de déploiement, notamment pour le placement des instances créées dynamiquement. Enfin certaines propriétés non fonctionnelles très importantes dans le cas des applications réparties telles que la sûreté de fonctionnement, la persistance ou la sécurité peuvent être définies sous forme de contraintes de déploiement. Le code nécessaire pour implémenter ces fonctionnalités relativement communes peut alors être généré automatiquement par un outil approprié.

Intégration de l'exécutif avec un intergiciel de plus haut niveau. Pour cela, il faut intégrer l'exécutif **L*f*P** avec un intergiciel de plus haut niveau. L'objectif est d'intégrer des composants développés avec la méthodologie **L*f*P** dans des environnements standards. Cela nécessite d'exporter l'interface des composants **L*f*P** pour que les autres composants puissent y accéder. Cela est nécessaire pour l'intégration d'applications développées à l'aide de la méthodologie **L*f*P** dans des environnements standards préexistants.

Meilleure utilisation des aspects dynamique d'UML . Les nouveautés apportées par la dernière version de la norme UML affinent fortement la sémantique des aspects dynamique du langage UML. Il devient donc possible d'analyser plus finement les aspects comportementaux d'une spécification UML. Dans ce contexte, **L*f*P** est une étape permettant de travailler sur des langages de plus haut niveau tout en conservant un lien avec la vérification formelle. Lors de la prochaine

étape, nous serons peut être en mesure d'utiliser directement des profils UML dédias. Cela supposerait qu'UML soit défini de manière suffisamment formelle et que les technologies de vérifications passent suffisamment à l'échelle pour traiter des modèles de grande taille. L'étude de cas réalisée à partir d'octobre 2005 dans le cadre de MORSE fournira des informations précises sur la manière d'envisager cette évolution.

Annexe A

BNF du langage L*f*P sous sa forme graphique

A.1 Présentation

Cette annexe présente la syntaxe des attributs textuels des éléments graphiques du langage **L*f*P**. Après les conventions de présentation énumérées dans la section A.2, ce document énumère les règles du langage **L*f*P** proprement dit dans les sections A.3 à A.5. La section A.3 énumère les éléments disponibles dans tous les diagrammes **L*f*P** ; la section A.4 présente les attributs des composants du diagramme d'architecture ; la section A.5 présente les attributs des éléments présents dans les diagrammes de comportement.

A.2 Conventions

Les conventions de présentation de la syntaxe sont les suivantes :

- les mots clés seront écrits en rouge, et en minuscule ;
- les règles de syntaxe non terminales sont écrites en pourpre, et en minuscules ;
- les règles terminales du parseur sont écrites en bleu et en majuscules ;
- les caractères faisant partie de la syntaxe du langage **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**.

A.3 Éléments communs à tous les diagrammes

A.3.1 Eléments de base

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

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.

Identificateurs valides

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

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

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

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 A.2.

$$INTEGER : [0-9] +$$

FIG. A.2 – Lexème définissant un nombre

Composant

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

A.3.2 Syntaxe des expressions

Cette section présente les opérateurs disponibles pour les expressions. Une expression définit tout élément lfp définissant une valeur de retour lors de son exécution, ce qui inclut les expressions arithmétiques, les variables, les appels de méthode etc...
Pour écrire les expressions, on dispose des opérateurs suivants :

1. `< > <= >= =` représentent respectivement pour les éléments d'un intervalle 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 A.3 et leur signification est explicitée ci-dessous (la règle `operator` désigne l'un des opérateurs binaire évoqué ci-dessus).

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

expr_variable : expr_variable . IDENTIFIER                           (A.12)
              | expr_variable ( expression [ , expression ] )        (A.13)
              | IDENTIFIER                                           (A.14)
```

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

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

La règle A.2 définit une expression entre parenthèses.

La règle A.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 le multi-ensemble afin de déterminer combien d'éléments semblables celui-ci contient.

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

La règle A.5 correspond à une valeur entière.

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

La règle A.7 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 A.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 A.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.

La règle A.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

[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.

l'attribut ; si nécessaire, une expression entre parenthèse spécifie l'expression sur laquelle l'attribut est appliqué.

La règle A.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 A.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 A.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 A.14 désigne le nom d'une variable.

Exemple 14 *Exemples d'expressions valides :*

objet@methode(3) ; —— *appel d'une méthode d'un composant externe*
tab(5).champs1 ; —— *accès au champs d'une structure stockée dans la* —— 5^{eme} *case d'un tableau*

A.3.3 Syntaxe des déclarations de types

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 A.4) ;
- les types range définis par énumération (règle 2 de la figure A.4) ;
- les types tableau (règle 3 de la figure A.4) ;
- les types ensemble (règle 4 de la figure A.4) ;
- les types multi-ensembles (règle 5 de la figure A.4) ;
- les types "record" (règle 6 de la figure A.4) ;
- les types "port" (règle 7 de la figure A.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 ; (A.1)
          | type IDENTIFIER is [ circular ] range ( IDENTIFIER
            [ , IDENTIFIER ]* ); (A.2)
          | type IDENTIFIER is array ( range_rule
            [ , range_rule ]* ) of IDENTIFIER ; (A.3)
          | type IDENTIFIER is set of IDENTIFIER ; (A.4)
          | type IDENTIFIER is bag of IDENTIFIER ; (A.5)
          | type IDENTIFIER is record
            [ IDENTIFIER : IDENTIFIER [ := expression ] ; ]+
            end ; (A.6)
          | type IDENTIFIER is port [ ( IDENTIFIER [ , IDENTIFIER ]* ) ] ; (A.7)
          | type IDENTIFIER is opaque [ exten_method ]* end (A.8)
extern_method : function IDENTIFIER [ parameters ] return IDENTIFIER (A.9)
              | procedure IDENTIFIER [ parameters ] (A.10)

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

FIG. A.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 A.10 page 174.

Les règles 11 et 12 permettent de définir un intervalle. 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).

Exemple 15 *Exemples de déclarations de types valides en **LfP** :*

```
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 t_record is record
  champ1 : tableau ;
  champ2 : integer := 8 ;
  champ3 : integer ;
end ;
type my_port is port (integer, essai);
```

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 **LfP**.*
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 (initialisé 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 valeur 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".*

A.3.4 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 A.5.

variable : *IDENTIFIER* [, *IDENTIFIER*]* : *IDENTIFIER* [:= *expression*] ; (A.1)
constant : const *IDENTIFIER* [, *IDENTIFIER*]* : *IDENTIFIER* := *expression* ; (A.2)

FIG. A.5 – déclaration de variables et de constantes

Les règles de la figure A.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.

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 créer une variable d'un type composant ne crée pas une instance de ce composant ; seul l'opérateur d'instanciation présenté à la section A.5.7 permet de créer une nouvelle instance de composant.

Exemple 16 *Exemples de déclarations de variables 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 ;
```

A.4 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 globaux du modèle.

La syntaxe des déclarations de constantes est donnée dans la section A.3. Dans cette section, nous étudieront successivement les syntaxes permettant de définir :

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

A.4.1 Déclaration des instances statiques

Les règles de syntaxes présentées par la figure A.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. A.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.

A.4.2 Définition des caractéristiques des binders

Les caractéristiques des binders dont définies à l'aide des quatre attributs suivants :
- multicplicité 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.
- liaison : cet attribut relie les instances de binders aux ports des classes. Sa syntaxe est donnée par la figure A.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.
- capacité : 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.
- ordonnancement : 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.

liaison : *IDENTIFIER* . *IDENTIFIER*

FIG. A.7 – association des binders

La figure A.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.

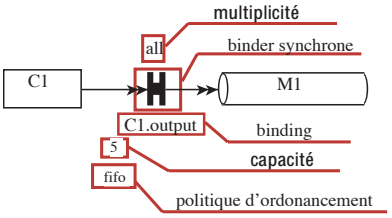


FIG. A.8 – Un exemple de déclaration de binder

A.5 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.

A.5.1 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 figureA.9

trigger : *trigger IDENTIFIER ;*

FIG. A.9 – Déclaration des triggers

A.5.2 Déclaration des méthodes d'une classe

Les règles de la figure A.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.

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 ;

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

FIG. A.10 – Déclarations des méthodes d'une classe

- 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.
- Éventuellement 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

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

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.

A.5.3 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 A.11.

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

FIG. A.11 – syntaxe d'une suite de déclaration

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 A.11.

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 A.12. l'IDENTIFIER est le nom du trigger dont le corps est donné par le sous diagramme.

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

FIG. A.12 – déclaration du corps d'un trigger

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 A.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. A.13 – Déclaration du corps d'une méthode

A.5.4 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 A.14. La syntaxe des gardes des transitions est définie en A.5.6 après avoir étudié l'ensemble des transitions du langage **L*f*P**

instructions : **[** *instruction* **]** $*$ (A.1)

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

FIG. A.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 A.4 page 169.

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 A.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.

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 ;
```

A.5.5 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 envoyés 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 figure A.15.

(a) (b) (c)

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

La transition A.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 A.15(b) permet de lire des informations dans les binders spécifiés. La transition de la figure A.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.

Opérations sur les binders depuis les classes

La syntaxe des opérations sur les binders au niveau des classes est définie sur la figure A.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.

Règle A.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

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

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

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

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

figure A.16 ne prend pas en compte les types de transition sur lesquelles chaque opération est autorisées. Ces limitations sont précisées dans le commentaire qui leur est associé.

Règles A.2, A.3, A.4, A.5, A.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 A.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 A.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 A.3).

Règle A.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 A.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 A.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 transfert de données typées. Cette règle ne peut être utilisée que sur une transition d'envoi (fig A.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 A.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.

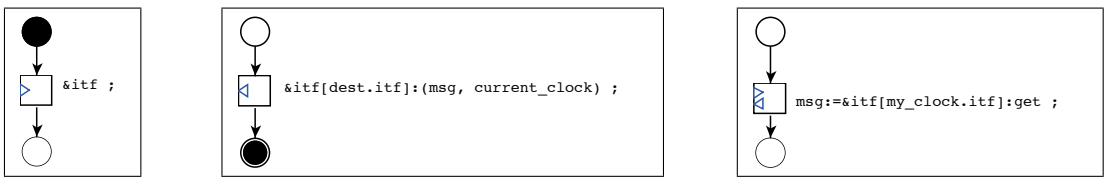
Règle A.13 : cette règle définit la syntaxe des envois de messages dits "de contrôle". Il s'agit de messages envoyés par la classe à destination du média. Le premier identificateur contient le port dans lequel le message est déposé ; et le discriminant contient le message destiné au média.

Règle A.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 A.15 : la règle expressions désigne une suite d'expressions arithmétique séparées par des virgules.La règle expression est définie sur la figure A.3 (page 167).

Exemples d'opérations réalisables depuis une classe

La figure A.17 donne trois exemples d'utilisation de transitions de communications dans une classe. La figure A.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 A.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 transmis qu'au média) est constitué d'une seule variable entre crochet. Enfin, la figure A.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. A.17 – Exemples d'opérations de communication depuis les classes **L*f*P**

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

Syntaxe pour les opérations sur les binders depuis les médias

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

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

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

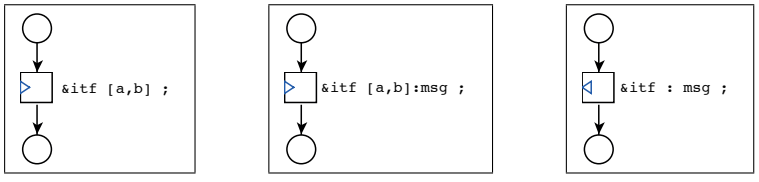
Règle 1 : cette règle précise la syntaxe des réceptions de messages de contrôle dans les médias. Cette règle ne peut être utilisée que sur une transition de réception.

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 A.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.

Exemples d'opérations réalisées depuis un média



(a) (b) (c)

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

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

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

A.5.6 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 A.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.

garde : \[*expression* \] (A.1)

FIG. A.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 :

- 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).

A.5.7 Instanciation dynamique de composants

Présentation de la syntaxe

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

instanciation : *expr_variable* **:=** *IDENTIFIER* **(** *IDENTIFIER* **=>** **expression** **;**
[**,** *IDENTIFIER* **=>** **expression** **]** ∗ **)** (A.1)
| *expr_variable* **:=** *IDENTIFIER* **;** (A.2)

FIG. A.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.

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) ;
```

Annexe B

BNF du langage L*f*P sous sa forme textuelle

B.1 Introduction

Cette annexe présente la syntaxe textuelle du langage **L*f*P** développée dans le cadre de MORSE. Ce travail est encore en cours de réalisation. Il s'agit d'un travail en cours de réalisation, son objectif est d'implémenter une partie des retours sur expériences présentés au chapitre 8, tout en permettant une implémentation rapide des outils associés au langage développés dans le cadre de MORSE.

La section B.2 rappelle les conventions utilisées dans les documents de syntaxe du langage **L*f*P**. La section B.5 présente la syntaxe de la partie déclarative du langage. Cette section définit principalement les déclarations de types du langage et la représentation du diagramme d'architecture. La section B.6 présente la syntaxe des instructions du langage **L*f*P**.

B.2 Conventions de présentation

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

- les mots clés seront écrits en rouge, et en minuscule ;
- les règles de syntaxe non terminales sont écrites en pourpre, et en minuscules ;
- les règles terminales du parseur sont écrites en bleu et en majuscules ;
- les caractères faisant partie de la syntaxe du langage **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.

B.3 Éléments de base du langage

B.3.1 Casse des caractères

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

B.3.2 Mots réservés et symboles utilisés par le langage L*f*P

Mots réservés du langage

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

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

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

Symboles utilisés par le langage

Les symboles utilisés par le langage **L*f*P** sont rassemblés sur la figure B.2.

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

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

B.3.3 Identificateurs valides

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

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

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

B.3.4 Chiffres et nombres

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

integer : [0 – 9]+

FIG. B.4 – Lexème définissant un entier

B.4 Syntaxe des expressions

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

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

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

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

La figure B.5 donne la BNF des expressions valides en **L*f*P**. La règle operator n'est pas définie explicitement par la figure B.5. Elle désigne l'ensemble des opérateurs listés dans les paragraphes précédents, en respectant leurs priorités.

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

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

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

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

expressionlist : *expression* **[** **,** *expression* **]**∗ (B.8)

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

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

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

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

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

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

La règle B.6 correspond à la déréférenciation qui peut être utilisée dans deux contextes :

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

La règle B.7 correspond à un appel de fonction. La syntaxe de cette instruction sera présentée à section B.6.12.

B.5 Syntaxe de la partie déclarative

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

– la partie déclarative globale du modèle ;
– la partie déclarative des composants.

La partie déclarative globale permet de représenter de manière textuelles les diagrammes d'architecture du langage, c'est à dire :

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

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

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

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

B.5.1 Déclaration des types tableaux

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

type_tableau : *type IDENTIFIER is array* (*range_list*) *of IDENTIFIER* ; (B.9)
range_definition : *expression* .. *expression* (B.10)
range_list : *range_definition* [, *range_definition*]∗ (B.11)

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

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

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

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

B.5.2 Déclaration des types énumérés

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

type_enumere : *type IDENTIFIER is enum* (*identifier_list*) ; (B.12)
| *type IDENTIFIER is circular enum*
(*identifier_list*) ; (B.13)

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

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

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

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

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

B.5.3 Déclaration des intervalles de valeurs

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

type_range : *type IDENTIFIER is range* (*range_definition*) *of IDENTIFIER* (B.15)
| *type IDENTIFIER is circular range* (*range_definition*)
of IDENTIFIER (B.16)

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

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

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

B.5.4 Déclaration des structures

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

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

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

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

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

La règle B.18 donne la BNF de la déclaration d'un champ pour un type record. Chaque champ est défini par son nom suivi de son type séparés par le caractère " : ". La syntaxe permet de déclarer plusieurs champs du même type en utilisant une liste d'identificateurs séparés par des virgules.

B.5.5 Déclaration des types port

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

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

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

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

B.5.6 Déclaration de variables et de constantes

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

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

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

FIG. B.11 – BNF des déclarations de variables et constantes

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

La règle B.21 définit la déclaration d'une ou plusieurs variables. Il
- Le premier identificateur correspond au nom de la variable déclarée ;
- la syntaxe permet de déclarer et d'initialiser plusieurs variables de même type en utilisant des identificateurs séparés par des virgules
- l'identificateur situé après le " : " définit le type de la ou des nouvelle(s) variable(s) ;
- enfin l'expression précise la valeur initiale des variables déclarées.

B.5.7 Déclaration des composants

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

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

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

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

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

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

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

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

Les règles B.25 et B.26 permettent de définir le type du composant qui peut être une classe ou un média.

B.5.8 Déclaration des méthodes des classes

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

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

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

La règle B.28 définit la syntaxe d'une déclaration de fonction.

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

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

mode : *in* (B.33)
| *inout* (B.34)
| *out* (B.35)
body : *is* **[** *variable* |*constant* **]**∗ *body* (B.36)

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

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

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

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

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

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

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

La règle B.36 définit le corps de la méthode :

– d'une série optionnelle de déclarations de variables et constantes locales ;
– de la suite d'instruction à effectuer lors de l'appel.
Les instructions valides sont définies à la section B.6.

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

B.5.9 Déclaration des triggers

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

declaration_trigger : *trigger IDENTIFIER* (*parametres*) *body* ; (B.37)
| *trigger IDENTIFIER* (*parametres*) *return IDENTIFIER*
body ; (B.38)

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

La règle B.37 définit la syntaxe de la déclaration d'un trigger sans valeur de retour. Il s'agit d'une procédure privée utilisable uniquement dans le composant qui la définit. L'identificateur désigne le nom du trigger, la syntaxe des paramètres et du corps du trigger sont les mêmes que pour les déclarations de méthode ou de fonctions (cf. règles B.29, B.32 et (B.36) de la figure B.13).

La règle B.38 donne la syntaxe de la déclaration d'un trigger qui retourne une valeur. Il s'agit d'une fonction privée utilisable uniquement dans le composant qui la définit :
– le premier identificateur est le nom du trigger ;
– la syntaxe des paramètres est la même que pour les déclarations de méthodes (cf. règles B.29, B.32 et (B.36) de la figure B.13) ;
– le mot clé return est suivi du nom du type de la valeur de retour.
Les appels de triggers retournant une valeur sont considérés comme des expressions, leur valeur de retour doit donc systématiquement être utilisée.

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

B.5.10 Déclaration des binders

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

La règle B.39 donne la BNF globale de la déclaration d'un binder en **L*f*P**
– le premier identficateur est le nom de la classe à laquelle le binder est rattaché ;
– Le second identificateur est le nom du port qui lui est associé ;
– ordering précise l'ordonnancement des messages dans la file ;
– l'expression entre parenthèses précise la capacité du binder en nombre de messages ;
– elle est suivie de son schéma de connexion.

Les règles B.40 et B.41 définissent l'ordonnancement des messages dans un binder :
– fifo les messages doivent être consommés dans la file dans l'ordre de leur arrivée (la lecture sur le binder est bloquante si le premier message de la file ne peut être consommé) ;

```
binder   :  binder IDENTIFIER . IDENTIFIER is ordering ( expression )
            conexion end ;                                              (B.39)

ordering :  fifo                                                        (B.40)
         |  bag                                                         (B.41)

conexion :  lecteurs ecrivains                                          (B.42)

lecteurs :  IDENTIFIER |-> IDENTIFIER [ , IDENTIFIER ]* ;               (B.43)
ecrivains:  IDENTIFIER <-| IDENTIFIER [ , IDENTIFIER ]* ;               (B.44)
```

FIG. B.15 – BNF de la déclaration des binders

– bag les messages présents dans la file peuvent être consommés dans un ordre quelconque (la lecture sur le binder est bloquante si aucun message de la file ne peut être consommé).

La règle B.42 permet de spécifier le schéma de connexion du binder, on spécifie toujours en premier les composants qui peuvent lire des messages dans le binder, puis les composants qui peuvent y écrire des messages.

La règle B.43 permet de spécifier la liste des composants pouvant lire un message dans le binder :
– le premier identificateur rappelle le nom du binder (il doit être égal au deuxième identificateur de la règle B.39 ;
– à droite du symbole |-> on trouve la liste des composants pouvant lire des messages dans le binder.

La règle B.44 permet de spécifier la liste des composants pouvant écrire des messages dans le binder :
– le premier identificateur rappelle le nom du binder (il doit être égal au deuxième identificateur de la règle B.39 ;
– à droite du symbole <-| on trouve la liste des composants pouvant écrire des messages dans le binder.

B.5.11 Déclaration de la topologie

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

La règle B.45 permet de spécifier que le média dont le nom est donné par le premier identificateur peut relier chacun des composants spécifiés dans la liste entre parenthèses.

La règle B.46 est une extension de la précédente. Elle permet de spécifier plusieurs liens pour un seul média en une seule déclaration. Chacune des listes définit un lien possible pour le média, elles sont séparées par des virgules et la "liste de liste" est elle même entre parenthèses.

La règle B.47 précise la syntaxe des listes de composants. Il s'agit d'une liste d'identificateur séparés par des virgules désignant une classe qui doit être déjà définie dans le modèle.

lien : *media IDENTIFIER component_list* ; (B.45)
| *media IDENTIFIER* (*component_list* [,
component_list]+) ; (B.46)

component_list : (*IDENTIFIER* [, *IDENTIFIER*]+) ; (B.47)

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

B.6 Syntaxe des instructions du langage L*f*P

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

B.6.1 Affectation

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

affectation : *expression* \ := *expression* ; (B.48)

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

La règle B.48 définit la BNF d'une affectation :
- l'expression de gauche doit désigner une variable ou bien un paramètre en mode out ou inout défini dans l'environnement courant ;
- l'expression de droite désigne la valeur qui lui sera affectée.

Les deux expressions doivent être de types compatibles.

B.6.2 Les blocs d'instruction

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

bloc_instruction : *declare declarations begin instructions end* ; (B.49)
| : *IDENTIFIER declare declarations begin instructions end* ; (B.50)

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

La règle B.49 définit un bloc d'instructions. Il est constitué en deux parties : une partie déclarative permettant de définir des variables locales, et la partie contenant les instructions elles même.

La règle B.50 définit la syntaxe d'un bloc d'instructions nommé. L'identificateur définit le nom du bloc ; ce nom peut servir de cible à une instruction de saut, le reste est identique à la règle B.49.

B.6.3 Les boucles *"for"*

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

boucle_for : **[** : *IDENTIFIER* **]** *for IDENTIFIER in expression .. expression*
begin instructions end ; (B.51)

FIG. B.19 – BNF d'une boucle *"for"*

La règle B.51 Définit la syntaxe d'une boucle for **L*f*P** :
- l'identificateur précédé du caractère " : " correspond au nom de la boucle, il peut être utilisé par une instruction break pour sortir explicitement d'une boucle ;
- l'identificateur suivant le mot clé for est le nom de la variable de boucle ;
- les deux expressions définissent l'intervalle de valeur parcouru par la variable de boucle ;
- le corps de la boucle est définit par les instructions situées entre les mots réservés begin et end .

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

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

B.6.4 Les boucles *"while"*

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

boucle_while : **[** : *IDENTIFIER* **]** *while expression begin instructions end* ; (B.52)

FIG. B.20 – BNF des boucles *"while"*

La règle B.52 définit la BNF d'une boucle while en **L*f*P** :
- l'identificateur précédé du caractère " : " définit le nom de la boucle ;
- l'expression définit la condition de sortie de boucle ;
- le corps de la boucle est donné entre les mots clés begin et end .

B.6.5 Instruction *"break"*

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

instruction_break : *break* **[** *IDENTIFIER* **]** ; (B.53)

FIG. B.21 – BNF de l'instruction break

La règle B.53 définit une instruction break. Cette instruction doit être définie à l'intérieur du corps d'une boucle. Si l'identificateur optionnel est omis, l'instruction provoque la sortie de la boucle courante. Si l'identificateur est précisé, il doit correspondre au nom d'une boucle, et l'instruction provoque la sortie du corps de cette boucle.

B.6.6 L'instruction d'alternative

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

$$\begin{aligned} \textit{alternative} \quad : \quad & \textit{if expression then instructions} \\ & [\ \textit{elsif expressionthen instructions}\ ]* \\ & [\ \textit{else instructions}]\ \textit{end ;} \end{aligned} \tag{B.54}$$

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

La règle B.55 donne la BNF d'une instruction d'alternative **L*f*P**. On distingue trois types de branches :

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

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

B.6.7 Instruction de saut

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

$$\textit{instruction\_goto} \quad : \quad \textit{goto IDENTIFIER ;} \tag{B.55}$$

$$\textit{label} \quad : \quad \textit{: IDENTIFIER :} \tag{B.56}$$

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

La règle B.55 présente la BNF de l'instruction goto. L'identificateur désigne le label cible du saut.

La règle B.56 présente la BNF de la déclaration d'un label en **L*f*P**. L'identificateur est le nom du label, il ne doit pas surcharger le nom d'une variable déclarée ou d'une boucle nommée.

B.6.8 Lecture d'un message

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

La règle B.57 décrit la syntaxe d'une opération de lecture :

- lorsque l'instruction est effectuée depuis un média, le discriminant peut être lu (cf. règle B.58) ;
- la liste d'expression entre parenthèse définit l'ensemble des variables ou paramètres en mode out ou inout qui vont recevoir les données contenues dans le message ;
- l'expression après le symbole "<- " définit le port sur lequel le message est lu ;

```
lecture       : discriminant ( expression [ , expression ]* ) <- expression
                [ with expression ] ;                                          (B.57)
discriminant  : \[ expression [ , expression ]* \]                             (B.58)
```

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

– il est possible de spécifier une garde sur le contenu du message en utilisant le mot clé with suivi d'une expression booléenne.

La règle B.58 définit la syntaxe d'un discriminant. Il s'agit d'une liste entre crochets d'expression séparées par des virgules.

B.6.9 Attente d'activation de méthode

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

```
attente_activation  : IDENTIFIER <- expression ;                               (B.59)
                    | IDENTIFIER <- ( ) ;                                      (B.60)
```

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

La règle B.60 définit la BNF d'une attente d'activation de méthode :
– l'identificateur donne le nom de la méthode activable ;
– l'expression donne le port sur lequel le message d'activation devra être lu ;

La règle B.60 définit la BNF d'une attente d'activation de méthode sur le port par défaut. L'identificateur désigne le nom de la méthode activable. Le message sera lu sur le port d'activation par défaut spécifié lors de la déclaration de la méthode.

B.6.10 Multiplexage des opérations de réception de messages

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

```
instruction_accept  : accept lecture [ , lecture ] end ;                       (B.61)
```

FIG. B.26 – BNF de l'instruction accept

La règle B.61 présente la BNF d'une instruction accept. Chaque opération de lecture peut être soit une lecture d'un message de donnée, soit une opération d'attente d'activation de méthode. Les syntaxes utilisées pour spécifier ces opérations sont celles vues en B.6.8, et B.6.9 sans le " ; " final. Les opérations multiplexées sont séparées entre elles par une virgule. Dans le cas d'une instruction de lecture, les expressions doivent définir des variables, et l'instruction doit être dans média.

B.6.11 Envoi d'un message de données

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

```
envoi_message : discriminant ( expression [ , expression ]* )
                -> expression ;                                   (B.62)
```

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

La règle B.62 définit l'envoi d'un message de donnée :

- le discriminant du message peut être spécifié en utilisant la syntaxe définie à la règle B.58 de la figure B.24 ;
- les données du message sont situées entre les parenthèses après le discriminant du message ;
- l'expression située après le symbole -> spécifie le port sur lequel le message est envoyé.

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

B.6.12 Appel d'une méthode d'un composant

On distingue en fait trois instructions d'appel de méthodes :

- l'appel de procédure asynchrone ;
- l'appel de procédure synchrone ;
- l'appel de fonction.

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

```
designation    : [ discriminant ] [ IDENTIFIER : ]
                 IDENTIFIER ( [ parametres ] )                    (B.63)
parametres     : expression [ , expression ]*                     (B.64)
appel_methode  : designation -> expression ;                      (B.65)
               | designation <-> expression ;                     (B.66)
               | designation <=> expression ;                     (B.67)
```

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

La règle B.63 définit la manière dont la méthode appelée est définie :

- le discriminant associé à l'appel de méthode est spécifié de la même manière que pour les envois de messages (cf. B.6.11) ;
- le premier identificateur permet de préciser le type du composant définissant la méthode appelée ;

– l'identificateur donne le nom de la méthode ;
– les paramètres sont donnés entre parenthèses

La règle B.64 donne la syntaxe de la liste des paramètres. Ils sont séparés par des virgules. Toutes les expressions sont acceptées, mais les expressions retournant des valeurs immédiates sont considérées comme invalides pour les paramètres en mode out .

Les règles B.65, B.66 et B.67 spécifient respectivement les appels de procédures asynchrones, de procédures synchrones et de fonctions. L'expression située après les symboles -> , <-> ou <=> définissent le port dans lequel le message d'activation est écrit.

Bibliographie

[1] J. Ø. Aagedal, J. Bezivin, and P. F. Linington. Model Driven Development, pages 148–157. Lecture Notes in Computer Science. Springer-Verlag, 2004.

[2] Adaptive, Alcatel, CBOP, Codagen Technologies Corp., DSTC, IBM, Softeam, Tata Consultancy Services, Thales, and TNI-Valiosys. Revised Submission for MOF 2.0 Query View Transformation RFP, 2004.

[3] David H. Akehurst and Stuart J. H. Kent. A Relational Approach to Defining Transformations in a Metamodel. In Jean-Marc Jezequel and Heinrich Hussmann, editors, <> 2002 - The Unified Modeling Language : Model Engineeing, Concepts, and Tools, volume 2460 of Lecture notes in computer science. Springer, October 2002.

[4] Robert J. Allen. A Formal Approach to Software Architecture. PhD thesis, School of Computer Science Carnegie Mellon University, 1997. CMU-CS-97-144.

[5] B. Rumpe, M.Schoenmakers, A. Radenmacher, and A. Schürr. UML + ROOM as a standard ADL ? In Proc. ICES'99 Fifth IEEE International Conference on Engineering of Complex Computer Systems, 1999.

[6] Benoit Caillaud, Jeant-Pierre Talpin, Jean-Marc Jezequel, Albert Benveniste, and Claude Jard. Bdl : A semantic backbone for uml dynamic diagrams. Research Report 4003, Institut National de Recherche en Informatique et en Automatique (INRIA), september 2000.

[7] O. Biberstein, D. Buchs, and N. Guelfi. CO-OPN/2 : A concurrent object-oriented formalism. In Proc. Second IFIP Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS), pages 57–72, London, 1997. Chapman and Hall.

[8] Olivier Biberstein. CO-OPN/2 : An Object-Oriented Formalism for the Specification of Concurrent Systems. PhD thesis, University of Geneva, July 1997.

[9] C. Binding, W.Bouna, M.Bauphin, G.karjoth, and Y.Yang. A common compiler for lotos and sdl specifications. In IBM Systems journal, volume 4, pages 668–690. IBM, 1992.

[10] JM Bruel, J Lilius, A Moreira, and RB France. Defining precise semantics for uml. In J. Malenfant, S. Moisan, and A. Moreira, editors, Object-Oriented Technology : ECOOP 2000, volume 1964 / 2000 of Lecture Notes in Computer Sciences. Springer-Verlag, 2000.

[11] Béatrice Bérard, Michel Bidoit, François Laroussinie, Antoine Petit, and Philippe Schoebelen. Vérification de Logiciels. Vuibert, 1999.

[12] CCIB. Common Criteria For Information Technology Security Evaluation, 1999.

[13] Stanislav Chachkov and Didier Buchs. From an abstract object-oriented model to a ready-to-use embedded system controller. In 12th International Workshop on Rapid System Prototyping. IEEE, 2001.

[14] Stanislav Chachkov and Didier Buchs. From formal specifications to ready-to-use software components : The concurrent object-oriented petri net approach. In Second International Conference on Application of Concurrency to System Design (ACSD'01), pages 99–110. IEEE, 2001.

[15] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. Stochastic well-formed colored nets and symmetric modeling applications. IEEE Transactions on Computers, 42(11) :1343–1360, 1993.

[16] R. Clark and A. Moreira. Use of E-LOTOS in adding formality to UML. Journal of Universal Computer Science, 6(11) :1071–1087, 2000.

[17] Jean-Michel Couvreur, Emmanuelle Encrenaz, Emmanuel Paviot-Adet, Denis Poitrenaud, and Pierre-André Wacrenier. Data decision diagrams for petri net analysis. In Javier Esparza and Charles Lakos, editors, Applications and Theory of Petri Nets 2002, 23rd International Conference, ICATPN 2002, Proceedings, number 2360 in Lecture notes in computer science, pages 101–120, Adelaide, Australia, June 24-30 2002. Springer Verlag.

[18] Krysztof Czarncki and Simon Helsen. Classification of model transformation approaches. In OOPSLA'03 Workshop on Generative Techniques in the context of Model-Driven Architecture, 2003.

[19] Réseau National des Technologies Logicielles. http ://www.telecom.gouv.fr/rntl/index.htm.

[20] P. Dissaux. UML 2.0 notation for the AADL. unpublished white paper, november 2003.

[21] ETSI. Methods for Testing and Specification (MTS) ; The testing and Test Control Notation Version3, 2003.

[22] European Telecommunications Standards Institute. Telecommunications Management Network (TMN) ;Universal Mobile Telecommunications System (UMTS) ;Management architecture framework ;Overview, processes and principles.

[23] Frédéric Gilliers, François Bréant, Denis Poitrenaud, and Fabrice Kordon. Model checking of high-level object oriented specifications : the lfp experience. In Proceedings of the Third Workshop on Modelling of Object, Components, and Agents, 2004.

[24] H. Garavel and M. Sighireanu. Towards a second generation of formal description techniques : Rationale for the design of e-lotos, 1998.

[25] Hubert Garavel, Mark Jorgensen, Radu Mateescu, Charles Pecheur, Mihaela Sighireanu, and Bruno Vivien. Cadp'97 - status, applications, and perspectives. In 2nd COST 247 International Workshop on Applied Formal Methods in System Design (Zagreb, Croatia), june 1997.

[26] W. Gibbs. Software's chronic crisis. Scientific American, pages 86–95, 1994.

[27] Object Management Group. Mof model to text transformation language. Request For Proposal ad/2004-04-07, OMG, 2004.

[28] David Harel. Statecharts : A visual formalism for complex systems. Sci. Comput. Programming, pages 231–274, 1987.

[29] Wai-Ming Ho, Jean-Marc Jézéquel, Alain Le Guennec, and François Pennaneac'h. Umlaut : an extendible uml transformation framework. In Automated Software Engineering, ASE'99, october 1999.

[30] INRIA. Site web du projet vasy http ://www.inrialpes.fr/vasy.

[31] Interantional Telecommunication Union Standardiation Sector (ITU-T). Specification and Description Language (SDL), 1999.

[32] Interantional Telecommunication Unition Standardiation Sector (ITU-T). SDL combined with UML, november 1999.

[33] International Telecommunication Union. Message Sequence Chart (MSC), 1999.

[34] International Telecommunication Union Standardization Sector (ITU-T). Specification and Description Language (SDL), Annex F2 : SDL formal definition : Static semantics, 1999.

[35] International Telecommunication Union Standardization Sector (ITU-T). Specification and Description Language (SDL), Annex F3 : SDL formal definition : Dynamic semantics, 1999.

[36] ISO. ISO 8807 :1989 LOTOS A formal description technique based on the temporal ordering of observational behaviour, 1989.

[37] ISO. Information thechnology – Programming Languages – Ada. Technical report, ISO, february 1995. ISO/IEC/ANSI 8653 :1995.

[38] ISO. Meta object facility (mof) specification. Technical report, ISO, april 2002.

[39] ISO/IEC. ISO/IEC 15437 :2001 Enhencements to LOTOS (E-LOTOS), 2001.

[40] J. -C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP : a protocol validation and verification toolbox. In Rajeev Alur and Thomas A. Henzinger, editors, Proceedings of the Eighth International Conference on Computer Aided Verification CAV, volume 1102, pages 437–440, New Brunswick, NJ, USA, / 1996. Springer Verlag.

[41] Java Community Process. Java Metadata Interface (JMI) Specification, 1.0 edition, June 2002.

[42] Jean Bézivin, Frédéric Jouault, and David touzet. An introduction to the atlas model management architecture. Technical report, Laboratoire d'informatique de Nantes Atlantique, February 2005.

[43] Johan Bengtsson, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL - a tool suite for automatic verification of real-time systems. In Hybrid Systems, pages 232–243, 1995.

[44] Stuart Kent. Model Driven Engineering, volume 2335 / 2002 of Lecture Notes in Computer Science, chapter p.286, pages 286–299. Springer-Verlag GmbH, 2002.

[45] J. Kerr and R. Hunter. Inside RAD. Mc Graw Hill, 1995.

[46] Alexander Knapp, Stephan Merz, and Christopher Rauh. Model checking timed UML state machines and collaborations. In W. Damm and E.-R. Olderog, editors, 7th Intl. Symp. Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT 2002), volume 2469 of Lecture Notes in Computer Science, pages 395–414, Oldenburg, Germany, September 2002. Springer-Verlag.

[47] F. Kordon and Luqi. An introduction to rapid system prototyping. IEEE Transaction on Software Engineering, 2003.

[48] Fabrice Kordon and Michel Lemoine, editors. Formal methods for embedded distributed systems : how to master the complexity. Kluwer Academic Publishing, first edition, 2004.

[49] L. Apvrille, J.-P. Courtiat, C. Lohr, and P. de Saqui-Sannes. Turtle : A real-time uml profile supported by a formal validation toolkit. IEEE Trans. Softw. Eng., 30(7) :473–487, 2004.

[50] Levi Lucio, Luis Pedro , and Didier Buchs. A methodology and a framework for model-based testing. In Proceedings of RISE 2004, University of Luxembourg, pages 52–63, 2004.

[51] Levi Lucio, Luis Pedro , and Didier Buchs. A test selection language for co-opn specifications. In To appear in 16th IEEE International Workshop on Rapid System Prototyping (RSP'04), 2005.

[52] Barbara Liskov and Jeanette M. Wing. A behavioral notion of subtyping. ACM Transaction on Programming Languages and Systems, 1994.

[53] Luqi and Joseph A. Goguen. Formal methods : Promises and problems. IEEE Softw., 14(1) :73–85, 1997.

[54] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. Software Engineering, 26(1) :70–93, 2000.

[55] Sofia N. Mejia. Le méta-Modèle LfP. Laboratoire d'Informatique de PARIS VI, 2004.

[56] OMG. Common Object Request Broker Architecture, 2001.

[57] OMG. Model Driven Architecture (MDA), Document number ormsc/2001-07-01, 2001.

[58] OMG. MDA Guide Version 1.0.1, 2003.

[59] OMG. Omg unified modeling language specification, version 1.5. Technical report, OMG, 2003.

[60] OMG. Unified Modeling Language Superstructure Version 2.0, October 2004.

[61] Paris Avgeriou, Nicolas Guelfi, and Nenad Medvidovic. Software architecture description and uml. In UML 2004 - Modeling Languages and Applications, pages 23–32, Lisbon - Portugal, 2004.

[62] Laurent Pautet. Intergiciels schizophrènes : une solution à l'interopérabilité entre modèles de répartition. Mémoire d'habilitation à diriger des recherches, upmc, décembre 2001.

[63] Peter H. Feiler, Bruce Lewis, and Steve Vestal. Improving Predictability in Embedded Real-Time Systems. Carnegie-Mellon Software Engineering Institute, December 2000.

[64] R. Probert and A. Williams. Fast functional test generation using an sdl model, 1999.

[65] D. Quartel, M. van Sinderen, and L. Ferreira Pires. A model-based approach to service creation. In 7th IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems, pages 102–110. IEEE Computer Society, 1999.

[66] Thomas Quinot. Conception et réalisation d'un intergiciel schizophrène. PhD thesis, Université Paris VI - Pierre et Marie Curie, mars 2003.

[67] Dan Marius Regep. LfP : un langage de spécification pour supporter une démarche de développement par prototypage pour les systèmes répartis. PhD thesis, Université Pierre et Marie Curie PARIS VI, 2003.

[68] W. Reisig. Petri nets and algebraic specifications. Theoretical Computer Science 80, pages 1–34, 1991.

[69] Requirements and Technical Concepts for Aviation (RTCA). Software Recommandations in Airborn Systems and Equipment Certification (DO-178B), 1992.

[70] John Rushby. Formal Methods and the Certification of Critical Systems. Technical Report CSL-93-7, Computer Science Laboratory SRI International, Decembre 1993. Egalemnt paru sous le titre Formal Methods and Digital Systems Validation for Airborn Systems as NASA CR 4551. Preparation of this report was sponsored by the Federal Aviation Administration an by the National Aeronotics and Space Administration under contract NAS1-18969.

[71] John Rushby. Formal Methods and their role in the Certification of Criticals Systems. Technical Report CSL-95-1, Computer Science Laboratory SRI International, Mars 1995. Also available as NASA Contractor Report 4673, August 1995, and to be issued as part of the FAA Digital Systems Validation Handbook (the guide for aircraft certification).

[72] John Rushby. Ubiquitous abstraction : A new approach for mechanized formal verification (extended abstract). In Second International Conference on Formal Engineering Methods (ICFEM '98), pages 176–178, Brisbane, Australia, dec 1998. IEEE Computer Society.

[73] John Rushby. Mechanized formal methods : Where next ? In Jeannette Wing and Jim Woodcock, editors, FM99 : The World Congress in Formal Methods, volume 1708 of Lecture Notes in Computer Science, pages 48–51, Toulouse, France, September 1999. Springer Verlag.

[74] John Rushby. Theorem proving for verification. In Franck Cassez, Claude Jard, Brigitte Rozoy, and Mark Dermot Ryan, editors, Modelling and Verification of Parallel Processes : MOVEP 2000, number 2067 in Lecture Notes in Computer Science, pages 39–57, Nantes, France, June 2000. springer Verlag.

[75] Vestal S. Metah user's manual. Technical report, Honneywell Inc., 1998.

[76] SAE Aerospace. ARCHITECTURE ANALYSIS & DESIGN LANGUAGE (AADL) v0.994, august 2004.

[77] M. Schmitt, M. Ebner, and J. Grabowski. Test generation with autolink and testcomposer. In 2nd Workshop of the SDL Forum Society on SDL and MSC - SAM', 2000.

[78] Carron Shankland and Alberto Verdejo. A case study in abstraction using e-lotos and the firewire. Comput. Networks, 37(3-4) :481–502, 2001.

[79] Mihaela Sighireanu, Alban Catry, David Champelovier, Hubert Garavel, Frédéric Lang, and Guillaume Schaeffer. LOTOS-NT User Manual. INRIA, june 2004.

[80] Mihaela Sighireanu and Radu Mateescu. Validation of the link layer protocol of the ieee-1394 serial bus (firewire) : an experiment with e-lotos. International Journal on Software Tools for Technology Transfer (STTT), 2 :68–88, 1998.

[81] A.J.H. Simons and L. Graham. 30 things that go wrong in object modelling with uml 1.3. In The Kluwer International Series in Engineering and Computer Science, volume 523, chapter 17. Kluwer Academinc Publishers, 1999.

[82] SUN MicroSystems. JavaBeans API Specification (Version 1.01), August 1997.

[83] B-CORE (UK). http ://www.b-core.com/btollkit.html.

[84] W3C. Simple object access protocol (soap) 1.2. Technical report, W3C, june 2003.

[85] J. Warmer and A. Kleppe. The Object Constraint Language : Precise Modeling with UML. Addison-Wesley, 1998.

[86] J. Warmer and A. Kleppe. Ocl : The constraint language of the uml. Journal of Object-Oriented Programming, 1999.

[87] Site web de la société Aonix. http ://www.aonix.fr.

[88] Site web de la société Telelogic. http ://www.telelogic.com/tau/sdl.

[89] Site web du projet MORSE. http ://www.lip6.fr/morse/.

[90] William E. McUmber and Betty H.C. Cheng. A general framework for formalizing uml with formal languages. In Proceedings of the IEEE International Conference on Software Engineering, 2001.