Approche dirigée par les modèles pour le développement de systèmes multi-agents

Selma Azaiez

To cite this version:

Selma Azaiez. Approche dirigée par les modèles pour le développement de systèmes multi-agents. Génie logiciel [cs.SE]. Université de Savoie, 2007. Français. NNT : . tel-00519195

HAL Id: tel-00519195
https://theses.hal.science/tel-00519195

Submitted on 18 Sep 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE DE l'UNIVERSITÉ DE SAVOIE

Spécialité : Informatique

présentée par

Selma AZAIEZ

pour obtenir le titre de

DOCTEUR DE L'UNIVERSITÉ DE SAVOIE
(Arrêté ministériel du 30 Mars 1992)

Approche dirigée par les modèles pour le développement de systèmes multi-agents

---

Soutenance prévue le 11 décembre 2007 devant le jury composé de :

Mme	Nicole LEVY	Rapporteur
M.	Juan PAVON MESTRAS	Rapporteur
M.	Yves DEMAZEAU	Examinateur
M.	Vincent DAVID	Membre Invité
M.	Flavio OQUENDO	Directeur de thèse
M.	Marc-Philippe HUGET	Co-encadrant

A mon cher père,
il aurait été si fier

Remerciements

J'adresse mes plus vifs remerciements à M. Juan PAVON MESTRAS, Professeur à l'Universidad Complutense de Madrid et Mme. Nicoles LEVY, Professeur à l'Université de Versailles Saint Quentin en Yvelines pour m'avoir fait l'honneur d'étudier mes travaux de thèse et pour les avoir cautionnés en qualité de rapporteurs.

Je tiens également à remercier M.Yves DEMAZEAU, Directeur de Recherche au Laboratoire d'Informatique de Grenoble (LIG) qui m'a fait l'honneur de présider le jury, ainsi que M. Vincent DAVID, Ingénieur Chercheur au CEA/Saclay pour avoir accepté de faire partie de ce jury.

Je tiens à exprimer toute ma gratitude à M. Flavio OQUENDO, Professeur à l'Université de Bretagne Sud, pour m'avoir donné la possibilité d'effectuer cette thèse et pour m'avoir intégré dans le projet ArchWare qui a constitué une expérience professionnelle très riche. Je tiens aussi à le remercier pour ses nombreux conseils durant mes travaux de thèse et la préparation de la soutenance.

Je tiens également à remercier M. Marc-Philippe HUGET, Maître de Conférences à l'Université de Savoie, pour avoir accepté de co-encadrer cette thèse et pour avoir contribuer à son aboutissement en apportant ses compétences dans le domaine des systèmes multi-agents.

Mes sincères remerciements vont à Georges HABCHI, Magali PRALUS et Jihène TOUNSI pour leur travail collaboratif durant le projet BQR qui m'a permise de valider mon approche. Je remercie également Philippe BOLON et Patrice MOREAUX pour m'avoir donné l'occasion de finaliser ma thèse dans des conditions convenables.

Un grand merci particulier à mes amis et collègues de l'ex-équipe LLP. Je nomme particulièrement Sorana CIMPAN, Ilham ALLOUI, Georges HABCHI, Hervé VERJUS, Frédéric POURRAZ, Lionel BLANC DIT JOLICOEUR et Fabien LEYMONERIE. Leurs nombreux encouragements, conseils ainsi que les nombreuses discussions ont été essentielles pour l'aboutissement de ces travaux. Un grand merci particulier à Valérie BRAESCH pour sa bonne humeur, son dynamisme et la relecture de ce mémoire de thèse.

Enfin, je remercie de tout mon coeur tous mes proches et amis et plus particulièrement ma mère Nouayra et mon frère Ahmed pour leur confiance, leur soutien et leur aide inconditionnels ainsi que leur présence inestimable à mes cotés durant toute la durée de cette thèse. Merci à vous, je n'y serais pas arrivé sans vous !

Table des matières

1 Introduction générale **1**
1 Contexte et problématique 2
2 Une démarche de développement flexible et sûre 4
3 Organisation du document 5

2 Etude du domaine **7**
1 Introduction 8
2 Paradigmes de développement vs Ingénierie des systèmes logiciels 9
2.1 Les paradigmes de développement 9
2.2 Les techniques d'ingénierie 10
3 L'évolution des paradigmes de développement 10
3.1 Le paradigme procédural 11
3.2 Le paradigme orienté objet 11
3.3 Le paradigme orienté composant 11
3.4 Le paradigme orienté service 12
4 Positionnement du paradigme orienté agent 13
4.1 Les problématiques traitées par le paradigme orienté agent 14
4.2 Les freins à l'utilisation des systèmes multi-agents . . 16
5 L'évolution des techniques d'ingénierie 17
5.1 Les techniques d'ingénieries classiques 17
5.2 L'ingénierie des systèmes basés sur le paradigme orienté composant 19
5.3 L'ingénierie dirigée par les modèles 22
5.4 Conclusions relatives à l'évolution des techniques d'ingénierie 23
6 Introduction à la problématiques de la thèse 24

3 Ingénierie des Systèmes Multi-agents **27**
1 Introduction 28
2 Les théories orientées agent 29
2.1 La vue agent 29

2.2 La vue environnement 35
2.3 La vue interaction 36
2.4 La vue organisation 39
3 Ingénierie des systèmes informatiques selon le paradigme orienté agent 43
3.1 Les langages de spécification 43
3.2 Les méthodologies de développement 44
3.3 Les plates-formes d'implémentation 46
4 Les langages de spécification orientés agent 46
4.1 Les langages basés sur une extension d'UML 46
4.2 Les langages réalisés par création de profil UML 47
4.3 Les notations graphiques 48
5 Les méthodologies orientées Agent 49
5.1 La méthodologie ADELFE 52
5.2 La méthodologie Gaia 59
5.3 La méthodologie Ingenias 64
5.4 La méthodologie PASSI 69
5.5 La méthodologie Tropos 73
5.6 Comparaison 76
5.7 Unification des métamodèles 77
6 Les plates-formes orientées Agent 78
6.1 La plate-forme ZEUS 78
6.2 La plate-forme JADE 79
6.3 La plate-forme MADKIT 80
6.4 La plate-forme AgentBuilder 81
7 Analyse et conclusion 82

4 Vers un développement flexible et sûr 85
1 Introduction 86
2 Problématique et objectifs de la thèse 87
3 Le développement dirigé par les modèles 89
3.1 Principes de l'approche 90
3.2 Les quatre niveaux de OMG 90
3.3 Les relations entre les modèles 93
3.4 La transformation de modèles 95
3.5 Processus de développement orienté modèles 97
3.6 Apports potentiels de l'approche orientée modèles dans l'ingénierie des systèmes multi-agents 100
4 Le développement orienté architecture 102
4.1 Définition d'une architecture logicielle 102
4.2 Mettre en œuvre l'approche orientée architecture . . . 103
4.3 Processus de conception orienté architecture 104
4.4 La notion de style architectural 106

4.5 Apports potentiels de l'approche orientée architecture dans l'ingénierie des systèmes multi-agents 108
5 Combinaison des deux approches 108
6 Conclusion 111

5 L'approche ArchMDE 113
1 Introduction 114
2 L'approche ArchMDE 114
2.1 Les modèles considérés 115
2.2 Le cycle de développement ArchMDE 119
3 Application du cycle de développement ArchMDE 120
3.1 Construction du cadre de développement par les experts de métamodélisation 121
3.2 Utilisation du cadre de développement par les développeurs 126
3.3 Les besoins relatifs à la mise en œuvre de l'approche . 130
4 L'environnement ArchWare 131
5 Les langages ArchWare 132
5.1 Le langage ArchWare ADL (π-ADL) 133
5.2 Le langage AAL 135
5.3 Le langage ArchWare ASL 135
5.4 Le langage ArchWare ARL 137
6 Les outils ArchWare 138
7 Formalisation du métamodèle du domaine 141
7.1 dataEntity 141
7.2 taskEntity 143
7.3 activeEntity 148
7.4 structureEntity 149
8 Le métamodèle orienté agent 151
8.1 Formalisation du style agent réactif 152
8.2 Définition d'une syntaxe 153
8.3 Propriétés d'un agent réactif 154
9 Le processus d'agentification 155
9.1 Extension du langage ARL 155
9.2 Le tissage des métamodèles 156
10 Génération de l'architecture 158
11 Validation des propriétés 158
12 Génération du code 162
13 Conclusion 162

6 Validation de l'approche ArchMDE - Agentification d'un système de production 167
1 Présentation du projet 168
2 Les systèmes de production 169

3 Métamodélisation d'un SdP 171
3.1 L'Entité Circulante (EC) 171
3.2 Le Système de Transformation du Produit (STP) . . . 172
3.3 Le Centre de Pilotage 178
4 Agentification du système de production 182
4.1 La ressource EC 183
4.2 Les agents hybrides STP-Stock et STP-Machine 184
4.3 Agentification des autres éléments du SdP 189
5 Conclusion 191

7 Conclusions et Perspectives 193
1 Rappel de la problématique 194
2 Contributions 196
2.1 Apports de l'approche IDM 196
2.2 Apports de l'approche centrée architecture 197
2.3 Apports de la démarche ArchMDE 197
2.4 Bilan 199
3 Perspectives 200

A Description des langages ArchWare 203
1 Le langage ArchWare ADL 204
1.1 Comportement 205
1.2 Les valeurs et les types 210
2 Le langage ArchWare AAL 212
2.1 Les opérateurs et les quantifieurs 212
2.2 Prédicats et fonctions sur les données 213
2.3 Prédicats sur les comportements 213
3 Le langage ArchWare ARL 215
4 Le langage ArchWare ASL 217
4.1 description générale d'un style 217
4.2 Types 218
4.3 Styles 218
4.4 Les constructeurs 219
4.5 Les contraintes 220
4.6 Les analyses 221

Bibliographie 223

Table des figures

2.1 La couche integiciel . . . 12
2.2 Le processus centré architecture . . . 20
2.3 L'évolution de développement des systèmes informatiques . . 24

3.1 L'architecture de Brooks . . . 32
3.2 L'architecture MANTA [DCF95] . . . 33
3.3 Réduction de la charge des réseaux . . . 35
3.4 Topologie à structure hiérarchique . . . 40
3.5 Topologie à structure de marché . . . 41
3.6 Topologie à structure de communauté . . . 41
3.7 Topologie à structure de société . . . 41
3.8 Les branchements définis dans AUML . . . 47
3.9 Une classification des méthodologies . . . 51
3.10 Le métamodèle d'ADELFE . . . 54
3.11 Le processus ADELFE . . . 58
3.12 Le métamodèle Gaia . . . 60
3.13 Exemple de modèle de rôle [ZJW03] . . . 61
3.14 Exemple de modèle d'interaction [ZJW03] . . . 62
3.15 Le processus de développement Gaia . . . 63
3.16 Un métamodèle Ingenias résumé [CBP05] . . . 65
3.17 Un résume du métamodèle Ingenias [Pav06] . . . 67
3.18 Le métamodèle PASSI [CBP05] . . . 70
3.19 Le processus PASSI [CP01] . . . 72
3.20 Le métamodèle Tropos . . . 73
3.21 Le métamodèle unifié [CBP05] . . . 77
3.22 Le métamodèle AALAADIN . . . 81

4.1 Les quatre niveaux de MDA . . . 92
4.2 La relation μ [BBB^{+}05] . . . 93
4.3 La relation χ . . . 94
4.4 Migration de modèle objet à un modèle relationnel . . . 97
4.5 Le cycle de vie de MDA . . . 98
4.6 Le cycle de vie en Y . . . 99

4.7 Un exemple de table de mapping PDM/PIM 100
4.8 Le processus centré architecture 105
4.9 Le processus centré architecture [Ley04] 107

5.1 Arbres de dérivation d'une expression arithmétique 116
5.2 La couche modèle 119
5.3 Le processus de développement ArchMDE 121
5.4 Définition des règles de transformation 123
5.5 Un métamodèle Orienté Agent 124
5.6 La phase d'analyse 128
5.7 Application des règles de transformation 129
5.8 Le langage ASL 136
5.9 Le langage ARL 138
5.10 Le processus de vérification effectué par Analyser 140
5.11 Le code ASL de dataEntity 142
5.12 Métamodèle d'Agent Réactif 152
5.13 Agentification d'une activeEntity en agentReactif 157
5.14 Processus de génération d'architecture 159
5.15 Génération d'un capteur dans un agent réactif 160
5.16 Vérification des propriétés 161
5.17 Vérification de l'interblocage 162

6.1 Modèle de Système de Production 170
6.2 Les trois fonctions fondamentales d'un STP 172
6.3 Modèle du CP 179
6.4 Le comportement du CP 181
6.5 Communication entre STP 188

Liste des tableaux

3.1 La notation AUML 48
3.2 Les concepts manipulés par AORML 49
3.3 Les concepts manipulés par OPM/MAS 50

4.1 Les apports relatifs à l'application de l'approche IDM dans le contexte multi-agents 101
4.2 Les apports relatifs à l'application de l'approche centrée architecture dans le contexte multi-agents 109
4.3 Combinaison de l'approche IDM et l'approche centrée architecture 110

5.1 Mapping entre métamodèle du domaine et métamodèle orienté agent 125
5.2 Mapping entre architecture en π-ADL et code Java 163

6.1 Agentification de l'Agent-CP 190

Chapitre 1

Introduction générale

1 Contexte et problématique

Les systèmes multi-agents constituent aujourd'hui une nouvelle technologie pour la conception et le contrôle de systèmes complexes. Les solutions proposées par les systèmes multi-agents sont prometteuses et permettent d'obtenir des systèmes flexibles et évolutifs. Cependant, leur mise en oeuvre reste difficile. Ceci est dû au manque de standardisation des techniques d'ingénierie adaptées à ce genre de système et qui permettent un développement fiable et cohérent.

Le cadre général de cette thèse est l'ingénierie des systèmes multi-agents. Nous nous proposons d'étudier les différents moyens mis à la disposition des développeurs pour construire des applications basées sur l'approche orientée agent. Au niveau de l'état de l'art, nous avons recensé différentes techniques d'ingénierie. Ainsi, nous distinguons :

- les méthodologies orientées agents qui fournissent un cadre conceptuel et méthodologique visant à guider les développeurs des systèmes multi-agents. Ces méthodologies structurent le processus d'ingénierie des systèmes multi-agents en proposant différentes activités qui doivent être menées lors du développement de ces systèmes. Certaines de ces méthodologies fournissent des notations adéquates permettant de spécifier les applications orientées agent. Elles fournissent aussi des outils supportant l'application de ces notations à différentes phases du cycle de développement,
- les langages de modélisation orientés agents qui étudient les différentes

notations pouvant être utilisées pour modéliser les systèmes multi-agents. Ces langages sont soit basés sur la notation UML, soit composés d'un ensemble de notations graphiques. Certains parmi ces langages sont incorporés dans des méthodologies,
– les plates-formes orientées agents qui sont des supports d'aide à la programmation. Elles fournissent une couche d'abstraction permettant de faciliter l'implémentation des concepts orientés agents.

L'étude des méthodologies, notations et plates-formes orientées agent ont fait ressortir plusieurs limites qui peuvent expliquer les difficultés des systèmes multi-agents à percer dans le monde industriel. Ces limites sont essentiellement liées à la diversité et parfois à l'ambiguïté qui entoure les concepts orientés agents. En effet, ces concepts sont abordés différemment au niveau de chaque méthodologie, notation et plate-forme orientée agents. Cette ambiguïté rend difficile la compréhension de la sémantique de ces concepts ainsi que leur application concrète. De plus, nous avons noté que ces techniques d'ingénierie ne couvrent qu'une vue partielle des systèmes multi-agents. De la même manière, les méthodologies existantes ne considèrent que certains aspects du cycle de vie (généralement l'analyse et la conception). La phase d'implémentation est, quant à elle, abordée par les plates-formes de développement. Cependant, un large fossé sépare les phases d'analyse/conception de la phase d'implémentation. Ce fossé est d'autant plus creusé par l'ambiguïté relative aux concepts puisque la sémantique adoptée au niveau de la méthodologie est généralement différente de celle appliquée au niveau de la plate-forme d'implémentation. Nous avons également constaté que la plupart des méthodologies sont basées sur l'approche orientée objet. Cependant, cette approche n'est pas bien adaptée car elle a un bas niveau d'abstraction par rapport aux concepts orientés agents et risque d'engendrer des systèmes fortement couplés puisque l'ingénierie des systèmes orientés objet se base généralement sur la desciption des invocations de méthodes [ZJW03]. Enfin, très peu d'outils sont fournis pour aider le développeur dans les différentes phases du cycle de vie.

Pour pallier ces limites, nous avons étudié les différentes avancées faites dans le domaine du génie logiciel pour aborder l'ingénierie des systèmes complexes basés sur des concepts ayant un haut niveau d'abstraction et composés d'éléments ayant une forte interaction. Nous avons retenu deux techniques d'ingénierie dédiées à des niveaux d'abstraction élevés :

– l'approche centrée architecture qui se concentre sur l'organisation des éléments qui composent le système. L'architecture offre une vue globale du système qui est lisible et qui expose les propriétés les plus cruciales ; ce qui permet un développement efficace.
– l'approche dirigée par les modèles (ou IDM) qui conçoit l'intégralité

du cycle de vie comme un processus de production, de raffinement itératif et d'intégration de modèles. L'utilisation des modèles permet de capitaliser les connaissances et le savoir-faire à différents niveaux d'abstraction qu'elles soient des connaissances du domaine métier ou du domaine technique.

Nous proposons dans cette thèse une démarche de développement basée sur la combinaison de ces deux approches afin d'adresser les différents aspects de développement des systèmes multi-agents.

2 Une démarche de développement flexible et sûre

Bien qu'il existe plusieurs propositions intéressantes aussi bien au niveau des méthodologies, des langages de spécification et des plates-formes d'implémentation, ces différentes propositions manquent de cohésion et font ressortir plusieurs différences aussi bien au niveau de la sémantique des concept utilisés mais aussi au niveau des démarches de développement.

Notre but durant cette thèse a été de proposer un cadre de développement flexible (prenant en compte la diversité des concepts orientés agent) et cohérent (conservant la sémantique des concepts tout au long du cycle de développement) se basant sur une combinaison de l'approche centrée architecture et de l'approche dirigée par les modèles. L'approche centrée architecture nous permet de raisonner sur les éléments qui structurent le système multi-agents ainsi que leurs interactions. Notre but en appliquant cette approche est d'identifier les patrons architecturaux nécessaires au développement des systèmes multi-agents en prenant en compte les différentes vues du système (vue organisationnelle, vue environnementale, etc.). L'approche orientée modèles nous permet d'exprimer de façon explicite la manière de combiner ces patrons architecturaux afin d'avoir une représentation globale du système multi-agents. D'autre part, IDM permet de couvrir les différentes phases du cycle de développement en adoptant une démarche basée sur la transformation de modèles. Cette démarche permet de garantir la cohérence du système durant les différentes phases du cycle de vie. Par ailleurs, celle-ci offre l'avantage de préserver le savoir-faire des développeurs en exprimant explicitement les opérations d'intégration (entre les patrons architecturaux) et de mapping (entre les modèles de conception et les modèles d'implémentation).

Pour appliquer notre approche, nous utilisons un cadre de développement formel. Le choix d'un cadre formel vise à réduire l'ambiguïté liée aux concepts multi-agents mais aussi à garantir une conception sûre afin de produire des logiciels de qualité. Ainsi, l'utilisation d'un langage formel donne la possibilité d'exprimer explicitement différentes propriétés. Les outils accompagnant ce langage permettent de vérifier et valider ces propriétés.

Nous adoptons au cours de cette thèse, le cadre de développement ArchWare qui est basé sur le π-calcul typé, polyadique et d'ordre supérieur [Mil99], ce qui permet de supporter les aspects communicatifs et évolutifs des systèmes multi-agents. Celui-ci offre différents langages, que nous allons utiliser pour spécifier les systèmes multi-agents. Ces langages sont accompagnés de différents outils qui nous seront utiles pour mettre en œuvre notre approche.

3 Organisation du document

Le suite de ce document est organisée en cinq chapitres, qui sont organisés comme suit :

- Chapitre 2 : ce chapitre situe le cadre général de la thèse. Nous distingons au niveau de ce chapitre la notion de paradigme de développement et celle de techniques d'ingénierie. La première partie de ce chapitre introduira les systèmes multi-agents et les positionne par rapport aux paradigmes de développement classiques. Au niveau de la deuxième partie, ce chapitre décrit l'évolution des techniques d'ingénierie. Cette description nous a permis de repérer les techniques d'ingénierie les plus adaptées au développement des systèmes multi-agents.
- Chapitre 3 : dans ce chapitre, nous passons en revue les différents langages de modélisation, méthodologies et plates-formes d'implémentation proposés dans le domaine des systèmes mult-agents. A la fin de ce chapitre, nous analyserons les avantages et les inconvénients de ces différentes techniques et nous introduirons les besoins liés à l'ingénierie des systèmes multi-agents.
- Chapitre 4 : la première partie de ce chapitre sera consacrée à une décription détaillée de la problématique et des objectifs de la thèse. Par la suite, nous décrirons et justifierons notre approche basée sur la combinaison de l'approche orientée architecture et de l'approche dirigée par les modèles.
- Chapitre 5 : ce chapitre décrira la mise en œuvre de notre approche que nous baptisons ArchMDE. La mise en œuvre sera d'abord décrite de manière générique, par la suite de manière plus spécifique en adoptant l'environnement ArchWare.
- Chapitre 6 : dans ce chapitre nous validons l'utilisation de ArchMDE en l'appliquant dans le cadre d'un projet BQR [HHP+07]. Ce projet étudie une nouvelle approche basée sur les systèmes multi-agents pour la simulation et le pilotage des systèmes de production. Nous utilisons notre approche ArchMDE pour concevoir un simulateur des systèmes de production appliquant le paradigme orienté agent.

Chapitre 2

Etude du domaine

1 Introduction

De nos jours, les applications informatiques sont de plus en plus complexes et difficiles à réaliser. Pour cette raison leur production doit être de plus en plus assistée et rigoureuse. Le génie logiciel est une discipline qui traite de la production et de la maintenance des logiciels. Elle a connu une grande évolution depuis l'apparition de l'informatique. Au niveau de cette discipline, des réflexions sont menées sur :

- la philosophie à adopter pour représenter les objets du monde réel sous une forme virtuelle. Cette philosophie est introduite par les **paradigmes de développement**,
- les cadres méthodologiques et techniques permettant de guider le développeur durant les différentes phases de production du logiciel. Nous appelons ces cadres **techniques d'ingénierie**.

Les techniques d'ingénierie sont différentes selon les paradigmes de développement appliqués. Dans le cadre de cette thèse, notre but est d'identifier les techniques d'ingénierie les plus adaptées aux paradigmes orientés agent ainsi que la manière de les appliquer.

Afin d'éviter toutes confusions, nous allons éclaircir dans ce chapitre la notion de paradigme de développement et celle de technique d'ingénierie. Nous positionnons, par la suite, le paradigme orienté agent par rapport aux paradigmes de développement classiques. Nous étudierons enfin, l'évolution des techniques d'ingénierie qui a accompagné celles des paradigmes de dé-

veloppement. Cette étude nous permettra d'identifier les techniques les plus appropriées pour appliquer le paradigme orienté agent.

2 Paradigmes de développement vs Ingénierie des systèmes logiciels

La notion de paradigme de développement et d'ingénierie de systèmes sont intimement liées. Pendant que l'un définit la philosophie générale appliquée tout au long du cycle de vie afin de représenter les objets du monde réel ; l'autre définit les techniques à utiliser afin d'appliquer ces paradigmes. Nous clarifions ces notions dans ce qui suit.

2.1 Les paradigmes de développement

Dès l'apparition des premiers ordinateurs, la fastidieuse entreprise d'écrire des instructions et des données sous forme binaire, a fait naître le besoin d'avoir un langage intermédiaire, facile à utiliser par l'homme et traduisible en langage machine. Depuis lors, plusieurs langages sont apparus, permettant de s'abstraire de plus en plus de la structure matérielle et améliorant le pouvoir expressif. Ces langages appliquent des concepts introduits par les paradigmes de développement.

Un paradigme est un mode de pensée qui permet de structurer nos connaissances, notre apprentissage et notre compréhension. Dans le contexte informatique, un paradigme de développement est une manière de représenter le monde dans le but de concevoir un programme informatique. Depuis l'apparition de l'informatique, plusieurs paradigmes ont été adoptés (paradigme procédural, paradigme orienté objet, etc.). Chaque paradigme identifie une classe de problèmes et fournit des techniques permettant de traiter ces problèmes. Par exemple, le problème traité par le paradigme procédural était de séparer la représentation des données du traitement qui va être effectué. Afin de régler ce problème, ce paradigme introduit un ensemble de concepts. Ceux-ci permettent de structurer les programmes informatiques.

Les concepts représentent des entités dénotées par un terme dans le langage et dont la sémantique permettra de décrire une partie de la philosophie introduite par le paradigme de développement.

Pour éclaircir ce point, prenons l'exemple du paradigme procédural. Séparer la représentation des données des traitements qui vont être effectués, ont mené à l'introduction des concepts de *structure de données* (pour représenter des données) et de *procédures*, *fonction* (pour représenter les traitement de données). Ces concepts ont été par la suite implantés dans divers langage de

programmation tel que Pascal, FORTRAN, C, etc.

2.2 Les techniques d'ingénierie

Les techniques d'ingénierie englobent les différents moyens mis à la disposition des développeurs afin de faciliter l'application des paradigmes. Une bonne application des paradigmes est conditionnée par le développement de techniques d'ingénierie appropriées fournissant des cadres conceptuels, technologiques et méthodologiques qui guident le développeur lors des activités de développement. Le développement d'une application informatique quelle qu'elle soit nécessite l'application de différentes activités allant de la collecte et l'analyse des besoins jusqu'au déploiement et la maintenance du système. Pour guider les développeurs lors de ces activités, des méthodologies sont proposées. Booch définie une méthodologie comme *"un ensemble de méthodes appliquées tout au long du cycle de développement d'un logiciel, ces méthodes étant unifiées par une certaine approche philosophique générale"* [Boo92]. L'approche philosophique générale est définie par les paradigmes de développement. Les méthodes quant à elles définissent la manière d'appliquer cette approche en fournissant des notations et des modèles permettant d'exprimer les spécifications relatives à chaque activité (modèle des besoins durant l'activité d'analyse, modèle architectural lors de l'activité de conception, code source lors de l'activité d'implémentation etc.). Les méthodes fournissent aussi des outils facilitant la production de ces systèmes. Les outils sont généralement intégrés dans des environnements de développement qui se composent d'éditeurs, d'outils de vérification, des machines virtuelles, etc.

Afin de mieux comprendre la problématique de cette thèse, nous allons dans ce chapitre, positionner le paradigme orienté agent par rapport aux principaux paradigmes de développement existants. Pour chaque paradigme, nous allons exposer les problématiques abordées ainsi que les solutions d'ingénierie qui ont été mises en place afin de faciliter son application. Cette étude nous permettra d'une part de montrer l'utilité du paradigme orienté agent dans le contexte informatique actuel ; d'autre part, de mieux cerner les besoins relatifs à l'ingénierie des systèmes basés sur ce paradigme. L'étude des différentes techniques d'ingénierie nous permettra, par ailleurs, d'identifier celles qui peuvent être réutilisées dans le contexte orienté agent.

3 L'évolution des paradigmes de développement

Différents paradigmes de développement ont vu le jour, introduisant de nouveaux concepts permettant de structurer les programmes informatiques. Les paradigmes de développement ainsi que les concepts qu'ils introduisent permettent d'augmenter le niveau d'abstraction afin de faciliter le dévelop-

pement. Le passage d'un paradigme à un autre est justifié par la nécessité d'encapsuler certains mécanismes de bas niveau rendant la tâche de développement moins complexe. Ainsi, les différents paradigmes de développement sont en fait complémentaires et s'inscrivent dans une logique de continuité.

3.1 Le paradigme procédural

Le paradigme procédural peut être considéré comme le paradigme de développement le plus ancien. Il a permis de s'affranchir du code machine et de renforcer une séparation entre la représentation des données et les traitements faits sur ces données. Une réutilisation des sous-routines (procédures) a été obtenue. Cependant, le développement des systèmes complexes utilisant ce paradigme a généré des structures en *plats de spaghetti*. En effet, l'interopérabilité entre les procédures se fait par les variables globales qui sont accessibles par différentes procédures. Ceci a engendré des effets de bord : si on oublie de mettre à jour certaines données partagées, si l'ordre chronologique des assignations par les différentes parties du logiciel est incorrect, ou si une zone de mémoire a été désallouée au mauvais moment, le programme se retrouve dans un état imprévu. Lors de la maintenance, la modification d'une procédure peut avoir des conséquences inattendues sur le résultat d'autres fonctions ou procédures *liées* par des variables globales. Ainsi, la maintenance des systèmes développés avec le paradigme procédural est difficile et coûteuse et la réutilisation du code se trouve souvent compromise. Un paradigme de développement plus modulaire est devenu nécessaire. C'est ainsi que le paradigme orienté objet a vu le jour.

3.2 Le paradigme orienté objet

Le paradigme orienté objet a permis une meilleure protection des données grâce à l'encapsulation des données et une meilleure réutilisation et extension des systèmes grâce à l'héritage. Mais devant la complexité sans cesse croissante des systèmes informatiques actuels, le paradigme orienté objet n'a pas réussi à limiter les difficultés de développement inhérentes aux systèmes distribués et concurrentiels compte tenu de sa faible granularité.

3.3 Le paradigme orienté composant

Le passage au paradigme orienté composant a permis d'augmenter le niveau d'abstraction et de changer la manière de développer les systèmes informatiques. Ceux-ci sont désormais construits par assemblage de composants logiciels réutilisables. L'objectif étant d'industrialiser la réalisation des systèmes informatiques comme c'est le cas dans d'autres secteurs tel que

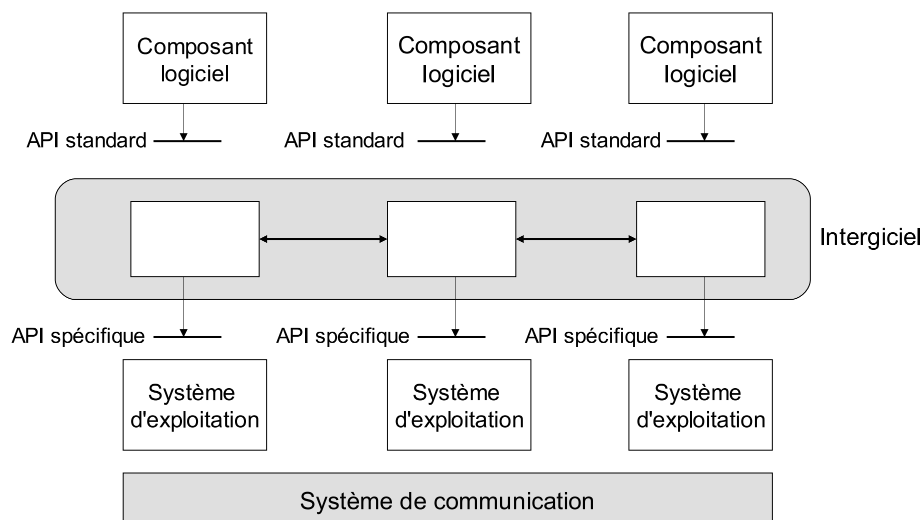


Fig. 2.1: La couche integiciel

l'automobile, la construction, etc. Bien que cet objectif soit en partie atteint, l'utilisation des intergiciels rend l'apporche difficile à appliquer. Les intergiciels visent à assembler les composants et les rendre interopérables (figure 2.1). Mais, leur prolifération et le manque de standardisation rend difficile leur application. L'administration des applications hétérogènes basées sur des composants distribués et utilisant des intergiciels hétérogènes soulève de nombreuses questions telles que le maintien de la cohérence, la sécurité, l'équilibre entre autonomie et interdépendance pour les différents sous-systèmes, la définition et la réalisation des politiques d'allocation de ressources, etc. De plus, la composition des composants est encore rigide et complexe ; basée uniquement sur les protocoles d'interaction et les flux d'entrée et de sortie des composants.

3.4 Le paradigme orienté service

Afin d'assouplir les lourdeurs dans la gestion des interopérabilités et réduire le couplage entre les composants, le paradigme orientée service a été proposé. Ce paradigme propose de découper les fonctionnalités d'une application en services métier, réutilisables dans d'autres applications. Un service métier peut être défini comme une fonction de haut niveau. Les opérations proposées par cette fonction encapsulent plusieurs autres fonctions et opèrent sur un périmètre de données large. Ainsi, les services ont un niveau de granularité plus élevé que les composants. La description d'un service est complètement indépendant des plates-formes et outils.

Dans le cas des services web, un service n'est autre qu'une application exposée par le biais d'une interface XML, connue sous le nom de *services Web*. Les services Web sont des couches d'invocation compréhensibles potentiellement par l'ensemble des systèmes. Les services Web interagissent les uns avec les autres par le biais d'envoi de messages. De ce fait, la modélisation de services Web met en jeu la description d'interactions et de leurs interdépendances, aussi bien d'un point de vue structurel (types de messages échangés) que d'un point de vue comportemental (flot de contrôle entre interactions). Trois types de modèles ont été distingués pour décrire les interactions entre les services :

- la chorégraphie : il s'agit de décrire les interactions entre les services Web sans avoir de coordinateur central,
- l'interface : il s'agit de décrire les actions d'envoi et de réception de message pour chaque service. Une interface permet de spécifier quelles sont les données nécessaires à l'exécution du service, et ce qu'il fournit en retour),
- L'orchestration : il s'agit de décrire un processus automatique d'organisation, de coordination, et de gestion des services Web.

Les applications développées selon le paradigme orienté services présentent de nombreux avantages parmi lesquels la flexibilité et la réactivité. Cependant, certains obstacles ne sont pas encore surmontés. Le chantier de standardisation relatif à l'orchestration et la chorégraphie est loin d'être achevé ce qui pose un problème d'interopérabilité. La mise en œuvre d'applications orientées service souffre d'un manque d'outils méthodologiques. Enfin, les services ne sont pas conscients de leur environnement et ne peuvent réagir en cas d'imprévus.

4 Positionnement du paradigme orienté agent

Bien qu'il soit né dans les années 80, le paradigme orienté agent reste encore d'actualité puisqu'il propose des mécanismes d'abstraction qui permettraient probablement de résoudre plusieurs problématiques rencontrées actuellement. En effet, les recherches menées autour de ce paradigme élaborent des mécanismes pour la construction de systèmes complexes évoluant dans un environnement dynamique et ouvert (c'est-à-dire dont la cardinalité et la topologie des composants le constituant peut être variable). Ces systèmes complexes comportent un grand nombre de composantes en interaction. L'organisation de ces composantes peut être *à priori* inconnue. Le contôle de tels systèmes ne peut pas se faire de manière centralisée.

Pour traiter ce type de système, le paradigme orienté agent propose une nouvelle philosophie de développement qui va au-delà d'une simple décom-

position conceptuelle ou fonctionnelle comme il se fait avec les paradigmes de développement actuels. Suivant le paradigme orienté agent, les composantes du système appelées *agents* sont des entités [Fer95], [Pic04], [JSW98] :

- autonomes : dont le comportement est guidé par des objectifs,
- réactives : capables de percevoir leur environnement et de réagir face aux évènements survenus dans celui-ci,
- proactives : capables d'agir sur leur environnement,
- sociales : capables d'interagir et de communiquer avec d'autres agents afin de coopérer en vue d'atteindre un but global.

D'autres caractéristiques peuvent être imputées aux systèmes multi-agents ; selon le contexte, ces caractéristiques ne sont pas toutes nécessaires. L'intégration de ces caractéristiques dans le système final dépend des besoins fonctionnels et non fonctionnels liés à chaque domaine. Nous recensons par exemple :

- le raisonnement : un agent peut décider quel but poursuivre ou à quel événement réagir, comment agir pour accomplir un but, suspendre ou abandonner un but pour se dédier à un autre,
- l'apprentissage : l'agent peut s'adapter progressivement à des changements dans des environnements dynamiques grâce à des techniques d'apprentissage,
- la mobilité : dans certaines applications déterminées il peut être intéressant de permettre aux agents de migrer d'un noeud à un autre dans un réseau tout en préservant leur état lors de leur migration.

Dans ce qui suit, nous allons expliquer comment ces caractéristiques peuvent être utiles pour répondre à certaines problématiques actuelles d'interopérabilité et d'évolution dynamique. Nous allons ensuite présenter les principaux freins empêchant une utilisation à grande échelle de ce paradigme.

4.1 Les problématiques traitées par le paradigme orienté agent

Dans le contexte actuel, le paradigme orienté agent trouve son utilité en s'attaquant à la modélisation de systèmes ouverts, flexibles et auto-adaptables. En effet, le paradigme orienté agent cherche à interconnecter des ressources logicielles de façon à créer un système intégré permettant de résoudre des problèmes complexes et de s'auto-adapter à différentes situations. Ainsi, les systèmes multi-agents visent à garantir :

- l'extensibilité : elle peut correspondre à de nouveaux objectifs pour le système, ce qui induit l'intégration de nouvelles entités qui vont coopérer avec celles qui existent (dans ce cas, les capacités sociales

des agents sont très utiles). Il pourrait s'agir aussi d'ajouts progressifs d'hypothèses sur des comportements locaux au sein des agents (ceci est rendu possible grâce notamment aux capacités cognitives des agents c'est-à-dire les capacités de raisonnement et d'apprentissage),
- la portabilité : c'est-à-dire l'aptitude d'un logiciel à être transféré d'un environnement à un autre. Les systèmes multi-agents sont axés sur l'interaction. Celles-ci se basent sur l'utilisation de langages de communication normalisés tels que ACL/KQML qui décrivent la sémantique des messages. Ainsi, les interactions dépendront de la compréhension du contenu des messages et non de la configuration des messages échangs. Ceci réduit fortement le couplage entre les composantes des systèmes,
- la performance : celle-ci est liée aux possibilités intrinsèques aux agents, d'exécution parallèle et d'asynchronisme. La technologie agent doit aussi fiabiliser l'exploitation des ressources disponibles en traitant les cas de concurrence entre agents,
- la robustesse et la tolérance aux pannes : celles-ci peuvent être garanties par les capacités d'adaptation des agents aux différentes situations qui peuvent surgir. Des études sont menées pour permettre à l'agent de diagnostiquer les pannes et d'adapter son comportement par le choix des meilleures manières d'atteindre les buts en fonction du contexte.

Nous pouvons donc conclure que le paradigme orienté agent vise à trouver des mécanismes permettant de simplifier l'utilisation et l'intégration de plusieurs systèmes hétérogènes. Il permet de réduire encore plus le couplage entre les différentes entités du système puisque les agents ont des capacités cognitives leur permettant de gérer leurs interactions. Les agents se distinguent aussi par leur capacité de perception de l'environnement. Cette capacité n'est pas prise en compte de manière explicite dans les autres paradigmes de développement où la distinction entre entités actives du système (qui vont correspondre aux agents) et entités passives du systèmes (qui vont correspondre aux ressources) est peu explicitée [ZJW03].

De cette manière, le paradigme orienté agent peut permettre un développement plus aisé des applications du futur telles que celles traitées au niveau de l'informatique diffuse (*l'Ubiquitous Computing*). Ce domaine utilise la large diffusion des processeurs dans les appareils de la vie courante (comme les chaînes hifi, les téléphones, les systèmes domotiques, etc.) pour proposer des fonctionnalités avancées obtenues par la composition de leurs fonctions en réponse aux désirs des utilisateurs : par exemple écouter du jazz avec une lumière tamisée de manière automatique tout en coupant le téléphone, etc. Seulement la fonctionnalité d'un tel système dépend énormément des utilisateurs, donc de l'environnement, et reste difficilement spécifiable avec les paradigmes classiques. Les services Web ne permettant pas de raisonner sur l'interaction avec l'environnement, il est plus aisé de traiter ce type de

système avec le paradigme orienté agent.

Le paradigme orienté agent s'est avéré utile dans d'autres domaines d'application. Nous pouvons les classer selon trois familles [BGG04a] :

- les systèmes purement informatiques, utilisés dans le domaine des télécommunications et de la simulation. On y trouve notamment : les supports d'aide à la décision, le commerce électronique, la supervision de réseaux, la simulation de systèmes sociaux et naturels, etc.
- les systèmes sociaux nécessitant une forte interaction avec l'être humain, tels que les collecticiels, le commerce électronique, la recherche d'information, l'automatisation des processus industriels ou administratifs etc.
- les systèmes mécaniques qui intègrent des composantes mécaniques opérant dans le monde réel, tels que la robotique, les systèmes industriels, les applications embarquées et sans fil etc.

Ainsi, les systèmes multi-agents peuvent être utiles dans plusieurs contextes. D'ailleurs, certains systèmes industriels tels que la gestion des ressources hydrauliques [BAP02], le contrôle du trafic aérien [LL92], les télécommunications [EQ96] ont appliqué ce paradigme. Cependant, il existe encore plusieurs freins empêchant leur utilisation à grande échelle. Nous allons décrire, dans ce qui suit, les principaux freins.

4.2 Les freins à l'utilisation des systèmes multi-agents

Bien que certaines applications industrielles appliquant le paradigme orienté agent aient commencé à émerger ; il manque encore l'expérience pour la conception et la construction de ces systèmes. La grande majorité des systèmes multi-agents ont été construits de manière *ad hoc* sans utiliser des composants agents réutilisables. Il devient donc nécessaire de développer des techniques d'ingénierie (méthodologies, méthodes, notations spécifiques, etc.) facilitant la réalisation de ces systèmes. Ces techniques d'ingénierie seront d'autant plus importantes qu'elles auront un rôle pédagogique. En effet, en dehors du milieu académique, très peu d'experts du paradigme orienté agent existent de par le monde. Ce genre de formation est dispensé au niveau du troisième cycle ; le manque de techniques d'ingénierie ne facilitant pas l'acquisition des compétences nécessaires à l'application de ce paradigme.

Dans le monde académique, plusieurs travaux sont proposés, tentant d'absorber ce problème. Ces travaux proposent généralement :

- des méthodologies de développement [IGG99, BCG+04, BKJT97, CP01, BPSM05, WJK00] qui permettent de guider les développeurs durant les phases de développement,

- des plates-formes orientées agents [BPR99, CNNL98, FG00] qui permettent de faciliter la phase d'implémentation,
- des langages de modélisation permettant de décrire des modèles selon les concepts orientés agents [BMO01, CLC+01, SDS03, WT03].

La diversité et le manque de maturité des travaux menés font que la plupart ne soient utilisés que dans le milieu académique. La difficulté provient essentiellement du manque de consensus concernant la sémantique des concepts orientés agent ainsi que leur utilisation concrète durant le cycle de développement. Par exemple, il arrive souvent que certains concepts soient supportés par certaines méthodologies ou plates-formes alors qu'ils sont complètement omis par d'autres. Ces omissions peuvent constituer un réel problème pour le développeur qui se voit obligé de modéliser ou d'implémenter certaines parties du système de manière *ad hoc*. A cela se rajoute la quasi inexistence de liens entre les méthodologies et les plates-formes d'implémentation. Le passage des modèles vers le code devient alors une lourde tâche réalisée pour la plupart du temps de manière *manuelle* par le développeur. Il devient alors nécessaire d'adopter une approche de développement plus cohérente permettant d'intégrer les différentes facettes du système multi-agent et de faciliter le travail des développeurs tout au long du cycle de vie. Ceci constitue le but de cette thèse. Pour y arriver, nous étudions dans la prochaine section les différentes techniques d'ingénierie existantes et choisir celles nous permettant de mieux gérer l'ingénierie des systèmes multi-agents.

5 L'évolution des techniques d'ingénierie

Dans les années 80, le développement était principalement considéré sous l'angle de la programmation. Il était alors effectué en interne par une seule personne ou un petit groupe de personnes. On parlait de *"programming in the small"* [DK75]. Actuellement, les systèmes sont amenés à devenir des systèmes ouverts et coopératifs, en constante interaction les uns avec les autres. Nous sommes donc passés à l'ère du *"programming in the large"*[DK75] où le développement est considéré à large échelle c'est-à-dire effectué par plusieurs équipes, parfois géographiquement éloignées et faisant coopérer plusieurs applications distribuées. Ce type de développement ne peut plus se concentrer uniquement sur la phase d'implémentation. De nouveaux outils deviennent alors nécessaires pour faciliter le travail de développement.

5.1 Les techniques d'ingénieries classiques

Depuis l'avènement du paradigme procédural, plusieurs environnements de développement intégrés (EDI) ont vu le jour. Ces environnements de développement aidaient les programmeurs à structurer leurs applications lors

de la phase d'implémentation. La première génération d'EDI ne supportait qu'un seul langage à la fois. Avec les mutations technologiques des années 90 (application de plus en plus étendue du paradigme orienté objet), plusieurs ajouts fonctionnels améliorant la productivité des développeurs ont été proposés. Ainsi, les environnements de développement intégrés ne sont plus dédiés à un seul langage mais à un ensemble de langages. C'est certainement, l'environnement Eclipse qui a le plus marqué cette mutation. Son but étant de fournir une plate-forme modulaire pour permettre le développement informatique de systèmes complexes et de grande taille. Cet environnement, écrit en Java, est extensible, universel et polyvalent, permettant de créer des projets mettant en œuvre n'importe quel langage de programmation.

Malgré l'efficacité des environnements de développement de deuxième génération ainsi que leurs différentes possibilités, leur aide reste limitée, dans la mesure où ils n'ont pas la capacité de couvrir la totalité du cycle de développement. En effet, ils restent pour la plupart dédiés à la phase d'implémentation. Une mutation actuelle des environnements de développement vise à intégrer les autres phases de développement telles que l'analyse, la conception et la validation des systèmes. Le principal but recherché est la génération automatique du code à partir de modèles conceptuels vérifiés et validés. Pour atteindre ce but, les EDI devraient permettre de couvrir la totalité du cycle de développement ce qui n'est pas actuellement le cas. Les premières phases de développement telles que l'analyse et la conception sont gérées par les Ateliers de Génie Logiciel (aussi appelés outils CASE - Computer Aided Software Engineering).

Les ateliers de génie logiciel (AGL) sont apparus dans les années 80 pour mieux structurer le travail de développement de logiciels durant les phases préliminaires de développement (principalement l'analyse et la conception). Ils intègrent un ensemble d'outils permettant de saisir les spécifications relatives au système en utilisant un formalisme graphique ou textuel. Le but recherché par ces outils est de pérenniser le savoir-faire en gardant une trace des spécifications réalisées durant les différentes phases de développement. Selon les outils disponibles dans l'environnement, ces spécifications peuvent être visualisées, validées et documentées par des rapports ou des dictionnaires de données. Certains AGL fournissent aussi des générateurs de code qui à partir des spécifications peuvent générer une partie du code (généralement c'est le squelette du code) dans un langage de programmation choisi.

L'utilisation des AGLs a connu un grand engouement avec l'apparition du paradigme orienté objet et surtout l'introduction du formalisme UML (Unified Modelling Language) [BRJ00] qui est un langage unifié de spécifications des applications orientées objet. UML est complètement indépendant de tout langage de programmation mais il intègre tous les concepts communs des

langages objets (héritage, appel de méthodes etc.). La simplicité du langage UML a vite conquis les industriels qui l'utilisent pour spécifier leurs applications. Cependant, malgré l'existence du standard UML, les spécifications sont produites pour documenter l'application et l'essentiel du développement reste concentré sur l'activité de codage. De plus, l'avènement des nouveaux paradigmes de développement a montré les limites de ces techniques d'ingénierie dans la modélisation de certains aspects tel que l'interaction entre composants logiciels, la gestion de la concurrence et la synchronisation.

5.2 L'ingénierie des systèmes basés sur le paradigme orienté composant

L'application du paradigme orienté composant nécessite la spécification du système à un niveau d'abstraction plus élevé que les concepts objet. Un raisonnement approfondi sur les composants à intégrer dans le système, la manière de les assembler et les contraintes qui doivent être respectées durant son exécution ainsi que celles qui doivent être conservées lors de son évolution est nécessaire. C'est l'architecture logicielle qui permet de représenter ces différents aspects. Au cours du développement d'un système, sa compréhension diminue tandis que le coût dû aux erreurs augmente. Le but d'un développement centré architecture (c.f. figure 2.2) est de préserver la compréhension du système au cours du temps, de détecter et de corriger les erreurs le plus tôt possible. Ainsi, d'après la définition fournie par d'ANSI/IEEE [Gro00], une architecture logicielle ne se contente pas de décrire uniquement l'organisation des composants du système en précisant leurs relations ; l'architecture doit être un cadre pour la satisfaction du cahier des charges. Elle est la base technique pour la conception comme la base gestionnaire pour l'estimation des coûts et du processus de gestion. Le but visé lors de la description de l'architecture est de permettre la pérennisation du savoir-faire et l'augmentation de la réutilisation.

Pour représenter ces différents aspects, on assiste, au début des années 90, à l'établissement d'un domaine consacré au développement centré architecture [PW92] [GS93]. Dans ce domaine, les recherches sont axées sur l'étude des langages et formalismes nécessaires à la représentation et la documentation des architectures logicielles.

Actuellement, il n'existe pas de notation unique standardisée permettant de représenter les architectures logicielles. Certains auteurs voient en UML un formalisme pertinent permettant de décrire différentes vues du système [Kru95]. Par exemple, Medvidovic *et al.* [MRRR02] proposent différentes manières de modéliser les architectures logicielles en UML (version 1.5). Il s'est avéré, à la suite de cette étude, que l'utilisation d'UML tel quel deve-

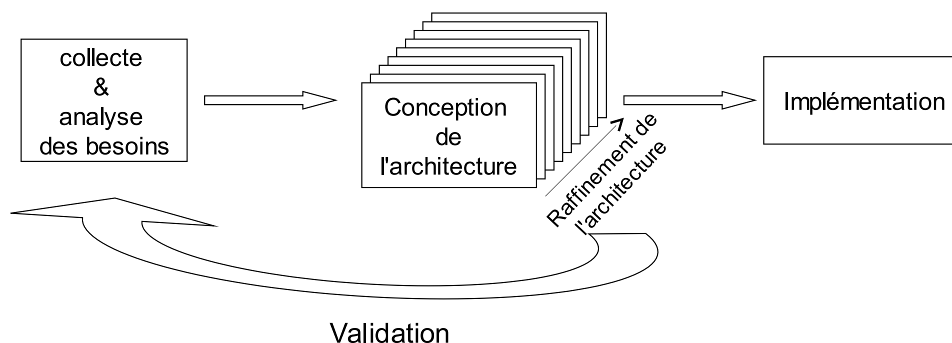


FIG. 2.2: Le processus centré architecture

nait assez complexe, du fait qu'UML ne fournissait pas de moyens simples permettant de décrire les composants, qui avaient un niveau d'abstraction plus élevé que les objets. Il a été alors nécessaire d'étendre UML avec le mécanisme des stéréotypes, les valeurs marquées et les contraintes OCL. Une deuxième approche a aussi été testée par ces mêmes auteurs qui consiste à étendre le langage UML avec de nouveaux concepts (méta-classes) pour les composants, les ports, les connecteurs, etc. L'utilisation des stéréotypes mais aussi des méta-classes a pour inconvénient de ne pas être conforme aux spécifications standardisées d'UML. De ce fait, les outils fournis par les environnements basés sur UML ne permettent pas de gérer la description d'architectures logicielles.

A l'opposé des formalismes basés sur UML, d'autres types de formalismes ont vu le jour ; ce sont des Langages de Description d'Architecture (appelés en anglais *Architecture Description Languages* ou ADL). Nous allons ici adopter cette appellation anglaise ADL, fréquemment utilisée dans la littérature. Un ADL fournit les abstractions selon lesquelles les architectures vont être décrites. Généralement, il décrit une architecture logicielle sous la forme de composants, de connecteurs et de configurations [MT00]. Les composants représentent les unités de calcul et de stockage de données dans un système logiciel. L'interaction entre ces composants est encapsulée par les connecteurs. Dans la configuration, l'architecte instancie un nombre de composants et un nombre de connecteurs. Il les lie entre eux afin de construire son système.

Il existe une multitude d'ADL dans le monde académique et industriel. Quelques exemples sont Wright, Darwin, ACME, Unicon, Rapide, Aesop, C2 SADL, MetaH, Archware, etc. Les concepts de composants, connecteurs et configurations sont abordés de manière différentes selon le langage. En effet, il n'existe pas, de nos jours de standards fixant une sémantique exacte à ces

concepts. De plus, chaque ADL se focalise sur des aspects particuliers de la conception d'architectures logicielles. Par exemple, Darwin [MDEK95] est plus axé sur la modélisation d'architectures distribuées et la prise en compte de leur reconfiguration dynamique. WRIGHT [All97] s'intéresse à la formalisation des connexions entre composants, tandis que Acme [GMW00] ou ArchWare représentent des exemples d'ADL à objectif général, c'est-à-dire offrant des abstractions génériques permettant de décrire différentes sortes d'architectures.

Certains ADL sont semi-formels (basés sur des notations graphiques normalisées) alors que d'autres sont strictement formels (disposant de règles syntaxiques et sémantiques ne souffrant d'aucune ambiguïté). Selon le type de langage, les outils intégrés dans les environnements basés sur les ADLs peuvent changer. Un langage formel donnera plus de possibilité à manipuler les spécifications exprimées. En effet, les langages formels permettent d'exprimer des propriétés sur la structure et le comportement du système. Dans ce cas, les environnements de développement basés sur ces langages offrent des outils permettant de vérifier et valider les propriétés. Etant donné que leurs syntaxes et sémantiques sont formelles, certains environnements offrent aussi des simulateurs (appelés aussi animateurs) permettant à l'architecte de simuler et valider le comportement décrit au niveau de l'architecture. D'autres environnements proposent des machines virtuelles permettant d'interpréter les spécifications décrites par l'ADL fourni. Enfin, certains environnements orientés architecture offrent la possibilité de générer automatiquement l'application correspondante (éventuellement après plusieurs étapes de raffinement).

En se basant sur les avancées faites au niveau des ADL, la nouvelle version 2.0 du langage UML apporte des avancées significatives à la modélisation des systèmes à base de composants. Dans cette nouvelle version, un composant peut être modélisé à tous les niveaux du cycle de développement. La structure interne du composant est cachée et accessible uniquement à travers les interfaces fournies. Un modèle de composant peut également comporter une vue interne (ou boîte blanche) sous la forme d'un ensemble de *classifiers* réalisant le comportement du composant (diagramme de structure composite).

Les ADLs offrent de larges possibilités de modélisation. Ils ont permis un nouveau mode de développement axé sur la phase de conception plutôt que sur la phase d'implémentation. L'utilisation de ces langages pour la modélisation des systèmes actuels est de plus en plus explorée.

5.3 L'ingénierie dirigée par les modèles

En parallèle du développement centré architecture, une autre technique d'ingénierie a vu le jour qui est l'Ingénierie Dirigée par les Modèles (IDM) ou Model Driven Engineering (MDE) en anglais. Ce domaine vise à proposer une approche unificatrice basée sur l'utilisation systématique des modèles tout au long du cycle de développement. Comme nous l'avons décrit depuis le début de cette section, chaque paradigme de développement a été accompagné par des langages de spécification permettant de représenter et de raisonner sur le système de manière abstraite. La nouveauté dans l'approche IDM consiste à passer des modèles purement *contemplatifs* aux modèles *productifs*[BBB$^+$05]. En effet, les techniques d'ingénierie traditionnelles présentées tout au long de cette section ont souvent été délaissées ou appliquées de manière peu rigoureuse durant le cycle de développement. Les modèles étaient produits dans le but de documenter l'application en cours de développement afin de faciliter la compréhension et la communication entre les différents participants au projet de développement. Cependant, ces modèles n'avaient pas de rôle productif, dans le sens où ils étaient délaissés dès que l'activité de codage commençait. Cette activité constituait le cœur du développement, le code étant considéré comme le seul représentant fiable du logiciel. Il arrivait souvent que des décisions conceptuelles soient prises (ou soient modifiées) au cours de l'activité de codage sans que les modèles soient remis à jour.

IDM vise à fournir un cadre conceptuel, technologique et méthodologique dans lequel les modèles sont au centre des activités de développement. Comme le souligne Jean-Marc Jézéquel [JGB06], chaque processus de développement, quel que soit son type, comporte un certain nombre de phases (comme l'expression des besoins, l'analyse, la conception, l'implantation ou encore la validation). Dans chaque phase un certain nombre de modèles (appelés aussi artefacts) sont produits qui peuvent prendre la forme de documents, de diagrammes, de codes sources, de fichiers de configuration, etc. Ces modèles qui représentent les différentes vues du système, sont souvent produits de manière indépendante. Il est alors important de s'assurer de la cohérence entre ces différentes vues. Ceci constitue le cheval de bataille du domaine de l'ingénierie dirigée par les modèles.

Pour ce faire IDM propose de faire évoluer l'usage de ces modèles en conservant une traçabilité entre les éléments des différents modèles et ceci quel que soit leur niveau d'abstraction. Il devient alors indispensable de faire un travail de fond sur la sémantique des modèles mais aussi la sémantique des langages avec lesquels ils sont exprimés. Il est aussi nécessaire de travailler sur les techniques de composition, de tissage et de transformation de modèles. Par exemple, il est utile de pouvoir exprimer explicitement les règles permettant le passage de l'expression des besoins vers les modèles concep-

tuels ou architecturaux ainsi que celles permettant le passage des modèles conceptuels ou architecturaux vers le code. Un autre volet du travail consiste à réaliser des avancées dans l'expression de propriétés qui seront attachées aux modèles, leur préservation ou modification lors des opérations de composition/tissage/transformation avec d'autres modèles.

L'évolution du développement logiciel vers une vision centrée modèles nécessite donc des avancées en matière de formalismes et outils supports :
- aux transformations de modèles,
- à la vérification d'adéquation entre les langages de modélisation et les applications elles-mêmes,
- de compositions des modèles

Une définition précise du processus de développement à base de modèles est aussi nécessaire précisant les artefacts devant être produits à chaque phase de développement, ainsi que les liens qu'ils peuvent avoir avec ceux produits dans les autres phases. Cette approche pose explicitement les problèmes rencontrés dans l'application des méthodes d'ingénierie précédentes. De plus, cette approche est basée sur l'intégration de modèles décrivant des vues différentes, ce qui fait d'elle une approche unificatrice pouvant intégrer des domaines différents.

5.4 Conclusions relatives à l'évolution des techniques d'ingénierie

L'évolution des techniques d'ingénierie a suivi celle des paradigmes de développement cf. figure 2.3. Avec l'augmentation des niveaux d'abstraction traités, les techniques d'ingénierie devaient s'adapter à de nouveaux besoins. Tout d'abord, la passage d'un développement interne à un développement sur une petite échelle (utilisant le paradigme procédural) a nécessité l'utilisation d'EDI de première génération (pour la phase d'implémentation) et d'AGL orientés base de données (utilisés essentiellement au cours des phases d'analyse/conception). Par la suite, l'utilisation du paradigme orienté objet s'est accompagné par la possibilité d'intégrer des applications développées avec différents langages de programmation. Les EDI de deuxième génération sont apparus accompagnés d'AGL plus sophistiqués basés sur le paradigme objet. L'avènement du paradigme orienté composant a permis une *"industrialisation"* de la production des logiciels en orientant le développement vers une intégration des composants. Ceci a dirigé les techniques d'ingénierie vers la prise en compte des architectures logicielles dès les premières phases de développement. Les architectures logicielles permettent de spécifier des systèmes à dynamique connue, c'est-à-dire que les interactions entre les composants sont spécifiées *à priori*. Cependant, cette approche a permis un développement à grande échelle où les composants logiciels sont produits par des équipes et organismes différents et rassemblés par la suite pour for-

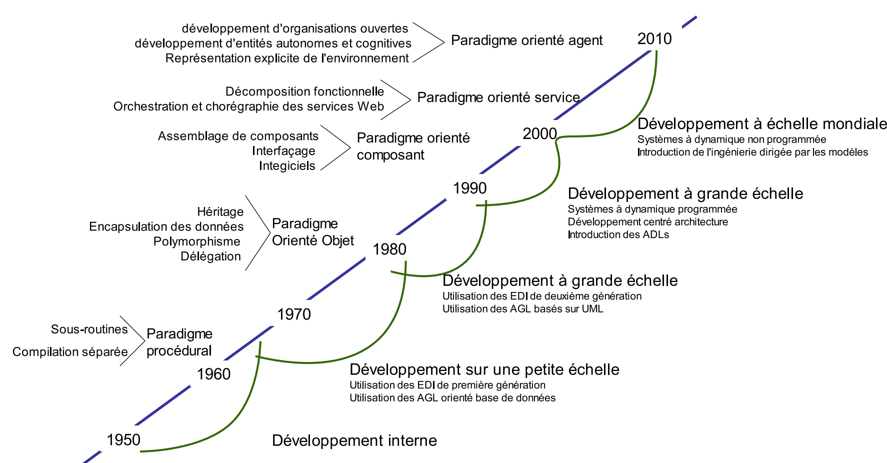


Fig. 2.3: L'évolution de développement des systèmes informatiques

mer une application. Cependant, à partir des années 2000, le développement des systèmes prend une autre tournure. Il s'oriente vers la prise en charge de systèmes à dynamique non programmée dont l'intégration doit être plus flexible afin de prendre en compte l'évolution rapide des processus métier. A ce niveau, les paradigmes orientés agent semblent être les plus adéquats pour régler cette problématique. Au niveau des techniques d'ingénierie, les travaux s'orientent vers une approche basée sur l'ingénierie dirigée par les modèles où un travail de fond est mené pour fixer la sémantique des modèles produits et la manière de les intégrer. Ainsi, cette technique d'ingénierie semble être la plus adaptées pour répondre aux problématiques soulevées dans le domaine des systèmes multi-agents (c.f. section 4.2).

6 Introduction à la problématiques de la thèse

Comme nous l'avons mentionné précédemment, l'un des plus grand frein à l'utilisation du paradigme orienté agent est le manque de techniques d'ingénierie standardisées qui facilitent son application. Les recherches dans ce domaine est en plein foisonnement, conduisant à l'élaboration d'un thème de recherche appelé *AOSE (Agent Oriented Software Engineering)*. Durant les conférences organisées autour de ce thème, plusieurs méthodologies, métodes, notations et plates-formes d'implémentation ont été proposées. Nous allons détailler quelques-uns de ces travaux dans le prochain chapitre. Cependant, celles-ci souffrent de plusieurs limites qui peuvent expliquer les difficultés des systèmes multi-agents à percer dans le monde industriel.

Afin de pallier ces limites, nous pensons qu'il est judicieux de tirer profit

des avancées réalisées au niveau des techniques d'ingénierie récentes. Les systèmes multi-agents traitent des systèmes complexes basés sur des concepts ayant un haut niveau d'abstraction. L'application de ces concepts peut faire appel à d'autres paradigmes de plus bas niveau tel que les objets, les composants et les services Web. De plus, le paradigme orienté agent est destiné à développer des systèmes complexes ayant une forte interaction, aussi bien entre les composantes du système qu'avec l'environnement. Dans ce contexte, il est nécessaire de pouvoir exprimer et vérifier les propriétés fonctionnelles et non fonctionnelles liées à ces systèmes. Ces propriétés devraient être concervées tout au long du cycle de vie.

Dans ce contexte, les techniques d'ingénierie les plus adaptées nous semble être :

- l'approche centrée architecture qui se concentre sur l'organisation des éléments qui composent le système. L'architecture offre une vue globale du système qui est lisible et qui expose les propriétés les plus cruciales ; ce qui permet un développement efficace.
- l'approche dirigée par les modèles (ou IDM) qui conçoit l'intégralité du cycle de vie comme un processus de production, de raffinement itératif et d'intégration de modèles. L'utilisation des modèles permet de capitaliser les connaissances et le savoir-faire à différents niveaux d'abstraction qu'elles soient des connaissances du domaine métier ou du domaine technique. Elle couvre ainsi les différentes vues du système.

Dans la suite de ce document, après une présentation de l'état de l'art des approches d'ingénierie orientées agent, nous allons expliquer comment nous pouvons utiliser l'approche centrée architecture et l'approche dirigée par les modèles pour développer des systèmes basés sur le paradigme orienté agent. Nous allons analyser l'apport de ces différentes approches et explorer leurs principaux inconvénients. Par la suite, nous allons exposer notre approche basée sur les techniques d'ingénierie citées ci-dessus.

Chapitre 3

Ingénierie des Systèmes Multi-agents

1 Introduction

Actuellement, une large panopli de travaux s'interesse au paradigme orienté agent. Tout au long de ce chapitre, nous allons les passer en revue. Dans la première section, nous exposerons les différentes théories liées à l'application du paradigme orienté agent. Ces théories englobent des modèles, des concepts et des notions ayant été minutieusement étudiés ou appliqués par certaines équipes de recherche. Ces théories visent à être réutilisables et à fournir un référent pour les développeurs.

Les théories orientées agent modifient les procédés de production de logiciel. Afin de mieux appréhender l'application de ces théories, un nouvel axe de recherche appelé AOSE (Agent Oriented Software Engineering) est né. Celui-ci est complètement dédié à la recherche des bonnes pratiques de développement qui aident à l'élaboration d'un système multi-agents. Cet axe de recherche couvre plusieurs aspects [BCP05] particulièrement liés à la dimension génie logiciel (et plus précisément aux techniques d'ingénierie). Ceci inclut la recherche de nouvelles méthodes, méthodologies, langages et outils adéquats nécessaires à l'analyse, la conception, l'implémentation, ainsi que la vérification et la validation des systèmes basés sur le paradigme orienté agent. Dans ce chapitre nous allons particulièrement nous intéresser aux :

- langages spécifiques au paradigme agent qui facilitent la spécification des systèmes multi-agents,
- méthodologies orientées agent qui guident les développeurs durant les phases de développement,

- plates-formes multi-agents qui facilitent l'implémentation et le déploiement des systèmes multi-agents.

Pour chaque technique, nous avons établi un cadre de comparaison qui nous permet de différencier les différents travaux. A la fin de ce chapitre, après une analyse de ces travaux, nous identifierons les différentes problématiques soulevées (et non encore résolues) liées à l'ingénierie des systèmes multi-agents.

2 Les théories orientées agent

Les systèmes développés selon le paradigme agent peuvent être considérés comme plus complexes que ceux basés sur les paradigmes orientés objets, orientés composants et orientés services. Cette complexité vient du fait que les systèmes multi-agents couvrent plusieurs vues et plusieurs niveaux d'abstraction. Pour gérer cette complexité, il est préférable de décomposer les systèmes multi-agents. Pour présenter de manière structurée les théories orientées agent, nous adoptons dans cette section, une décomposition basée sur quatre vues principales [Dem01] :

- une vue Agent,
- une vue Environnement,
- une vue Interaction,
- une vue Organisation.

Différentes théories ont été proposées pour l'élaboration de chaque vue. Ainsi, plusieurs modèles ou types d'agents existent. L'environnement peut être considéré sous plusieurs angles. Les interactions peuvent être modélisées par des protocoles réutilisables dans différents contextes. Enfin, pour élaborer l'organisation, plusieurs notions doivent être prises en compte tel que l'émergence, l'auto-adaptation, la topologie, etc. Dans ce qui suit, nous exposerons, les différentes théories liées à chaque vue.

2.1 La vue agent

Il n'existe pas, actuellement, une définition de l'agent qui fasse l'unanimaté. Pour avoir une bonne vision du concept agent, nous confrontons les définitions les plus utilisées dans la littérature. La plupart des travaux francophones font référence à la définition fournie par Ferber [Fer95] qui stipule qu'un agent est une entité physique ou virtuelle :

- qui est capable d'agir dans un environnement,
- qui peut communiquer directement avec d'autres agents,
- qui est mu par un ensemble de tendance (sous la forme d'objectifs individuels ou d'une fonction de satisfaction, voire de survie, qu'elle cherche à optimiser),

- qui possède des ressources propres,
- qui est capable de percevoir (mais de manière limitée) son environnement,
- qui ne dispose que d'une représentation partielle de cet environnement (et éventuellement aucune),
- qui possède des compétences et offre des services,
- qui peut éventuellement se "reproduire",
- qui a un comportement qui tend à satisfaire ses objectifs, en tenant compte des ressources et des compétences dont elle dispose, et en fonction de sa perception, de ses représentations et des communications qu'elle reçoit.

Par ailleurs, une autre définition fréquemment citée est fournie par Jennings *et al.* [JSW98] qui stipule qu'un agent est un système informatique, situé dans un environnement, et qui agit d'une façon autonome et flexible pour atteindre les objectifs pour lesquels il a été conçu. Les notions "situé", "autonomie" et "flexible" sont définies comme suit :

- situé : l'agent est capable d'agir sur son environnement à partir des entrées sensorielles qu'il reçoit de ce même environnement. Exemples : systèmes de contrôle de processus, systèmes embarqués, etc,
- autonome : l'agent est capable d'agir sans l'intervention d'un tiers (humain ou agent) et contrôle ses propres actions ainsi que son état interne,
- flexible : par flexibilité, les auteurs entendent :
  - Réactivité : un système réactif maintient un lien constant avec son environnement et répond aux changements qui y surviennent.
  - Pro-activité : un système pro-actif génère et satisfait des buts. Son comportement n'est donc pas uniquement dirigé par des événements.
  - Capacités sociales : un système social est capable d'interagir ou coopérer avec d'autres systèmes.

Ces définitions sont cependant trop généralistes et ne donnent pas une idée sur la manière d'appliquer concrètement le concept agent. Pour avoir une meilleure vision de ce concept, nous explorons d'autres travaux qui parlent de modèles d'agents, d'architectures d'agents et d'implémentation d'agents. Ces termes explicitent les différents niveaux de description d'un agent :

- le modèle qui décrit comment l'agent est compris, ses propriétés et comment on peut les représenter,
- l'architecture qui est un niveau intermédiaire entre le modèle et l'implémentation. Elle précise la manière dont un modèle peut être appliqué en spécifiant explicitement ses composants,
- l'implémentation qui s'occupe de la réalisation pratique de l'architecture des agents à l'aide de langages de programmation.

Les modèles d'agents peuvent être classés par famille. Chaque famille se focalise sur une ou plusieurs caractéristiques que possède l'agent. Il définissent la manière de concevoir l'entité agent. On trouve également dans la littérature des architectures qui raffinent les modèles en donnant une idée plus concrète sur les composants nécessaires à l'implémentation. Ces architectures sont parfois prises en compte au niveau des plates-formes d'implémentation.

Nous présentons dans ce qui suit, les principaux modèles d'agents classés par famille. Ainsi, nous distinguons :
– les agents réactifs
– les agents cognitifs
– les agents hybrides
– les agents mobiles

Modèle d'agents réactifs

La structure des agents purement réactifs tend à la simplicité. Ce sont des agents qui fonctionnent selon un mode stimuli/réponse. Dès qu'ils perçoivent une modification de leur environnement, ils répondent par une action programmée. L'agent réactif ne possède pas une représentation complète de son environnement et n'est pas capable de tenir compte de ses actions passées. De ce fait, il ne peut avoir des capacités d'apprentissage. Ainsi, les agents réactifs sont souvent considérés comme n'étant pas "intelligents" par eux-mêmes. Chaque individu pris séparément possède une représentation faible de l'environnement et n'a pas de buts. Mais ces derniers peuvent être capables d'actions évoluées et coordonnées. C'est le cas d'une société de fourmis, étudiée par Alexis Drogoul [Dro93].

Quand on parle d'agents réactifs, plusieurs travaux font référence à l'architecture de Brooks [Bro86], appelée architecture de subsomption. Une architecture de subsomption comporte plusieurs modules, qu'on appelle *modules de compétence*. Chaque module étant responsable de la réalisation d'une tâche spécifique. Les couches de bas représentent des comportements primitifs, par exemple "éviter les obstacles". Les couches de haut niveau correspondent à des tâches plus abstraites et sont prioritaires sur les couches de plus bas niveau. La figure 3.1 illustre un exemple d'application de l'architecture de Brooks.

Dans cet exemple, le robot perçoit son environnement à travers ses capteurs. Les capteurs transmettent l'information aux différentes couches. Cette information est filtrée par un inhibiteur qui permet d'intercepter les messages venant de la couche supérieure. Ceci provoque l'annulation de la tâche relative à la couche de l'inhibiteur. A la sortie de chaque couche, un supresseur

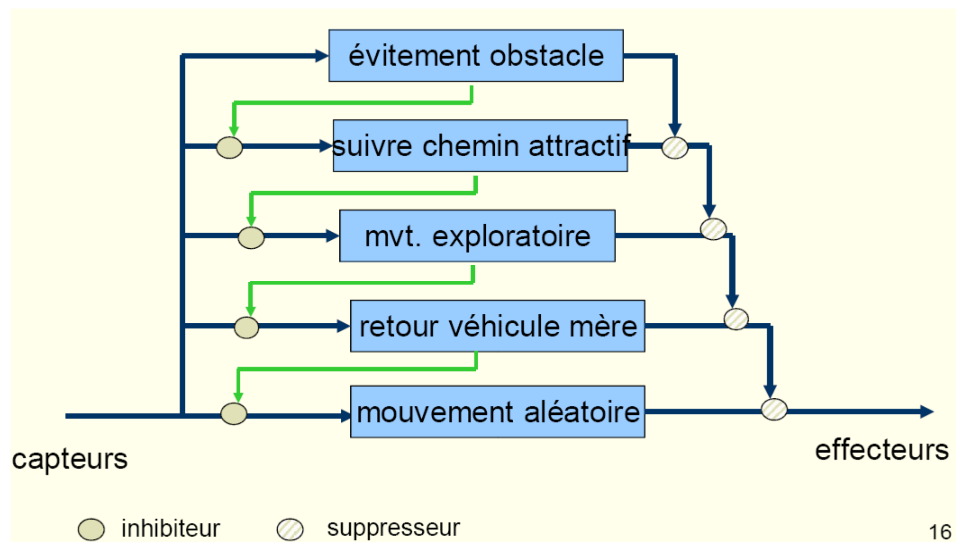


Fig. 3.1: L'architecture de Brooks

intercepte les messages de la couche supérieur, ce qui a l'effet d'annuler la tâche de plus bas niveau. L'architecture de Brooks est parmi celles qui ont été les plus utilisées pour développer des agents réactifs. D'autres architectures existent telles que l'architecture Manta [DCF95] décrite dans la figure ci-dessous 3.2.

Les travaux sur les agents réactifs ont fait apparaître des concepts devant être pris en compte lors de l'élaboration de l'architecture interne de l'agent tel que :

- les capteurs : qui servent à détecter un évènement survenu dans l'environnement,
- les tâches : qui décrivent une partie du comportement de l'agent,
- les effecteurs : qui permettent de réaliser les tâches.

Plusieurs plates-formes implémentant ces concepts ont été proposées dans la littérature telles que MANTA [DCF95], DIET [MBWH03], SimuLE [MP05].

Modèle d'agents cognitifs

Ils se basent sur un modèle qui provient d'une métaphore du modèle humain. Ces agents possèdent une représentation explicite de leur environnement, des autres agents et d'eux-mêmes. Ils sont aussi dotés de capacités de raisonnement et de planification ainsi que de communication. Le travail le plus représentatif de cette famille d'agent porte sur le modèle BDI (Believe Desire Intention) [BIP88]. Les sources de ces travaux sont les sciences humaines et sociales.

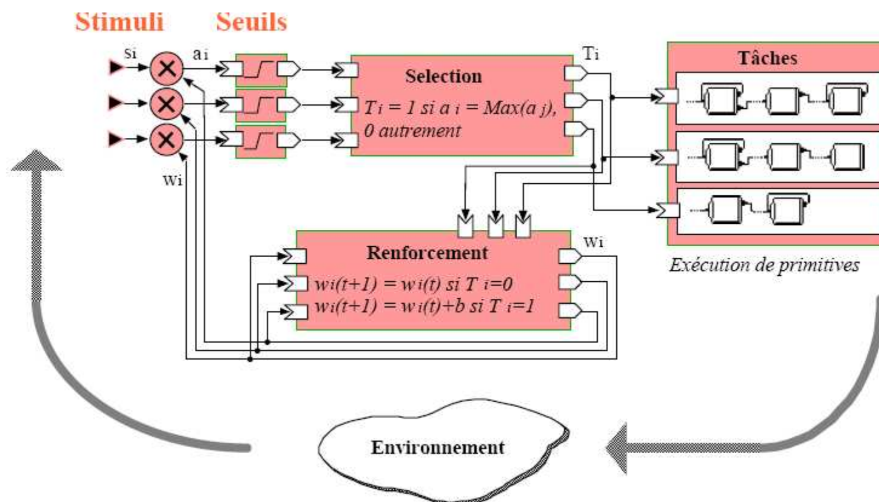


FIG. 3.2: L'architecture MANTA [DCF95]

La logique BDI (Belief, Desire, Intention) est énoncée dans les années 80 par [BIP88] et fait l'objet jusqu'à maintenant de plusieurs travaux. Ce modèle se fonde sur trois attitudes qui définissent la rationalité d'un agent intelligent :

- B = Belief = Croyance : ce sont les informations que possède l'agent sur son environnement et sur les autres agents agissant sur le même environnement. Ceci constitue les connaissances supposées vraies de l'agent.
- D = Desire = Désir : ce sont les états de l'environnement et parfois de lui-même, qu'un agent aimerait voir réaliser. Ce sont les objectifs que se fixe un agent.
- I = Intention = Intention : ce sont les actions qu'un agent a décidé de faire pour accomplir ses désirs. Ils forment des ensembles de plans qui sont exécutés tant que l'objectif correspondant n'est pas atteint.

Le modèle BDI a inspiré beaucoup d'architectures d'agents cognitifs. Nous pouvons citer : PRS (Procedural Reasoning System) développée par Georgeff et Lansky en 1987 [GL87]. dMARS (the distributed Multi-Agent Reasoning System) est une évolution de l'architecture PRS, réalisée par Wooldridge *et al.* en 1997 [dKLW97]. Enfin, nous pouvons citer IRMA (Intelligent Resource-bounded machine Architecture) développée par Bratman *et al.* en 1988 [BRA87], etc.

Modèle d'agents hybrides

Ce sont des agents ayant à la fois des capacités cognitives et réactives. Ils conjuguent la rapidité de réponse des agents réactifs ainsi que les capacités de raisonnement des agents cognitifs. Ce modèle a été proposé par plusieurs auteurs afin de palier aux problèmes liés au temps de décision et au temps d'action relativement élevé pour les agents réactifs. Dans ce type de modèle, les agents sont conçus comme étant composés de niveaux hiérarchiques qui interagissent entre eux. Chaque niveau gère un aspect du comportement de l'agent [OJ96] :

- bas niveau : couche réactive qui est en relation directe avec l'environnement et qui raisonne suivant les informations brutes de ce dernier,
- niveau intermédiaire : couche mentale qui fait abstraction des données brutes et qui se préoccupe d'un aspect de l'environnement plus évolué,
- niveau supérieur : couche sociale qui tient compte de l'interaction avec les autres agents.

Pour illustrer cette famille, nous pouvons citer l'architecture ASIC [BD94] utilisée pour le traitement numérique d'images, l'architecture ARCO [Rod94] créée dans le cadre de la robotique collective et l'architecture ASTRO [OBDK01] développée pour être utilisée dans les systèmes multi-agents soumis à des contraintes de type temporel.

Modèle d'agents mobiles

Un agent mobile est un agent capable de se déplacer d'un site physique à un autre. Ce type d'agent s'oppose aux agents statiques (ou stationnaires), qui s'exécutent seulement dans le système où ils ont commencé leur exécution. Les agents stationnaires communiquent avec leur environnement selon des moyens conventionnels tels que RPC (Remote Procedure Calling). Ces moyens peuvent être coûteux en terme de charge de communication. Par contre, un agent mobile n'est pas lié au système dans lequel il débute son exécution [LO98]. Il est capable de se déplacer d'un hôte à un autre dans le réseau. Il peut transporter son état et son code d'un environnement vers un autre où il poursuit son exécution. Ceci a l'avantage de réduire le transfert des flux de données volumineuses dans le réseau : quand un grand volume de données est stocké dans un hôte éloigné, ces données seront traitées localement plutôt que transférées à travers le réseau. Il s'agit donc de déplacer les traitements vers les données plutôt que les données vers les traitements (figure 3.3).

Un agent mobile peut avoir la même architecture qu'un agent réactif ou

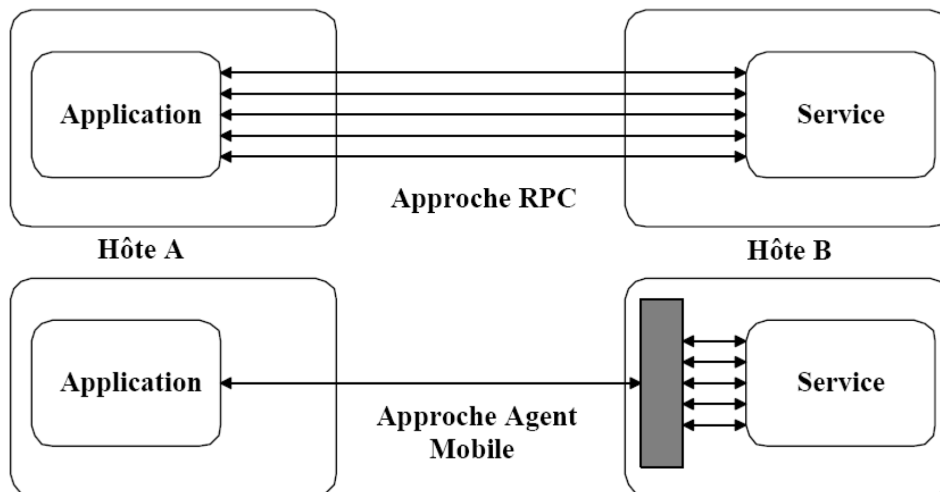


FIG. 3.3: Réduction de la charge des réseaux

cognitif, bien qu'il y ait une nécessité d'avoir un module supérieur gérant la mobilité de l'agent. Il existe à l'heure actuelle beaucoup d'implémentations différentes des plates-formes à agents mobiles mais qui sont incompatibles entre elles. Certaines écrites en Java pur, bénéficient des avantages de ce langage (portabilité, sécurité d'exécution, chargement dynamique de classes, programmation multithread, etc.). Les plates-formes basées sur Java les plus connues sont Odyssey de General Magic Inc, Aglets d'IBM, Concordia de Mitsubishi, Voyager de ObjectSpace. D'autres plates-formes ont été développées avec d'autres langages : Agent Tcl, Tacoma et Telescript (General Magic).

2.2 La vue environnement

L'environnement d'un agent désigne tout ce qui est extérieur à l'agent. Il définit *l' espace* commun aux agents du système et est doté d'un ensemble d'objets. Dans la littérature, l'environnement est distingué sous plusieurs angles :

- environnement social (par opposition à l'environnement physique) : l'environnement social correspond aux autres agents avec lesquels un agent est en interaction par envoi de message. L'environnement physique est quant à lui constitué des ressources matérielles que l'agent peut percevoir ou sur lesquels il peut agir. Par exemple, dans une société de fourmis, l'environnement physique est composé de nourriture et d'obstacles.
- environnement accessible (par opposition à "inaccessible") : le système peut obtenir une information complète, exacte et à jour sur l'état de

son environnement. Dans un environnement inaccessible, seule une information partielle est disponible. Par exemple, un environnement tel que l'Internet est inaccessible car il est impossible de tout connaître à propos de lui. Un robot qui évolue dans un environnement possède des capteurs pour le percevoir et, en général, ces capteurs ne lui permettent pas de connaître tout de son environnement qui est alors considéré comme inaccessible.

– environnement continu (par opposition à "discret") : dans certains environnements, le nombre d'actions et de perceptions possibles est infini. Dans ce cas, l'environnement est considéré comme continu. A l'inverse, si le nombre d'actions et de perceptions possibles est limité, l'environnement est dit discret. Ceci est le cas dans un environnement simulé tel qu'un éco-système.
– environnement déterministe (par opposition à "non déterministe") : une action effectuée dans un environnement déterministe a un effet unique et certain. Dans ce cas, l'état suivant de l'environnement est complètement déterminé par l'état courant. Dans un environnement non déterministe, une action n'a pas un effet unique garanti. Par nature, le monde physique réel est un environnement non déterministe.
– environnement dynamique (par opposition à "statique") : l'état de cet environnement dépend des actions des processus appartenant à cet environnement mais aussi des actions d'autres processus externes. Aussi, les changements ne peuvent pas être prédits par le système. Par exemple, l'Internet est un environnement hautement dynamique, son changement n'est pas simplement lié aux actions d'un seul utilisateur.

Actuellement, l'environnement est une dimension encore peu modélisée dans les SMA à quelques exceptions dans le domaine de la simulation. En fait, la modélisation de l'environnement est souvent intégrée dans les connaissances du domaine de l'agent.

2.3 La vue interaction

L'interaction a comme objectif de mettre en relation dynamique deux ou plusieurs agents par le biais d'une ou plusieurs actions réciproques. Elle peut se faire selon plusieurs manières suivant la situation à travers laquelle interagissent nos agents.

On distingue différents types d'interaction que les agents peuvent adopter comme : la communication, la coopération, la négociation et la coordination. L'interaction se base sur des protocoles qui permettront aux agents d'échanger des messages structurés.
Le fait d'interagir va permettre à l'agent de partager de l'information, d'être au courant de ce qui se passe autour de lui, d'atteindre ses buts et d'éviter

autant que possible les conflits [GHB00]. Dans ce qui suit, nous allons détailler les différents modes d'interaction qui sont évoqués dans la littérature ainsi que les protocoles sur lesquels ils se basent.

La communication

La communication est un des éléments importants du système multi-agents. Elle permet l'échange des informations, la coopération et la coordination. Dans la communication agent, l'intention de communiquer est de produire un effet sur les destinataires : ils exécuteront probablement une action demandée par l'émetteur ou répondront à une question [Hug05]. La communication entre agents peut être une communication directe ou indirecte.

La communication indirecte est adoptée par les agents réactifs. Elle se fait à travers l'environnement ou bien par le biais d'un tableau noir. Dans une communication par environnement, les agents laissent des traces ou des signaux qui seront perçus par les autres agents. Dans ce genre de communication il n'y a pas de destinataire bien défini. La communication directe, quant à elle, est spécifique aux agents cognitifs. Elle est intentionnelle et se fait par envoi de messages à un ou plusieurs destinataires. La communication directe se base sur trois éléments essentiels :

- le langage de communication : il permettent de structurer les messages échangés entre les agents. Les langages de communication les plus utilisés et les plus connus sont KQML [FLM95] et FIPA ACL [FIP99]. Ils sont basés sur la théorie des actes de langages [Aus62],[Sea69].
- l'ontologie : elle sert à fournir un vocabulaire et une terminologie compréhensible par tous les agents. Cette sémantique sera régie par des règles et des contraintes qui permettront de définir un consensus sur le sens des termes. En effet, l'ontologie concernera les symboles non-logiques du contenu du message.
- les mécanisme de communication : ils permettent de stocker, rechercher et adresser les messages aux agents. Ces mécanismes sont présents dans les plates-formes multi-agents comme Jade (cf. section 6.2) ou MadKit **??**.

La coopération

La coopération se traduit par le fait qu'un ensemble d'agents travaillent ensemble pour satisfaire un but commun ou individuel. L'ajout ou la suppression d'un agent influe considérablement sur la performance du groupe. Le besoin de faire coopérer des agents, vient essentiellement du fait qu'un agent ne peut atteindre son objectif individuellement et a, par conséquence

besoin de l'aide des autres agents du système. La coopération a comme devise "diviser pour régner", c'est-à-dire que la tâche va être décomposée entre plusieurs agents et ceci peut être réalisé par plusieurs mécanismes comme le Contract Net [Smi88] ou encore la planification multi-agents qui consiste en une technique adoptant un organe centralisé qui se charge de la réalisation des plans et de la gestion des conflits.

La coordination

La coordination [Cia96] est présente lorsqu'il existe une interdépendance dans les actions des agents, leurs buts ou même les ressources qu'ils utilisent. A ce stade, coordonner les activités des agents devient un aspect essentiel afin d'éviter des problèmes dans le comportement global du système. En effet, la coordination met de l'ordre dans le processus global effectué par des agents. Elle consiste à synchroniser leurs activités ou à régler les conflits qui existent entre eux.

La négociation

Un système multi-agents est composé de plusieurs agents. Ces derniers peuvent entrer en conflit pour plusieurs raisons : conflits d'intérêts ou de buts, accès à des ressources ou proposition de plusieurs solutions différentes à un seul problème. Afin de résoudre ces conflits et de trouver une situation qui satisfasse tout le monde, les agents négocient entre eux en faisant des concessions ou en cherchant des alternatives. Dans la littérature, Smith [Smi88] définit la négociation par :

> "By negotiation, we mean a discussion in which the interested parties exchange information and come to an agreement. For our purposes negotiation has three important components (a) there is a two-way exchange of information, (b) each party to the negotiation evaluates the information from its own perspective, (c) final agreement is achieved by mutual selection".

A travers cette définition, nous remarquons que les deux grands piliers de la négociation sont la communication (a) et la prise de décision (b) (c). Afin de résoudre les conflits, les systèmes multi-agents peuvent adopter plusieurs méthodes que nous pouvons classer dans deux grandes familles :

- Les méthodes centralisées : elles reposent sur la négociation via un médiateur ou sur des protocoles tels que le Contract Net Protocol,
- Les méthodes décentralisées : elles sont plus utilisées par les agents BDI telles que la recherche négociée (TEAM) [LL93], la méthode heuristique [JPSF00] ou encore la négociation basée sur l'argumentation (ANA) [JFL+01].

Les protocoles d'interaction

Afin de structurer la communication entre les agents, des protocoles d'interaction sont élaborés. Les protocoles d'interaction permettent de décrire explicitement les échanges de messages entre les agents. Ils représentent un schéma commun de conversation utilisé pour exécuter une tâche ou appliquer une stratégie de haut niveau. Un protocole précise qui peut dire quoi à qui et les réactions possibles à ce qui est dit. Il décrit également les enchaînements de messages qui peuvent être possibles entre les agents. Il existe différents types de protocoles d'interaction, on cite particulièrement :

- les protocoles de coordination,
- les protocoles de coopération,
- les protocoles de négociation.

Parmi les protocoles de coordination les plus utilisés et les plus connus dans le système multi-agents, on trouve le *Contract Net Protocol* [Smi88] qui décrit un protocole dont les agents coordonnent leurs activités en établissant des contrats afin d'atteindre leurs buts ou encore *Auction protocol* [HOB04] qui se base sur la notion d'enchères croissants (English Auction) ou enchères décroissants (Dutch Auction).

2.4 La vue organisation

L'organisation permet de structurer les différentes composantes du système ainsi que leur mode de fonctionnement global. L'organisation s'intéresse aussi aux dimensions sociales des agents [Hab68]. Elle peut être considérée comme une structure décrivant les interactions et autres relations qui existent entre les membres de la dite organisation [Fox81] (dans le but d'assouvir un objectif commun). L'organisation peut donc apparaître comme une structure de coordination et de communication [Mal87].

Il existe de nos jours, deux principales visions du concept organisation [CSB05]. Dans la première vision, l'organisation est une entité à part entière qui est composée par un groupe d'agents. La structure organisationnelle ainsi que le comportement global sont pré-établis. Plusieurs contraintes peuvent être exprimées au niveau de la topologie, des protocoles d'interaction, des normes utilisés ainsi que le déroulement des activités [FGM03], [HSB02], [VSDD04].

Dans la deuxième vision, l'organisation émerge du comportement global des différentes composantes du système [Dro93], [RB98], [GGG05]. Les différents éléments faisant partie de l'organisation sont préalablement connus et leur comportement individuel est spécifié. Cependant, le comportement

global est inconnu. Ces cas sont surtout rencontrés dans les applications d'ingénierie sociale qui consistent à étudier la dynamique des structures sociales. L'objectif de ces applications est de découvrir le modèle organisationnel généré suite aux activités et décisions des différents acteurs sociaux ainsi que leur interaction. Nous pouvons citer par exemple, le cas des sociétés de fourmis [RB98]. Dans cette vision, il est intéressant de spécifier de manière explicite au niveau de l'organisation, les règles de gestion de conflits ou d'interférences générés suite aux activités de chacun des agents.

Quand on parle d'organisation au niveau des systèmes multi-agents, différents aspects sont étudiés en profondeur. Ce sont principalement :
- la topologie,
- l'emergence,
- l'auto-organisation.

Les topologies de l'organisation

Les organisations multi-agents peuvent être décrites selon une certaine topologie. Ces topologies décrivent les structures organisationnelles et permettent d'imposer certaines contraintes au niveau des interactions entre les agents appartenant à l'organisation. Dans sa thèse, Le Strugeon [Str95] identifie différents types de topologies :
- les topologies à structure hiérarchique : ces organisations sont souvent rigides. Le contrôle est centralisé sur un agent qui communique les ordres aux autres agents qui se contentent de les exécuter,

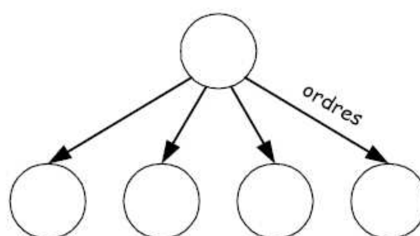


Fig. 3.4: Topologie à structure hiérarchique

- les topologies de type marché : les organisations respectant ce type de topologie sont composées d'agents coordinateurs et d'agents exécutants. Cette structure est un peu plus décentralisée autour de plusieurs objectifs opérationnels. L'architecture MAGIQUE [RM01] se base sur ce type de topologie,

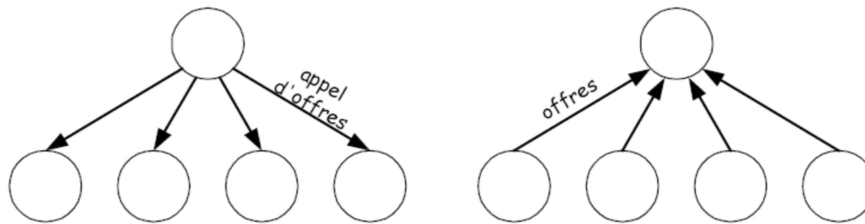


FIG. 3.5: Topologie à structure de marché

– les topologies de type communauté : le contrôle est fortement distribué et dont les membres possèdent les mêmes capacités,

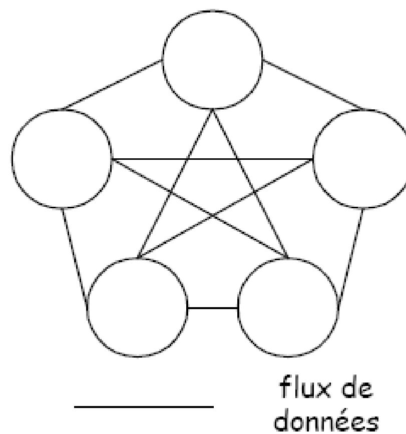


FIG. 3.6: Topologie à structure de communauté

– les topologies de type société : dans ces organisations, le contôle est décentralisé. Les agents poursuivent chacun un objectif opérationnel et le comportement global est ajusté par des principes de négociation.

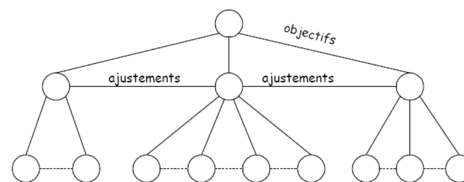


FIG. 3.7: Topologie à structure de société

L'émergence

Un système multi-agents est un ensemble d'agents situés dans un même environnement et qui interagissent. De ces interactions peuvent émerger différents phénomènes. L'émergence traite donc de l'apparition soudaine non

programmée et irréversible de phénomènes dans un système. Cette notion se base sur la maxime grecque "le tout est plus que la somme de chaque partie". Les phénomènes qui apparaissent peuvent être classés en trois catégories [MC97] : l'émergence structurelle, l'émergence de comportements et l'émergence de propriétés.

Une définition fournie par Müller dans [Mül02], énonce qu'un phénomène est émergent si :
- il y a un ensemble d'agents en interaction entre eux et via un environnement dont l'expression des états et de la dynamique n'est pas exprimée dans les termes du phénomène émergent à produire mais dans un vocabulaire ou une théorie D,
- la dynamique des agents en interaction produit un phénomène global qui peut être une structure stable, une trace d'exécution ou n'importe quel invariant statique ou dynamique,
- ce phénomène global est observé soit par un observateur extérieur (sens faible) soit par les agents eux-mêmes (sens fort) en des termes distincts de la dynamique sous-jacente, c'est à dire avec un autre vocabulaire ou théorie D'.

L'observation peut se faire par un observateur extérieur ou par les agents eux-mêmes à travers un mécanisme d'inscription et d'interprétation fondé sur une théorie de l'émergence *D*. La raison d'être de cette théorie de l'émergence étant d'exprimer la façon de relier les théories D et D'.

Les travaux sur l'émergence tentent de développer des outils, des méthodes et des constructions qui rendent le processus d'émergence moins opaque et par conséquent plus compréhensible. Pour cela, ils se basent, entre autre, sur :
- la théorie des systèmes complexes adaptatifs : l'émergence fait référence aux patterns du macro niveau apparaissant dans les systèmes d'agents en interaction,
- la théorie des systèmes dynamiques non linéaires issue de la théorie du chaos. Le travail porte en grande partie sur les attracteurs,
- l'école synergétique fondée par Haken (1981) qui étudie l'autoorganisation dans les systèmes physiques,
- la thermodynamique Loin et autour de l'Équilibre introduite par Prigogine. Les phénomènes émergents sont les structures dissipatives apparaissant à partir de conditions "loin de l'équilibre".

L'auto-organisation

Plusieurs travaux se sont penchés sur l'adaptation dans les systèmes multi-agents et étudient de ce fait, leur capacité d'auto-adaptation. Ces travaux considèrent que le système doit s'adapter à son environnement par

auto-organisation des agents. L'organisation finale observée est alors la plus adaptée à l'état courant de l'environnement.

L'auto-organisation a été initialement étudiée et appliquée dans les secteurs de la biologie, de la physique et de la chimie. L'objectif de l'auto-organisation est de permettre l'évolution dynamique de l'existant en fonction du contexte de façon à assurer la viabilité du système [Cam94]. L'auto-organisation est un mécanisme très interessant permettant de réaliser des systèmes tolérant aux pannes. Ce mécanisme permet aussi aux composantes du système de s'adapter à leur environnement soit par spécialisation des fonctions (apprentissage), soit par modification de la topologie de l'organisation. Les études sur l'auto-organisation portent sur la définition des critères de réorganisation.

3 Ingénierie des systèmes informatiques selon le paradigme orienté agent

Comme nous l'avons mentionné dans la section 1, plusieurs techniques d'ingénierie orientées agent existent. Nous classifions ces différentes techniques dans trois grandes classes :
- les langages de spécification orientés agent,
- les méthodologies orientées agent,
- les plates-formes d'implémentation orientées agent.

Avant de procéder à la présentation et la comparaison des travaux portant sur ces différentes techniques, il convient de définir ce que nous entendons par langages de spécification, méthodologie et plates-formes d'implémentation.

3.1 Les langages de spécification

Les langages de spécification (comme son nom l'indique) permettent d'exprimer les spécifications liées à un système. Différents types de notation existent. Elles vont des "plus naturelles" aux plus formelles.

Les énoncés informels

Cette catégorie de notation se base sur des descriptions faites en langage naturel. Bien que pouvant être standardisées, elles sont bien souvent propres à une entreprise voire un projet ou une équipe de développeurs. L'inconvénient majeur de ce type de techniques est dû à l'ambiguïté du langage. Des risques d'incohérence, de non complétude, de difficulté d'organisation et de redondance d'information peuvent donc apparaître. Pour pallier l'ambiguïté, certains adoptent des présentations formatées telles que les dictionnaires de

données ou glossaires, dans lesquels est défini le vocabulaire du domaine d'application. Ceci reste cependant insuffisant et les risques cités précédemment perdurent.

Les notations semi-formelles

Ces notations sont relativement simples et permettent ainsi aux différents participants à un projet de communiquer facilement et efficacement. Les représentations classiques de cette catégorie de notations sont :

- les diagrammes d'états-transitions dont l'objectif est, comme pour les tables états-transitions, de représenter les différents états du système et les évènements qui déclenchent les changements d'états,
- les diagrammes de flots de données qui intègrent les flots de données et des processus de transformation. Ils montrent comment chaque processus transforme les données qu'il reçoit en flots de sortie,
- l'ensemble des notations utilisées dans UML (diagramme de cas d'utilisation, de séquence, de collaboration, d'états-transitions, etc.).

Les notations formelles

Cette famille de notations qui repose sur les mathématiques permet, lors d'une étape de spécification, d'éliminer les ambiguïtés des notations informelles ou semi-formelles et les mauvaises interprétations qui en découlent. Les langages formels possèdent trois composantes :

- une syntaxe qui précise la notation utilisée pour procéder à la spécification,
- une sémantique qui donne une définition non ambiguë de cette syntaxe,
- un ensemble de relations qui définissent les règles donnant les propriétés des objets mathématiques satisfaisant la spécification.

Des représentants des notations formelles sont, par exemple, LOTOS [Gar89], le langage B [Abr91], le langage Z [Spi92] et le π-calcul [Mil99]. Certaines spécifications non formelles peuvent se voir adjoindre des langages formels pour l'expression des contraintes. On peut citer par exemple OCL [BKPPT00]dans le cas d'UML.

3.2 Les méthodologies de développement

Souvent, il est d'usage de confondre les termes *méthodologie* et *méthode* et de les utiliser comme signifiant la même chose. Cependant, ces termes sont différents. Une méthode correspond en réalité à une description étape par étape de ce qui doit être fait pour accomplir une tâche ; alors qu'une méthodologie désigne un ensemble de principes généraux qui guident le choix d'une méthode adaptée aux besoins d'un projet ou d'une tâche. Dans son ouvrage *"Fundamentals of Software Enginerring"*, Carlos Ghezzi *et al.* [GJM03] fait

la distinction entre méthodes et méthodologies. Selon ces auteurs une méthode détermine la manière d'effectuer une tâche durant le développement du logiciel. Elle peut être comparée à une recette de cuisine qui indique les ingrédients nécessaires ainsi que la manière de les cuisiner afin de réussir un plat. Une méthodologie, quant à elle, a été considérée par ces mêmes auteurs comme un système de méthodes. Nous retiendrons la définition proposée par Booch [Boo92] , qui est plus explicite et qui stipule qu'une méthodologie est :

> "un ensemble de méthodes appliquées tout au long du cycle de développement d'un logiciel, ces méthodes étant unifiées par une certaine approche philosophique générale".

A titre de comparaison, une méthodologie va correspondre à l'ensemble des recettes de cuisine qui vont être utilisées pour préparer et servir un grand repas organisé selon un thème particulier. Ce thème correspond à l'approche philosophique générale, adoptée lors du développement. Comme nous l'avons déjà mentionné dans le chapitre 2, ce sont les paradigmes de développement qui définissent cette approche philosophique. Ainsi, il existe par exemple des méthodologies liées au paradigme orienté objet (OMT (Object Modeling Technique) [RBE+95], OOSE (Object Oriented Software Engineering) [Jac93],etc.).

De manière plus explicite, une méthode regroupe :

– une démarche qui consiste en un ensemble ordonné de phases et d'étapes décrites en termes d'activités de production. La démarche a pour but de guider le travail des développeurs pour arriver à un résultat autrement dit à un délivrable,
– des concepts basés sur les paradigmes de développement et qui permettent de créer des modèles suffisamment explicites pour que l'on en ait une représentation unique (exemple : une classe est un concept introduit par le paradigme orienté objet),
– des notations qui permettent d'introduire une syntaxe et une sémantique permettant d'exprimer les spécifications (voir section 3.1),
– des outils qui permettent la production des spécifications selon des modèles et des diagrammes définis par les notations. Ces outils permettent d'avoir des spécifications propres, lisibles, facilement modifiables. Ils permettent aussi la vérification de leurs syntaxes, leurs cohérences et de leur complétude. Les outils peuvent être regroupés dans des environnements.

Une méthodologie va regrouper différentes méthodes conformes à un paradigme de développement. Ces méthodes vont être appliquées tout au

long du cycle de vie. Rappelons qu'un cycle de vie est un ensemble ordonné d'étapes de développement. Le cycle de vie de logiciel passe par différentes étapes concernant des points aussi divers que la faisabilité, la spécification, la production, les tests, l'utilisation du logiciel, sa vente et sa maintenance. Le cycle de vie est souvent organisé (de manière explicite ou pas) selon un modèle qui peut être en cascades [Boe76], en spirale [Boe88], en V, etc. Le modèle de cycle de vie peut être explicité ou pas au niveau de la méthodologie. Le modèle de cycle de vie caractérise le processus logiciel qui décrit l'enchaînement des étapes de la conception à la maintenance d'un produit logiciel. Une méthodologie peut couvrir une partie ou l'ensemble du processus de développement.

3.3 Les plates-formes d'implémentation

L'implémentation d'un système consiste en sa réalisation pratique sur une plate-forme particulière. La plate-forme est une infrastructure de logiciels utilisée comme environnement pour le déploiement et l'exécution du système. Elle doit fournir des manières confortables pour créer et tester des entités logiciels selon un paradigme de développement particulier. Ainsi, nous trouvons des plates-formes dédiées aux paradigmes procéduraux, objets, composants, agents etc. Elle peut être vu comme une collection de services offerts aux développeurs, mais également aux entités logicielles en exécution. Ainsi, une plate-forme devrait agir en tant qu'un médiateur entre le système d'exploitation et les applications développées sur cette plate-forme. Les plates-formes peuvent être considérées comme des EDI dédiés à la phase d'implémentation du système. Dans ce qui suit, nous allons présenter un panorama comportant des exemples de langages, méthodologies et plates-formes orientées agents.

4 Les langages de spécification orientés agent

Plusieurs langages de modélisation ont été proposés dans le contexte de l'ingénierie orientée agent. Différents courants ont été appliqués pour la construction de ces langages. Ainsi, nous distinguons :

- les langages basés sur une extension d'UML, par exemple AUML [BMO01], AML [CT04],
- les langages réalisés par création de profil UML, par exemple AORML [WT03],
- les notations graphiques, par exemple OPM/MAS[SDS03].

4.1 Les langages basés sur une extension d'UML

L'extension du langage UML s'obtient grâce à la notion de stéréotype. Un certain nombre de stéréotypes sont prédéfinis. C'est le cas par exemple

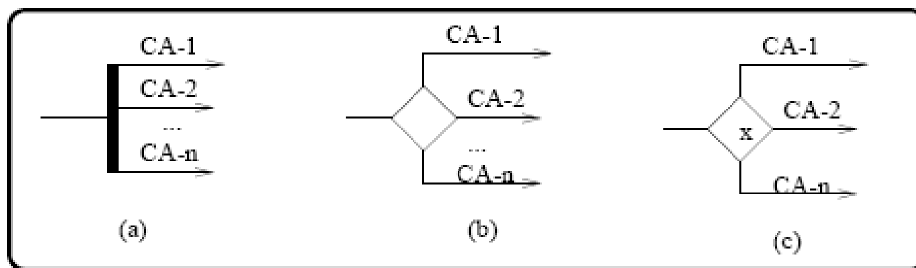


FIG. 3.8: Les branchements définis dans AUML

du stéréotype *"interface"*, mais un utilisateur peut en définir d'autres en fonction de ses besoins.

Ceci permet d'étendre le nombre de diagrammes utilisables. Dans notre contexte, nous allons prendre l'exemple de deux langages qui ont utilisé cette alternative pour définir une nouvelle notation supportant certains concepts des systèmes multi-agents ; c'est le cas notamment des langages AUML [BMO01], [HOB04] et AML[CTCG05].

AUML porte essentiellement sur les protocoles d'interactions entre agents. Ce langage a défini des types de branchements dans les diagrammes de séquence afin de prendre en compte l'indéterminisme du comportement d'un agent. Ces branchements peuvent être AND (ET logique), OR (OU logique) XOR (OU exclusif)(voir figure 3.8).

AUML prend en compte d'autres considérations telle la spécification des rôles dans les différents diagrammes, où les classes du diagramme de séquences générique sont annotées Classe/Rôle. Le tableau 4.1 résume les concepts pris en compte dans la notation AUML.

Du fait qu'AUML soit une extention d'UML, il peut être appliqué avec les outils UML classiques. De la même façon, AML [CT04] [CTCG05] est aussi un langage de modélisation des systèmes multi-agents basés sur une extension de UML 2.0. AML fournit un certain nombre d'éléments conçus pour décrire différents aspects. On trouve notamment la modélisation des ontologies, des aspects sociaux et mentaux, des interactions, du comportement ainsi que la décomposition du comportement sur plusieurs agents etc.

4.2 Les langages réalisés par création de profil UML

Une autre manière de créer des langages basés sur UML est de créer des profils. Un profil UML permet de spécialiser UML à un domaine particulier. Un profil est composé de stéréotypes, de *tagged values* et de contraintes. Généralement, la création de profil UML est accompagnée par la définition de règles de génération de code. Par exemple, les profils standards (EJB,

Concepts Modélisés	**Notation AUML**
Agent	Diagramme de classes : stéréotype *"Agent"* description d'états, actions, méthodes, capabilité, protocoles supportés, contraintes
Rôle	Diagramme de classes : stéréotype *"Role"*
Interaction	Spécification d'un template UML Diagramme d'interaction étendu d'une branche XOR pour prendre en compte l'indéterminisme
Message	Message asynchrone : flèche simple Message synchrone : flèche remplie
Action	Décrite au niveau du diagramme de séquences avec un rectangle aux angles arrondis

Tab. 3.1: La notation AUML

CORBA) proposent quasiment tous des règles de génération de code.

Dans le contexte orienté agent, nous pouvons prendre l'exemple de AORML (pour Agent-Object Relationship Modeling Language) [WT03] qui utilise cette technique. AORML est un profil UML pour l'analyse et la conception de systèmes d'information organisationnels basés sur les agents. Deux sortes de modèles sont proposés par cette notation : externes et internes. Les modèles externes présentent les interactions entre les agents ainsi que leur relation avec les objets de leur environnement. Les modèles internes sont le moyen de définir le comportement interne des agents . Les actions et comportements des agents, dans la vue interne, sont décrits grâce à des réseaux de Petri. Par contre, cette notation ne prend pas en compte l'auto-organisation ou la coopération. Le tableau 4.2 résume les concepts pris en compte dans la notation AORML.

L'application de cette notation se fait à travers l'outil Microsoft Visio template.

4.3 Les notations graphiques

Ces notations ne sont pas basées sur le langage UML. Elles définissent des icônes traduisant les concepts orientés agent. Le langage graphique orienté agent le plus connu est OPM/MAS [SDS03]. Celui-ci est considéré comme un langage complet. En effet, plus d'une vingtaine de types de flèches, de

	Vue Externe
Interaction entre agents	Passage de messages
Interaction entre objets et agents	Evénements non communicatifs Evénements de non action Evénements d'actions
	Vue Interne
Agent	Représentation interne de l'environnement chez l'agent
Action	Utilisation des réseaux de Petri

TAB. 3.2: Les concepts manipulés par AORML

noeuds ou de liens ayant des sens différents ou proches sont définis par cette notation. Le langage supporte notamment la modélisation de la vue statique du système ainsi que la vue dynamique.

Une vue d'ensemble sur les concepts supportés par OPM/MAS ainsi que la sémantique qui a été adoptée est fournie dans le tableau 4.3.

Ce langage a une grande expressivité mais il est très difficile à manipuler vu le nombre élevé des éléments graphiques qui le contituent. Son application demande donc du recul.

5 Les méthodologies orientées Agent

La particularité des concepts utilisés dans les systèmes multi-agents rend l'utilisation des méthodologies traditionnelles obsolètes. En effet, ces concepts ont particulièrement élevé le niveau d'abstraction, ils couvrent une sémantique très large et ont des interrelations très complexes. Ainsi, pour aider le concepteur à développer ce type de système, plusieurs méthodologies orientées agent ont été proposées. Il n'existe pas à l'heure actuelle une méthodologie unifiée pour assister le développement des systèmes multi-agents. Actuellement, les travaux sur la dimension méthodologique prend en compte divers aspects. Il est difficile dans ce contexte de pouvoir les comparer.

La problématique de la comparaison des méthodes et méthodologies a été souvent soulevée dans la littérature pour aider le concepteur à choisir celles qui sont les plus adaptées au problème donné. Shehory et Sturm ou bien Dam et Winikoff comparent les méthodes suivant quatre ou cinq axes : les concepts manipulés, les notations utilisées, le processus de développement et la pragmatique [SS03] [DW03]. Bernon *et al.* [BGPG02], utilisent huit critères pour comparer les méthodologies : l'étendu de la couverture du processus, la spécialisation de la méthode à une application, l'architecture d'agent sous-jacente, l'utilisation de notations existantes, le domaine d'appli-

	Vue statique
Organisation	composé d'un ensemble de rôles
Société	Exhibe des organisations, des agents et des règles
Rôle	Le rôle est joué dans des organisations par des agents et doit respecter les règles de la société
Protocole	Le protocole est exhibé par des agents
Faits	Peuvent représenter des croyances, des désirs ou des intentions
Buts	Les désirs peuvent être considérés comme des buts
Services	Ils sont exhibés par les agents et exhibent à leur tour des taches. Un service peut être composé par un autre service.
Règles	Elles sont exprimées au niveau des sociétés et doivent être respectées au niveau de l'élaboration des rôles
	Vue dynamique
Agents	Processus composé de tâches et qui fournit des services, participe à des protocoles et joue des rôles
Tâches	Elles sont réaliées par les rôles ou les services. Une tâche peut être composée par une autre. Elles peuvent effectuer des changements sur les objets ou les faits. Pour leur exécution, elles peuvent avoir besoin des conversations.
Messages	Lance un processus messaging qui peut affecter des objets. Ce processus peut avoir besoin des protocoles et des conversations.

Tab. 3.3: Les concepts manipulés par OPM/MAS

cation (dynamique ou non), le modèle de rôle, le modèle de l'environnement et l'identification des agents. Plus récemment, Cernuzzi *et al.* [CCZ05] se focalise sur les modèles de cycles de vie qui ont été adoptés dans les différentes méthodologies. Ils ont aussi précisé le degré de recouvrement de chaque phase de développement (voir figure 3.9).

Phases → **Process Model** and **Methodology** ↓	Requirements Elicitation	Requirements Analysis	Design	Coding and Implementation	Verification & Testing	Deployment
Waterfall Like						
Gaia		X	X			
Roadmap	X (partially)	X	X			
Prometheus	X (partially)	X	X	X	X	
MaSE	X (partially)	X	X	X	X(partially)	
AOR		X	X	X		
Evolutionary and Incremental						
OPM/MAS	X	X	X			X
MASSIVE		X	X	X	X	X
Ingenias		X	X	X		
Tropos	X	X	X	X		
PASSI and Agile PASSI		X	X	X	X	X
Transformation						
DESIRE	X	X	X	X	X(partially)	
Spiral						
MAS-CommonKADs	X	X	X	X	X(partially)	X

[CCZ05]

FIG. 3.9: Une classification des méthodologies

Enfin, nous pouvons aussi citer les travaux de Santos *et al.* [SRB06] qui proposent une table d'évaluation multi-critères. Ces travaux peuvent être considérés comme les plus complets. Ils se basent en effet sur 12 critères d'évaluation qui sont :

- la définition du processus,
- l'utilisation d'UML,
- la prise en compte des caractéristiques d'adaptation (l'agent est-il capable de s'adapter à diverses situations en apprenant de son expérience ?), d'apprentissage (l'agent est-il capable d'apprendre à appréhender son environnement ?)et de composition (est-il possible de composer plusieurs agents dans un environnement ?), de mobilité (les agents sont-ils capables de migrer d'une plate-forme à une autre ?),
- la représentation explicite de la structure organisationnelle,

– le traçage des modèles (c'est-à-dire garder la trace les différents modèles développés tout au long du développement),
– la capture des besoins,
– l'identification des agents,
– l'interaction entre les agents,
– le modèle interne des agents,
– la plate-forme d'implémentation,
– la modélisation du processus organisationnel,
– la structure des messages.

Notre but dans cette étude est de vérifier la complétude ainsi que la cohérence des méthodologies (aussi bien au niveau de la sémantique des concepts qu'au niveau du développement tout au long du cycle de vie). Nous allons nous focaliser sur les critères suivants :

– les types de systèmes ciblés : comme nous l'avons vu, les systèmes multi-agents traitent divers types de systèmes comme les systèmes adaptatifs, les systèmes distribués, les systèmes à base de connaissances, etc. Certaines méthodologies sont prescrites pour traiter certains types de systèmes alors que d'autres sont plus généralistes,
– le métamodèle : plusieurs concepts ont été introduits pour traiter les systèmes multi-agents. Ces concepts sont diffèrents d'une méthodologie à une autre. L'élaboration d'un métamodèle associé à chaque méthodologie a permis de décrire les concepts qui y sont utilisés,
– les notations : les notations adoptées pour décrire les modèles orientés agent sont différentes. Certaines se basent sur UML et les langages basés sur des extensions d'UML, d'autres se basent sur d'autres langages formels ou informels,
– les outils : certaines méthodologies sont fournies avec des outils spécifiques,
– le processus de développement : le processus de développement (qui spécifie la manière dont le développement est organisé en phases indépendamment du type de support technologique utilisé dans le développement [DKW99]) est différent selon les méthodologies proposées.

Dans notre comparaison, nous nous sommes focalisés sur des exemples de méthodologies tels que ADELFE [BCP05], Gaia [ZJW03], Ingenias [PGS03], PASSI [CGSS05] et Tropos [BPSM05].

5.1 La méthodologie ADELFE

ADELFE (Atelier pour le DEveloppement de Logiciels à Fonctionnalité Emergente) [BCP05] est une méthodologie dédiée à la conception des systèmes adaptatifs. Ces systèmes sont capables de s'adapter à un environnement fortement dynamique. Leur conception impose une méthode rigoureuse qui se distingue de l'approche globale descendante. ADELFE est une métho-

dologie qui a pour but de guider les développeurs lors du développement de ces systèmes en appliquant les théories adaptatives des systèmes multi-agents. Elle propose un processus, des notations et des outils.

Le métamodèle

Les systèmes adaptatifs sont basés sur des agents coopératifs qui exhibent des comportements sociaux. Lors du développement, les différents agents du système sont définis. Ceux-ci sont dotés d'aptitude pour décider de leurs interactions. Ainsi, le concepteur de l'application implémente les agents ; puis le système en fonctionnement se reconfigure de manière automatique par auto-organisation sans l'intervention du concepteur. Pour définir ce genre de système, ADELFE se focalise principalement sur la constitution des agents coopératifs.

Le métamodèle adopté par ADELFE (voir figure 3.10) met en évidence ce type d'agents. Selon ce métamodèle, un système multi-agent adaptatif est composé par un ou plusieurs agents coopératifs qui observent des règles de coopération et qui ont une représentation du monde. Les règles de coopération sont reliées à des situations non coopératives, c'est-à-dire des situations qui ne sont pas prises en charge. Celles-ci peuvent être de plusieurs types : incompréhension, ambiguïté, incompétence (la requête ne peut être satisfaite), improductivité (ne sait pas traiter les informations perçues), concurrence et conflit. Quand l'agent est face à ces situations, il devrait être capable de les résoudre, en cas d'incompréhension par exemple, il pourrait envoyer la requête vers un autre agent qu'il croit capable de la résoudre.

ADELFE considère que les agents ont une représentation du monde réel qui est constituée par sa perception de l'environnement physique, ses croyances par rapport aux autres agents et sa perception de lui-même. Cette perception est décrite au niveau du composant *"Representation"*. Les agents sont aussi constitués d'autres composants : les aptitudes, les compétences, les caractéristiques et les communications. Les aptitudes sont les capacités de raisonnement de l'agent, elles sont exprimées soit par des systèmes d'inférence soit par des données simples. Un agent a ses propres compétences qui lui permettent de réaliser ses fonctions. Les caractéristiques sont les propriétés de l'agent. Enfin, le composant communication décrit toutes les communications possibles entre les différents agents. Les patterns de communication correspondants peuvent être définis au niveau des AIPs (Agent Interaction Protocol).

L'environnement est explicitement défini dans le métamodèle ADELFE. Il contient un ou plusieurs éléments que les agents peuvent percevoir. Ce

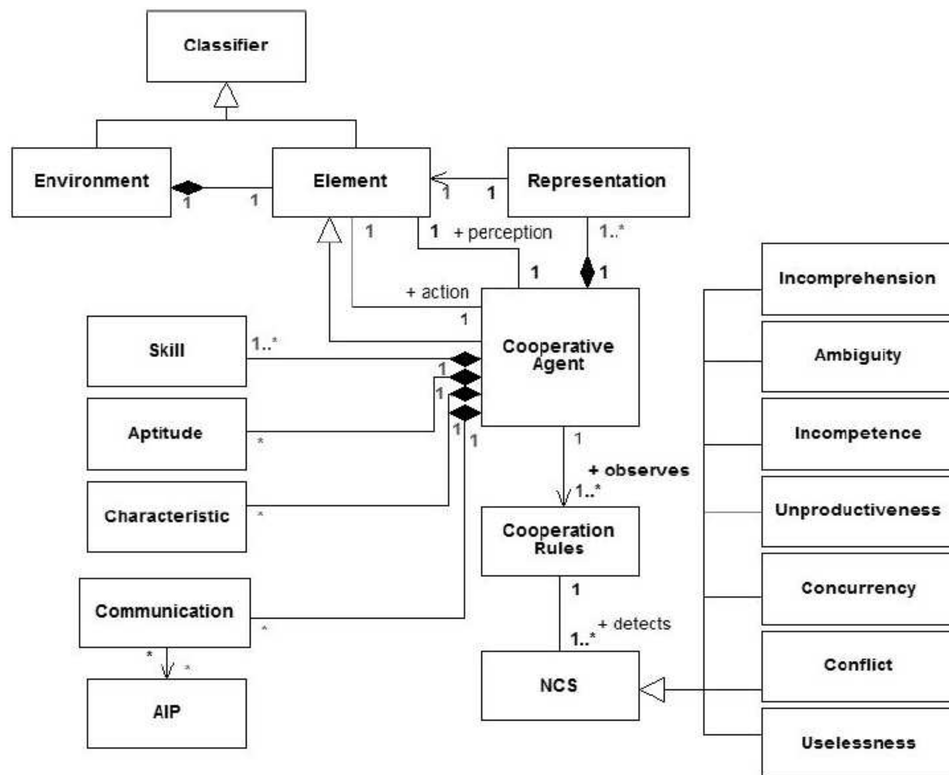


Fig. 3.10: Le métamodèle d'ADELFE

métamodèle est intéressant puisqu'il décrit d'une manière assez complète les caractéristiques des agents coopératifs. Cependant, il reste très spécifique aux systèmes basés sur la coopération des agents. Ce métamodèle n'est donc pas générique. De plus, ADELFE ne peut pas être considéré comme un métamodèle exhaustif puisqu'il est principalement centré sur la coopération entre les agents. Il ne décrit pas les autres concepts qu'on peut trouver dans la littérature tels que l'organisation, les plans, les tâches, les actions.

Les notations

Dans ADELFE, le choix a été fait d'utiliser les notations dédiées à l'objet et plus particulièrement la notation UML. Ce choix est justifié par le fait qu'un agent peut être vu comme une évolution de l'objet et possède de nombreux points communs avec ce dernier [Ode02]. Cependant, l'objet, en tant que tel, n'est pas suffisant pour exprimer les concepts propres aux agents. Dans ADELFE, les agents coopératifs sont composés de modules spécifiques, suivent un cycle de vie "perception-décision-action" et peuvent participer à des négociations complexes dans lesquelles peuvent survenir des situations non coopératives. Ces aspects ont été traités en utilisant un profil UML [Des00] qui permet de définir des stéréotypes de manière flexible.

Dans ADELFE, neuf stéréotypes ont été définis pour exprimer comment un agent est formé et/ou comment son comportement peut être exprimé. Ces stéréotypes permettent de traiter les agents comme des classes ayant une sémantique particulière.

Le stéréotype *"cooperative agent"* exprime qu'une classe est un agent qui a une attitude coopérative. Cette classe abstraite possède une opération *run* simulant le cycle de vie de l'agent. D'autre part, cette classe possède également les opérations *perceive*, *decide* et *act*. L'agent est activé par l'opération *run*.

Le stéréotype *"characteristic"* est utilisé pour pointer les propriétés intrinsèques ou physiques d'un agent coopératif. Un attribut ainsi stéréotypé représente la valeur de cette propriété. Une caractéristique peut être lue ou modifiée à n'importe quel moment du cycle de vie de l'agent ou par une opération également stéréotypée *characteristic*. Ce type d'opération peut être appelé par les autres agents. Pour mieux expliciter la notion d'un attribut stéréotypé *characteristic*, nous pouvons prendre comme exemple une application de robotique. Les robots peuvent être modélisés en tant qu'agents, les attributs caractéristiques peuvent être le nombre de roues, leur couleur, leur poids, leur position dans l'espace, la vitesse maximum de déplacement, etc.

Le stéréotype *"perception"* exprime la perception de l'environnement physique ou social. Ce stéréotype s'applique sur les attributs et les opérations. Les attributs représentent les données provenant de l'environnement alors que les opérations sont des moyens de mettre à jour ces données.

Le stéréotype *"action"* est utilisé pour signaler un moyen d'agir sur l'environnement durant la phase d'action. Les opérations sont les actions possibles pour un agent, alors que les attributs sont des paramètres de ces actions. Un agent est le seul à pouvoir activer ses propres actions afin de conserver son autonomie. Ainsi, une opération ou un attribut stéréotypé *"action"* est nécessairement privé.

Le stéréotype *"skill"* est utilisé pour étiqueter les compétences qui sont des connaissances spécifiques à un domaine. Les opérations représentent les règles de raisonnement que peuvent avoir les agents. Les attributs sont des données (ou faits) sur le monde ou les paramètres des opérations stéréotypées *skill*. Ces attributs ou opérations sont également privés afin de conserver l'autonomie de l'agent. Les opérations peuvent être appelées uniquement lors de la phase de décision de l'agent.

Le stéréotype *"aptitude"* exprime la capacité d'un agent à raisonner sur ses perceptions, ses connaissances et ses compétences. Les opérations expriment le raisonnement qu'un agent est capable de faire, par exemple de l'inférence sur ses compétences (qui peuvent alors être des règles et des faits). Les attributs représentent les données de fonctionnement ou les paramètres du raisonnement. Une opération ou attribut stéréotypé *aptitude* peut seulement être accessible à l'agent lui-même, pour exprimer son autonomie de décision.

Le stéréotype *"representation"* est un moyen d'indiquer les représentations du monde que possède l'agent. Les attributs stéréotypé *representation* sont des unités de connaissances. Les opérations *representation* sont des moyens de les manipuler. Etant donné que les connaissances d'un agent sont locales et personnelles (à moins qu'il ne les communique via des messages par exemple), les attributs et les opérations *representation* sont privés.

Le stéréotype *"interaction"* étiquette les outils qui permettent à l'agent de communication. Ainsi, les opérations *interaction* expriment la capacité d'un agent à interagir avec les autres alors que les attributs *interaction* représentent les données de fonctionnement ou les paramètres des interactions.

Le stéréotype *"cooperation"* implante les règles de résolution des situations non coopératives. Ces règles permettent à la fois de détecter ces situations et de les résoudre. Les opérations *cooperation* sont prioritaires sur les

actions choisies en cas de fonctionnement normal.

Les aspects dynamiques ont été modélisés grâce à des machines à états finis et des diagrammes d'interaction. Les machines à états finis modélisent le comportement interne de l'agent. Afin de représenter les interactions entre les agents, ADELFE utilise la notation A-UML (c.f. section 4.1).

Les outils

ADELFE possède deux sortes d'outils : OpenTool pour prendre en charge les notations et AdelfeToolkit pour prendre en charge le processus.

OpenTool qui a été proposé par TNI(www.tni.fr), est un outil de conception UML semblable à Rational Rose. Il a l'avantage d'être facilement modifiable en fonction des besoins des utilisateurs. Une version d'OpenTool a pu ainsi être modifiée pour répondre aux spécifications d'ADELFE : OpenTool UML1.4 / Adelfe 1.2. Cette modification concerne principalement l'intégration des stéréotypes spécifiques à ADELFE ainsi que leurs règles de bonne utilisation et l'intégration des diagrammes de protocole A-UML.

Le suivi du processus de développement est réalisé par l'outil Adelfe-Toolkit. Celui-ci permet de faire une synthèse sur les tâches déjà réalisées, les artefacts déjà effectués et ceux qui restent à faire.

Le processus de développement

Le processus de développement d'ADELFE est basé sur le processus *RUP(Rational Unified Process).* Au niveau de la méthodologie ADELFE, quatre phases de développement sont couvertes : la collecte des besoins préliminaires, la collecte des besoins finaux, l'analyse et la conception. ADELFE a représenté son processus de développement de manière explicite en utilisant la notation SPEM (Software Process Engineering Meta-model) [Gro02] qui est un profil UML dédié à la représentation des processus [GMP03]. La notation SPEM introduit trois concepts selon lesquels a été décrit le processus ADELFE :

- définitions de travail (WorkDefinition ou *WDi*) : pour représenter les phases du RUP (Expression des besoins, Analyse et Conception),
- activités (Activity ou *Aj*) : qui décomposition chaque phase en plusieurs activités,
- étapes (Step ou *Sk*) : qui décompose chaque activité en plusieurs étapes.

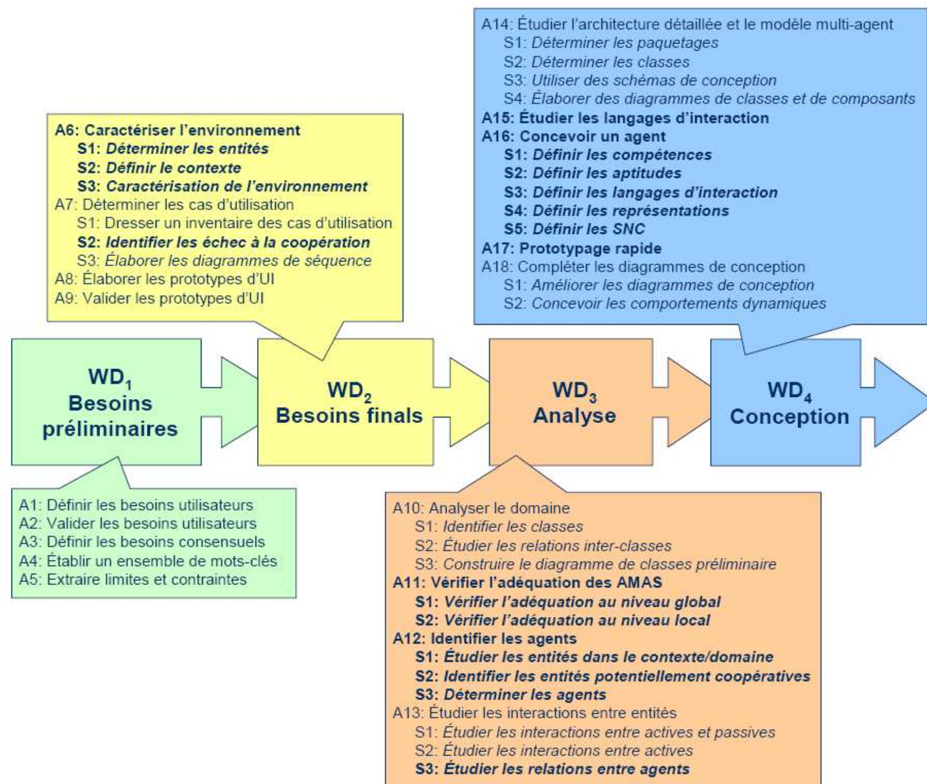


Fig. 3.11: Le processus ADELFE

Pour tenir compte des spécificités des agents, certaines étapes et activités ont été rajoutées aux quatre phases citées préalablement. Le processus détaillé d'ADELFE est décrit dans la figure 3.11 (dans cette figure, la notation utilisée n'est pas celle de SPEM).

Analyse et conclusion

ADELFE fournit une méthodologie interessante aidant au prototypage rapide des systèmes multi-agents adaptatifs. Les modèles de ADELFE sont exprimés par une évolution des langages UML et AUML. Ceci permet au développeurs familiarisés avec ces notations de facilement l'adopter. De plus, contrairement à certaines méthodologies, ADELFE modélise explicitement son processus de développement. Le déroulement de ce processus est pris en charge par l'outil AdelfeToolkit, ce qui facilite sa mise en œuvre et son suivi. Cependant, nous pouvons noter que le processus de développement n'est pas entièrement couvert. Ainsi, les phases postérieures à la conception telles que l'implémentation ou la validation ne sont pas encore traitées. Actuellement, le prototype d'AdelfeTookit vérifie uniquement la production, mais non la validité ou la pertinence des productions. Nous relevons donc l'absence de

vérification sémantique.

Dans les concepts introduits par ADELFE, nous remarquons l'absence de l'aspect organisationnel (structure organisationnelle, règles organisationnelles, rôle etc.) ce qui oblige le concepteur à assurer la synchronisation entre les différents agents suivant leurs comportements et leurs compétences. Le concepteur doit exprimer sa structure organisationnelle sous forme de règles de coopération propres à chaque agent. Ainsi, l'organisation dans ADELFE est connue à travers un comportement émergent.

Enfin, un des principaux points faibles est le manque de fondements théoriques formels sur lesquels ADELFE peut se reposer pour aider à une conception sûre en terme de spécification des comportements des agents. Ces fondements formels peuvent aider à étudier les propriétés des systèmes multi-agents et contribuer ainsi à une meilleure maîtrise des théories de l'émergence et de l'auto-organisation.

5.2 La méthodologie Gaia

La méthodologie Gaia [WJK00] [ZJW03] utilise une approche centrée sur l'organisation pour analyser et concevoir un système multi-agents. Elle est considérée comme une méthodologie générique permettant le développement de tous types de systèmes. Gaia prend en considération deux niveaux : un macro-niveau (qui modélise une société d'agents) et un micro-niveau (qui se focalise sur l'agent).

Le métamodèle

Le métamodèle fourni par Gaia (cf. figure 3.12) est principalement basé sur deux concepts : l'organisation et le rôle. Un système multi-agents est structuré par une ou plusieurs organisations. Ces organisations définissent des règles organisationnelles telles que la vivacité et la sûreté. Ces règles vont permettre de structurer les communications et les rôles. Les organisations sont contenues dans une structure. Elles obéissent à une topologie et un régime de contrôle.

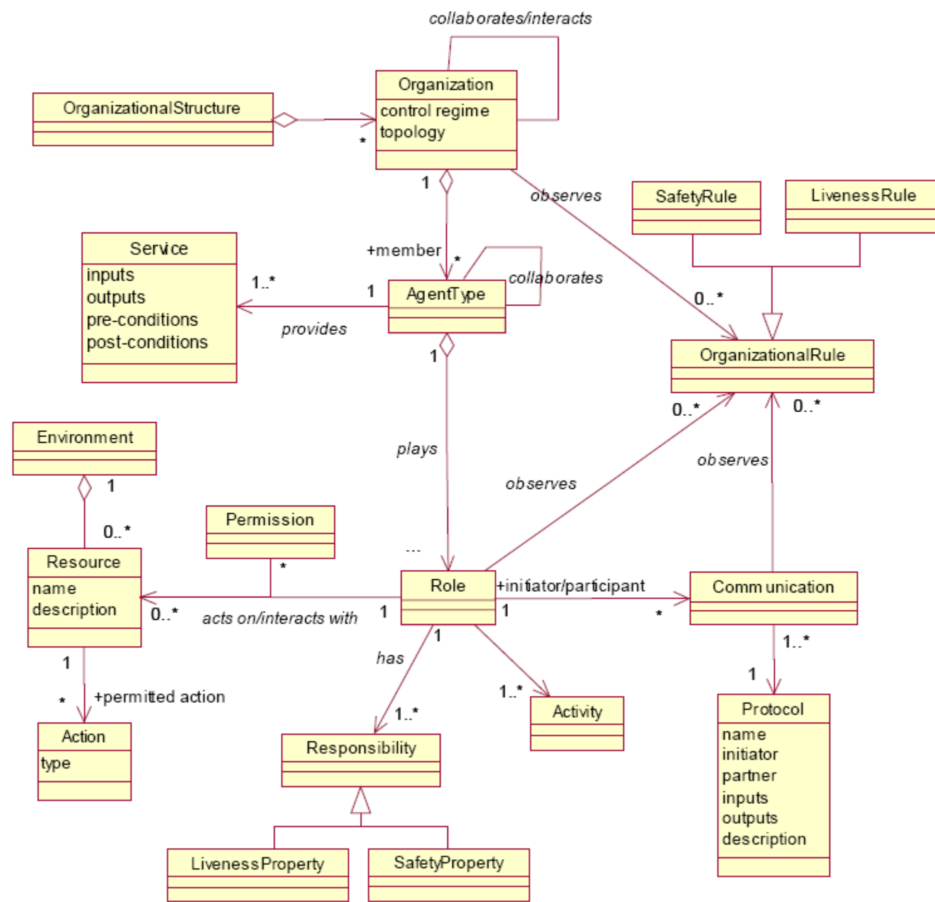


FIG. 3.12: Le métamodèle Gaia

L'organisation contient plusieurs agents qui fournissent des services et qui collaborent entre eux. Un agent joue un ou plusieurs rôles. Le rôle a des responsabilités et respecte les propriétés de vivacité et de sûreté. Le rôle permet également à l'agent - si la permission lui est accordée - d'accéder aux ressources de l'environnement. Le rôle définit des activités. Selon la sémantique adoptée par Gaia, les activités sont l'équivalent des méthodes dans le paradigme orienté objet. Elles correspondent à une unité d'action que l'agent exécute sans qu'il n'ait recours à une interaction avec un autre agent. Enfin, le rôle permet à l'agent de participer ou d'être l'initiateur de communication. Ces communications sont structurées selon un protocole qui est considéré comme une activité qui nécessite l'interaction avec d'autres agents.

Les notations

Gaia ne fournit pas de notations graphiques à proprement parler. Elle fournit cependant des tables permettant de décrire certains modèles. En se basant sur le métamodèle présenté précédemment, Gaia introduit six principaux modèles d'analyse et de conception :

- le modèle de rôle : il identifie les différents rôles devant être joués par les agents du système. Le modèle de rôle est décrit par une table comprenant quatre parties. La première partie correspond à une description textuelle du rôle. La deuxième partie décrit les activités et les protocoles correspondant au rôle. La troisième partie décrit les permissions. Dans cette partie les ressources sont représentées par des variables labellisées et les permissions par des attributs décrivant les droits octroyés pour manipuler les ressources (lecture, écriture, etc.). Enfin la quatrième partie décrit les responsabilités du rôle à travers des propriétés de sécurité et de sûreté. Celles-ci sont exprimées en utilisant les opérateurs de logique temporelle. Un exemple du modèle de rôle représentant le rôle d'un *reviewer* est donné dans la figure 3.13.

Role Schema: REVIEWER

Description:
This preliminary role involves receiving papers for review from some conference official, reviewing that paper, and sending back a completed review form.

Protocols and Activities:
ReceivePaper, ReviewPaper, SendReviewForm

Permissions:
reads *Papers* // *all the papers it receives*
changes *ReviewForms* // *one for each of the papers*

Responsibilities
Liveness:
REVIEWER = (ReceivePaper.ReviewPaper.SendReview)^{maximum_number}

Safety:
- $number_of_papers = number_of_reviewforms$

FIG. 3.13: Exemple de modèle de rôle [ZJW03]

- le modèle d'interaction : il définit les protocoles de communication entre agents. Dans Gaia, un protocole définit une action demandant une interaction entre deux agents. Le modèle d'interaction est défini par une table comprenant un label, un expéditeur, un récepteur, une description, des entrées et des sorties et une description textuelle. La figure 3.14 représente un exemple de table d'interaction illustrant la réception d'un papier par un reviewer,

Protocol Name: **Receive Paper**		
Initiator: **??** (PC Chair or PC Member)	Partner: **Reviewer**	Input: **Paper info**
Description: When a paper has to be assigned to a reviewer it (by someone undefined at this stage) it will be proposed by sending paper info to one of the potential reviewer		Output: **No, don't review** **OR** **Yes, I review it, send me the full paper**

Fig. 3.14: Exemple de modèle d'interaction [ZJW03]

- le modèle d'agent : il attribue les rôles aux différents agents du système. Il identifie les différents types d'agents et leurs instances. Ce modèle est une arborescence dont les racines correspondent aux agents et les feuilles correspondent aux rôles qui leur sont associés,
- le modèle de service : il définit les différents services offerts par le système. Un service est vu comme une fonction qu'un rôle offre. Ainsi, chaque protocole identifié dans le modèle de rôle va se transformer en service. Chaque service possède les caractéristiques suivantes : des entrées, des sorties, des pré-conditions d'exécution et des post-conditions. Les pré et post-conditions sont extraites à partir des propriété de sûreté définies dans le modèle de rôle,
- le modèle d'organisation ou d'accointances : il définit les liens de communication entre les différents types d'agents. La structure de l'organisation est représentée par des graphes orientés dont les nœuds correspondent aux types agents alors que les arcs représentent les liens de communication entre ces noeuds,
- le modèle environnemental : il décrit les différentes ressources accessibles caractérisées par les types d'actions que les agents peuvent entreprendre pour les modifier.

La notation introduite par Gaia est considérée comme semi-formelle. Elle a été appliquée avec une grande facilité et plusieurs exemples pédagogiques ont été publiés (dont l'organisation d'une conférence ou l'étude d'un système manufacturier [ZJW03]). Nous remarquons dans cette notation que les aspects dynamiques ne sont pas réellement modélisés. Ces aspects sont uniquement introduits au niveau des propriétés de sûreté et de vivacité. Pour pallier ce problème, l'utilisation de AUML a été suggérée [CZ04].

Les outils

A ce jour, il n'existe pas d'outils supportant l'application de la méthodologie Gaia. Par conséquent, le développeur ne peut pas exploiter ou réutiliser les modèles Gaia.

Le processus de développement

Tout comme ADELFE, le processus de développement de Gaia ne couvre que quelques phases de développement : la collecte des besoins, l'analyse et la conception. La figure 3.15 montre comment est organisé ce processus et quels sont les modèles à élaborer dans chaque phase.

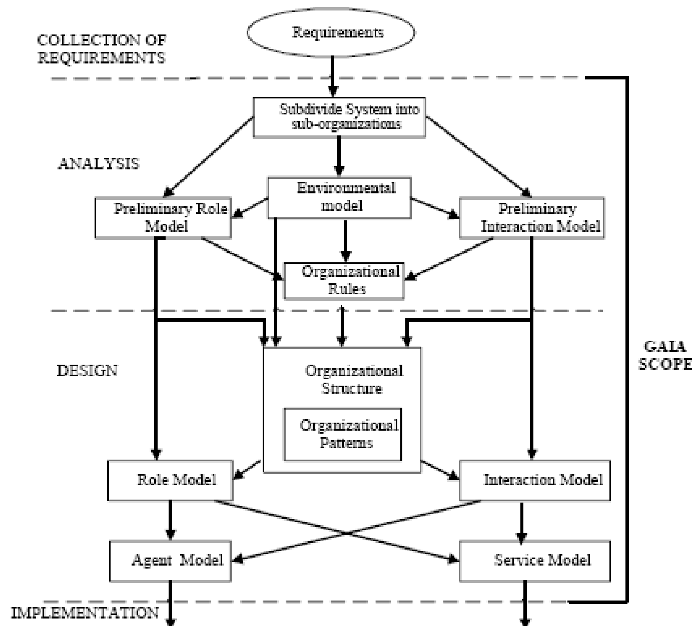


FIG. 3.15: Le processus de développement Gaia

A partir de la collecte des besoins, le système est divisé en plusieurs sous-organisations. Pour chaque sous-organisation, les modèles préliminaires de rôle et d'interaction sont réalisés ainsi que le modèle d'environnement. Au niveau de ces différents modèles, les règles organisationnelles (sûreté et vivacité) sont identifiées.

Au niveau de la conception, les modèles de rôle et d'interaction sont raffinés et les modèles d'agent, d'organisation et de service sont élaborés ce qui permet d'identifier les patterns organisationnels qui peuvent être réutilisables. La phase d'implémentation est aussi mentionnée bien qu'elle ne soit pas encore réellement traitée. Moraitis *et al.* [MPS02] proposent cependant une démarche qui permet de passer d'une modélisation Gaia à une implémentation avec la plate-forme JADE.

Analyse et conclusion

La méthodologie Gaia oriente le développement vers la spécification des organisations, ce qui est considéré comme une manière plus *"naturelle"* d'ap-

préhender le développement des systèmes multi-agents [FGM03]. Les modèles proposés par Gaia rendent la manipulation des concepts de base tels que l'agent, l'environnement, l'organisation et l'interaction facile ce qui a certainement fait sa popularité. Le fait de pouvoir exprimer les propriétés de vivacité et de sûreté permet d'avoir une conception précise. Cependant, ceci nécessite une grande expertise de la logique temporelle.

Par contre, plusieurs inconvénients peuvent être relevés. Tout d'abord la notation adoptée dans Gaia est plutôt limitée. Elle peut être bien adaptée pour l'analyse des systèmes mais ne peut couvrir une conception détaillée. Nous pouvons aussi noter que la gestion de la complexité organisationnelle ou fonctionnelle n'est pas prise en compte. Gaia est limitée aux applications à agents à forte granularité, peu nombreux, et avec une organisation statique, ce qui rend le passage à l'échelle difficile. De plus, elle ne peut permettre l'application des théories d'émergence, d'auto-adaptation et d'intelligence or ces théories constituent une forte valeur ajoutée des systèmes multi-agents.

Le processus de développement couvert par Gaia est très limité et ne dépasse pas la phase de conception. Il n'y a donc pas de phase de vérification/validation. La gestion de projet, comme celle élaborée dans ADELFE n'est pas prise en compte. Néanmoins, les délivrables et différents modèles à fournir au cours du processus sont très clairement définis.

5.3 La méthodologie Ingenias

Dans Ingenias [PGS03], l'approche générale pour spécifier un système multi-agents consiste à diviser le problème en plusieurs aspects plus concrets qui forment les différentes vues du système. Le but global d'Ingenias est de fournir un ensemble de méthodes et d'outils de développement génériques permettant le développement des systèmes multi-agents.

Ingenias est fondée sur la définition de métamodèles décrivant les différents aspects d'un système multi-agents ainsi que leurs relations. Sur cet aspect, Ingenias s'inscrit dans la continuité de la méthodologie MESSAGE [CLC+01] [FGSP04] qui a appliqué les techniques de métamodélisation dans le domaine des systèmes multi-agents. Ingenias a fait évoluer les métamodèles en incorporant d'autres concepts. Les modèles sont ensuite construits avec des instances des entités métamodèles. Ainsi, le métamodèle est considéré comme la définition des langages de modélisation pour la description des systèmes multi-agents.

Le métamodèle

Ingenias considère cinq perspectives (ou vues) dans un système multi-agents :

- l'agent
- l'organisation
- les interactions
- les tâches et les buts
- l'environnement

La figure 3.16 résume les métamodèles adoptés par Ingenias. En effet, au niveau de la méthodologie, chaque perspective est décrite de manière détaillée par un métamodèle.

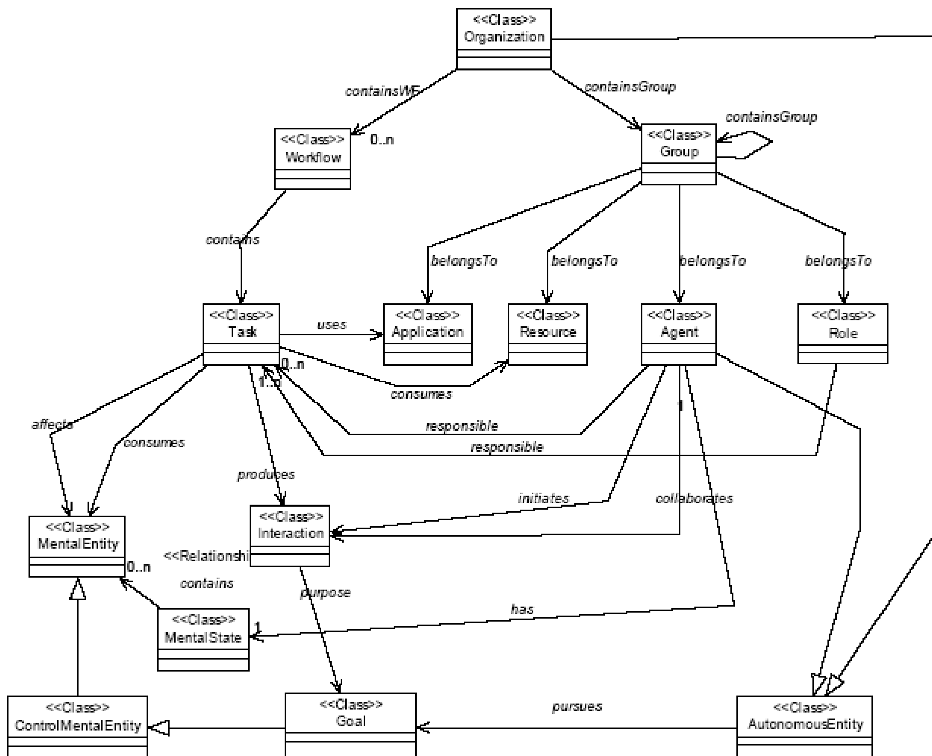


FIG. 3.16: Un métamodèle Ingenias résumé [CBP05]

La perspective Agent décrit l'agent comme étant une entité autonome faisant parti d'un groupe. Il est responsable de l'exécution de certaines tâches. L'agent a un état mental et des buts qui permettent de définir le contrôle de l'agent. Les entités mentales sont produites et consommées par les tâches. L'agent peut initier ou collaborer à une interaction.

La perspective Organisation établit l'architecture du système. La vision de l'organisation adoptée dans Ingenias partage celle présentée par Ferber et Gutknecht [FG98] et qui stipule qu'il y a des relations structurelles qui ne peuvent pas être restreintes à des hiérarchies entre rôles. Ces structures sont déléguées à des entités spécialisées qui sont les groupes. Les groupes incluent des applications, des ressources, des agents et des rôles. Il n'existe pas dans ce métamodèle de liens directs entre ces différents concepts. Le lien se fait à travers les tâches. Ces tâches sont définies soit par les rôles, soit par les agents. Pendant leur exécution, celles-ci vont utiliser des applications, consommer des ressources, affecter et consommer des entités mentales et produire des interactions. Tout comme l'agent, l'organisation est aussi considérée comme une entité autonome poursuivant un but. Les fonctionnalités de l'organisation sont exprimés par des workflows qui montrent les associations producteur-consommateur entre tâches ainsi que les responsabilités pour leur exécution, et les ressources associées à chacune.

La perspective Environnement identifie les ressources et les applications existantes.

La perspective Buts-Tâches identifie les buts et les tâches. La réalisation des buts s'effectue à travers l'exécution de certaines tâches. Aussi, il faut identifier les ressources nécessaires pour l'exécution des tâches ainsi que les composants logiciels (applications) nécessaires. Certaines tâches produisent des interactions nécessaires à la réalisation d'un but.

Enfin, la perspective Interaction décrit comment se réalise la coordination entre les agents. La rélisation d'une interaction est motivée par la réalisation d'un but. Compte tenu qu'une interaction est réalisée par une tâche reliée à l'agent ou au rôle, celle-ci peut affecter l'état mental des agents.

Ce métamodèle donne une vue globale des concepts associés à Ingenias. Comme nous l'avons mentionné au début du paragraphe, chaque perspective est décrite par son propre métamodèle. Ceux-ci sont ensuite utilisés pour générer les modèles correspondants. Cette opération n'est cependant pas triviale vu les différentes dépendances qui existent entre les perspectives. Par exemple, les tâches dans le workflows du modèle organisationnel devraient être décrites aussi dans la perspective buts-tâches.

Les notations

Les perspectives décrites précédemment constituent la base du développement et sont raffinées tout au long du cycle de développement. Ingenias n'introduit pas une notation spécifique pour produire les spécifications, mais

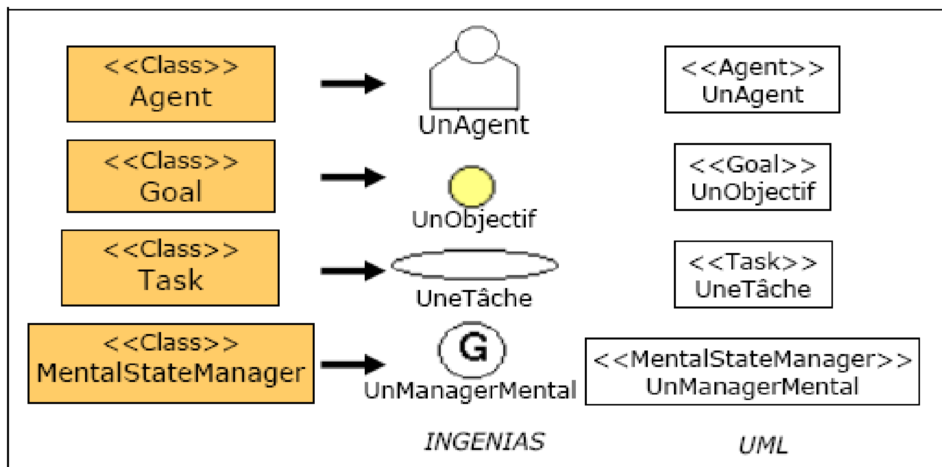


Fig. 3.17: Un résume du métamodèle Ingenias [Pav06]

se contente de bien définir les sémantiques des concepts qui vont être utilisés. Ainsi, Ingenias offre une grande flexibilité concernant les notations à utiliser. Divers diagrammes peuvent être utilisés pour décrire les spécifications. Ces diagrammes doivent respecter la sémantique décrite au niveau du métamodèle. Ingenias adopte deux sortes de notations :

- des notations basées sur une extension du langage UML,
- des icônes spécifiques respectant la sémantique des concepts décrits dans le métamodèle.

Par exemple, pour décrire la perspective Agent, Ingenias propose l'utilisation des stéréotypes UML qui font référence aux entités de métamodèle, ou bien un ensemble d'icônes spécifiques (cf. figure 3.17).

De même, la perspective interaction peut être spécifiée avec différentes notations : avec un diagramme de collaboration d'UML, le diagramme d'interaction de GRASIA (GRupo de Agentes Software :Ingenieria y Aplicaciones), ou avec un diagramme de protocole d'AUML. Ceci fournit une grande flexibilité aux développeurs puisque ceux-ci peuvent utiliser les notations qui leur sont familières.

Les outils

Ingenias fournit un environnement de développement appelé IDK (Ingenias Development Kit). Ce kit est un ensemble ouvert d'outils basés sur les métamodèles d'Ingenias. L'IDK offre :

- l'éditeur Ingenias : c'est l'outil de développement principal d'Ingenias.

C'est un support de modélisation facilitant la création et la modification des diagrammes. Afin de gérer le grand nombre de diagrammes pouvant être produits pour modéliser un système multi-agents, l'éditeur permet de les structurer en une hiérarchie de paquets. L'IDK a une représentation des métamodèles d'Ingenias via XML. Celle-ci définit les concepts, les relations, les diagrammes de métamodèle, ainsi que les associations entre les concepts et les éléments graphiques qui les représentent.
– les modules IDK : ce sont des outils pour la vérification et la validation des modèles, ainsi que des moteurs de transformation qui visent à dériver le code pour la plate-forme finale. Ainsi, ces outils traitent des spécifications pour produire en sortie : le code source (des templates pour chaque plate-forme cible sont spécifiées ainsi que des procédures d'extraction d'information des modèles), des rapports (analysant les spécifications pour recueillir une information statistique sur l'utilisation des différents éléments), et des transformations vers d'autres modèles.

L'IDK est très important dans le processus de développement d'Ingenias puisqu'il permet le développement rapide d'applications, le découplage des spécifications de l'implémentation, et la vérification de la spécification suivant les besoins d'implémentation.

Le processus de développement

Ingenias couvre les phases d'analyse, de conception, de codage et d'implémentation. Son processus de développement est basé sur un cycle de développement standard : le Processus Unifié (Unified Process). Cependant, Ingenias raffine ce processus de développement en décrivant avec plus de détail les activités pouvant être réalisées dans la phase d'analyse et de conception [Pav06]. Ces activités sont en fait, basées sur l'approche dirigée par les modèles (brièvement présentée dans la section 5.3 du chapitre 2) qui se base elle-même sur la spécification des métamodèles et la transformation successives des modèles jusqu'au code source. Ingenias identifie deux types principaux de rôles dans le processus de développement :

– le développeur des SMA qui utilise l'éditeur de l'IDK pour spécifier les modèles qui décrivent le système multi-agents. Ces modèles peuvent être vérifiés et validés avec des modules d'IDK. Une fois validés, les modules de génération de code facilitent l'implémentation pour la plateforme cible.
– l'ingénieur Ingenias qui connaît le métamodèle Ingenias et peut travailler sur les outils de l'IDK. Celui-ci va être responsable de la production de nouveaux modules pour vérifier les propriétés de modèles basés sur de nouveaux concepts qui viennent étendre les métamodèles d'Ingenias ou pour la génération de code en une plate-forme cible spé-

cifique. Il est possible également de personnaliser l'éditeur pour un propos spécifique, par exemple, pour l'adapter à un langage de modélisation spécifique du domaine d'application. Ce dernier exige de modifier le métamodèle et demande une connaissance plus approfondie du métamodèle Ingenias.

Analyse et conclusion

La méthodologie Ingenias offre une manière très flexible pour le développement des systèmes multi-agents. Elle applique une technique d'ingénierie basée sur l'approche IDM (Ingénierie Dirigée par les Modèles) qui permet le passage des modèles vers le code. Un des grands avantages d'Ingenias est le fait d'avoir un environnement de développement offrant les outils nécessaires pour l'application de la méthodologie. Cependant, nous pouvons reprocher à Ingenias le manque de formalisme permettant de spécifier et de vérifier les propriétés relatives aux SMA telles que les propriétés de vivacité et sûreté. Ces propriétés, bien qu'elles soient considérées comme des propriétés comportementales devraient être prise en compte très tôt au niveau du cycle de vie. Leur conservation devraient être garanties au niveau du code source généré. De plus, l'IDK n'offre pas de formalisme pouvant exprimer explicitement les règles de transformation et de génération de code, ce qui ne permet pas de conserver le savoir-faire.

5.4 La méthodologie PASSI

PASSI (Process for Agent Societies Specification and Implementation) [CGSS05] est une méthodologie se basant sur des concepts provenant de l'ingénierie orientée objet et de l'intelligence artificielle. PASSI est une méthodologie générique pouvant s'appliquer à n'importe quel domaine. PASSI a la particularité de prendre en compte la modélisation des agents mobiles. Cette modélisation respecte les spécifications FIPA.

Le métamodèle

Le métamodèle PASSI est décomposé en trois domaines(cf. figure 3.18) :

- le domaine de la solution
- le domaine d'agence
- le domaine du problème

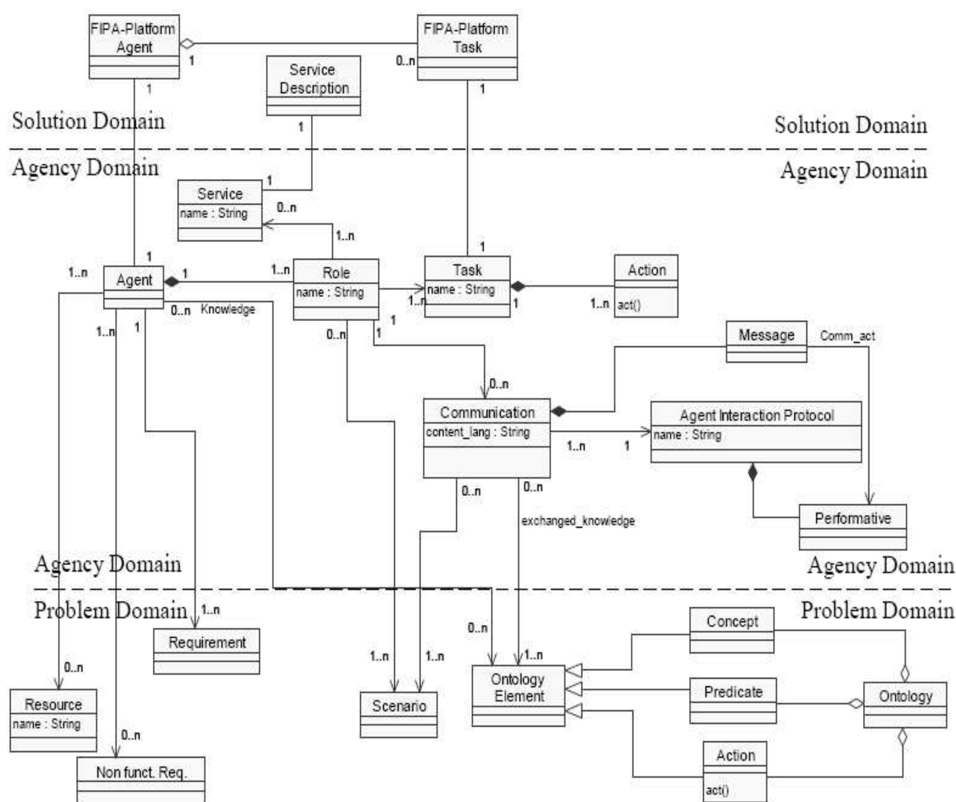


Fig. 3.18: Le métamodèle PASSI [CBP05]

Au niveau du *domaine du problème*, la capture des besoins est effectuée et traduite en terme de scénarios (qui décrivent les interactions entre les acteurs et le système) et d'environnement. L'environnement est défini par un ensemble d'éléments ontologiques : les concepts qui définissent les différentes catégories du domaine, les actions qui regroupent les opérations pouvant être exécutées et les prédicats qui permettent d'ajouter des éléments dans le domaine. L'environnement contient aussi des ressources auxquelles l'agent va accéder.

Le *"domaine d'agence"* représente les agents qui peuvent avoir connaissance des éléments ontologiques, des besoins fonctionnels et non fonctionnels définis dans la couche précédente. Les agents accèdent aux ressources. Ils ont au moins un rôle qui propose des services, effectuent des tâches, définissent des communications et exécutent les scénarios. Les communications sont constituées de messages définis par des performatives qui vont spécifier la sémantique du contenu. Les performatives sont contenues dans les protocoles d'interaction qui structurent les communications.

Au niveau du *domaine de solution* la structure de l'implémentation est spécifiée. Selon PASSI, l'implémentation doit être compatible à la plateforme FIPA. Ainsi, trois éléments vont être choisis : FIPA-Plateform Agent qui définit la classe qui va implémenter l'agent, FIPA-Platform Task qui définit la structure d'implémentation des tâches et Service Description qui va implémenter les services.

Les notations

PASSI réutilise fortement UML [CP01]. En se basant sur cette notation, PASSI fournit cinq modèles permettant de décrire les concepts identifiés au niveau du méta-métamodèle. Ces modèles sont :

- modèle de collecte des besoins : celui-ci est utilisé dès les premières phases relatives à la collecte des besoins. Il permet "d'agentifier" le système à développer c'est-à-dire de le décrire sous forme d'agents, de rôles et de taches. De plus ce modèle permet de décrire le domaine d'application. Pour cela il utilise le diagramme de cas d'utilisation d'UML, de paquetages stéréotypés, le diagramme de séquences (associés aux cas d'utilisation) et le diagramme d'activités,
- modèle de société d'agents : regroupe la description de trois concepts qui sont les ontologies, les rôles et les protocoles d'interaction. Ce modèle est focalisé sur la description des interactions sociales ainsi que les dépendances entre les agents,
- modèle d'implémentation : permet de définir les constituants des agents en terme de ressources internes, de connaissances (attributs) et de tâches. Ce modèle permet également la définition du comportement des agents grâce à des diagrammes d'activités où apparaissent les tâches des agents ainsi que leurs interactions. Ce modèle est réalisé par les diagrammes de classes et d'activités classiques,
- modèle de code : permet d'identifier les modules réutilisables dans le cas où un développement préalable a été réalisé. Au même titre que le modèle d'implémentation, le modèle de code est réalisé par les diagrammes de classes et d'activités,
- modèle de déploiement : permet de spécifier la configuration du système. Ce modèle est réalisé par les diagrammes de déploiement UML.

Les outils

La modélisation avec PASSI est supportée par l'outil PTK (PASSI Toolkit) [CCS04]. PTK est un add-in de Rational Rose. En effet, tous les diagrammes et stéréotypes spécifiques sont intégrés dans une extension au logiciel Rational Rose, afin de réaliser les modèles d'analyse et conception de manière ergonomique.

Le processus de développement

Le processus de PASSI est un processus pas-à-pas, mais réitérable. Il est composé de cinq phases, composées d'étapes, correspondant aux spécifications des cinq modèles présentés précédemment.

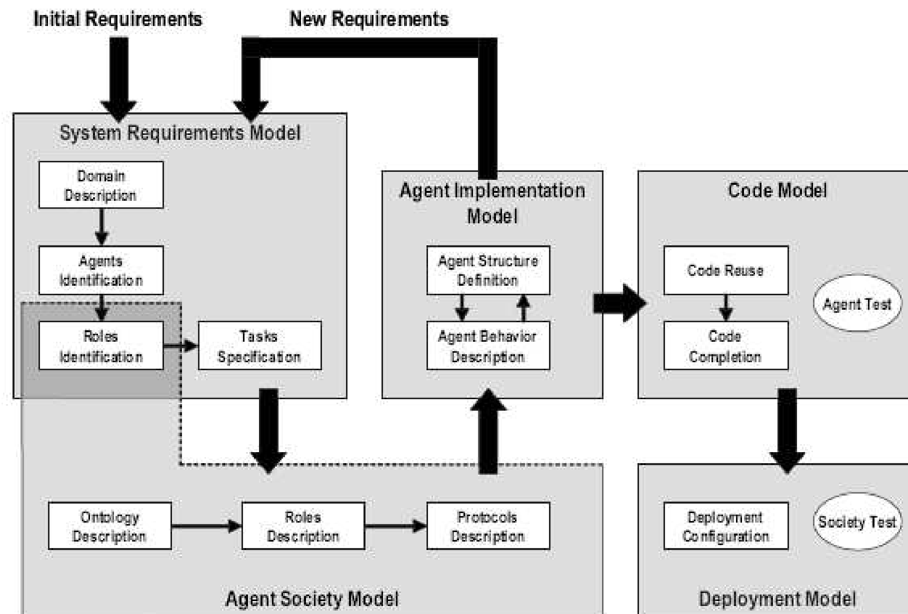


FIG. 3.19: Le processus PASSI [CP01]

Analyse et conclusion

PASSI couvre pratiquement la totalité des phases du cycle de développement. Son utilisation est facile puisqu'elle est basée sur une extension du langage UML. Son application ne nécessite donc pas une expertise particulière (bien que la connaissance des spécifications FIPA soit préférable). PASSI est complètement dissociée des plates-formes d'implémentation bien que l'outil PTK soit orienté vers la génération de code en Java afin d'être compatible avec JADE et FIPA-OS.

Nous pouvons reprocher à PASSI, le manque de formalisme qui rend difficile l'expression des propriétés telles que la vivacité et la sûreté. Aussi, le fait d'être basée sur un langage semi-formel tel que UML rend les phases de tests et de vérification peu efficaces.

5.5 La méthodologie Tropos

Tropos [CGSS05] est une méthodologie de développement fondée sur les concepts utilisés en ingénierie des besoins [GKMP04]. Elle adopte une approche transformationnelle, dans le sens où, à chaque étape, les modèles vont être raffinés de manière itérative par ajout ou suppression d'éléments ou de relations dans les modèles.

Le métamodèle

Tropos a été conçu en se basant sur une approche orientée buts. Le métamodèle [GKMP04] défini par cette méthodologie met ainsi en évidence le concept but qui est défini par une classe *"Goal"*. Deux sous-classes émanent de celle-ci : *"HardGoal"* et *"SoftGoal"*. *"HardGoal"* décrit les besoins fonctionnels tandis que *"SoftGoal"* décrit les besoins non fonctionnels. La réalisation de ces buts est effectuée par l'application d'un ou plusieurs plans.

[CBP05]

FIG. 3.20: Le métamodèle Tropos

Ces plans sont reliés aux acteurs, c'est à dire les différents participants au système (agent, acteur humain, etc.). Les agents héritent de la classe acteur, de même que le rôle et la position. La position est en fait un ensemble de rôle. Un agent peut occuper une ou plusieurs positions qui vont lui assigner plusieurs rôles. Les acteurs utilisent des ressources et lancent l'exécution d'un plan.

Une classe *"dépendance"* spécifie les différentes relations de dépendances entre les agents. Elle indique qu'un acteur dépend d'un autre afin d'atteindre un but, exécuter un plan, ou fournir une ressource. Le premier acteur source s'appelle le *depender*, le deuxième acteur est un *dependee* et l'objet de leur dépendance est le *dependum* qui peut être un but, une ressource ou une tâche.

Ce métamodèle définit les concepts de la notation i*, qui est le langage adopté par Tropos (voir 5.5). Cependant, d'autres concepts sont manipulés qui n'apparaissent pas dans ce métamodèle tels que les capacités (qui modélisent les capacités d'un acteur à définir, choisir et exécuter un plan pour remplir un but), les croyances (qui sont une représentation de la connaissance d'un acteur sur le monde) et les contributions (qui permettent de notifier si les plans contribuent de manière positive ou négative dans la réalisation d'un but).

Les notations

Tropos utilise principalement i* pour modéliser les systèmes. Ce langage se focalise sur l'ingénierie des besoins centrée sur les caractéristiques intentionnelles des agents (leurs buts) [YM94]. Ce langage manipule les différents concepts du métamodèle. C'est un langage graphique qui est facilement compréhensible. Cependant, les diagrammes qui modélisent de réelles applications complexes peuvent vite devenir illisibles. Par ailleurs, i* ne peut être utilisé que dans la phase d'analyse puisqu'il permet uniquement d'exprimer les besoins.

Pour cette raison, Tropos a recours à l'utilisation d'autres notations telles que les diagrammes d'états/transitions pour modéliser les capacités et les plans. Les interactions entre agents sont spécifiées grâce à des diagrammes de protocole AUML.

Une approche plus formelle, appelée Formal Tropos [PPRS03], propose des descriptions textuelles et logiques des besoins. Les descriptions logiques sont basées sur la logique temporelle. Ces descriptions textuelles et logiques peuvent être par la suite traduites automatiquement vers i*. Ceci ajoute de la précision aux notations et langages et rend les diagrammes vérifiables.

Ces différentes notations, sont utilisées pour produire les principaux diagrammes suivants :

- diagramme d'acteur : il décrit les acteurs, leurs buts et leurs dépendances. Ce type de diagramme est employé pour définir les besoins des utilisateurs du système, pour montrer leurs intentions ainsi que les relations entre les acteurs. Ce diagramme représente un acteur par un noeud et les dépendances par les liens entre les nœuds,
- diagramme de but : il montre la structure interne d'un acteur (ses buts, ses plans et ses ressources) et les rapports entre eux,

- diagramme de compétence : chaque compétence (ou ensemble corrélé de compétences) est modélisée par un diagramme d'états/transitions. Les événements externes déclenchent l'état du diagramme de compétence ; les plans sont décrits par les noeuds du diagramme, les arcs des transitions modélisent les événements et les croyances,
- diagramme de plan : chaque noeud de plan du diagramme de compétences peut être encore détaillé par un diagramme d'activités UML,
- diagramme d'interaction d'agent : pour décrire l'interaction des agents, Tropos utilise des diagrammes AUML.

Les outils

Plusieurs outils aident à l'application de la méthodologie Tropos. T-Tool [KPR03] est un outil utilisable durant les phases préliminaires de collecte des besoins. Il permet l'analyse et la validation des spécifications décrites durant cette phase en utilisant les notations formelles introduites par Formal Tropos. T-Tool applique pour cela la technique du *model checking*. T-Tool permet par ailleurs d'animer certains modèles afin d'être validés par les développeurs.

GR-Tool [SPM04] est aussi fourni par la méthodologie Tropos. C'est un outil de conception graphique permettant de spécifier les diagrammes de buts et exécuter les algorithmes qui leur correspond. Tropos fournit aussi ST-Tool (Secure Tropos tool) [GMMZ04] qui est un éditeur graphique qui permet de spécifier des diagrammes de sécurité. Ces diagrammes, qui viennent étendre la méthodologie Tropos, permettent de spécifier les règles de sécurité. Enfin, TAOM4E est une plate-forme permettant l'intégration de différents outils de manière flexible. Le but étant d'appliquer l'approche dirigée par les modèles. [BGG+04b].

Le processus de développement

Tropos couvre une grande partie du cycle de développement. Elle suit une approche incrémentale par raffinement itératif. La démarche proposée par Tropos se décompose en cinq phases [GMP02] :

- l'analyse des besoins initiaux permet une compréhension du problème. Cette phase permet d'identifier les principaux acteurs et leurs buts respectifs dans le système. A ce niveau les diagrammes d'acteurs et de buts sont élaborés,
- l'analyse des besoins finaux fournit une description du système dans son environnement opérationnel et modélise le système en tant qu'ensemble d'acteurs possédant des dépendances. Cette phase permet de raffiner les modèles précédents,
- la conception architecturale définit l'architecture globale du système en termes de sous-systèmes interconnectés par des flots de données et

de contrôle. Ces sous-systèmes sont représentés comme des acteurs et les interconnexions comme des dépendances entre acteurs. Plusieurs styles d'architectures (structure plate, pyramidale, etc.) sont fournis,
- la conception détaillée définit chaque composant architectural en termes d'entrées, de sorties, de contrôle, etc. Cette phase permet de détailler la communication entre acteurs et le comportement des acteurs en différents niveaux. Un niveau détaille un dialogue intra-acteur ou inter-acteurs. Le niveau le plus bas décrit le traitement interne d'un acteur via des diagrammes de plans,
- l'implantation fournit une architecture d'agents BDI sur la plate-forme Jack [BRHL99].

Analyse et conclusion

Tropos est une méthodologie générique pouvant être appliquée à des contextes différents. Cette méthodologie offre une manière flexible pour développer des systèmes multi-agents assistées par plusieurs outils de développement. Tropos couvre la totalité du cycle de développement, et se base sur une notation formelle (Formal Tropos) qui lui permet de vérifier et valider ses modèles. Cependant, celles-ci ne sont applicables que durant les phases d'ingénierie des besoins. D'un autre coté, les activités et les délivrables associés au sein des phases ne sont pas si clairement définis et la lecture de nombreux articles est nécessaire. De plus, Tropos ne couvre qu'un ensemble limité de concepts orientés agent. La dimension organisationnelle, par exemple, n'est pas prise en compte de manière explicite.

5.6 Comparaison

La comparaison des méthodologies fait ressortir toute la diversité des techniques d'ingénierie orientée agent.

Ces méthodologies se limitent parfois à des aspects très spécifiques des systèmes multi-agents ou bien ne couvrent pas la totalité du processus de développement. Généralement, ce sont les phases d'analyse et de conception qui sont les plus couvertes. Ceci nécessite parfois le passage d'une méthodologie à une autre afin de pouvoir couvrir la conception de tout le système. On parle alors de fragments de méthodes pour construire des méthodologies flexibles [CGGS07]. Ceci serait réellement facilité si ces différentes méthodologies sont conformes à un même métamodèle générique. Cependant, dans l'état actuel des choses, les méthodologies ont défini leur propre métamodèle afin de spécifier et de définir les concepts qu'elles utilisent. Ces métamodèles restent orientés vers les caractéristiques de chaque méthodologie et sont ainsi très différents les uns des autres. Un courant vers une unification des méthodologies commence à apparaître. Nous détaillons ceci dans la section suivante.

5.7 Unification des métamodèles

C. Bernon, M. Cossentino et M.P. Gleizes [BCG+04] proposent d'unifier les métamodèles proposés par les méthodologies ADELFE, PASSI et Gaia. L'unification des métamodèles consiste à récupérer les concepts utilisés dans chacun d'eux. Ainsi, pour la structure de l'agent, les concepts utilisés dans ADELFE ont été mixés avec ceux de PASSI et ceux de Gaia. La même chose a été faite pour les interactions. Les structures organisationnelles et sociales ont été décrites quant à elles selon les concepts de Gaia et PASSI. Enfin l'implémentation a aussi été prise en compte selon ce qui a été défini dans PASSI. La figure suivante montre le résultat de cette unification :

Fig. 3.21: Le métamodèle unifié [CBP05]

Le métamodèle généré suite à l'unification d'ADELFE, PASSI et Gaia est plus complet et plus générique. Cependant, celui-ci mixe des concepts de haut niveau et des concepts liés à l'implémentation. Nous pensons que ceci prive le concepteur de son libre choix de la plate-forme d'implémentation. De plus, certains concepts ne sont pas pris en compte. Nous citons principalement les buts, les plans, les normes, etc.

6 Les plates-formes orientées Agent

Tout comme les méthodologies, il existe une multitude de plates-formes multi-agents dédiées à différents modèles d'agent. Les plates-formes fournissent une couche d'abstraction permettant de facilement implémenter les concepts des systèmes multi-agents. D'un autre coté, elle permet aussi le déploiement de ces systèmes. Ainsi, elles constituent un receptacle au sein duquel les agents peuvent s'exécuter et évoluer. En effet, les plates-formes sont un environnement permettant de gérer le cycle de vie des agents et dans lequel les agents ont accès à certains services.

Comme le choix d'une plate-forme d'agent a une grande influence sur la conception et la mise en œuvre des agents, FIPA a produit les normes qui decrivent comment une plate-forme d'agent devrait être. Ces normes existent pour assurer une conception uniforme des agents indépendamment de la plate-forme.

Dans ce qui suit, nous présentons quelques plates-formes. Cette liste n'est pas exhaustive. Elle représente cependant les plates-formes les plus utilisées et les plus citées dans la littérature. Pour comparer les différentes plates-formes, nous appliquons un cadre de comparaison qui se base sur les éléments suivants :

- les types de systèmes visés : certaines plates-formes sont conçues pour développer des applications dans des domaines spécifiques tel que la simulation, les systèmes distribués, etc.
- les modèles d'agents suportés : certaines plates-formes imposent un modèle d'agent spécifique (agent BDI, agent collaboratif, etc.).
- les méthodologies : certaines plates-formes proposent une méthodologie de développement, d'autres plates-formes sont utilisées par certaines méthodologies orientées agent.
- le langage d'implémentation appliqué : généralement, le langage d'implémentation qui est adopté par les plates-formes est Java. Nous étudierons les classes abstraites qui sont fournies par la plate-forme, qui aident à architecturer les applications.

6.1 La plate-forme ZEUS

Zeus [CNNL98] est une plate-forme dédiée pour la construction rapide d'applications à base d'agents collaboratifs. Elle se prête bien aux systèmes économiques qui utilisent des applications de planification ou d'ordonnancement.

Pour implémenter les agents collaboratifs, Zeus se base principalement sur les concepts agents, buts, tâches (que les agents doivent réaliser pour atteindre leurs buts) et faits (qui représentent les croyances des agents). Un

agent dans Zeus est constitué en trois couches : la couche de définition, qui contient les capacités de raisonnement et des algorithmes d'apprentissage, la couche organisationnelle, qui contient la base de connaissances et des accointances de l'agent, et la couche de coordination, qui définit les interactions avec les autres agents. Zeus propose aussi un ensemble d'agents utilitaires (serveur de nommage et facilitateur) pour faciliter la recherche d'agents.

Zeus fournit aussi une méthodologie qui se base sur quatre phases pour la construction d'agents :

- l'analyse du domaine : consiste à modéliser des rôles. A ce stade aucun outil logiciel n'est fournit et le langage UML est utilisé.
- la conception : consiste à spécifier des buts et des tâches qui permettent de les réaliser. Cette spécification se fait par le langage naturel et n'est supportée par aucun outil.
- le développement : Cette phase est réalisée en quatre étapes : création de l'ontologie, création des agents, configuration de l'agent utilitaire, configuration de l'agent tâche et implémentation des agents.Des outils supportant des notations graphiques sont fournies afin d'aider à l'élaboration de ces étapes.
- le déploiement : dans cette phase, le système multi-agents est lancé. Un outil de visualisation permet le suivi de l'exécution. Cet outil permet de visualiser l'organisation, les interactions ayant lieu dans la société d'agents, la décomposition des tâches et l'état interne des agents.

Zeus fournit un environnement de développement d'agents grâce à un ensemble de bibliothèques Java que les développeurs peuvent réutiliser pour créer leurs agents.

6.2 La plate-forme JADE

La plate-forme JADE (Java Agent DEvelopment framework) [BPR99] est certainement celle qui est la plus utilisée par la communauté des systèmes multi-agents. JADE permet de développer et d'exécuter des applications distribuées basées sur le concept d'agents et d'agents mobiles. Elle est compatible à la plate-forme FIPA.

Les agents dans JADE sont implémentés selon 6 propriétés :

- Autonomie : les agents ont leurs propres thread de contrôle qui leur permet de contrôler leurs actions, de prendre leurs propres décisions afin de réaliser leurs buts mais aussi de contrôler leur cycle de vie.
- Réactivité : les agents peuvent percevoir les évènements de leur environnement et réagissent en fonction de ces évènements,
- Aspects sociaux : les agents exhibent des aspects sociaux qui leur permettent de communiquer et d'interagir entre eux. La communication

se fait à travers le passage de messages asynchrones. La communication est considérée comme un type d'actions et peuvent de ce fait intégrer un plan d'actions. Les messages ont une sémantique et une structure définis par le standard FIPA.
- Dynamicité : les agents ont la possibilité de découvrir dynamiquement d'autres agents et de communiquer avec eux.
- Offre de service : chaque agent offre un ensemble de services, il peut enregistrer ses services et les modifier. Il a aussi la possibilité de chercher des agents qui offrent les services dont il a besoin.
- Mobilité : les agents dans Jade ont la possibilité de se déplacer. Les agents sont implémentés dans des conteneurs et ils peuvent se déplacer.

C'est un composant complémentaire ajouté à Jade pour permettre d'intégrer l'implémentation de l'architecture interne des agents. Jadex [PBL05] [SBPL06] prend en compte l'architecture des agents hybrides (c'est-à-dire des agents qui sont à la fois réactifs et proactifs). Pour les agents proactifs, jadex se base sur le modèle BDI. Jade assure la sécurité en offrant aux applications des systèmes d'authentification qui vérifient les droits d'accès des agents.

Jade n'offre pas de méthodologie, par contre plusieurs méthodologies la prennent comme plate-forme cible lors de la génération de code tel que Gaia et PASSI.

L'implémentation de Jade est basée sur Java. La plate-forme peut être répartie sur un ensemble de machines et configurée à distance. La configuration du système peut évoluer dynamiquement puisque la plate-forme supporte la mobilité des agents.

6.3 La plate-forme MADKIT

La plate-forme MadKit (Multi-Agents Developement kit) [FG00] est développée à l'Université de Montpellier II. Bien qu'elle puisse supporter le développement de divers systèmes, elle semble bien adaptée pour les application de simulation.

La plate-forme MADKIT est basée sur les concepts agent, groupe et rôle (figure 3.22). Ces concepts sont appliqués de la manière suivante :
- l'agent : quasiment aucune contrainte n'est posée sur l'architecture interne ou le modèle de comportement de l'agent. L'agent est simplement décrit comme une entité autonome communicante qui joue des rôles au sein de différents groupes. La faible sémantique associée à l'agent est volontaire ; l'objectif étant de laisser le choix au concepteur pour choisir l'architecture interne appropriée à son domaine applicatif.
- le groupe : chaque agent peut être membre d'un ou plusieurs groupes.

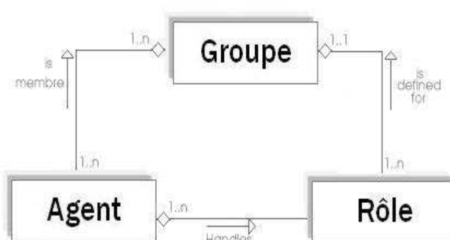


FIG. 3.22: Le métamodèle AALAADIN

Il sert à identifier la structure organisationnelle d'un système multi-agents usuel.
- le rôle : il est considéré comme une représentation abstraite d'une fonction ou d'un service. Chaque agent peut avoir plusieurs rôles et un même rôle peut être tenu par plusieurs agents. Les rôles sont locaux aux groupes.

Cette implémentation correspond à la conception réalisée au niveau de la méthodologie AALAADIN. En effet, à partir des concepts AGR, cette méthodologie définit une démarche de développement axée sur la spécification du cadre organisationnel des applications multi-agents. Cette démarche définit l'ensemble des rôles possibles, spécifie les interactions, et décrit les structures abstraites de groupe.

MadKit fournit une API permettant la construction d'agent en spécialisant une classe d'agent abstraite. Les agents sont lancés par le noyau de MadKit, qui propose notamment les services de gestion des groupes et de communication. L'échange des messages se fait à travers le rôle que l'agent est en train de jouer.

6.4 La plate-forme AgentBuilder

AgentBuilder est une suite intégrée d'outils permettant de construire des agents intelligents. Cette plate-forme est adaptée pour tous types de systèmes.

L'élaboration du comportement des agents se fait à partir du modèle BDI et du langage AGENT-0. KQML est utilisé comme langage de communication entre les agents. AgentBuilder est composé d'une interface graphique et d'un langage orienté agent permettant de définir des croyances, des engagements et des actions. Il permet également de définir des ontologies et des protocoles de communications inter-agents. Les agents sont décrits avec le langage Radl (Reticular Agent Definition Language), qui permet de définir les règles du comportement de l'agent. Les règles se déclenchent en fonction

de certaines conditions et sont associées à des actions. Les conditions portent sur les messages reçus par l'agent tandis que les actions correspondent à l'invocation de méthodes Java. Il est aussi possible de décrire des protocoles définissant les messages acceptés et émis par l'agent.

AgentBuilder propose l'utilisation de la méthode OMT durant la phase d'analyse. OMT permet de spécifier des objets du domaine et les opérations qu'ils peuvent effectuer. A partir de cette spécification une ontologie du domaine est créée. Des outils graphiques sont fournis pour effectuer ces tâches. Durant la phase de conception, les agents sont identifiés et des fonctionnalités leur sont assignées. Leurs rôles et leurs caractéristiques sont définis ainsi que les protocoles d'interaction auxquels ils participent. Durant la phase de développement, le comportement de l'agent est défini ainsi que ses croyances, intentions et capacités. Durant le déploiement, le code de l'agent est interprété par le *Run-Time Agent Engine.*

7 Analyse et conclusion

L'étude des différentes techniques d'ingénierie présentées tout au long de ce chapitre, fait ressortir plusieurs constats que nous allons exposer dans ce qui suit.

1. Différentes vues du système peuvent être spécifiées
   Pour simplifier la compréhension ainsi que le développement des systèmes multi-agents, il est préférable de considérer plusieurs vues. Lors de la première section, nous avons adopté la décomposition Voyelles qui considère la vue Agent, Environnement, Interaction et Organisation. Nous avons remarqué qu'au niveau des méthodologies, d'autres vues sont considérées. Par exemple, dans Ingenias, six vues (cas d'utilisation, agent, organisation, interaction, "tâches & buts" et environnement) ont été adoptées. Au niveau des autres méthodologies, les vues n'ont pas été explicitées. Nous pouvons cependant déduire qu'ADELFE met plus en valeur la vue Agent et Interaction. Gaia est plus orientée vers l'Organisation. PASSI définit une vue du domaine, une vue sur le systèmes multi-agent (incluant agents, services et interaction) et une vue sur l'implémentation du système. Enfin Tropos s'oriente plus vers les buts et les interactions. La difficulté dans le développement orienté agent est le fait que ces différentes vues sont fortement interreliées et la définition des concepts liés à chaque vue n'est pas chose aisée. Ainsi, mettre en valeur une vue plutôt qu'une autre peut dépendre entièrement du domaine d'application.

2. Définition de différents concepts

Les métamodèles adoptés dans chaque méthodologie, prennent en compte des concepts différents. Ceci implique que les modèles utilisés au niveau de l'analyse et de la conception sont différents sémantiquement et structurellement. Aucun problème particulier ne sera posé, si les méthodologies couvraient entièrement le processus de développement ainsi que les différentes vues du système. Mais tel n'est pas le cas. Comme nous avons pu le constater, les méthodologies couvrent généralement une partie du cycle de développement ou une vue partielle du système. Prenons l'exemple de la méthodologie PASSI. Celle-ci peut être considérée parmi les plus complète. Nous remarquerons cependant que les phases de validation et de maintenance ne sont pas couvertes ce qui constitue un réel problème dans le cas des systèmes multi-agents qui sont considérés comme des systèmes complexes et évolutifs. Un développement sûr et cohérent est important dans ce cas. Pour cette raison, des étapes de validation doivent être proposées à chaque phase de développement. De plus, une réelle prise en charge de la maintenance permet de contrôler l'exécution du système ainsi que son évolution.

3. Cycle de développement
   Nous remarquons aussi qu'un large fossé sépare les phases d'analyse/-conception des phases d'implémentation. En effet, rares sont les méthodologies qui prennent en compte la phase d'implémentation et la génération de code vers une plate-forme spécifique. Au niveau des méthodologies que nous avons exposées dans ce chapitre, l'outil PTK de PASSI permet de générer du code vers JADE alors que Tropos génère du code vers la plate-forme JACK. Cependant, les plates-formes ne couvrent pas tous les concepts pris en compte au niveau des modèles produits par la méthodologie. C'est donc au développeur que revient la charge de créer les entités correspondantes.

4. Notations orientées Agents
   Il est aussi important de soulever le fait que les notations orientées agent restent complètement déconnectées des méthodologies et plateformes d'implémentation (bien que AUML soit utilisé dans plusieurs méthodologies). Dans ce contexte, l'utilisation de ces notations devient difficile.

Suite aux constats listés précédemment, nous avons identifié plusieurs besoins liés à l'ingénierie des systèmes multi-agents. Dans le cadre de cette thèse, nous nous concentrons sur deux besoins particuliers :

1. Cadre de développement flexible
   Nous pouvons déclarer que l'application des systèmes multi-agents

souffre de l'absence de cadre de développement flexible et cohérent qui couvre les différentes vues d'un système multi-agent ainsi que les différentes phases de développement. Bien qu'il existe plusieurs propositions intéressantes aussi bien au niveau des méthodologies, des langages de spécification et des plates-formes d'implémentation, ces différentes propositions manquent de cohésion. Loin de préférer une technique sur une autre, nous soulignons que dans le contexte des systèmes multi-agents, des techniques flexibles sont nécessaires. Dans ce contexte, nous adhérons à l'approche adoptée par Ingenias qui consiste à diviser le problème en plusieurs aspects qui seront modélisés par des métamodèles, les modèles, les notations et les outils nécessaires seront alors générés à partir de ces métamodèles. L'environnement de développement est ainsi construit en fonction des besoins du développeur.

2. Vérification et validation des modèles
   Nous pouvons aussi constater que la validation des modèles n'est pratiquement pas abordée durant les différentes phases de développement. Cette situation peut être dûe à deux causes principales : les méthodologies sont pratiquement toutes basées sur des langages informels ou semi-formels ce qui rend la vérification et la validation des modèles difficiles. La deuxième raison évoque le manque d'outillage permettant la vérification et la validation des propriétés. Gaia est une méthodologie qui intègre l'utilisation de la logique temporelle lors de la conception, ce qui permet d'exprimer des propriétés de sûreté et de vivacité. Or, aucun outil de validation n'est fourni au sein de cette méthodologie qui permet de vérifier si la conception faite respecte les propriétés spécifiées. Par ailleurs, Tropos permet d'exprimer et vérifier les propriétés en utilisant Formal Tropos et les outils qui lui sont associés (tel que T-Tool). Cependant, cette vérification n'est valable que durant les phases d'ingénierie des besoins. Or, la vérification et validation des propriétés ne trouvent leur interêt que lorsque celles-ci sont conservées jusqu'à la phase d'implémentation voire de déploiement et de maintenance. Ainsi, il y a un grand besoin relatif aux notations permettant d'exprimer différentes propriétés ainsi qu'aux outils de vérification et validation. Il est en effet nécessaire d'avoir une conception sûr afin de produire des logiciels de qualité surtout dans le contexte des systèmes orientés agent.

Suivant les besoins que nous avons identifiés dans cette section, nous exposons dans le chapitre suivant, les éventuelles approches d'ingénierie qui peuvent nous aider à résoudre ces deux points.

Chapitre 4

Vers un développement flexible et sûr

1 Introduction

Nous avons vu lors du précédent chapitre que les principaux besoins liés à l'ingénierie des systèmes multi-agents consistent dans la spécification d'un cadre de développement, dont les caractéristiques sont :

- flexible pour couvrir les différentes vues d'un système multi-agent,
- capable de gérer la cohérence des différentes vues et la préservation de leurs propriétés lors de leur intégration,

Afin de répondre à ces besoins, et proposer au final un cadre de développement sûr, cohérent et formel, nous avons analysé les différentes avancées faites au niveau des techniques d'ingénierie actuelles (c.f. chapitre 2). Les problématiques soulevées dans le cadre des systèmes multi-agents rencontrent celles qui sont soulevées lors de l'application d'autres paradigmes de développement récents tels que les composants logiciels et les services Web. Les avancées effectuées au niveau du génie logiciel pour résoudre ces problématiques sont diverses (cf. chapitre 2). Notre choix s'est porté sur deux principales approches :

- L'ingénierie dirigée par les modèles : qui nous permet d'appliquer les techniques de métamodélisation afin d'obtenir une flexibilité dans la définition des composants liés aux systèmes. L'approche dirigée par les modèles conçoit l'intégralité du cycle de vie comme un processus de production, de raffinement itératif et d'intégration de modèles ; ce qui rejoint notre objectif.
- Le développement orienté architecture : qui se penche sur les techniques de description des modèles architecturaux ainsi que leur intégration.

En s'abstrayant des détails de l'implémentation, l'architecture offre une vue globale du système qui est lisible et qui expose les propriétés les plus cruciales ; ce qui permet un développement efficace. De plus, comme nous allons le décrire lors de la section 3 de ce chapitre, le développement orienté architecture offre des moyens de formalisation qui facilitent l'expression et la vérification des propriétés.

Dans ce chapitre, nous commençons par une présentation détaillée de la problématique, déclinée en plusieurs objectifs. Par la suite, nous présentons plus profondément chacune de ces approches avec leurs apports pour le developpement des systèmes multi-agents. Nous concluons ce chapitre par une mise en relation de ces deux approches, à partir de laquelle nous construisons notre proposition (cf. chapitre 5).

2 Problématique et objectifs de la thèse

Nous avons vu lors du précédent chapitre que les principaux besoins liés à l'ingénierie des systèmes multi-agents consistent à la spécification d'un cadre de développement flexible et cohérent qui couvre les différentes vues d'un système multi-agent ainsi que les différentes phases de développement. En effet, nous avons passé en revue dans le chapitre 3 les concepts orientés agent. Ceux-ci sont nombreux et leurs sémantiques peuvent avoir plusieurs déclinaisons. Par exemple, un agent peut être cognitif, réactif ou hybride. Il peut avoir une ou plusieurs interfaces à travers lesquelles il peut percevoir son environnement ou communiquer avec d'autres agents. Il peut poursuivre des buts ou offrir des services. Il peut être situé dans un environnement et peut appartenir à une organisation sociale etc. La sémantique d'agent qui va être utilisée dépend largement du domaine d'application. Ainsi, la spécification d'un système particulier nécessite une certaine flexibilité où le développeur peut définir la sémantique qui convient le plus à son domaine d'application.

D'autre part, afin de garantir un développement de qualité, le cadre de développement devrait permettre l'expression et la validation des propriétés au cours du cycle de développement. Les systèmes multi-agents sont en effet des système complexes nécessitant un grand nombre de composants interconnectés. Dans un pareil contexte, il faut s'assurer que les propriétés fonctionnelles et non fonctionnelles restent conservées tout au long du cycle de développement.

Dans cette thèse, nous proposons de construire un cadre de développement qui permettra aux développeurs de :

- s'adapter aux différents types de systèmes multi-agents,

- définir les différentes vues nécessaires au développement de leur application ainsi que leurs concepts respectifs,
- définir les mécanismes de raffinement et de transformation qui permettent le passage de l'analyse/conception vers l'implémentation et le déploiement,
- définir les propriétés ainsi que les moyens de vérification et de validation de ces propriétés qui devraient être conservées tout au long du cycle de développement.

La flexibilité d'un cadre de développment peut être obtenue grâce à l'utilisation des métamodèles. En effet, tout domaine d'application (télécom, commerce électronique, etc.) possède des concepts spécifiques. Ces concepts ainsi que leurs relations sont décrits au niveau du métamodèle. Celui-ci est utilisé pour préciser ce qu'est un composant logiciel pour un domaine particulier. Etant donné qu'il n'y a pas de définition unique des composants multi-agents, ce point nous semble important. De plus, nous avons noté qu'au niveau des systèmes multi-agents, une multitude de métamodèles peuvent être exprimés pour des utilisations diverses. Il est important, dans ce contexte, d'accorder la liberté au développeur de définir le métamodèle qui convient le plus à ses besoins.

Par ailleurs, nous avons vu que la complexité des systèmes multi-agents nécessite leur décomposition en plusieurs vues. Chaque vue peut être définie dans un métamodèle propre. Par la suite, les vues vont être intégrées pour obtenir le métamodèle complet d'un système particulier. De par cette manière de définir un métamodèle, il est possible de disposer d'un support complet, adapté à une application particulière et respectant la séparation des préoccupations. Notons que la séparation des préoccupations permettra de capturer les propriétés de chaque vue de manière plus aisée.

> Le premier objectif de cette thèse consiste à offrir les moyens au développeur de définir les métamodèles représentant les vues de leur système. Au niveau de ces métamodèles, nous pensons que les contraintes structurelles et comportementales relatives aux entités contenues dans chaque vue doivent être explicitement définies.

Dans ce contexte, INGENIAS adopte la même approche basée sur la métamodélisation, ce qui permet de définir les modèles nécessaires à chaque vue du système multi-agents. Ces modèles vont être instanciés au niveau des phases d'analyse/conception afin de décrire l'application qui est en cours de développement. Des *templates* sont générés à partir de ces vues qui sont intégrés et compilés afin de vérifier leur validité. Cependant, nous avons soulevé que les propriétés structurelles et comportementales ne sont pas vérifiées

dans les phases d'analyse et de conception. Ceci est principalement dû au manque de formalisation. En effet, INGENIAS utilise des notations semi-formelles, ce qui ne facilite pas la vérification de telles propriétés.

Le deuxième objectif de cette thèse consiste à offrir les moyens aux développeurs d'exprimer les propriétés fonctionnelles et non fonctionnelles de leur système. Ces propriétés doivent être préservées tout au long du cycle de développement. Pour cette raison, les moyens d'exprimer ces propriétés doivent alors être accompagnés d'outils de vérification et de validation.

Pour atteindre cet objectif, nous utiliserons au cours de cette thèse des langages formels qui garantissent l'expression des propriétés, leur vérification et leur validation durant les différentes phases de développement.

Nous avons aussi soulevé le fait que les phases d'analyse/conception soient complètement dissociées des phases d'implémentation/déploiement. Pour résoudre cette problématique, il est nécessaire d'adopter un cycle de développement permettant de guider les développeurs durant les différentes phases en explicitant les règles de passage d'une phase à une autre. Ceci peut correspondre par exemple, à la définition de règles permettant de générer les modèles à partir du métamodèle ou spécifier des règles permettant le passage des modèles vers le code, etc. Ces règles sont en général appelées règles de transformation et règles de raffinement (nous allons approfondir ces deux notions au cours de ce chapitre).

Le troisième objectif de cette thèse consiste à décrire les règles de raffinement et de transformation permettant le passage d'une phase de développement vers une autre.

3 Le développement dirigé par les modèles

Le développement dirigé par les modèles a été brièvement introduit dans la section 5.3 du chapitre 2. Cette approche vise à fournir un cadre conceptuel, technologique et méthodologique dans lequel les modèles sont au centre des activités de développement. Dans cette section, nous allons rappeler brièvement les principes de cette approche avant d'introduire les principes de son application et consiste en l'étude :

- des niveaux d'abstraction considérés,
- des relations entre les niveaux,
- des transformations de modèles,
- des processus de développement à mettre en œuvre

3.1 Principes de l'approche

IDM spécifie l'intégralité du cycle de vie du logiciel comme un processus de production, de raffinement itératif et d'intégration de modèles. En ce sens, IDM fait évoluer l'usage des modèles. En effet, une traçabilité entre les éléments des différents modèles est gardée et ceci quel que soit leur niveau d'abstraction. L'utilisation des modèles permet de capitaliser les connaissances qu'elles soient des connaissances du domaine métier ou du domaine technique. Par ailleurs, la traçabilité (entre les différentes vues du système) ainsi que la transformation de modèles (entre les différentes phases de développement) permet de capitaliser le savoir-faire.

Les travaux menés sur l'IDM font suite à la définition de l'approche Model Driven Architecture (MDA) par l'OMG (Object Management Group) [OMG03]. MDA a recours aux différents standards de l'OMG afin de décrire les démarches basées sur l'ingénierie des modèles. IDM entend ne pas se limiter au jeu de standards spécifiques à un organisme particulier, et désire proposer autour de la notion de modèle et de métamodèle une vision unificatrice. Dans ce contexte, MDA est considéré comme un exemple particulier d'ingénierie dirigée par les modèles. Néanmoins, la plupart des travaux sur l'IDM font référence à MDA. Cet état de fait provient du fait que l'OMG catalyse, centralise, et synthétise bon nombre de travaux sur l'IDM. Pour cette raison, nous détaillerons dans ce qui suit plusieurs travaux proposés dans le cadre de MDA.

3.2 Les quatre niveaux de OMG

Afin de mieux comprendre l'approche de l'IDM, il convient de préciser plus en détail la nature des modèles utilisés tout au long du cycle de vie, les divers usages de ces modèles, ainsi que les possibles intentions du concepteur au moment de leur construction. MDA a proposé une architecture à quatre niveaux qui structure les différents modèles pouvant être produits lors de l'application de l'approche de l'IDM. Cette architecture fait maintenant l'objet d'un consensus [Sei03], [BBB+05], on retrouve ainsi des références à cette architecture dans de nombreux travaux désirant se situer par rapport à l'IDM [IEV05]. Cette architecture (cf. figure 4.1) comporte quatre niveaux d'abstraction que nous allons détailler dans ce qui suit.

Le niveau M0

Le niveau M0, représente les objets du monde du réel. Il représente, par exemple, un compte bancaire avec son numéro et son solde actuel (cf. figure 4.1).

Le niveau M1

C'est au niveau M1 que les modèles sont édités. Ces modèles sont conformes aux métamodèles définis au niveau M2. Ainsi, MDA considère que si l'on veut décrire des informations appartenant au niveau M0, il faut d'abord construire un modèle appartenant au niveau M1. De ce fait, un modèle UML (comme le diagramme de classes ou le diagramme d'état/transition) est considéré comme appartenant au niveau M1. Il représenterait des objets manipulés dans le monde réel (décrits au niveau M0).

Le niveau M2

Le niveau M2, est le lieu de définition des métamodèles. Un métamodèle peut être considéré comme un langage spécialisé pour un aspect du système. Il peut aussi décrire les aspects spécifiques aux différents domaines, chaque aspect étant pris en compte dans un métamodèle spécifique. Les métamodèles contenus dans le niveau M2 sont tous des instances du niveau M3 (notons qu'au niveau M3, il ne peut y avoir qu'un seul méta-métamodèle). Dans le cadre de MDA, c'est le métamodèle d'UML [OMG04] qui est le plus utilisé, celui-ci définit la structure interne des modèles UML.

Le niveau M3

Le niveau supérieur, M3 correspond au méta-métamodèle. Il définit les notions de base permettant l'expression des métamodèles (niveau M2), et des modèles (M1). Pour éviter la multiplication des niveaux d'abstraction, le niveau M3 est réflexif, c'est-à-dire qu'il se définit par lui-même. Le plus souvent, c'est le méta-métamodèle MOF (Meta Object Facility) qui est utilisé. Celui-ci est standardisé par l'OMG [OMG06]. Cependant d'autres méta-métamodèles ont été proposés tels que eCore définit dans le cadre du "Eclipse Modeling Framework" (EMF) [BSE+03] et OWL [DGS06].

Des plates-formes de modélisation génériques implémentant MOF permettent la production de métamodèles spécifiques à des domaines et de produire ensuite des modeleurs spécifiques à ce domaine. Par exemple, la plate-forme de métamodélisation GME [LBM+01] est compatible avec eCore d'EMF [BBB+05].

Afin d'avoir une idée plus précise de l'architecture à quatre niveaux d'OMG, nous illustrons par la figure 4.1, les modèles pouvant appartenir à chaque couche.

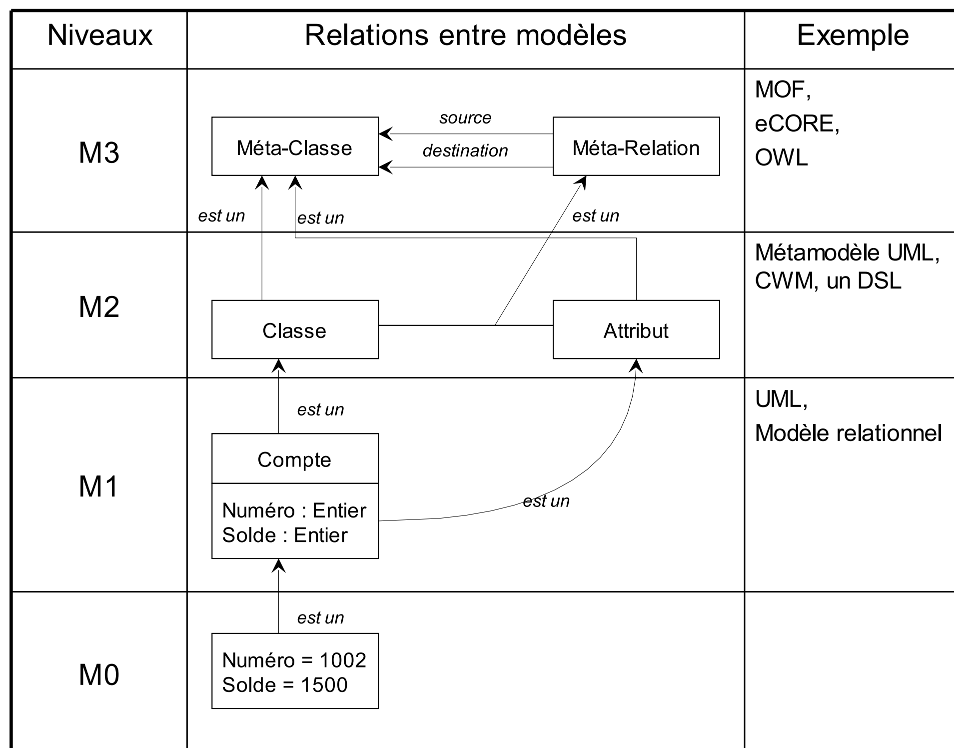


FIG. 4.1: Les quatre niveaux de MDA

3.3 Les relations entre les modèles

Les relations entre les modèles appartenant aux quatre niveaux ont fait l'objet de plusieurs débats [Fav04]. Les flèches que nous avons annotées *est un* sur la figure 4.1 méritent un approfondissement de leur sémantique. Nous pouvons aller jusqu'à déclarer que l'approche de l'IDM repose sur la sémantique adoptée au niveau de ces flèches. Actuellement, trois relations fondamentales sont identifiées : "représentation de", "être conforme à" et "être instance de".

la relation *"représentation de"*

Cette relation (notée μ) lie le modèle au système qu'il représente. Cette relation traduit la sémantique qui existe entre un système et un modèle. En effet, il est communément admis qu'un modèle est la simplification subjective d'un système. Dans ce contexte, le modèle sert à obtenir des réponses par rapport au système qu'il représente [BG01], [Béz04]. Par exemple, une carte géographique peut jouer le rôle de modèle, alors que la région étudiée jouera celui de système modélisé. De même qu'une carte elle-même peut jouer le rôle du système modélisé comme le montre la figure 4.2. Dans cette figure, la carte est elle même représentée par un schéma XML.

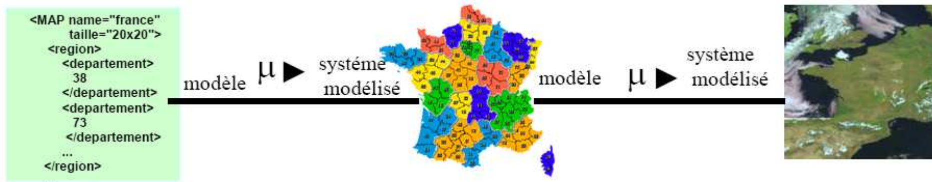


Fig. 4.2: La relation μ [BBB+05]

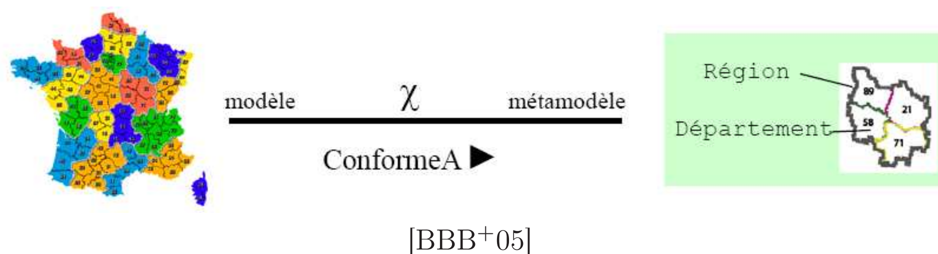
Il faut spécifier par ailleurs, que dans la figure précédente, le fichier modélisant la carte ne peut être considéré comme un métamodèle. En effet, il existe une nuance entre la définition d'un modèle et la définition d'un métamodèle. Alors qu'un modèle est une description d'un système ou d'une partie du système [KWB03], un métamodèle est un modèle qui définit ***le langage*** qui exprime le modèle. Ainsi, la relation *"représentation de"* peut décrire le lien entre la couche M0 et M1, mais ne décrit pas le lien entre les couches M1, M2 et M3.

la relation *"être conforme à"*

La relation *"être conforme à"* (notée χ) est plus appropriée pour la description de liens entre les couches M1, M2 et M3. Elle spécifie en effet la

relation existante entre les modèles et leur métamodèle. Manipuler informatiquement et opérationnaliser des modèles nécessitent qu'ils soient exprimés dans un langage clairement défini. L'architecture à quatre niveaux permet d'avoir un cadre permettant cette définition : un modèle appartenant au niveau M1 est conforme à un métamodèle défini au niveau M2, qui est lui-même conforme à un méta-métamodèle défini au niveau M3.

La relation χ est considérée comme le point clé de l'approche de l'IDM. En effet, celle-ci permet d'assurer d'un point de vue théorique mais aussi opérationnel qu'un modèle est correctement construit. A partir de ce moment, il devient envisageable de lui appliquer des transformations automatisées. La figure 4.3 démontre la relation entre une carte géographique et son métamodèle qui est constitué d'une notation graphique représentant les régions et les départements.



[BBB+05]

FIG. 4.3: La relation χ

la relation *"InstanceDe"*

MDA est principalement fondée sur l'approche orientée objet. Ceci a généré de nombreuses confusions par rapport aux relations existantes entre les différentes couches de modèles. La relation *"InstanceDE"* est un exemple qui incarne cette confusion. Au niveau du standard MDA, il a été largement relayé par l'OMG qu'un modèle serait une "instance d'un" métamodèle. Or, ceci fait référence à l'hypothèse implicite que modèles et métamodèles sont exprimés dans une technologie orientée-objet. Il n'existe aucune contrainte de cette sorte dans l'IDM. L'approche IDM se veut indépendante de toute contrainte technologique, pour cette raison, la relation *InstanceDe* est considérée comme une incarnation particulière de la relation *ConformeA* dans la technologie orientée objet [BBB+05].

3.4 La transformation de modèles

Les transformations de modèles [1] sont au coeur de l'approche de l'ingénierie dirigée par les modèles. Elles représentent l'un des grands défis à relever d'un point de vue technique pour envisager une large diffusion de l'ingénierie dirigée par les modèles. La transformation de modèles intéresse aussi bien les chercheurs que les industriels. En effet, leur mise en œuvre augmentera considérablement la productivité et la qualité des logiciels produits. Cependant, à l'heure actuelle, il n'existe pas encore de consensus sur la définition et la mise en œuvre d'une transformation. Une transformation est une opération qui sert à produire un modèle (appelé modèle cible) à partir d'un autre modèle (appelé modèle source). Plusieurs sortes de modèles cibles et sources peuvent exister. Selon la nature des modèles, plusieurs types de transformation sont répertoriés [MCG05] :

- Transformation du modèle vers le code : dans l'ingénierie dirigée par les modèles, le code source est considéré comme un modèle ayant un bas niveau d'abstraction (modèle concret). Une telle transformation consiste à générer du code à partir d'une spécification.
- Transformation du code vers le modèle : dans une transformation le code source peut aussi jouer le rôle de modèle source. Il s'agit alors de réingénierie des systèmes.
- Transformation *endogènes* et *exogènes* : la transformation endogène implique des modèles sources et cibles conformes au même métamodèle ; alors qu'une transformation exogène implique des modèles issus de métamodèles différents. Dans ce dernier cas, on parle de *translation.* La génération de code, la réingénierie ainsi que les migrations (cf. figure 4.4) sont des exemples typiques de translation (ou transformation exogène). Les transformations endogènes peuvent quant à elles servir à l'optimisation (pour améliorer la qualité du modèle), la refactorisation (pour améliorer la lisibilité du modèle), la simplification ou normalisation (pour limiter la complexité syntaxique relative à un modèle, il s'agit par exemple de rajouter du sucre syntaxique).
- Transformation verticale et transformation horizontale : une transformation horizontale est une transformation où le modèle cible et le modèle source appartiennent au même niveau d'abstraction (un exemple de cette transformation est la refactorisation). La transformation verticale s'applique sur des modèles appartenant à des niveaux d'abstraction différents (exemple génération de code).

MDA définit le langage QVT (Query/View/Transformation) [OMG05] comme standard pour la spécification des règles de transformation. Ce lan-

[1] Dans cette section, le terme modèle est utilisé dans le sens large, et non pas en faisant référence à la couche M1.

gage est formé de trois parties :

- la définition de requêtes qui permet une navigation dans les modèles,
- la spécification de vues qui permet de focaliser sur une partie des modèles,
- la spécification de règles de transformations de modèles dont l'application permet de générer un modèle source d'un modèle cible.

Différents langages de transformation implémentent QVT (exemples SmartQVT [SMA06], Medini QVT [HSE06], OptimalJ [JSHL07]) ou s'en rapprochent (principalement ATL [JK06]). Ces langages se comparent selon différents critères, définis par Czarnecki *et al.* [CH03] :

- les règles de transformation : avec ce critère, nous pouvons regarder la structure de construction des règles de tranformation. Les règles peuvent être paramétrées, décrites d'une manière déclarative, impérative ou hybride (à la fois déclarative et impérative).
- la portée d'application des règles : les règles de transformation peuvent avoir comme portée la totalité du modèle ou une partie du modèle.
- les relations entre modèle source et modèle cible : ce critère compare la nature de relations qui peut exister entre le modèle cible et le modèle source. Celles-ci peuvent être de deux types ; le modèle cible est un nouveau modèle différent, créé à partir du modèle source ou le modèle cible est le même que le modèle source sur lequel des modifications ont été faites.
- la stratégie d'application des règles : l'application des règles peut être déterministe (c'est-à-dire que l'exécution des règles est faite de manière préétablie ou standard comme par exemple le parcours classique d'un arbre) ; ou bien indéterministe (c'est-à-dire pouvant être effectuées dans un mode interactif voire concurrent).
- l'ordonnancement des règles : ce critère vérifie si l'ordre d'application des règles peut être explicité. Dans le cas contraire c'est l'outil interprétant les règles qui détermine l'ordre.
- L'organisation des règles : ce critère vérifie s'il est possible de rassembler et d'organiser les règles dans des modules. Il vérifie également si elles peuvent être réutilisées par d'autres règles en utilisant le mécanisme d'héritage ou de composition.
- La traçabilité : durant la transformation de modèles, les liens de traçabilité entre modèles peuvent être conservés.
- La direction : les règles de transformation peuvent être appliquées de manière unidirectionnelle ou bidirectionnelle (dans ce cas la traçabilité entre les modèles doit être conservée).

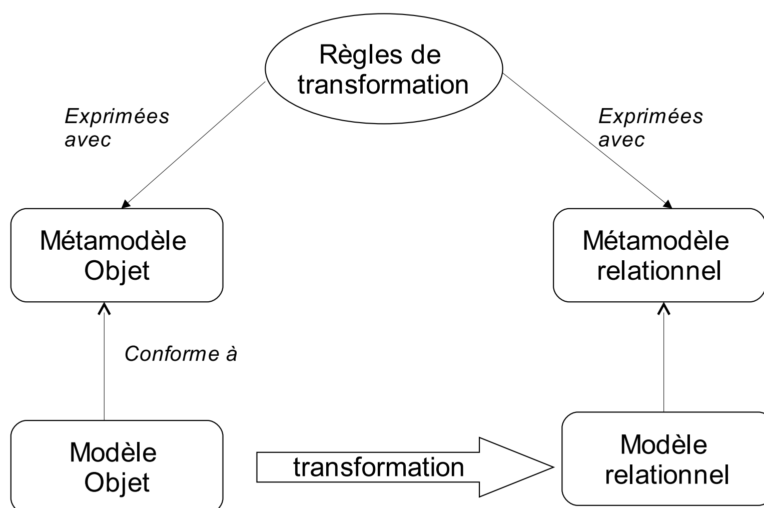


FIG. 4.4: Migration de modèle objet à un modèle relationnel

3.5 Processus de développement orienté modèles

Le modèle de cycle de développement ne fait pas encore l'unanimité dans l'ingénierie dirigée par les modèles. Cependant, nous avons pu distinguer deux sortes de modèles de cycle de vie. Le premier a été proposé dans le cadre de l'approche MDA. Le deuxième est considéré plus généraliste et est basé sur une adaptation du cycle de vie en Y. Nous allons présenter dans ce qui suit, chacun de ces modèles.

Le cycle de développement de MDA

Le cycle de développement de MDA peut être comparable à un cycle de développement classique tel que le processus en cascade. On y trouve en effet les mêmes phases de développement à savoir l'expression des besoins, l'analyse, la conception, l'implémentation, la validation et le déploiement. La différence réside cependant dans la nature des artefacts produits dans chacune des phases. La figure 4.5 décrit ce processus de développement.

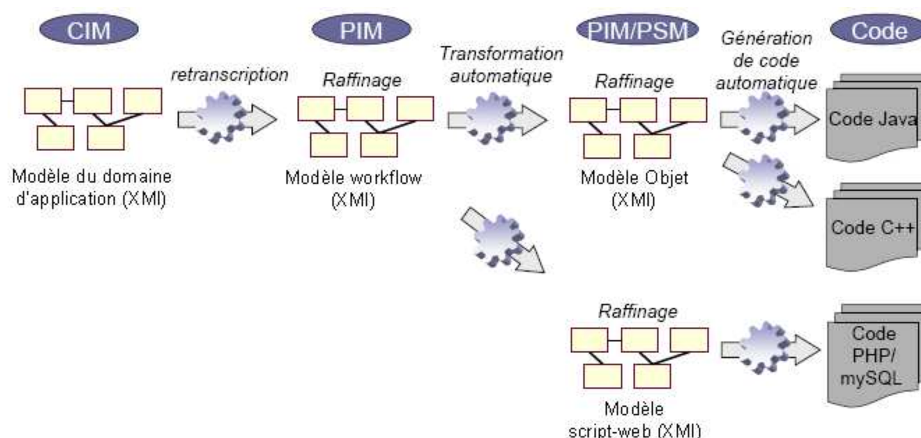


Fig. 4.5: Le cycle de vie de MDA

Le premier modèle produit appelé Computer Independent Model (CIM) correspond à la description en langage naturel du domaine d'application. Dans ce modèle aucune considération informatique n'apparaît. Par exemple, dans le domaine Bancaire, il pourrait s'agir d'un texte où il serait indiqué que certaines compensations peuvent être traitées le soir. Un modèle CIM sert de base à la définition du modèle PIM (Platform Independent Model) où le métamodèle sous-jacent est cette fois-ci de type informatique mais non rattaché à une technologie particulière (ex : un modèle objet). Le passage d'un modèle CIM à un modèle PIM est normalement manuel et nécessite plusieurs discussions entre experts du domaine et informaticiens bien que certains travaux préconisent une projection automatique [ZMZY05]. Le modèle PIM est, dans une troisième étape raffiné afin d'indiquer des aspects propres au paradigme de développement qui va être utilisé (ex : l'objet, le procédural, etc.). La quatrième étape consiste à produire, en appliquant une transformation automatique, un modèle PSM (Platform Specific Model) c'est-à-dire un modèle spécifique à une plate-forme. Un modèle EJB (modèle UML utilisant le profil EJB) est un exemple de modèle de type PSM. Après raffinement de ce type de modèles, celui-ci sert à générer automatiquement une partie du code final qui va être par la suite raffiné et testé (manuellement) par le développeur.

Ce processus de développement a cependant fait l'objet de plusieurs critiques. Bien qu'il semble simple de premier abord, plusieurs travaux actuels portant sur les compositions de modèles ont montré la difficulté de relier entre eux les différents modèles, représentant les différents aspects d'une application [EVI05]. D'autres difficultés ont été relevées quant à l'utilisation de UML et de MOF. Ceux-ci sont considérés comme très généralistes et difficilement manipulables par des non spécialistes. Microsoft par exemple a fait le choix de privilégier des langages de domaines (Domain Specific Languages ou

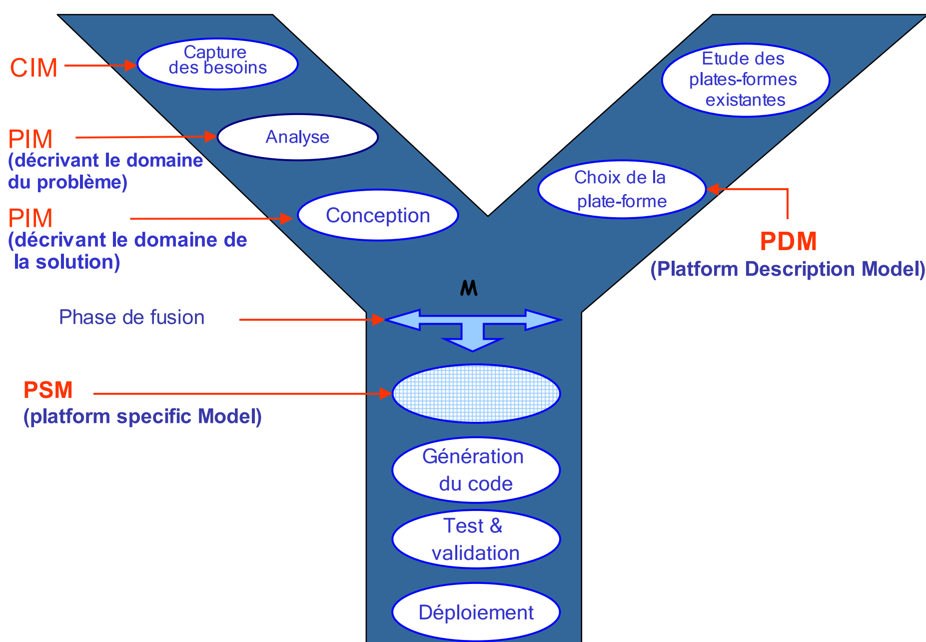


FIG. 4.6: Le cycle de vie en Y

DSL) de petite taille, facilement manipulables et transformables [JSCK04].

Processus de développement en Y

Plusieurs travaux concernant IDM font référence au cycle de développement en Y [BDD+02], [Kad05]. Initialement, ce processus de développement avait pour objectif une maîtrise des coûts et délais grâce à une parallélisation des tâches d'analyse et de conception [Lar94] mais certains travaux l'ont adapté dans le cadre d'IDM [BBF06]. Dans ce contexte, il s'agit de mettre en parallèle l'élaboration du PIM et la description de la plate-forme. Une jonction entre les deux branches va constituer le PSM. La figure 4.6 illustre ce processus.

Selon ce processus, les phases d'analyse/conception sont menées en parallèle avec l'étude des différentes plates-formes existantes. Durant l'analyse et la conception, les modèles CIM et PIM sont produits comme le préconise le processus fourni par MDA. Notons qu'un premier PIM est élaboré décrivant le domaine du problème. Celui-ci est ensuite raffiné pour produire un PIM décrivant le domaine de la solution. Ce raffinement doit cependant conserver l'indépendance des modèles par rapport aux détails d'implémentation. Il s'agirait de spécifier l'échange de flux de données entre les différentes entités du système. Parallèlement, les aspects techniques de chaque plate-forme

PIM entity	Mapping Decision	PDM entity
BadSyntaxException	*InheritsFrom*	Exception
Button	*IsReplacedBy*	Button
DataGrid	*IsReplacedBy*	DataGrid
Document	*uses*	String
Doublet_array	*IsReplacedBy*	ArrayList
Evaluation_Stack	*IsReplacedBy*	Stack
LibraryDatabase	*uses*	OracleConnection
LibraryDatabase	*uses*	OracleCommand
LibraryDatabase	*uses*	OracleDataReader
Menu	*IsReplacedBy*	Menu
MenuItem	*IsReplacedBy*	MenuItem

FIG. 4.7: Un exemple de table de mapping PDM/PIM

sont étudiés et le choix de la plate-forme d'implémentation est effectué. Les caractéristiques techniques de celle-ci sont modélisés dans le PDM (Platform Description Model). Celui-ci va décrire par exemple, les types supportés par la plate-forme (exemple : une chaîne de caractère en JAVA correspond à un objet qui instancie la classe String). Un mapping est ensuite effectué entre les entités décrites dans le PIM et celles qui leur correspond dans le PDM. La figure 4.7 illustre un exemple de table de mapping entre les entités du PIM et celles du PDM.

C'est à la suite du mapping PDM/PIM que le PSM est généré. Celui-ci est donc le résultat de la fusion du PDM et du PSM. Une fois le PSM effectué, le code est généré. Ensuite il est raffiné, testé et validé par le développeur avant que le déploiement n'ait lieu.

3.6 Apports potentiels de l'approche orientée modèles dans l'ingénierie des systèmes multi-agents

Dans cette section, nous nous replaçons dans le contexte du développement orienté agent. L'ingénierie dirigée par les modèles peut être utile à divers niveaux dans ce contexte. Nous résumons dans le tableau 3.6, tous les avantages que nous pouvons tirer de l'application d'IDM dans l'ingénierie des systèmes multi-agents.

Nous remarquons cependant que l'expression et la vérification des pro-

Problématiques des systèmes multi-agents	**Théories IDM**
Plusieurs méthodologies, langages de modélisation, plates-formes orientées agents sont proposés, chacun adoptant une sémantique particulière des concepts.	**L'architecture à quatre niveaux** de IDM permet de structurer et d'unifier le cadre de développement en définissant les métamodèles et les méta-métamodèles utilisés. La transformation de modèles permet de combiner les modèles produits selon des métamodèles différents.
La sémantique des concepts utilisés est très ambiguë.	**La couche métamodèle et méta-métamodèle** permettent de définir la sémantique des concepts utilisés.
Application de l'approche à différents domaines.	**L'architecture à quatre niveaux de IDM permet une grande flexibilité** : la métamodélisation permet de définir les concepts de manière indépendante. Ainsi, les concepts orientés agent seront définis indépendemment des concepts du domaine. L'unification des deux métamodèles se fera par la suite lors de la définition du PIM.
Connecter les phases d'analyse et de conception à la phase d'implémentation.	**Le PIM** est un modèle élaboré pendant les phases d'analyse et de conception tandis que **le PSM** est élaboré pendant la phase d'implémentation. Des règles de transformation doivent être réalisées pour passer du PIM vers le PSM ce qui permet de connecter ces différentes phases.
Interopérabilité entre les plates-formes.	Théoriquement, la **transformation de modèles** facilitera l'échange de modèles entre plusieurs plates-formes.

TAB. 4.1: Les apports relatifs à l'application de l'approche IDM dans le contexte multi-agents

priétés sur les modèles, ne sont pas réellement traitées au niveau de IDM. En effet, IDM s'oriente vers la validation des modèles par rapport au métamodèle. Ce genre de validation permet de déterminer si le modèle est correctement construit mais ne peut raisonner sur les propriétés comportementales telles que la sûreté, la vivacité ou le *deadlock*. Les deux modèles de cycle de vie présentés précédemment, ne prennent pas en compte l'étape de vérification au niveau des phases d'analyse et de conception. De plus, les formalismes utilisés sont essentiellement basés sur MOF et UML dont la syntaxe et la sémantique ne sont pas formalisées de manière mathématique. Il en découle que des propriétés structurelles et comportementales sont difficilement exprimables et vérifiables. Ainsi, tout en suivant les instructions d'IDM, il nous paraît judicieux de combiner cette approche avec l'approche centrée architecture dans laquelle les phases d'analyse et de conception ont été traitées avec plus de rigueur. Nous présentons cette approche dans ce qui suit.

4 Le développement orienté architecture

Le développement centré architecture a aussi été abordé dans la section 5.2 du chapitre 2. Dans cette section nous allons approfondir cette approche. Nous commençons d'abord par rappeler les principes de bases de cette approche, avant d'expliquer sa mise en œuvre et détailler le processus de développement utilisé. A la fin de cette section, nous résumons les avantages que nous pouvons tirer de l'utilisation de cette approche dans les systèmes multi-agents.

4.1 Définition d'une architecture logicielle

Le développement orienté architecture est une discipline récente focalisant sur la structure, le comportement et les propriétés globales d'un logiciel. Elle s'adresse plus particulièrement à la conception de systèmes logiciels de grandes tailles ou de familles de systèmes logiciels. Le but du développement orienté architecture est de permettre aux concepteurs d'effectuer des analyses préliminaires sur ces systèmes. Ces analyses visent à découvrir et à résoudre les problèmes de conception dès les premières étapes du développement.

Plusieurs définitions ont été attribuées aux architectures logicielles [PW92], [Boa95] , [BCK99]. De manière générale, une architecture logicielle est définie par un ensemble de composants (ex : filtres, objets, bases de données, serveurs, etc.) ainsi que par la description des interactions entre ceux-ci (ex : appels de procédures, envois de messages, émissions d'évènements, etc.) [GS93]. Mais l'architecture logicielle ne se limite pas à ces descriptions. La description architecturale d'un système informatique permet en outre de spécifier :

- sa structure : éléments de traitement et leurs interactions (pas de détail d'implémentation),
- son comportement : fonctionnalités et protocoles de communication, dynamisme,
- ses propriétés globales : sécurité, vivacité, etc.
- les règles régissant sa conception : il pourrait s'agir de règles de raffinement, de transformation (vers le code) ou de vérification.

L'évaluation d'une architecture mène à une meilleure compréhension des besoins, des stratégies d'implémentation, et des risques potentiels. De plus, l'utilisation d'une conception orientée architecture comporte des avantages pour toutes les personnes intervenant dans le cycle de vie d'un projet logiciel :

- pour le client : l'architecture permet une estimation du budget, de la faisabilité et du temps de développement.
- pour le chef de projet : elle fournit un support pour la traçabilité des besoins, l'évaluation du système et le suivi des progrès,
- pour le développeur : l'architecture offre une ligne de conduite et les spécifications nécessaires pour l'implémentation de nouveaux composants. De plus, elle offre les garanties pour l'interopérabilité avec les systèmes existants,
- pour le mainteneur : l'architecture est un guide pour faire évoluer le système.

Pour avoir une idée plus claire sur l'architecture logicielle, nous allons, dans la suite de cette section exposer les différents apports de la spécification des architectures logicielles.

4.2 Mettre en œuvre l'approche orientée architecture

La mise en œuvre de l'approche orientée architecture nécessite divers techniques. Malencontreusement, dans l'industrie encore aujourd'hui, les descriptions architecturales sont encore basées sur des diagrammes informels (sous forme de lignes et de boîtes) et sont souvent ambiguës, incomplètes, inconsistantes, et non-analysables. En effet, de simples diagrammes comportant des annotations ne sont pas suffisamment expressifs pour spécifier une architecture. Pour être exploitable, une architecture doit clairement définir :

- quels traitements les "boîtes" et leurs annotations représentent,
- quelles relations de contrôle/flux de données sont indiquées par les "lignes" ou "flèches",
- comment le comportement global du système est déterminé par celui de ses composants,

– quelles sont les propriétés structurelles et comportementales qui doivent être conservées tout au long du développement.

Afin de définir avec rigueur l'architecture des systèmes logiciels, les langages de description d'architectures - ADLs (Architecture Description Languages) - émergent comme le support de notation pour les modèles d'architectures [MT01], [WH05]. Ils utilisent des notations textuelles et graphiques ayant une sémantique bien définie. Ces langages sont souvent accompagnés d'outils de création, de modification, de navigation, de simulation et d'analyse. Certains ADLs sont basés sur des notations formelles. Ceci comporte de multiples bénéfices : le fait de décrire formellement l'architecture rend possible l'utilisation (parfois automatique) des mécanismes d'analyse, de raffinement, et de génération du code correspondant.

L'analyse de l'architecture permet l'évaluation des propriétés d'un système au niveau architectural ; ce qui permet de prévenir les erreurs et de diminuer les coûts dûs à celles-ci. Les types d'analyses applicables dépendent de l'ADL utilisé et de son modèle sémantique. En effet, certains ADL se contentent de décrire la structure du système, alors que d'autres incluent aussi des caractéristiques comportementales. Ainsi, selon la nature de l'ADL, il est possible d'exprimer des contraintes structurelles et comportemantales. Ces contraintes constituent des propriétés fonctionnelles que le système final doit respecter. Dans certains ADL, ils est aussi possible de pouvoir exprimer des propriétés non fonctionnelles telles que la sécurité, la qualité etc.

Lors du raffinement, les propriétés de l'architecture abstraite doivent être conservées par l'architecture concrète. Dans ce contexte, certains ADLs permettent d'exprimer explicitement les règles de raffinement, en spécifiant l'opération qui doit être effectuée lors du pas de raffinement ainsi que les propriétés qui doivent être conservées. L'usage d'outils adéquats est indispensable dans ce cas.

Le but ultime de la conception logicielle est de produire un système exécutable. Un modèle architectural "élégant" et efficace a peu de valeur s'il n'est pas convertible en application exécutable. Cela peut se faire manuellement ; cas dans lequel de nombreux problèmes de consistance et de traçabilité entre l'architecture et son implémentation peuvent apparaître. Il est préférable d'utiliser des outils de génération de code qui devraient être fournis par chaque ADL.

4.3 Processus de conception orienté architecture

Au niveau du cycle de développement, la spécification des architectures logicielles se situe essentiellement dans les phases de conception et d'implémentation. Le processus de développement centré architecture inclut trois

sortes d'acteurs : l'architecte d'application, le développeur et l'analyste qui est responsable de vérification et la validation des architectures (cf. figure 4.8).

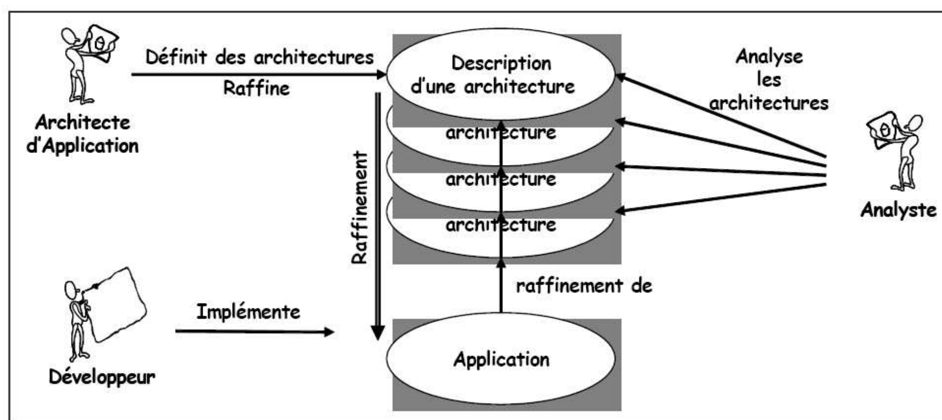


FIG. 4.8: Le processus centré architecture

L'architecte a pour rôle de définir l'architecture qui servira de base au développement du système. Le développeur raffine cette architecture de façon à s'approcher, petit à petit, du système final. On parle alors du passage d'une architecture abstraite vers une architecture concrète. Ce passage se fait par raffinement successif ; ce qui consiste à ajouter de plus en plus de détails à l'architecture abstraite jusqu'à ce que l'architecture soit assez détaillée pour pouvoir être implémentée sans ajouter d'informations nouvelles. A chaque étape du raffinement, l'analyste doit être en mesure de vérifier que l'architecture raffinée est conforme à l'architecture du niveau d'abstraction supérieur. Ce processus permet de garantir que l'application obtenue respecte les propriétés fonctionnelles, structurelles et comportementales définies par l'analyste en accord avec le client et les utilisateurs.

Le nombre de raffinements successifs peut être très différent d'une application à l'autre. En effet, il dépend de la précision nécessaire de l'architecture pour pouvoir passer au code. Cette étape de raffinement est nécessaire pour les grosses applications qui ne peuvent pas être construites en une seule passe, mais on peut très facilement s'en passer pour des applications de plus petites tailles. Dans ce cas, l'application pourrait directement être implémentée à partir de la première architecture si celle-ci est assez précise.

4.4 La notion de style architectural

Les principes d'organisation architecturale pour les systèmes logiciels sont appelés styles architecturaux. Un style formalise la connaissance et l'expérience dans un domaine logiciel spécifique. Le but principal des styles architecturaux est de simplifier la conception des logiciels et la réutilisation, en capturant et en exploitant la connaissance utilisée pour concevoir un système [MKMG97]. Un style architectural définit une famille des systèmes logiciels ayant un vocabulaire commun pour désigner les composants, des caractéristiques topologiques et comportementales communes, et un ensemble de contraintes sur les interactions parmi les composants. Un style architectural est moins contraignant et moins complet qu'une architecture spécifique. Il spécifie uniquement les contraintes les plus importantes, au niveau par exemple de la structure, du comportement, de l'utilisation des ressources des composants [AA96]. D'une façon générale, les styles architecturaux permettent à un développeur de réutiliser l'expérience concentrée de tous les concepteurs qui ont précédemment fait face à des problèmes similaires [KK99].

Un style architectural fournit typiquement [Gar95] :

- un vocabulaire pour spécifier les types d'éléments de construction : Agent, Groupe, Client, Serveur, etc.
- des règles de configuration, ou contraintes, qui déterminent comment les éléments architecturaux peuvent être composés. Une contrainte peut interdire à deux éléments de communiquer, elle peut définir aussi des patrons spécifiques (une organisation hiérarchique ou en étoile).
- une interprétation sémantique par laquelle les compositions des éléments architecturaux ont une signification bien définie.
- des analyses qui sont associées au style et qui identifient des caractéristiques de différentes sortes (par exemple vérifier une mesure telle que le nombre d'éléments ou vérifier la satisfaction d'une propriété etc.).

Les styles architecturaux ont un très haut niveau d'abstraction. Ils ont émergé naturellement de l'expérience du développement logiciel et en particulier de la conception architecturale. Ils sont utilisés très tôt dans le processus de développement d'un système logiciel, au début de la conception architecturale. Ceci va être illustré dans la section suivante.

Les styles architecturaux ont longtemps été définis et utilisés comme des concepts, des guides de conception informels. Puisqu'un style architectural représente une famille entière de systèmes logiciels, il est désirable de formaliser le concept de style architectural pour à la fois avoir une définition précise de la famille de systèmes et pour étudier les propriétés architectu-

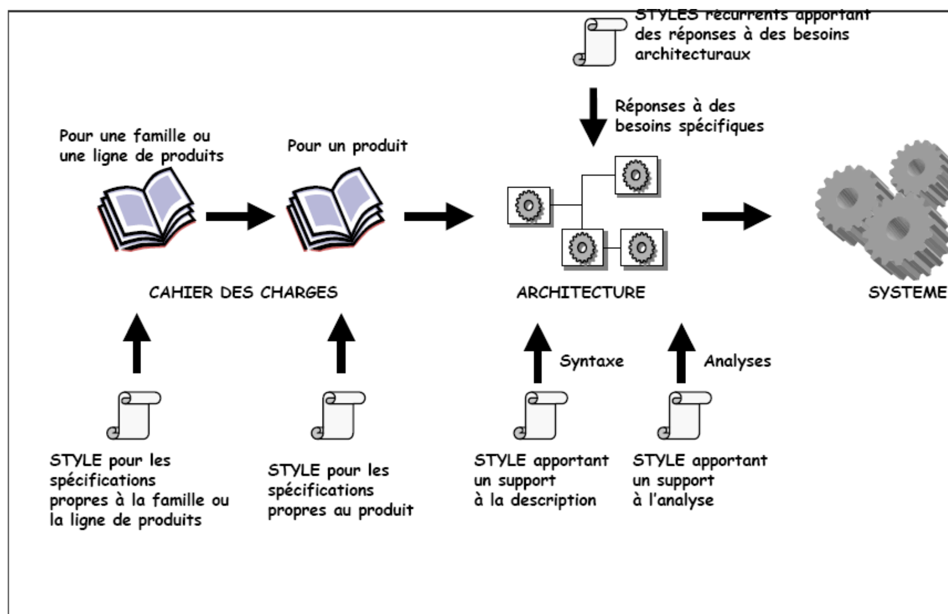


FIG. 4.9: Le processus centré architecture [Ley04]

rales communes à tous les systèmes de la famille [BGPG02]. Les travaux menés depuis lors ont souligné l'intérêt de formaliser les styles [AAG95], et de nombreux langages dédiés à cette tâche ont été développés. Les ADLs devraient permettre la définition des styles architecturaux [NR99]. De plus ils devraient fournir un mécanisme pour exploiter un style dans la définition d'une architecture. Il est aussi utile de pouvoir définir des sous-styles. De nombreux ADLs sont spécifiques à un domaine et à un style en particulier. Par exemple, le style MetaH [Ves92], [Ves94] est un langage spécifique aux architectures des systèmes multiprocesseurs temps réel pour l'aviation.

Les styles sont présents tout au long du cycle de développement centré architecture, depuis la spécification des besoins jusqu'à l'exécution du système (4.9). Les styles offrent un cadre et un support à la conception architecturale, mais un style en particulier peut promouvoir un aspect plutôt que l'autre suivant sa définition. Le schéma ci-dessous montre que les styles peuvent être utilisés pour formaliser les spécifications d'une famille (ou d'une ligne) de produits à partir d'un cahier des charges, pour apporter un support à la description avec des notations spécifiques, pour apporter un support analytique ou pour apporter un support à la conception avec des styles répondant à des problèmes spécifiques.

4.5 Apports potentiels de l'approche orientée architecture dans l'ingénierie des systèmes multi-agents

Comme nous l'avons mentionné dans la section 3.6 l'ingénierie dirigée par les modèles permet de mieux structurer le cycle de développement. En utilisant la métamodélisation, les concepts multi-agents seront mieux définis, ce qui enlèvera certaines ambiguïtés. La séparation des aspects permettra de définir les concepts multi-agents et les concepts liés au domaine d'application de manière indépendante. Ceci simplifie le développement puisque les détails de chaque aspect seront dégagés. Les modèles de cycle de développement de IDM suggèrent de spécifier les PIMs à partir des métamodèles. Ces PIMs peuvent parfaitement correspondre aux architectures logicielles puisque leur rôle consiste à spécifier en détail les différentes vues du système. La plupart des travaux utilisent pour cela le formalisme UML. Bien que UML soit parfois considéré comme un formalisme adéquat pour la description des architectures (cf. section 5.2), plusieurs difficultés ont été recensées quant à son efficacité à produire des modèles de qualité qui soient corrects, extensibles, réutilisables et opérationnels.

Rappelons que dans notre problématique, nous avons souligné l'absence de phases de vérification et validation dans les diverses méthodologies orientées agent. Nous avons aussi précisé que dans le cas des systèmes complexes comme ceux développés avec le paradigme agent, diverses propriétés doivent être vérifiées si nous voulons aboutir à un logiciel sûr. Malgré l'utilisation du langage OCL, certaines propriétés comportementales ne peuvent être exprimées et vérifiées sur les modèles UML. C'est pour cela que nous pensons qu'utiliser les ADLs peut être adéquat à la spécification des architectures logicielles au niveau du PIM. Les avantages des ADLs sont résumés dans le tableau 4.5.

5 Combinaison des deux approches

L'utilisation de l'approche centrée architecture est tout à fait envisageable au niveau de l'IDM. Plusieurs similarités entre les deux approches existent. Avant de discuter les détails techniques qui peuvent être mis en œuvre pour appliquer une telle combinaison, nous résumons dans le tableau 5, les similarités des deux approches.

Problématiques des systèmes multi-agents	**Approche Orientée Architecture**
La sémantique des concepts utilisés est très ambiguë	La spécification **d'un style architectural** permet d'éviter toute ambiguïté par rapport à l'application d'un concept et la définition du composant qui l'implémente puisque celui-ci permet de fournir le vocabulaire, les contraintes, l'interprétation sémantique et les analyses liées aux concepts
La plupart des méthodologies sont basées sur l'approche orientée objet cependant celle-ci n'est pas bien adaptée puisqu'elle a un bas niveau d'abstraction par rapport aux concepts orientés agents	L'approche orientée architecture fournit **un niveau d'abstraction plus élevé**. Le développeur réfléchit sur les éléments composant le système ainsi que sur leurs interactions.
Les méthodologies n'offrent pas de supports consistants pour la vérification et la validation des modèles lors de la phase de conception	L'utilisation des **ADLs formels** rend possible **la vérification et la validation des propriétés structurelles et comportementales** et ce lors des différentes étapes de développement, **grâce aux règles de raffinement.**
Les phases d'analyse/conception ne sont pas connectés à la phase d'implémentation	**Le cycle de développement par raffinement successif** permet de limiter les écarts entre les différentes phases de développement.

Tab. 4.2: Les apports relatifs à l'application de l'approche centrée architecture dans le contexte multi-agents

L'approche IDM	**L'approche Centrée Architecture**
Développement centré sur l'utilisation des modèles	L'approche IDM permet de manipuler des modèles or l'architecture est un modèle qui permet de décrire à la fois la structure et le comportement global du système. Tout système se caractérise par une architecture particulière. Ainsi, le modèle architectural peut décrire les PIMs et les PSMs introduits par l'IDM.
Transformation de modèles	Le développement centré architecture est basé sur la notion de raffinement, cette notion rencontre celle des transformations de modèles d'IDM.
La métamodélisation	L'approche IDM met en avant la métamodélisation qui permet de définir la sémantique des concepts. Ceci rejoint le principe des **styles architecturaux** qui décrivent de manière plus abstraite une architecture particulière d'un système

Tab. 4.3: Combinaison de l'approche IDM et l'approche centrée architecture

Nous situons l'architecture logicielle au niveau du PIM et les styles architecturaux au niveau des métamodèles. Notre objectif durant cette thèse est de décrire les mécanismes qui à partir des métamodèles (décrits par des styles architecturaux) vont permettre de définir les patrons architecturaux nécessaires à la spécification de chaque vue du système multi-agents. Ces patrons architecturaux - qui vont constituer notre PIM - devraient respecter la sémantique définie au niveau des métamodèles. Nous étudierons aussi la manière de combiner ces différents patrons afin d'avoir une description globale du système. Une partie importante de notre travail sera consacrée à la définition de certaines propriétés structurelles et comportementales liées aux patrons architecturaux. Une autre partie correspondra à la défintion des règles de raffinement permettant d'obtenir ces combinaisons tout en respectant la vérification et la validation des propriétés. Cette solution va être expliquée avec plus de détail dans le prochain chapitre.

6 Conclusion

Ce chapitre nous a permis de préciser la problématique à laquelle s'attaque cette thèse. A partir de cette problématique, l'étude des différentes techniques d'ingénierie existantes (exposées dans le chapitre 3) nous a amené à choisir deux approches principales afin de mieux structurer le développement des systèmes multi-agents. Le développement centré architecture nous permettra de focaliser sur l'architecture logicielle des systèmes multi-agents. Le but recherché à l'application de cette approche est l'obtention de logiciels corrects, extensibles et réutilisables. L'ingénierie dirigée par les modèles servira à spécifier le cadre de développement permettant de réaliser les différentes phases du cycle de développement. Trois avantages majeurs ressortent de l'utilisation de cette approche :

- la pérennisation du savoir relatif à l'utilisation des concepts (aussi bien les concepts multi-agents que les concepts du domaine) ;
- la pérennisation du savoir faire grâce à l'application de règles de transformation permettant de combiner divers modèles ;
- un gain de productivité grâce à la spécification des règles de transformation servant à la génération de code.

Dans le prochain chapitre, nous abordons les moyens techniques relatifs à la mise en œuvre de notre approche.

Chapitre 5

L'approche ArchMDE

1 Introduction

Dans ce chapitre, nous décrivons notre démarche de développement des systèmes multi-agents qui est basée sur la combinaison de l'approche IDM et de l'approche centrée architecture. Nous baptisons notre approche ArchMDE pour Architecture Model Driven Engineering.

Dans la première partie de ce chapitre, nous allons détailler les principes de ArchMDE en précisant les artefacts considérés au niveau de notre approche ainsi que les différentes manières de les utiliser. Cette première partie reste générique et ne fait pas de suppositions quant à l'utilisation d'outils techniques particuliers. La deuxième partie illustre l'application de notre démarche ArchMDE à l'aide de l'environnement ArchWare qui sera présenté dans la section 4 de ce chapitre.

2 L'approche ArchMDE

ArchMDE combine l'approche IDM et l'approche centrée architecture. En respectant les principes d'IDM, ArchMDE est basée sur la construction et la transformation des modèles. Dans ArchMDE, les modèles sont des constructions architecturales. Le métamodèle est représenté par un style architectural, le PIM par une architecture particulière et le PSM par le code source.

Dans cette première section, nous commençons par présenter les modèles considérés dans ArchMDE. Par la suite, nous précisons comment ces modèles

sont réalisés dans le cadre d'un processus de développement ArchMDE.

2.1 Les modèles considérés

ArchMDE repose sur l'architecture à quatre niveaux de MDA (cf. section 3.2). Dans ce qui suit, nous allons redéfinir cette architecture dans notre contexte.

Le méta-métamodèle

Rappelons que le méta-métamodèle sert à définir la syntaxe et la sémantique selon lesquelles les métamodèles vont être exprimés (cf. section 3.2). Un langage générique et réflexif doit être adopté à ce niveau. Un langage réflexif est un langage qui se définit par lui-même. Cette caractéristique permet d'une part d'éviter de multiplier les niveaux d'abstraction et d'autre part de rendre le langage facilement extensible ; ce qui permet de définir de nouveaux concepts appartenant à des domaines différents. Traditionnellement, dans les approches IDM ce sont les langages basés sur le formalisme de MOF qui sont utilisés. Nous reprochons à ces langages le manque de formalisation ainsi qu'une ambiguïté quant à la définition de certains concepts et leurs utilisations. Le fait que MOF soit exprimé en utilisant le diagramme de classe UML brouille les frontières entre les deux niveaux d'abstraction (méta-métamodèle et métamodèle). De plus, MOF est orienté objet et est principalement défini en terme de classes, d'associations, de paquetages, de types de données, d'exception, de constantes et de contraintes, ce qui peut limiter considérablement notre approche, puisque les agents comme nous l'avons démontré au chapite 2 ont un niveau d'abstraction plus élevé que les objets logiciels. L'utilisation des objets dans notre contexte risque d'engendrer des systèmes fortement couplés. Enfin, MOF est très difficile à maîtriser : les spécifications de OMG sur ce sujet restent vagues et compliquées à comprendre ; ce qui rend son utilisation difficile.

Pour ces raisons, nous favorisons l'utilisation des langages formels pour exprimer les métamodèles. Nous avons démontré dans le chapitre précédent, qu'il y avait une grande similarité entre le rôle des métamodèles et des styles architecturaux (cf. tableau 5 du chapitre 4). L'utilisation d'un ADL permettant la définition des styles est donc tout à fait envisageable à ce niveau. Cependant, il faut prendre soin de choisir un ADL générique permettant la description de divers concepts appartenant à des domaines différents. Ce langage ne doit pas imposer des restrictions qui peuvent être incohérentes ou conflictuelles avec celles sous-jacentes à un domaine particulier [1].

[1] Par exemple, C2SADEL [MRT99] ne permet pas à deux interfaces (ports) d'être connectées directement à un même troisième.

S = <exp> ::= x
<exp> ::= <exp> - <exp>

E = x - x - x

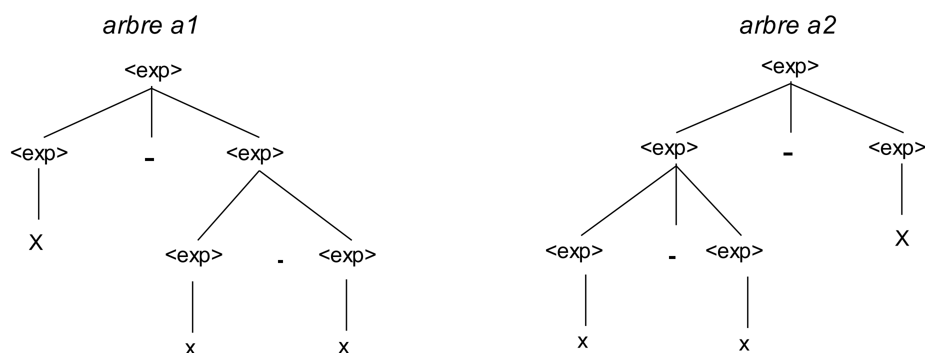


Fig. 5.1: Arbres de dérivation d'une expression arithmétique

De plus, les langages formels permettent l'expression et la vérification des propriétés. En effet, ces langages sont à la fois définis par une syntaxe mais également par une sémantique formelle associée aux différentes constructions du langage. Une syntaxe d'un langage formel est souvent exprimée par une grammaire EBNF (Extensible BNF). Cependant, celle-ci ne traduit que la structure sans s'attaquer à la sémantique. Dans notre cas, la sémantique est en effet décrite par les contraintes et les analyses que le style introduit, ce qui permet de réduire les arbres de dérivation qui peuvent être générés lors de l'interprétation d'une expression du langage. Nous allons illustrer nos propos par un exemple simple, lié à la formulation d'une expression arithmétique. Bien que cet exemple puisse être considéré de bas niveau, il illustre bien les problèmes sémantiques qui peuvent apparaitre lors des formalisations.

Prenons donc l'exemple de la syntaxe S décrivant une opération algébrique, la soustraction (cf. figure 5.1). En appliquant cette syntaxe, nous pouvons obtenir une expression arithmétique E. L'interprétation de cette expression peut être réalisée de deux manières différentes décrites par l'arbre *a1*, soit par l'arbre *a2*. Pour éviter cette situation, le méta-métamodèle devrait donner la possibilité d'exprimer des règles sémantiques relatives à l'utilisation des expressions syntaxiques. Dans notre exemple, ceci revient à préciser deux règles sémantiques permettant de préciser le sens de l'associativité. Une première règle *R1* qui correspond à l'arbre *a1* décrit une associativité à droite et une règle *R2* qui correspond à l'arbre *R2* décrit une associativité à gauche. Pour éviter toute confusion entre les niveaux, nous précisons que la syntaxe (S), peut être considérée comme un méta-métamodèle. L'expression E dé-

crit un style (métamodèle) qu'on pourrait appeler *additionTroisNombres*. Ce style peut en fait être spécialisé en deux sous-styles. Le premier imposera la règle R1 alors que le deuxième imposera la règle R2. Lors de l'instanciation du style (au niveau architecture), si l'expression respecte le style R1, elle va être interprétée par x-(x-x). Si elle respecte le style R2, elle sera interprétée par (x-x)-x.

Les métamodèles

Dans notre contexte, le métamodèle nous permet de définir les concepts utilisés dans un système multi-agents. Comme nous l'avons vu dans le chapitre 3, plusieurs vues regroupant des concepts orientés agents peuvent être définies. Nous avons aussi souligné que ces vues peuvent largement dépendre des domaines d'application. Une question nous est donc parue essentielle dans ce contexte. Comment associe-t-on les concepts du domaine d'application aux concepts orientés agent ? Dans les méthodologies orientées agent que nous avons parcourues, très peu traitent les concepts du domaine indépendamment des concepts orientés agent. Le métamodèle agent est généralement fixé, et les concepts du domaine vont être directement décrits en utilisant les concepts orientés agent. Ceci peut compliquer davantage la tâche des développeurs qui se retrouvent à manipuler les concepts agents et les concepts du domaine en même temps. Dans ce contexte, certaines propriétés du domaine d'application risquent d'être omises ou traitées d'une manière inappropriée.

Un des principes de l'approche IDM est la séparation des aspects qui stipule que chaque aspect d'un problème soit traité indépendemment afin de pouvoir se concentrer plus efficacement sur chacun. Dans notre contexte, il nous parait indispensable de pouvoir représenter de manière indépendante les concepts du domaine d'application et de spécifier leurs propriétés afin de pouvoir les *"agentifier"* par la suite d'une manière correcte. Cette séparation des aspects est d'ailleurs illustrée dans le métamodèle de PASSI (cf. section 5.4 du chapitre 3) qui représente trois couches : la couche domaine, la couche orientée agent, et la couche plate-forme d'implémentation.

Ainsi, c'est en s'appuyant sur la décomposition adoptée par PASSI que nous avons identifié trois types de métamodèles :

- *le métamodèle du domaine* : il décrit les concepts utilisés dans le domaine d'application. Il s'agit de définir les entités "métier", faisant parti du domaine d'application, leurs relations ainsi que leurs principales propriétés architecturales.
- *le métamodèle orienté agent* : il décrit les concepts orientés agent ainsi que leurs principales relations et propriétés architecturales. Au niveau de ce métamodèle différentes vues peuvent être intégrées telles que les

vues adoptées dans Ingenias (section 5.3 du chapitre 3), à savoir l'agent, l'organisation, l'interaction, les "tâches & buts" et l'environnement.
- *le métamodèle de la plate-forme d'implémentation* : il décrit les concepts qui sont implémentés par une plate-forme orientée objet ou autre plate-forme *"classique"*.

Ces différents métamodèles permettent d'appliquer la séparation des préoccupations. L'application développée par le paradigme orienté agent est abordée par partie, ce qui réduit la complexité de conception et de réalisation. De plus, cette séparation entre les trois métamodèles nous permet de capturer les propriétés liées à chaque aspect de manière indépendante. La difficulté réside ensuite dans l'intégration des différents aspects afin de générer l'architecture concrète du système. Ce point sera discuté plus loin dans ce chapitre (section 2.2).

Les métamodèles adoptés dans ArchMDE sont spécifiés en utilisant le langage de style de la couche méta-métamodèle. Notons par ailleurs que la représentation de ces métamodèles par des styles architecturaux permet - si le langage de style le permet - de définir aussi bien des propriétés que des contraintes [2] structurelles et comportementales liées à chaque aspect.

Le PIM et le PSM

Dans le cadre de ArchMDE, le PIM et le PSM (cf. section 3.5 du chapitre 4) appartiennent à la couche M1. Ces deux modèles sont obtenus suite au tissage des aspects appartenant aux métamodèles que nous avons définis précédemment. Le tissage doit conserver les propriétés spécifiées dans chaque métamodèle. Il est réalisé grâce à des règles de transformation. C'est à la suite de ce tissage que nous obtenons le PIM et le PSM du système.

Le PIM est obtenu par tissage des concepts du domaine avec les concepts orientés agent. Nous obtenons dans ce modèle, une description agentifiée d'un domaine d'application. A ce niveau, nous sommes pourtant loin d'avoir un PSM. En effet, un domaine agentifié se situe dans la phase de conception et les concepts d'implémentation ne sont pas encore introduits. Ceux-ci sont en effet décrits par le métamodèle de la plate-forme d'implémentation. Une autre transformation est nécessaire qui associe les concepts orientés agents aux concepts de la plate-forme. L'application de ces règles de transformation génèrera le PSM constitué par le code source de l'application (cf. figure 5.2).

[2]Les contraintes peuvent être considérées comme des propriétés particulières devant être absolument satisfaites

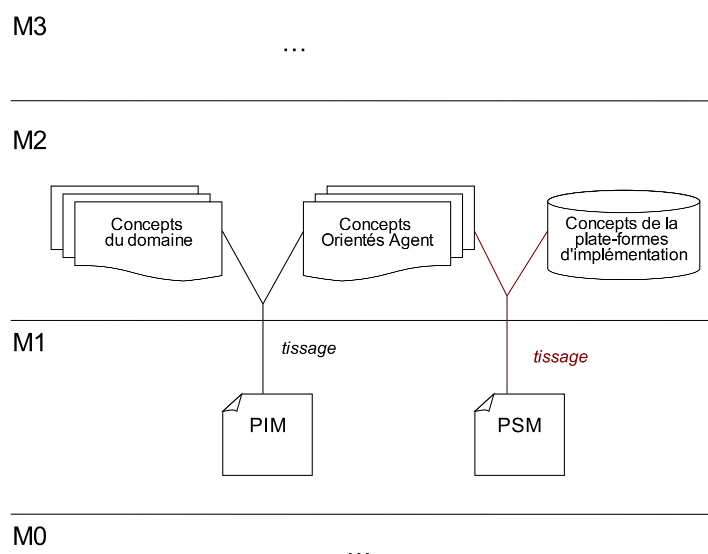


FIG. 5.2: La couche modèle

La couche système

Le déploiement du système à partir du code source généré dans le PSM est effectué au niveau de cette couche. Le système généré doit conserver les propriétés décrites au niveau des métamodèles et des modèles. Après le déploiement du système, un aspect interessant peut être étudié : le monitoring du système ainsi que le contrôle de son évolution. Ces deux points peuvent être considérés d'un point de vue de réingénierie. Que se passe-t-il si durant l'exécution, on fait évoluer le système pour intégrer de nouvelles fonctionnalités ? Ces points ne seront pas étudiés dans cette thèse mais feront partie des perspectives de nos travaux.

2.2 Le cycle de développement ArchMDE

Le but de l'approche ArchMDE est de permettre au développeur de créer son cadre de développement qui soit spécifique à son application. Le cadre de développement va contenir :

- les concepts et les propriétés spécifiques au domaine d'application,
- les concepts et les propriétés spécifiques aux systèmes multi-agents qu'il va utiliser,
- les règles de transformation entre les concepts du domaine et les concepts orientés agents,
- les concepts et les propriétés sur lesquels se basent les plates-formes d'implémentation,
- les règles de transformation entre les concepts orientés agents et les

concepts de la plate-forme d'implémentation.

L'application de la démarche de ArchMDE est située à deux niveaux :

- le premier niveau consiste à définir le cadre de développement formé par les métamodèles et les règles de transformation,
- le deuxième niveau consiste à utiliser le cadre de développement en créant des modèles basés sur les métamodèles définis et en appliquant des règles de transformation entre ces modèles.

Comme nous l'avons défini précédemment, nous proposons trois métamodèles. Ces métamodèles constituent la base du cycle de développement. Nous avons vu précédemment que la principale difficulté de notre approche réside dans le tissage des aspects appartenant à différents métamodèles. Deux principales phases de tissage sont nécessaires dans le cadre de notre approche. La première phase concerne le tissage entre le métamodèle orienté agent et le métamodèle du domaine. Nous appelons ce processus "agentification". La deuxième phase représente une translation des modèles "agentifiés" vers le PSM. Durant cette translation, il s'agit de redéfinir l'architecture décrite au niveau du PIM selon les concepts du métamodèle de la plate-forme d'implémentation. Nous appelons ce processus de transformation un "mapping". Ces deux processus forment en fait des jonctions qui permettent d'associer deux branches. Chaque branche représente une phase de développement. Pour cela, nous décrivons notre cycle de développement par un double Y (cf. figure 5.3). Selon le niveau auquel la demarche ArchMDE est appliquée (développement ou utilisation du cadre de développement), les étapes du cycle de vie en double Y prennent des interpretations différentes, comme nous allons le présenter dans la section suivante.

3 Application du cycle de développement ArchMDE

L'application du cycle de développement ArchMDE peut se faire de deux manières différentes. Chaque manière nécessite des acteurs différents. Dans la première, le processus doit être déroulé par des experts de métamodélisation dont le rôle est de spécifier le cadre de développement pour des domaines particuliers. Ces experts vont définir :

- le métamodèle selon lequel le domaine va être spécifié,
- le métamodèle orienté agent qui va être adopté,
- les règles de transformation entre ceux-ci,
- les règles de génération de code vers une plate-forme spécifique.

La démarche ArchMDE peut aussi être utilisée par des développeurs. A ce niveau, le métamodèle du domaine, le métamodèle orienté agent ainsi que

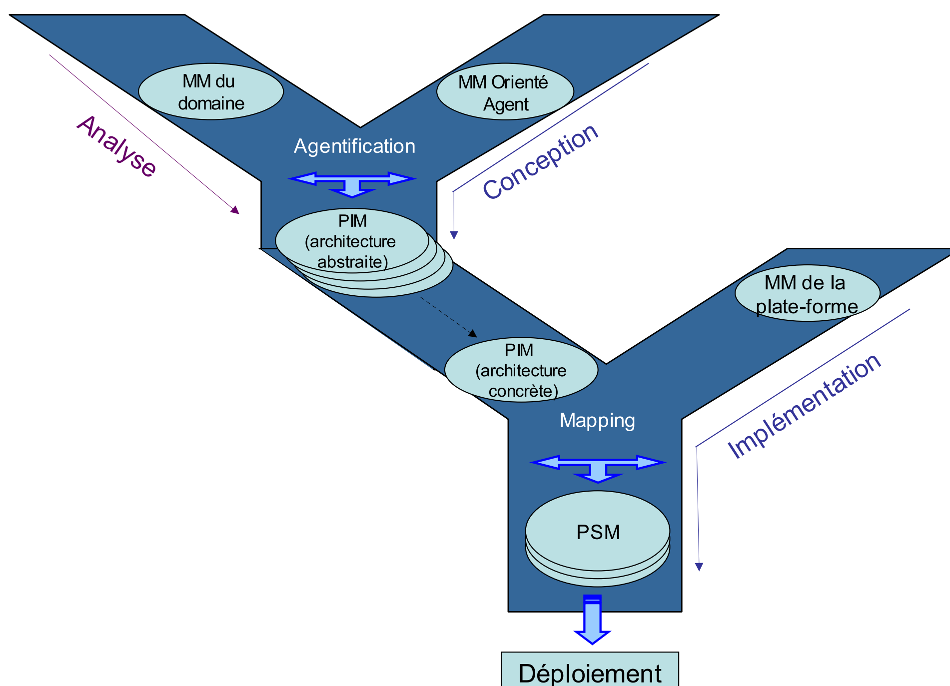


FIG. 5.3: Le processus de développement ArchMDE

les règles de tissage sont déjà spécifiés et le développeur n'aura qu'à les utiliser. Ces utilisateurs auront alors la tâche de décrire leur domaine avec le métamodèle du domaine fourni. Ils utiliseront par la suite les règles de transformation fournies pour agentifier leur domaine. Enfin, ceux-ci utiliseront les règles de génération de code pour générer leur application.

3.1 Construction du cadre de développement par les experts de métamodélisation

Comme nous l'avons spécifié, le rôle des experts de métamodélisation est de spécifier les méta-entités qui vont être utilisées par les développeurs. Dans ce cas, la démarche ArchMDE va être appliquée afin de spécifier le cadre de développement des applications appartenant à des domaines spécifiques (par exemple, un domaine économique, le domaine des systèmes embarqués etc.).

Nous associons la spécification des méta-entités du domaine à la phase d'analyse, la spécification du métamodèle d'agent ainsi que des règles de tissage à la phase de conception. Enfin, la phase d'implémentation concerne la métamodélisation de la plate-forme et la spécification des règles de tissage entre métamodèle de la plate-forme et métamodèle orienté agent. Dans ce qui

suit nous allons détailler les tâches à effectuer durant ces différentes phases.

L'analyse

Durant cette phase, le métamodéliste va produire un métamodèle permettant au développeur de facilement décrire son domaine indépendamment de tout paradigme de développement. Ce métamodèle est à un niveau d'abstraction intermédiaire entre le langage de style et les concepts du domaine. Plusieurs métamodèles peuvent être fournis à ce niveau, par exemple le métamodèle entité-relation [Che76], les cas d'utilisation adoptés par UML, les modèles de flots de données etc.

Dans le cadre de cette thèse, nous avons spécifié un métamodèle de domaine formé par les entités suivantes :

- dataEntity : décrivent les données relatives au domaine. Ces données peuvent être simples ou composées. Par exemple, un contrat de vente peut être considéré comme une *dataEntity*.
- taskEntity : elles décrivent les tâches et les fonctionnalités qui sont effectuées dans le domaine. Par exemple, passer une commande peut être considérée comme une taskEntity.
- activeEntity : ce sont les entités qui décrivent les acteurs du système. Nous incluons aussi bien les acteurs humains ou logiciels. Ceux-ci sont responsables de l'exécution d'une ou plusieurs tâches (taskEntities) selon un processus déterminé.
- structureEntity : ces entités peuvent être considérées comme des structures organisationnelles regroupant toutes les autres entités. Elles représentent les éléments composites du système où plusieurs activeEntities vont fonctionner et interagir ensemble. La différence entre une structureEntity et une activeEntity réside dans le niveau de granularité considéré. Par exemple, une *structureEntity* va représenter une entreprise fournisseur (elle contiendra un acteur chargé de la vente, un acteur chargé de la production et un acteur chargé de la gestion des stocks). Cette structureEntity, peut elle-même être considérée comme une *activeEntity* qui forme un maillon d'une supply chain (travaux en cours de Jihene Tounsi [TBH07]).

Le métamodéliste aura comme mission de formaliser ces méta-entités par un langage de style et de saisir les contraintes relatives à chacunes d'elles. Par exemple, une taskEntity doit être exécutée par au moins une activeEntity ou une structureEntity. Nous allons traiter ce point plus loin dans le chapitre (section 7).

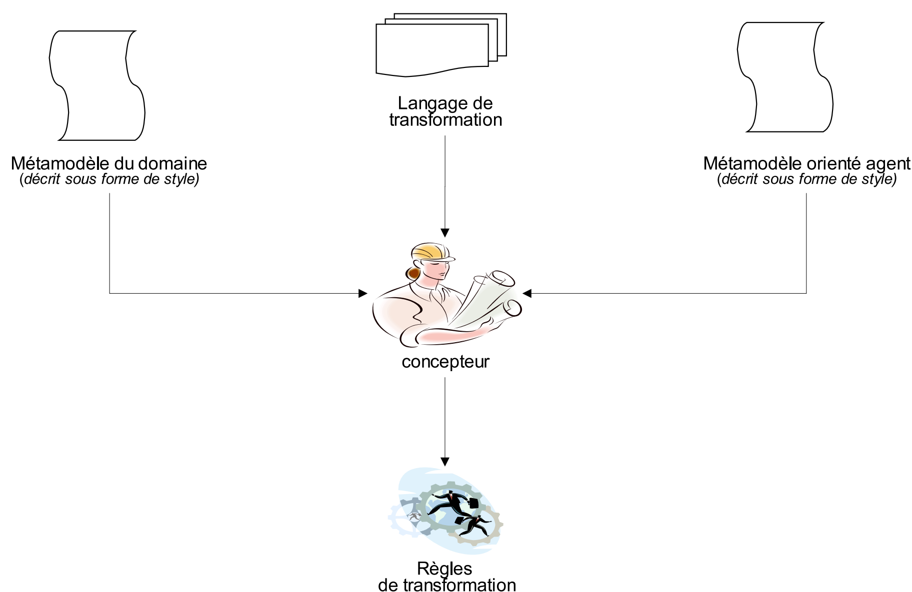


FIG. 5.4: Définition des règles de transformation

La conception

Durant cette phase, les experts de métamodélisation auront à exécuter deux types de tâches :

1. définition du métamodèle orienté agent : en utilisant le langage de style, un expert de système multi-agent peut introduire le métamodèle orienté agent qui lui parait le mieux adapté à sa vision.
2. définition des règles de transformation : quand les deux métamodèles (du domaine et orienté agent) sont spécifiés, la deuxième étape de conception va consister à déclarer les règles de transformation entre les concepts du métamodèle du domaine et ceux du métamodèle agent. Pour réaliser cette étape, le concepteur a besoin d'un langage de transformation. Cette étape est décrite par la figure 5.4.

Durant notre thèse, nous avons spécifié un métamodèle orienté agent qui traduit notre vision [AHO06]. Une version simplifiée de ce métamodèle est illustrée dans la figure 5.5.

Ce métamodèle exprimé avec un langage semi-formel, ne traduit que partiellement les caractéristiques des entités décrites. La formalisation grâce à un langage de style va permettre de saisir toutes les propriétés de manière plus fine. Nous allons traiter l'exemple d'un agent réactif dans la section 8 de ce chapitre.

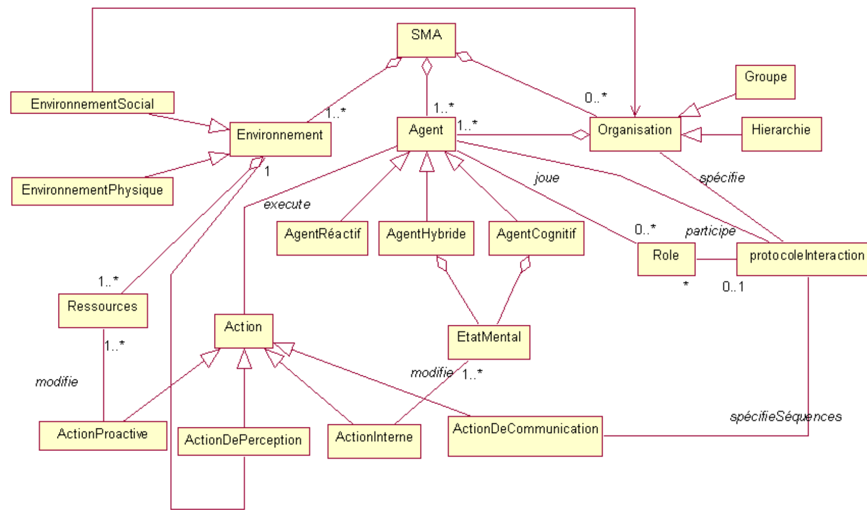


FIG. 5.5: Un métamodèle Orienté Agent

Après la spécification des deux métamodèles du domaine et orienté agent, des règles de transformation génériques seront nécessaires pour permettre la redéfinition d'une entité du domaine vers une entité du métamodèle agent. Si nous reprenons, le métamodèle du domaine que nous avons spécifié dans la sous-section précédente, plusieurs règles de transformation peuvent exister. Nous donnons quelques exemples dans le tableau 3.1.

Au cours de cette phase, un travail important doit être mené qui consiste à définir les règles de transformation des concepts métier vers les concepts orientés agent. Cette transformation doit respecter les propriétés décrites dans les deux métamodèles afin de maintenir la cohérence globale du système.

L'implémentation

Le principe de base durant cette phase, est de transformer les concepts orientés agent vers les concepts implémentés dans une plate-forme d'implémentation quelconque. Cette phase est très délicate et nécessite une grande expertise de la plate-forme d'implémentation. La tâche des experts de métamodélisation à ce niveau consiste à identifier les règles de transformation entre les concepts orientés agent et ceux implémentés dans la plate-forme.

Entités du domaine	**Entités agentifiées**
dataEntity	Etat Mental d'un agent Ressource dans un environnement Attribut d'un agent
taskEntity	Tâche réalisée par un agent Tâche réalisée par une ressource active dans un environnement processus dans un agent Tâche inclue dans le rôle de l'agent Tâche réalisée collectivement
activeEntity	Agent (faible granularité) Groupe d'agents (grande granularité) Ressource active dans un environnement
structureEntity	Organisation d'agents

Tab. 5.1: Mapping entre métamodèle du domaine et métamodèle orienté agent

Cette phase correspond alors à la spécification d'un générateur de code pour un langage donné. Ce générateur de code peut être fourni par l'environnement de description de style.

3.2 Utilisation du cadre de développement par les développeurs

Suite à la construction du cadre de développement par les experts en métamodélisation, les développeurs peuvent l'utiliser pour décrire leurs applications spécifiques. Les tâches à réaliser durant les phases de développements sont alors différentes de celles des experts en métamodélisation.

L'analyse

Au niveau de la phase d'analyse, l'utilisation du cadre de développement ArchMDE va permettre d'acquérir le maximum d'informations concernant les concepts et les processus "métier". Nous supposons que ces informations sont décrites au niveau d'un cahier des charges. Il s'agit durant cette phase de décrire les entités du domaine selon le métamodèle fourni par le cadre de développement. En utilisant ce métamodèle, le développeur n'a pas besoin de maîtriser les concepts orientés agent. Supposons par exemple, que le domaine d'application concerne une application de vente par Internet. Ce domaine comporte plusieurs concepts tels que :

- *les acteurs* : deux sortes d'acteurs sont considérés de prime abord, le client et le fournisseur.
- *le flux de données* : la commande constitue un flux de données échangé entre le client et le fournisseur. Elle peut avoir des éléments descriptifs tels que le prix, le délai de livraison, etc.
- le processus : il peut exister des processus de commande ou de négociation de prix entre le fournisseur et le client.

En utilisant le métamodèle fourni précédemment, les acteurs vont être considérés comme des activeEntities échangeant des flux de données dont les types vont être décrits en tant que dataEntities. Les processus vont être décomposés en plusieurs taskEntities. Celles-ci vont être ordonnancées au niveau d'une structureEntity. Les processus décrivent partiellement le comportement du système. Nous pensons cependant que ceux-ci doivent figurer au niveau du métamodèle. En effet, ils imposent des contraintes comportementales, que le système doit correctement exécuter lors de son déploiement. Le développeur devrait avoir à sa disposition un langage lui permettant de décrire ces entités ainsi que leurs propriétés. Les propriétés deviennent des contraintes lorsqu'elles doivent être absolument satisfaites par le système. Notons par ailleurs que les propriétés ont plusieurs niveaux :

- *niveau structurel* : à ce niveau deux types de propriétés peuvent être prises en compte, les propriétés topologiques qui précisent par exemple le nombre d'occurences possibles d'une entité. Dans notre système, il devrait y avoir au moins un client et un fournisseur. Le deuxième type de propriétés désigne les propriétés d'attribut. Les attributs décrivent généralement les données manipulées par le système. Elles peuvent être contraintes par leur type, l'intervalle de valeurs possibles, etc.
- *niveau comportemental* : il décrit des contraintes comportementales telles que la séquence d'actions qui doivent être exécutées lors d'un processus, etc.
- *niveau interne* : ce niveau décrit les contraintes dont la portée concerne une seule entité du domaine [3] ; par exemple le prix d'un produit ne doit pas dépasser 5 euros, ou un processus récursif doit être relancé à la suite de sa dernière action,
- *niveau global* : ce niveau concerne des propriétés ayant une portée sur le système global ou une partie de celui-ci (une partie d'un système contient plusieurs entités). Par exemple, un protocole d'interaction définit une séquence d'envois de messages effectuée par plusieurs acteurs du système.

A la fin de la phase d'analyse, le métamodèle décrit sous forme de style architectural va introduire (cf. figure 5.6) :
- le vocabulaire lié au domaine et qui spécifie ses différents concepts,
- les contraintes liées à chaque concept,
- les interprétations sémantiques de chaque concept.

La conception

La phase de conception, telle qu'elle est décrite dans la figure 5.3 permet de faire la jointure entre le métamodèle du domaine et le métamodèle agent. L'utilisation du cadre de développement ArchMDE durant cette phase implique deux principales étapes :

1. *application des règles de transformation* : l'application des règles de transformation peut se faire automatiquement si le langage de transformation est fourni avec un outil permettant d'interpréter les règles de transformation. A la suite de l'application des règles de transformation, nous obtenons un style décrivant le métamodèle du domaine agentifié.
2. *génération de l'architecture* : la dernière étape concerne la génération de l'architecture qui va être conforme au style du domaine agentifié.

[3] Nous considérons qu'une entité décrit un ou plusieurs concepts reliés.

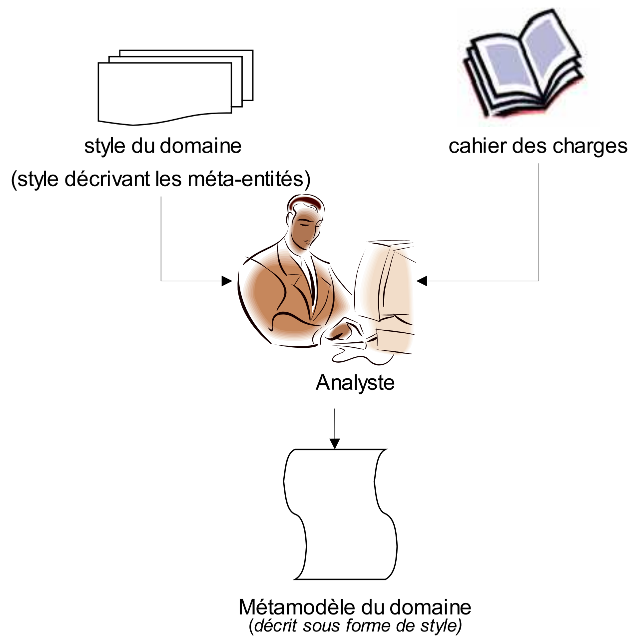


FIG. 5.6: La phase d'analyse

> La génération d'une architecture peut se faire soit manuellement soit à l'aide d'un outil qui permet de configurer le style en vue de l'instancier (cf. figure 5.7).

Reprenons l'exemple du client/fournisseur de la section précédente. Le client est considéré comme un acteur du système. L'entité acteur du métamodèle du domaine peut être associée à l'entité agent du métamodèle orienté agent. Si le client se contente de recevoir un ordre d'achat et de lancer une commande à la suite de cet ordre, celui-ci va être considéré comme un agent réactif. Par contre, supposons que le client avant de lancer sa commande, doit étudier les prix pratiqués lors des transactions ultérieures afin de fixer un seuil à ne pas dépasser. Cela suppose que le client est capable de garder un historique de ses actions. De ce fait, le client ne peut être considéré comme un agent réactif puisque ce dernier n'a ni un état mental ni un historique. Cependant, le client peut être conçu comme un agent cognitif ou hybride. Si les propriétés sont validées, l'entité du domaine devient alors une spécification de l'entité agent. A ce niveau, le concepteur utilise les règles de transformation fournies paramétrées avec les entités de son domaine. L'application de ces règles va générer un nouveau style dédié au domaine agentifié. Le concepteur va par la suite configurer ce style afin de générer son architecture concrète.

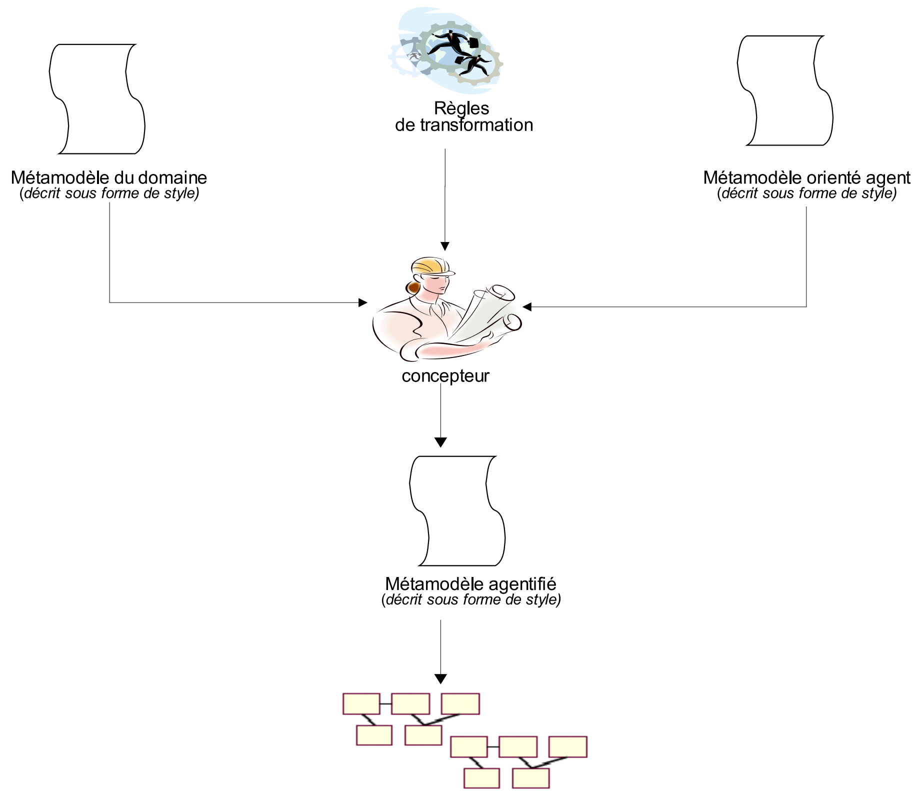


Fig. 5.7: Application des règles de transformation

L'implémentation

Durant la phase d'implémentation, le développeur n'aura qu'à choisir le langage cible avec lequel il voudrait générer son application. Le générateur de code étant fourni par le cadre de développement, le développeur n'aura qu'à l'utiliser.

Dans la suite de ce chapitre nous allons explorer comment on peut mettre en œuvre l'approche ArchMDE à l'aide d'un environnement de développement centré architecture.

3.3 Les besoins relatifs à la mise en œuvre de l'approche

Pour mettre en œuvre ArchMDE, il est nécessaire de disposer d' *"outillages"* qui aideront particulièrement dans la construction du cadre de développement. Ces outillages seront regroupés dans un environnement de modélisation. Ils sont composés de :

- langage constituant le méta-métamodèle (langage d'expression de style) et qui va être utilisé pour créer les différents métamodèles. Il doit être assez générique pour permettre la capture des spécificités de chaque domaine. Par ailleurs, ce langage doit être facile à manipuler et compréhensible par tous les participants au projet de développement afin d'encourager le travail d'équipe,
- un langage de transformation/raffinement : ce langage doit faciliter la définition des règles de transformation/raffinement et ce entre les concepts du domaine et les concepts orientés agents. Les règles de transformation doivent définir le contexte sur lequel va s'appliquer la transformation, l'opération à effectuer et les propriétés à vérifier, ce qui nécessite un système de navigation sur les différents modèles,
- un langage d'expression de propriétés : ces propriétés peuvent être soit des contraintes structurelles ou comportementales exprimées au niveau du métamodèle (et devant être respectées au niveau des modèles) ; soit des propriétés architecturales qui régissent le comportement du système modélisé et qui doivent être respectées lors de son déploiement,
- des outils d'interprétation et d'application des règles de transformation ainsi que des outils de vérification des propriétés sont nécessaires afin de raffiner et valider les modèles (architectures) produits.

Pour fournir ces différents éléments, nous adoptons l'environnement d'ingénierie logicielle ArchWare, développé en partie par le laboratoire LISTIC dans le cadre d'un projet européen. Nous présentons cet environnement dans la section suivante.

4 L'environnement ArchWare

L'environnement ArchWare a été développé au sein du projet européen ArchWare (ArchWare European RTD Project IST-2001-32360). L'objectif de ce projet consistait à fournir un environnement d'ingénierie logicielle qui permet de répondre aux problématiques de développement actuelles. Dans ce contexte, ArchWare fournit des solutions consistant en des langages, des modèles de conception et des outils pour l'ingénierie des architectures logicielles dynamiques (c'est à dire dont la topologie peut changer) ayant une forte interaction entre ses composants. Ceci est le cas des systèmes multi-agents puisque les agents sont des entités communicantes, ayant de fortes interactions avec leur environnement ; et qui sont susceptibles de faire évoluer la configuration du système dans lequel ils opèrent (théorie d'émergence et d'auto-adaptation - chapitre 3).

Plusieurs partenaires ont participé à l'élaboration de cet environnement : The University of Manchester (Angleterre), The University of St Andrews (Ecosse), Engineering Ingegneria Informatica S.p.A. (Italie), CPR - Consorzio Pisa Ricerche (Italie), INRIA Rhônes-Alpes (France), Thésame - Mécatronique et Management (France) et InterUnec - University of Savoie (France).

Dans la mesure où nous avons activement participé à ce projet, il nous paraît judicieux de l'utiliser dans le cadre de cette thèse. Mais ceci ne constitue pas le seul argument de notre choix. En effet, l'environnement ArchWare fournit plusieurs outils qui peuvent nous aider à mettre en œuvre notre approche ArchMDE. Ceci va être discuté dans la section **??** où nous montrerons que la vision d'ArchWare rejoint sur plusieurs point celle de ArchMDE. En effet, le but d'ArchWare est de fournir un environnement personnalisable d'ingénierie logicielle qui peut être utilisé pour créer des environnements logiciels centrés architecture. Les mécanismes clés de cette personnalisation sont les styles architecturaux qui rappelons-le forment aussi la base de notre approche ArchMDE. Dans le cadre d'ArchWare, un langage de style est fourni permettant de décrire les concepts, leurs caractéristiques et leurs contraintes. Ce langage permet aussi de définir une syntaxe pour appliquer ces concepts ainsi qu'un mécanisme de génération d'architecture. Ceci rejoint donc nos besoins pour la mise en œuvre de ArchMDE. De plus, l'environnement ArchWare est basé sur une approche formelle améliorant ainsi l'exactitude du système modélisé et garantissant une certaine qualité de développement à travers une analyse rigoureuse des propriétés.

Le projet ArchWare fournit un environnement personnalisable centré architecture structuré en deux couches distinctes, à savoir un cadre conceptuel visant à produire les modèles architecturaux et un cadre d'exécution visant

à exécuter ces modèles afin de valider leurs comportements.

Le cadre conceptuel d'ArchWare inclut notamment :

- un éditeur textuel permettant la définition des styles et des architectures ;
- un animateur graphique ;
- des outils de vérification de propriétés (statiques et dynamiques) ;
- un raffineur permettant l'interprétation et l'exécution des règles de raffinement ;
- un générateur de code permettant de générer le code correspondant à l'architecture dans le langage souhaité.

Le cadre d'exécution inclut un moteur d'exécution d'architectures, un processus de raffinement de description d'architecture (qui effectue un raffinement en cours d'exécution) et des mécanismes supportant l'interopérabilité des outils de l'environnement. Nous n'allons pas utiliser le cadre d'exécution au niveau de cette thèse. Nous allons en effet, nous focaliser sur les phases d'analyse et de conception ainsi que sur la production de leurs artefacts.

De ce fait, ArchWare offre un environnement complet pour l'application de notre approche ArchMDE. Avant d'expliquer l'utilisation de l'environnement ArchWare dans le cadre de ArchMDE, nous présentons les langages fournis par cet environnement.

5 Les langages ArchWare

Les langages ArchWare possèdent une grande souplesse dans les descriptions d'architectures dynamiques ayant une forte interaction entre ses composants. Ceci est dû au fait que les langages ArchWare soient basés sur le π-calcul [Mil99]. Ces langages sont :

- ArchWare Architecture Description Language (π-ADL) : c'est un langage pour la description d'architectures ayant des caractéristiques dynamiques et évolutives. En effet, le langage permet de capturer les mécanismes permettant à l'architecture de changer de topologie et de configuration durant l'exécution,
- ArchWare Architecture Analysis Language (AAL) : c'est un langage pour la spécification de divers propriétés architecturales (structurelles et comportementales),
- ArchWare Style Language (ASL) : l'utilisation de ce langage permet la personnalisation de l'environnement puisqu'il permet la description de styles architecturaux. A travers ce langage nous pouvons introduire la description de divers concepts en leur associant des contraintes. Ce langage permet aussi l'introduction de syntaxe adaptée à ces concepts.

Enfin, ArchWare ASL permet la description de générateurs d'architecture (appelés constructeurs),
- ArchWare Architecture Refinement Language (ARL) : c'est un langage pour la description de règles de raffinement/transformation.

Nous allons dans cette section introduire brièvement chacun de ces langages. Une description plus approfondie est fournie dans l'annexe A. Notons aussi que ces langages sont tous définis par une EBNF.

5.1 Le langage ArchWare ADL (π-ADL)

Le projet ArchWare propose une famille de langages de description d'architecture. Cette famille est structurée en couches, à partir d'un langage noyau, le langage ArchWare π-ADL. Les différentes couches sont construites grâce à un mécanisme d'extension, le style architectural. Le langage π-ADL [OACV02], [COB+02] est un langage formel conçu pour supporter la spécification exécutable d'architectures logicielles dynamiques et évolutives. Il est fondé sur le π-calcul [Mil99]. π-ADL est défini comme une extension du π-calcul typé d'ordre supérieur : c'est une extension bien formée pour définir un calcul d'éléments architecturaux mobiles et communicants. Ce langage permet de formaliser la structure et le comportement au sein d'une même description.

En π-ADL, une architecture est un ensemble d'éléments, appelés éléments architecturaux, qui sont reliés par des liens de communication. Ces éléments sont définis en terme de comportement (***behaviour***). Le langage définit aussi un mécanisme de réutilisation de comportements paramétrés. Ce mécanisme est appelé ***abstraction***. Le comportement est défini par un ensemble d'actions ordonnancées qui spécifient le traitement interne de l'élément (actions internes) et les interactions avec son environnement (actions de communication). Un élément architectural communique avec les autres par une interface caractérisée par un ensemble de connexions (***connection***) qui permet de faire transiter des données. Un mécanisme de composition et un mécanisme de liaison, appelé unification (c'est une substitution au sens du π-calcul) permettent la mise en relation des éléments architecturaux. Ces éléments peuvent interagir lorsqu'ils sont composés et liés. Un élément architectural peut être défini comme une composition d'autres éléments. En d'autres termes, un comportement peut être défini comme un ensemble d'autres comportements interconnectés. π-ADL permet la description d'architectures dynamiques. En effet, les éléments architecturaux peuvent, d'une part, être composés ou décomposés à la volée et, d'autre part, des éléments architecturaux peuvent être créés et liés dynamiquement. Enfin, des éléments architecturaux peuvent transiter comme des données à travers les connexions. Ce qui fournit un mécanisme adéquat pour décrire la mobilité (exemple les agents mobiles).

ArchWare ADL permet de décrire à la fois la structure du système et son comportement.

ArchWare ADL permet la description d'architectures dynamiques. D'abord, les éléments architecturaux peuvent être composés ou décomposés à la volée. Ensuite, des éléments architecturaux et des connexions peuvent être créés dynamiquement. Enfin, des éléments architecturaux et des connexions peuvent transiter comme des données à travers les connexions.

L'exemple suivant définit une architecture décrivant un groupe de deux agents. Ces agents communiquent entre eux : *Agent1* envoie une requête à travers la connexion *call* et *Agent2* reçoit la requête à travers la connexion *request*. Le comportement de chaque agent est décrit dans une abstraction. Ces abstractions sont ensuite appliquées [4] au niveau du groupe qui est lui même décrit par l'abstraction *Groupe2Agents*. Au niveau de cette abstraction les connexions sont unifiées afin que les deux agents puissent communiquer. L'*Agent1* effectue un appel à l'*Agent2*, puis il attend une réponse. L'*Agent2* est en attente de l'appel de l'*Agent1*. Lorsqu'il reçoit cet appel celui-ci lui répond. *Groupe2Agents* est alors considéré comme un élément composite.

```
value Agent1 is abstraction ()
{
via call send; via wait receive;
unobservable
};

value Agent2 is abstraction ();
{via request receive; unobservable;
 via reply send)};

value Group2Agents is abstraction ();
{
value call, wait, request, reply is connection();
compose { A1 is Agent1()
         and
          A2 is Agent2() where {
                    A1::call unifies A2::request
                    and
                    A2::reply unifies A1::wait}
        }
}
```

[4] L'application d'une abstraction génère un comportement.

5.2 Le langage AAL

AAL [AGMO02] est un langage formel pour exprimer les propriétés structurelles et comportementales des architectures logicielles évolutives modélisées en π-ADL. Il est défini comme une extension du μ-calcul [Koz83] qui est limité à l'expression de propriétés comportementales. Pour exprimer les propriétés structurelles, l'AAL s'appuie sur la logique des prédicats.

En AAL, les propriétés sont définies comme des formules prédicatives. Ce langage fournit des prédicats prédéfinis et des mécanismes (opérateurs et quantificateurs) pour construire de nouveaux prédicats. Un support de vérification des propriétés est fourni à travers les outils Analyser [AO05] et model Checker [BCD+04].

L'expression suivante décrit une contrainte qui est vraie si le prédicat p est vrai pour chacun des éléments d'une collection.

```
to collection apply forall{element | p(element)}
```

5.3 Le langage ArchWare ASL

Ce langage est considéré comme un mécanisme d'extension qui représente une famille d'architectures ayant des caractéristiques communes et obéissant à un certain nombre de contraintes. Des concepts peuvent être introduits par un style, et formeront le vocabulaire du style. Il est ainsi possible de définir des styles architecturaux selon une tour de méta-niveaux : en utilisant la couche π-ADL de l'environnement ArchWare, ASL permet de définir une couche n+1, fournissant un vocabulaire lié aux concepts d'un domaine donné. Une architecture définie en utilisant le langage π-ADL a son correspondant dans le langage défini par ASL. Les contraintes associées aux concepts contenus dans les styles sont décrits par des propriétés AAL. Ainsi, ASL forme une couche supérieure incluant π-ADL et AAL (cf. figure 5.8)

Nos travaux se basent sur ce langage. ASL a été contruit en ciblant deux principaux axes. Le premier concerne l'expression des contraintes. Celles-ci permettent de délimiter un espace de conception qui caractérise la famille d'architectures, de vérifier la satisfaction d'une architecture à un style ou de révéler les écarts au style. Le deuxième axe concerne un support à la description architecturale en permettant de spécifier une syntaxe spécifique au système. Celle-ci peut être utilisée comme un DSL (Domain Spécific Language), ce qui est très utile pour les non spécialistes d'ArchWare qui peuvent décrire leur architecture sans avoir à connaitre les mécanismes d'ArchWare.

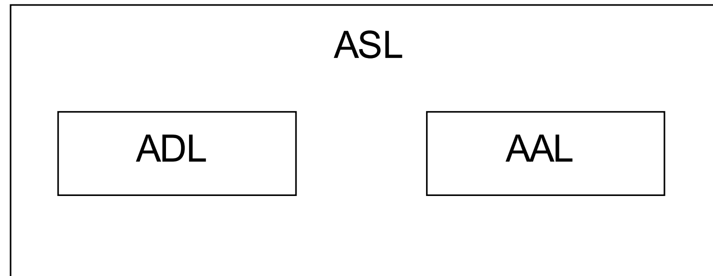


FIG. 5.8: Le langage ASL

De plus, ASL offre des mécanismes pour la réutilisation des styles. Parmi ces mécanismes, nous citons :

- **l'instanciation** : l'instanciation d'un style est la définition d'une architecture telle que celle-ci satisfasse ce style. ASL fournit des constructeurs qui permettent l'instanciation des architectures[5],
- **le sous-style** : lorsqu'un style représente un sous-ensemble d'une famille d'architectures représentée par un style plus abstrait, on parle de sous-style.

Dans le cadre de nos travaux, ASL nous permettra de décrire des styles architecturaux fournissant :

- Un vocabulaire pour spécifier les concepts, par exemple, Agent, Role, etc.
- Des règles de configuration, ou contraintes, qui déterminent comment les éléments architecturaux peuvent être reliés,
- Une interprétation sémantique par laquelle les compositions des éléments architecturaux ont une signification bien définie,
- Des analyses qui peuvent être appliquées sur les systèmes construits dans ce style.

Dans ce contexte, le langage ASL jouera le rôle de méta-métamodèle. et les styles définis selon ce langage joueront le rôle de métamodèle. Les architectures générées à partir de ces styles appartiennent à la couche M1 avant d'être instanciés au niveau de la couche M0. La syntaxe permettant la description d'un style est la suivante. Celle-ci est expliquée en détail au niveau de l'annexe.

```
NomDuStyle is style extending Style_parent where {
    types { Définitions de types }
```

[5]Comme nous allons le voir par la suite le mécanisme de constructeur est bien plus génerique. Il permet l'instatiantion des fragments d'architecture ou des types de données.

```
    styles { Définitions de styles }
    constructors { Définitions de constructeurs }
    constraints { Définitions de contraintes }
    analyses { Définitions d'analyses }
}
```

Notons cependant que *types* définit un ensemble de types manipulés au niveau des styles.*Styles* définit les styles contenus dans *NomDuStyle* (relation d'agrégation). *Constructors* décrivent les mécanismes de génération de l'architecture. Au niveau des constructeurs, une syntaxe spécifique peut être définie qui sera interprétée selon le code spécifié par le constructeur. *Constraints et Analyses* décrivent les propriétés qui doivent être respectées par les architectures définies selon ce style. Ces différentes notions sont décrites de manière détaillée dans l'annexe A.

5.4 Le langage ArchWare ARL

ArchWare ARL [Oqu03] est un langage formel basé sur la logique de réécriture, dédié au raffinement/transformation d'architectures logicielles avec d'une part, la prise en compte du raffinement/transformation des données, des comportements, des connexions et des structures et, d'autre part, la préservation des propriétés architecturales lors de l'exécution des règles de raffinement/transformation. Le noyau du langage est un ensemble d'opérations appelées actions de raffinement/transformation [Meg04]. Chaque étape de raffinement/transformation implique l'application d'une action qui fournit une solution architecturale correcte. Dans ARL, les actions de raffinement/-transformation sont exprimées par des pré-conditions, des transformations et des post-conditions. Les pré-conditions sont des conditions qui doivent être satisfaites dans une architecture avant l'application d'une action de raffinement. Les post-conditions doivent être satisfaites suite à l'application d'une action de raffinement. La transformation décrit l'opération à réaliser.

Un modèle d'architecture peut être raffiné/transformé vers un autre modèle d'architecture. Ceci établit une relation de raffinement/transformation binaire notée ► :

```
archetype architectureDefId is architecture {
types is { typeDeclarations }
ports is { portDeclarations }
behaviour is { compositionOfComponents}
}
```

►

```
archetype architectureRefId is architecture {
types is { typeDeclarations' }
ports is { portDeclarations' }
```

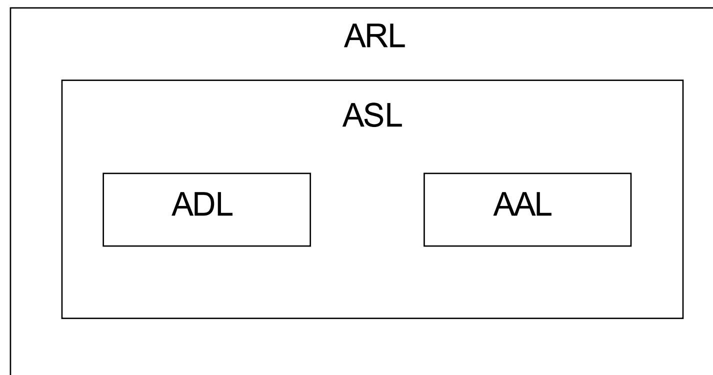


FIG. 5.9: Le langage ARL

```
behaviour is { compositionOfComponents' }
}
```

où :

- typeDeclarations et portDeclarations sont des déclarations de types et de ports avant le raffinement et typeDeclarations' et portDeclarations' sont des déclarations de types et de ports après le raffinement.
- portDeclarations contient les connexions libres avec lesquelles les composants communiquent avec l'extérieur.
- compositionOfComponents est le comportement composite avant le raffinement et compositionOfComponents' est le nouveau comportement composite après le raffinement.

Dans le cadre du projet, le langage ARL a été dédié au raffinement/-transformation de l'architecture (c'est-à-dire la couche M1). Pour nos besoins spécifiques dans l'approche ArchMDE, nous avons étendu ce langage afin qu'il prennent en compte le rafinnement des styles. Dans ce cas, ARL formera une couche supérieure incluant les trois langages ASL, π-ADL et AAL (cf. figure 5.9). Nous allons éclaircir ce point dans la section 9.

6 Les outils ArchWare

Les langages ArchWare sont accompagnés de plusieurs outils qui nous seront indispensables lors du déroulement du processus ArchMDE. Ces outils sont :

- l'ASL Toolkit,
- l'Analyser,
- l'Animator,
- le Refiner,

– le Code Synthesizer.

Pour la création des métamodèles et la génération de l'architecture l'ASL Toolkit est nécessaire. L'analyse des propriétés d'une architecture se fait grâce à l'outil Analyser. Le Refiner permet d'exécuter les règles de raffinement au niveau architectural (niveau M1). Enfin, le code Synthesizer aidera le développeur à implémenter l'architecture.

L'outil ASL Toolkit

L'outil ASL Toolkit [Ley04] permet la personnalisation d'un environnement en s'appuyant sur la définition de styles architecturaux. Cet outil contient deux modules : le module "style compiler" et le module "style instantiator". Le premier module compile des définitions de styles écrites en ASL, tandis que le second génère des instances du style. Celles-ci constituent des architectures (abstraites ou concrètes selon le niveau d'abstraction du style). Ces architectures seront décrites en π-ADL. ASL Toolkit est implémenté en Java et en XSB Prolog.

Cet outil sera utilisé lors de la création des métamodèles. Chaque métamodèle sera compilé afin de vérifier s'il est correctement construit. A la fin du processus d'agentification, l'architecte, pourrait utiliser le module "style instantiator" pour générer son architecture.

L'outil Analyser

Nous avons activement participé à l'élaboration de cet outil [AO05]. Au cours du processus ArchMDE, l'Analyzer va être utilisé pour vérifier que l'architecture générée à partir du style respecte bien toutes les propriétés et contraintes déclarées par celui-ci. Pour cela, l'outil Analyzer s'appuie sur les démonstrations logiques pour réaliser des vérifications de propriétés exprimées en AAL. Il est implémenté en Java et XSB Prolog.

Cet outil comporte deux modules, le module *reifier* et le module *moteur d'analyse*. Le module reifier permet de réifier l'architecture décrite en π-ADL en faits Prolog, afin de pouvoir raisonner sur celle-ci et vérifier les propriétés. Cette réification est faite suite à une analyse syntaxique suivie d'une analyse sémantique de l'architecture décrite en π-ADL. Des faits prolog sont alors générés en fonction de la sémantique de chaque expression.

Le moteur d'analyse est un module qui prend en entrée un fichier contenant la description architecturale en π-ADL et un fichier contenant les propriétés décrites en AAL. Les propriétées en AAL vont être analysées syntaxiquement et sémantiquement avant d'être transformées en prédicats prolog.

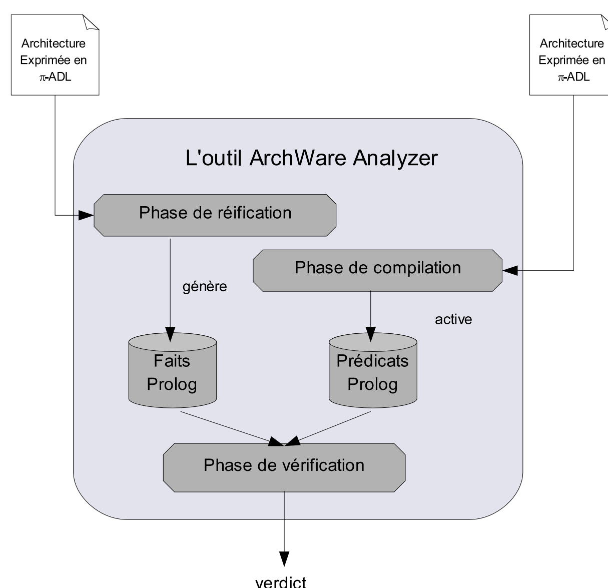


FIG. 5.10: Le processus de vérification effectué par Analyser

Les prédicats Prolog vont raisonner sur les caractéristiques structurelles de l'architecture ainsi que sur les caractéristiques comportementales. En effet, le comportement de l'architecture va être traduit par des arbres reliés chacun à son abstraction. La vérification des propriétés comportementales se fait en parcourant ces arbres.

Le processus de vérification des architectures se fait donc selon deux phases (cf. figure 5.10) :
- processus de réification de l'architecture,
- processus vérification.

L'outil Analyser va être utilisé par l'analyseur après la génération de l'architecture et à chaque pas de raffinement, où l'outil va être évoqué pour vérifier l'architecture.

L'outil Animator

L'outil Animator [APVO03], [PVA+05] anime des descriptions d'architectures écrites en π-ADL, il est implémenté en Java et en XSB Prolog. Cet outil, contrairement à l'analyseur, ne permet pas de vérification formelle. Il s'agit en réalité de générer les traces d'exécution d'une architecture et de les interpréter graphiquement. Ceci permet à l'architecte ou l'analyste de valider le comportement de l'architecture.

Dans le cadre de notre approche l'animateur peut être utilisé après la génération de l'architecture pour interpréter graphiquement les traces d'exé-

cution de l'architecture.

L'outil Refiner

L'outil Refiner [Meg04] se base sur le langage ARL pour construire (par raffinement) une architecture, il est implémenté sous MAUDE [CDS+03]. Il s'appuie sur les actions de raffinement pour générer, de façon formelle, une nouvelle architecture.

Dans notre contexte, l'outil Refiner pourrait être utilisé par l'architecte pour ajouter de nouveaux détails à l'architecture ou pour ajuster celle-ci afin qu'elle respecte certaines propriétés. Cependant, dans le cadre d'ArchWare, cet outil a été conçu pour être utilisé au niveau de l'architecture et n'est pas adéquat pour prendre en compte la transformation de style.

L'outil Code Synthesizer

L'outil Code Synthesizer [BFDP05] s'appuie sur des règles de transformation pour générer la description d'une architecture dans un langage cible (exécutable). Concrètement, cet outil permet de transformer du code écrit en π-ADL en n'importe quel langage cible pour peu que des règles de transformation soient écrites.

De par les langages et les outils qu'il offre, l'environnement ArchWare est bien adapté pour l'application de notre approche ArchMDE. Cet environnement sera utilisé par les experts en métamodélisation afin de construire le cadre de développement spécifique à un domaine. Cet environnement devrait être transparent pour les autres développeurs. Dans la suite de ce chapitre, nous expliquons comment utiliser les langages et les outils durant le processus ArchMDE.

7 Formalisation du métamodèle du domaine

Nous avons formalisé avec le langage ASL le métamodèle du domaine fournis dans la section 3.1, ce qui nous a permis de générer une syntaxe relative à ces concepts, simplifiant ainsi leur application. Nous allons décrire ci-dessous la formalisation de ces entités avec ArchWare ASL. Afin de simplifier la compréhension, nous adoptons un exemple pédagogique décrivant un client et un fournisseur exécutant un processus de commande.

7.1 dataEntity

Une dataEntity n'a aucun comportement mais se contente de décrire certaines données du système. Les *dataEntities* peuvent être simples ou compo-

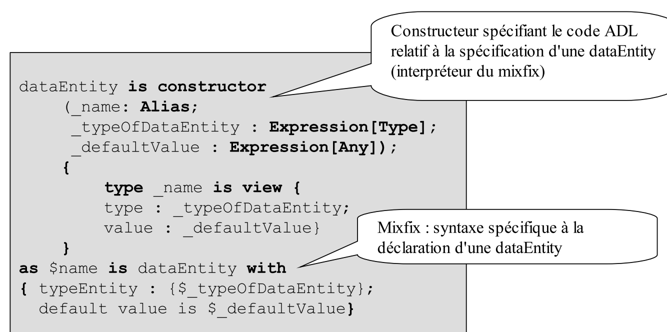


FIG. 5.11: Le code ASL de dataEntity

sites. Pour les décrire, nous utilisons les types de base d'ArchWare ADL.

Une dataEntity est définie par son *name*, qui permet de la référencer. Elle définit aussi les types des données qui la constituent(*typeOfDataEntity*). Accessoirement, une dataEntity peut aussi spécifier les valeurs par défaut qui seront assignées lors de la création/instanciation de la donnée (*defaultValue*). Nous donnons ci-dessous une description du constructeur relatif à *dataEntity* exprimée en ArchWare ASL. A ce constructeur est associé une syntaxe mixfix, qui définit un langage dédié que les spécialistes du domaine d'application peuvent utiliser sans être experts du domaine. Le style correspondant à dataEntity s'exprime de la manière suivante en ArchWare ASL 5.11

Quand il va décrire une dataEntity particulière, le développeur va la syntaxe introduite par le mixfix. Celle-ci permet d'introduire trois paramètres : le nom attribué à celle-ci (variable $_name), le type qui permet de la décrire (variable $_typeOfDataEntity) et optionnellement la valeur par défaut qu'elle prend lors de l'instanciation (variable $_typeOfDataEntity). Par exemple, dans le système que nous traitons, le fournisseur manipule l'attribut stock dans lequel est définie la valeur du stock. A l'initiation, celui-ci contient 100 éléments. Ceci est décrit de la manière suivante, en utilisant la notation mixfix fournie par le constructeur :

```
stock is dataEntity with {typeEntity :  integer;
            default value is 100};
```

Comme nous le constatons, les mécanismes fournis par ASL sont faciles à utiliser. Le constructeur permettra de créer un nouveau type dont le nom est _name contenant son type et sa valeur de base. La notation mixfix facilite l'application de ce type. En effet, l'utilisateur final n'aura pas à connaître la syntaxe de base de ASL. Il utilisera directement la syntaxe définie par le mixfix.

Contraintes et Analyse

Nous avons identifié plusieurs propriétés relatives aux dataEntity :

- *propriétés sur les valeurs* : cette classe de propriétés impose des contraintes sur les valeurs assignées aux instances de dataEntity. Par exemple, la valeur stock ne doit jamais dépasser la capacité maximale d'un dépôt. Ceci peut s'exprimer en AAL de la manière suivante :

  ```
  to stock.Instances apply
  {stock.Instances.value <= depot.Instances.capacity}
  ```

  L'expression de cette contrainte suppose bien sûr qu'un style dépôt soit défini et que celui-ci contienne une valeur *capacité*.

- *propriétés sur la visibilité* : cette classe de propriétés permet de definir le degré de visibilité de la dataEntity à partir d'une activeEntity/taskEntity ou une structureEntity. Ces propriétés portent ainsi sur le lien entre la dataEntity et les autres entités. En effet, une dataEntity peut être visible (et donc accessible) à certaines entités et complètement invisible à d'autres. Au niveau de AAL, nous avons créé un prédicat *isInvisible(dataEntity, Entity)* qui permet de notifier qu'une instance de dataEntity est invisible à l'autre. Un exemple d'utilisation de ce prédicat est fourni dans la propriété suivante qui stipule qu'un fournisseur n'a nullement accès au stock d'un client :

  ```
  to client.Instances apply
   {isInvisible(stock.instances, fournisseur.Instances) }
  ```

- *propriétés sur la cardinalité* : cette classe de propriétés permet de contraindre le nombre d'instances crées pour une dataEntity. Par exemple, la propriété suivante exprime qu'il peut y avoir entre une et cinq instances de clients dans une structureEntity appelée *"clientsFournisseur"*.

  ```
  to clientsFournisseur.Instances.elements apply {
  exists ([1..5] x | x in style Client)
  ```

7.2 taskEntity

Le concept taskEntity décrit les actions atomiques de base qui doivent être réalisées au niveau du domaine. Celles-ci vont être par la suite regroupées selon un certain ordonnancement pour décrire le comportement d'une entité active (*activeEntity*). Une taskEntity décrit donc une partie du comportement. Cette décomposition nous permettra de faciliter l'intégration de ces fragments du comportement au sein d'un processus. Par ailleurs, ceci facilitera l'identification des tâches à assigner aux agents lors de la phase *"d'agentification"*.

Une taskEntity peut recevoir des paramètres en entrée (*_inputParameters-OfTaskEntity*) et produire des résultats en sortie(*_outputOfTaskEntity*) suite à une opération effectuée (*_operationOfTaskEntity*). Les résultats en sortie seront communiqués par des connexions libres. En effet, ArchWare ADL ne permet pas de modéliser des fonctions qui vont directement communiquer leur résultat. Cette entité est formalisée en ASL de la manière suivante :

```
taskEntity is constructor (
        _name: Alias;
        _inputParametersOfTaskEntity : set[Attributes];
        _operationOfTaskEntity : Expression[Behavior];
        _outputOfTaskEntity : set[Connection]);
{
type taskEntity is view [
        name : _name;
        parametersOfTaskEntity : _parametersOfTaskEntity;
        behaviourOfTaskEntity : _operationOfTaskEntity;
        outputOfTaskEntity : _outputOfTaskEntity]
}
}as $_name is task with
{
[input : { $_parametersOfTaskEntity;}]
operation : {$_operationOfTaskEntity;}
[output :{$_outputOfTaskEntity}]
}
```

Nous pouvons observer que le constructeur prend quatres paramètres : le nom, la liste des paramètres en entrée, l'opération et la liste des paramètres en sortie de la taskEntity. Ce constructeur sera dans notre cas utilisé en utilisant la notation mixfix qui lui est associée. L'exemple suivant de l'utilisation du constructeur *taskEntity* représente un comportement qui consiste à créer un bon de commande et l'envoyer. Ceci va s'exprimer de la manière suivante :

```
passerCommande is task with
{
input : {
        _numCommande : String
    _nomClient : String;
    _refProduit : Integer;
    _quantité : Integer;
    _dateDeLivraison : String}
operation : {
value operationPasserCommande is behaviour
  {value Commande is
   view(
   numCommande is _numCommande;
   nomClient is _nomClient;
   refProduit is _refProduit;
```

```
    quantité is _quantité;
    dateDeLivraison is _dateDeLivraison)
  )}
value passerCommande_connection is free connection(Commande);
via passerCommande_connection send Commande;
output :{passerCommande_connection: connection(Commande)}
}}
```

La liste de paramètres d'entrée de cette taskEntity contient un numéro de commande (*_numCommande*), le nom du client(*_nomClient*), la référence du produit (*_refProduit*), la quantité (*_quantité*) et une date de livraison (*_dateDeLivraison*). Le comportement lié à cette opération consiste à créer une instance d'une commande sous forme d'une vue en fonction des paramètres qui ont été passés. Par la suite, cette commande est envoyée au fournisseur via la connexion *passerCommande_Connection*. Cette connexion est donnée dans liste de paramètres de sortie.

Contraintes et Analyse

Les contraintes relatives à taskEntity sont aussi variées. Nous présentons quelques unes :

– la relation inclusion : cette relation est établie entre deux taskEntity [6] et signifie que tout comportement représentant l'instanciation de *taskEntity2* inclut un comportement représentant l'instanciation de *taskEntity1*. Cette propriété peut s'exprimer comme suit :

```
include is analysis {
kind AAL
input {taskEntity1 : taskEntity,
    taskEntity2 : taskEntity}
output{Boolean}
body{
 to taskEntity2.Instances apply {
    exists {t1: taskEntity1.instance | true*.t1.true*}
        }
}}
```

Nous avons modélisé cette propriété par une analyse. En effet, il est utile de vérifier cette propriété au niveau architectural pour savoir si l'architecture analysée répond bien aux exigences du style.

[6] Ici par taskEntity nous faisons référence au style taskEntity. Une instance de ce style peut etre construite en utilisant le constructeur qui porte le même nom et pour lequel nous avons donnée une définition ASL auparavant.

– succession directe : cette contrainte existe aussi entre deux taskEntity et signifie qu'un comportement représentant l'instantitation de *taskEntity1* est immédiatement suivi par un autre comportement représentant l'instanciation de *taskEntity 2*.

```
successionDirecte is analysis {
kind AAL
input {taskEntity1 : taskEntity,
       taskEntity2 : taskEntity}
output{Boolean}
body{
 to style.Instances apply{
   forall{ t1 : taskEntity1.Instances |
    exists{ t2 : taskEntity2.Instances
      | true*.t1.t2.true*}
   }
}
}
```

– succession indirecte : cette contrainte signifie qu'un comportement représentant l'instanciation de *taskEntity1* est indirectement suivi par un autre comportement représentant l'instanciation de *taskEntity2*.

```
successionIndirecte is analysis {
kind AAL
input {taskEntity1 : taskEntity,
       taskEntity2 : taskEntity}
output{Boolean}
body{
 to style.Instances apply{
   forall{ t1 : taskEntity1.Instances |
    exists{ t2 : taskEntity2.Instances
      | true*.t1.true*.t2.true*}
   }
}
```

– choix conditionnel : cette contrainte peut être appliquée pour vérifier que deux taskEntities sont mutuellement exclusives selon une condition *C*. Ceci s'exprime comme suit :

```
choixConditionnel is analysis {
kind AAL
input { taskEntity1: taskEntity,
        taskEntity2: taskEntity}
output{Boolean}
body{
```

```
to style.Instances apply{
  forall{ t1 : taskEntity1.Instances |
   exists{ t2 : taskEntity2.Instances
    |  exists{c: boolean
        |  true*.((c== true).t1)
                or
          ((c==false).t2).true*}
   }
  }
 }
}
}
```

– choix indéterministe : cette contrainte vérifie que deux taskEntity sont mutuellement exclusives sans qu'une condition ne soit fixée. Ceci est exprimé comme suit :

```
choixIndéterministe is analysis {
kind AAL
input { taskEntity1: taskEntity,
      taskEntity2: taskEntity}
output{Boolean}
body{
    to style.instance apply{
    forall{ t1 : taskEntity1.Instances |
        t1.{not (exists {t2 : taskEntity2.Instances
                                | true*.t2}))}
        or
    forall{ t2 : taskEntity2.Instances |
        t2.{not (exists {t1: taskEntity1.Instances
                                | true*.t1}))}
    }
}}
```

– parallélisme : cette contrainte vérifie si deux taskEntity sont mises en parallèle. Elle s'exprime de la manière suivante :

```
parallélisme is analysis {
kind AAL
input { taskEntity1: taskEntity,
      taskEntity2: taskEntity}
output{Boolean}
body{
to style.instance apply{
  forall{ t1 : taskEntity1.Instances |
   exists{ t2 : taskEntity2.Instances |
    true*. compose{
    (true*.t1.true*)
```

```
    and
    (true*.t2.true*)
    }}}}
}}
```

7.3 activeEntity

Les *activeEntities* décrivent les acteurs du système responsables de l'exécution d'une ou plusieurs tâches selon un processus bien défini. Ce processus contient au moins une ou plusieurs taskEntities. Les activesEntities peuvent aussi contenir quelques attributs permettant par exemple, de passer les paramètres aux taskEntities qu'ils vont activer. Ainsi, les activeEntities sont formalisées de la manière suivante :

```
activeEntity is constructor (
    _name: Alias;
    _attributes : set[Attributes]
    _process : Expression[taskEntity]);
{
type activeEntity is view [
   name : _name;
   attributesOfActiveEntity : _attributes;
   processOfActiveEntity : _process]
}
}as $_name is activeEntity with
{
attributes : { $_attributes;}
process : {$_process;}
}
```

Nous continuons notre exemple du client qui va passer une commande à un fournisseur. A la suite de cette commande, le client attend la réponse du fournisseur qui va accepter ou pas de traiter la commande. Cette entité active va avoir un processus séquentiel qui consiste à passer une commande au fournisseur, celui-ci va ensuite attendre la réponse du fournisseur. Si la réponse est positive, le client va recevoir la livraison de la commande et procéder par la suite au paiement. Si la réponse est négative, nous supposons que le processus va s'arrêter. Le processus décrit donc l'ordre d'instanciation des *taskEntities*. Pour traduire cette relation entre les *taskEntities* et les *activeEntities*, nous utiliserons l'opérateur ***new*** proposé dans le langage ASL [Ley04]. La structure syntaxique de cet opérateur est la suivante :

```
new nom_meta-entité [named nom_entite]
            [initialised with {parameters}]
```

Notons que *nom_ entite* est optionnel. Voici maintenant la description du client en ASL :

```
Client is activeEntity with
{
attributes : {
numCommande is String default value is "num0001";
    nomClient is String default value is "Client1";
    refProduit is String default value is "ref001";
    quantité is Interger default value is 100;
    dateDeLivraison is String default value is "11dec2007";}
process :
{
 new passerCommande with
 {parameters : set(numCommande;nomClient; refProduit;
        quantité; dateDeLivraison)};
 new attendreReponse;
choose{
    done
    or
    behavior is {
         new recevoirLivraison;
         new payementFacture;
         done}
        }
}
}
```

Contraintes et Analyse

Une activeEntity va en premier lieu respecter les contraintes décrites par les taskEntity et les dataEntity. D'autres contraintes peuvent être décrites à ce niveau :

- *communicationForbidden* : cette contrainte impose que deux activeEntities ne doivent pas directement communiquer ensemble. Ceci veut dire que les activeEntities en question n'ont aucune unification de connexions.
- *communicationPrescribed* : cette contrainte impose que deux ou plusieurs activeEntities doivent communiquer ensemble. Au contraire de *communicationForbidden*, ces activeEntities doivent unifier leurs connexions.

7.4 structureEntity

La *structureEntity* peut être considérée comme l'élément composite auquel appartient différentes *activeEntities*, *taskEntities* et *dataEntities*. Dans notre exemple, il s'agit du système regroupant les deux entités client-fournisseur.

Ce système va représenter un maillon d'une *supply chain*. Cette structure est responsable de l'instanciation des deux entités actives. Au sein de cette structure, les entités actives tournent en parallèle. L'entité structurelle est responsable de la synchronisation des entités actives si celles-ci doivent communiquer. Rappelons que les langages ArchWare synchronisent les processus en unifiant les connexions sur lesquelles elles sont sensées communiquer. Ainsi, ***structureEntity*** contient deux éléments :

- l'ensemble des entités actives lui appartenant ;
- leur configuration qui consiste à l'unification de leurs connexions libres.

La formalisation de ***structureEntity*** est décrite ci-dessous :

```
structureEntity is constructor
    (_name: Alias;
     _activeEnties : set[ActiveEntity]
     _configuration : set[view[c1: connection[Any],
                  c2: connection[Any]]);
{
type structureEntity is abstraction[];
{
  value _Name is abstraction()
  { project _activeEnties as n and setActiveEntity;
   iterate  sequence(1..n) by i:Natural;
   from value composingElements is behaviour()
   accumulate
   { compose setActiveEntity::i
     and
     composingElements}
   where {_configuration::c1::i unifies
      _configuration::c2::i;}

}as $_name is structureEntity with
{
activeEntities : { $_activeEnties}
configuration : {$_configuration}
}
```

En utilisant notre système de client fournisseur sera modélisé comme suit :

```
clientFournisseur is structureEntity with
{
activeEntities : {new client; new fournisseur}
configuration : {set(
     view(client::passerCommande_connection;
          fournisseur::recevoirCommande_connection);
     view(client::recevoirNotification_connection;
          fournisseur::notifierCommande_connection);
```

```
                        ...          )}
}
```

Contraintes et Analyse

A ce niveau, les contraintes décrites au niveau des dataEntities, taskEntities et activeEntities doivent toutes être respectées. Il faut aussi vérifier que la vivacité et la sureté du système sont garanties. Il est nécessaire aussi que ce système ne contient pas d'interblocage. L'absence d' interblocage peut être vérifiée de la manière suivante :

- toutes les connexions sont unifiées (c'est-à-dire que chaque entité communiquante a un interlocuteur),
- les connexions unifiées transportent le même type de données,
- à une action d'envoi correspond une action de récéption.

8 Le métamodèle orienté agent

Ce métamodèle introduit les différents mécanismes appliqués au niveau des systèmes multi-agents indépendemment des domaines qu'ils sont en train de traiter. Nous avons vu au cours du chapitre 3 que plusieurs métamodèles sont proposés par les méthodologies et que plusieurs sémantiques peuvent correspondre à ces concepts. Nous ne pouvons à ce jour, affirmer qu'un tel métamodèle est meilleur qu'un autre. D'ailleurs, ceci ne constitue pas l'objet de nos travaux. Ainsi, il est important de donner une certaine flexibilité dans la construction du métamodèle orienté agent [AHO06]. Cette flexibilité implique qu'un expert de système multi-agents peut proposer une bibliothèque de styles décrivant différents concepts orientés agent, que l'architecte peut utiliser à sa guise lors du processus *d'agentification.*

La description des concepts ainsi que leurs relations se fait aussi en utilisant le langage ArchWare ASL qui permet de définir les constituants d'un concept, ses mécanismes de fonctionnement, sa manière de le construire (c'est-à-dire la manière de créer une architecture à partir d'un style), ses contraintes ainsi que ses analyses qui représentent les propriétés qu'une architecture doit respecter. Par ailleurs, nous avons vu que le style nous permettra de définir une syntaxe spécifique au domaine multi-agent qui peut facilement être utilisable par un non spécialiste ArchWare.

Il nous serait impossible d'énumérer tous les concepts orientés agent et de les formaliser avec le langage ArchWare ASL. Nous avons donné un exemple de métamodèle orienté agent dans la section 3.1. Au cours de cette section, nous allons formaliser une partie de ce métamodèle illustrant un agent réactif. Nous avons choisi ce type d'agent pour des raisons pédagogiques, étant donné que c'est le type d'agent le plus simple à mettre en œuvre. Une vue

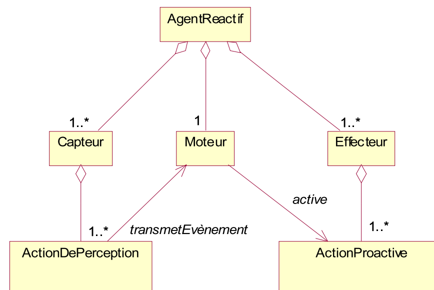


FIG. 5.12: Métamodèle d'Agent Réactif

plus détaillée sur l'agent réactif est décrite dans la figure 5.12.

Un agent réactif est composé de capteurs et d'effecteurs. Les capteurs contiennent des actions de perception permettant de percevoir les évènements survenus dans l'environnement (et plus particulièrement sur les ressources de l'environnement). Les effecteurs contiennent des actions proactives permettant de changer l'état de l'environnement. Les capteurs et les effecteurs sont liés au moteur interne de l'agent qui permet d'associer une action de perception à une action proactive.

A la suite de cette section, nous allons présenter comment le style d'un agent réactif est formalisé. Par la suite, nous présenterons la manière d'associer une syntaxe à un style architectural.

8.1 Formalisation du style agent réactif

Pour représenter le style de l'agent réactif tel qu'il est décrit dans la figure 5.12, nous profitons des mécanismes d'aggrégation fournis par le langage ASL qui nous permet de facilement décrire la structure d'un agent réactif :

```
agentReactif  is style
{
 capteur is style {...}
 moteur is style {...}
 effecteur is style {...}
}
```

Ce style peut décrire différentes sortes d'architectures d'agent réactif. Ainsi, une architecture peut limiter le nombre de capteurs utilisés à un seul capteur percevant un seul évènement ou au contraire à plusieurs capteurs percevant plusieurs sortes d'évènements. De même un effecteur peut agir sur une ou plusieurs ressources de l'environnement. Dans ASL, un style peut définir des constructeurs. Le constructeur introduit les mécanismes de fonc-

tionnement d'un concept. Concrètement, il définit le comportement qui sera généré au niveau de l'architecture. Par exemple, le constructeur suivant va prendre en paramètres les capteurs, moteurs, effecteurs et configuration. Il va composer (c'est-à-dire mettre en parallèle) ces différents éléments, ensuite configurer leurs liens en appliquant *bindings*.

Un constructeur d'un agent réactif contenant un capteur, un moteur et un effecteur va être décrit de la manière suivante :

```
reactiveAgent is constructor
( _ID : Alias,
 captor : captor.Instance
 engine : engine.Instance
effector : effector.Instance
binding : set [tuple[Expression[Connection],
                Expression[Connection]]])

{
 abstraction();{
 compose {
  captor()
  and
  engine()
  and
  effector()
 } where{
    iterate bindings by i: Natural do
    project bindings::i as c1, c2 do
    c1 unifies c2;
 }
}
}}
```

A l'application de ce style (en utilisant l'outil ASL TOOLKIT), une abstraction va être générée dont le comportement consiste à mettre en parallèle le capteur, le moteur et l'effecteur. Ces différents éléments sont eux-mêmes décrits par des styles ayant des constructeurs permettant ainsi de décrire le comportement de chaque entité.

8.2 Définition d'une syntaxe

Afin de faciliter la tâche de l'architecte durant le processus d'agentification, le styliste peut fournir une syntaxe liée à un constructeur. Cette syntaxe va faciliter l'utilisation du style par un utilisateur lambda. Voici un exemple de syntaxe pour un agent réactif :

```
$_ID is reactiveAgent with {
 captors {$captors}
```

```
 engine {$engine}
 effectors {$effector}
 bindings {$binding}
}
```

Quand il va utiliser cette syntaxe, l'architecte déclare un agent réactif en définissant son identifiant, sa liste de capteurs, sa liste d'effecteurs, son moteur et sa configuration que nous avons nommée ici *bindings*. Cette configuration permet de décrire le lien qui existe entre les actions de perception et les actions proactives. Au niveau de ASL, ce lien va être décrit par une synchronisation des connexions internes partagées d'une part entre les actions de perception contenues dans les capteurs et le moteur interne de l'agent et d'autre part entre le moteur et les actions proactives contenues dans les effecteurs. Rappelons que la synchronisation dans ArchWare ADL se fait en unifiant les connexions sur lesquelles la communication entre deux entités va s'effectuer. *Bindings* est donc un ensemble qui contient des tuples[7] de connexions allant être unifiées.

8.3 Propriétés d'un agent réactif

Nous pouvons définir plusieurs propriétés au niveau de l'agent réactif. Ces propriétés peuvent concerner la configuration de l'agent. Ainsi, l'agent réactif doit posséder au moins un capteur et un effecteur. Ceci est décrit comme suit :

```
Captor_and_effector_instances is constraint {
    to reactiveAgent.Instances apply{
    exists {c | c is captor.Instances }
    and
    exists {e | e is effector.Instances }
    }
```

Les seules connexions libres d'un agent réactif doivent appartenir soit à un capteur soit à un effecteur mais jamais au moteur de l'agent, étant donné que celui-ci ne peut communiquer avec l'extérieur qu'à travers son capteur ou son effecteur. Cette propriété va être exprimée comme suit :

```
Reactive_agent_connections is constraint {
    to reactiveAgent.Instance apply{
    forall{ c: free connection[Any]|
            (c isIn captor.Instances)
            or
            (c isIn effector.Instances)}
        }
}}
```

[7]un tuple est un type composé fourni par archWare ADL et qui permet de regrouper plusieurs valeurs (voir annexe A)

Après la spécification des styles du domaine et du style orienté agent, il s'agit maintenant de spécifier les règles de transformation qui vont nous permettre d'agentifier le domaine. Ceci est décrit dans la section suivante.

9 Le processus d'agentification

Au cours de ce processus, les styles définis dans le métamodèle du domaine et ceux définis dans le métamodèle orienté agent vont être combinés afin d'obtenir un style du domaine agentifié. Ce style va permettre de définir la syntaxe relative à la conception d'un domaine agentifié. C'est en utilisant ce style que le concepteur va générer l'architecture de son système. Le résultat de cette combinaison doit conserver les propriétés des deux précédents métamodèles. Pour expliquer notre démarche, nous nous appuyons sur les deux métamodèles définis dans les sections 7 et 8.

Nous avons spécifié dans le tableau 3.1 de ce chapitre quelques exemples de tissage entre les entités du métamodèle du domaine et les entités du métamodèle agent. Au cours de cette section, nous allons formaliser quelques règles de tissage en employant le langage ArchWare ARL. Cependant, avant d'utiliser ce langage des extensions sont nécessaires dans la mesure où ce langage permet uniquement le raffinement des expressions architecturales au niveau M1. De plus, celui-ci est basé sur un style composant/connecteur.

9.1 Extension du langage ARL

Afin d'avoir la possibilité d'utiliser le langage ARL dans notre contexte, deux possibités s'offrent à nous dans ce cas :

- utiliser le langage tel qu'il est, ce qui nous obligera à exprimer nos entités avec le style composant/connecteur,
- étendre le langage ARL afin de prendre en compte la syntaxe exprimée au niveau du métamodèle du domaine et du métamodèle orienté agent.

Nous pensons que la première méthode alourdira le processus d'agentification, dans la mesure où cela obligera l'architecte à bien connaître le fonctionnement du style composant/connecteur. Nous avons ainsi opté pour la deuxième solution qui consiste à étendre le langage ARL pour prendre en compte les nouvelles syntaxes. Cette solution est tout à fait faisable dans la mesure où les règles de transformation sont basées sur des opérateurs génériques qui peuvent s'appliquer à toutes les entités structurelles. Ces opérateurs sont :

- *becomes* : pour modifier une entité
- *includes* : pour introduire une nouvelle entité (exprimée avec une syntaxe spécifique
- *excludes* : pour supprimer un élément

Sur la base de ces éléments nous modifions la BNF du langage comme suit :

```
archetypeDefinition ::= archetype identifier is {
             ...
             [styleExpression]
             }

refAction ::= ...
       | StyleAction

StyleAction ::= includes styleExpression
         |  excludes styleExpression
         |  becomes styleExpression
```

Dans ce qui suit, nous allons utiliser cette extension du langage pour tisser les styles du domaine et les styles orientés agent.

9.2 Le tissage des métamodèles

Le choix du concept orienté agent qui va représenter l'entité du système se fait en fonction des propriétés qui ont été définies sur celles-ci. Par exemple, une dataEntity va être considérée comme une ressource de l'environnement de l'agent si elle est accessible par plusieurs activeEntity alors qu'elle va être considérée comme un état mental ou un attribut d'agent si elle est utilisée par une seule activeEntity. Ainsi pour transformer une dataEntity en ressource de l'environnement, la précondition suivante doit être vérifiée :

```
to data : dataEntity.Instances apply
{ forall{ a : activeEntity |
    iterate a by i: Natural do
    {if isVisible(data,a::i) do
     a::i includes activeEntitySet}
  and
 activeEntitySet.size > 1
}
```

Cette propriété parcourt les entités actives du système. Si celles-ci accèdent à la donnée considérée, elles sont incluses dans l'ensemble *activeEntitySet*. Si la taille de cet ensemble est strictement supérieure à 1, dataEntity peut être transformée en une ressource dans l'environnement grâce à l'opération suivante :

```
domainMetamodel::data includes agentifiedMM::Environment
```

Une taskEntity peut être exécutée par une seule entité active. Si l'entité active est représentée par un agent, celui-ci va intégrer l'entité active au

FIG. 5.13: Agentification d'une activeEntity en agentReactif

niveau de ses actions. Par ailleurs si la taskEntity est exécutée par plusieurs entités actives, celle-ci va intégrer une entité rôle.

Combinaison des syntaxes

Un métamodèle d'un domaine agentifié permettra de générer une syntaxe (qui pourrait être utilisée comme un DSL(Domain specific Language)) qui aidera à concevoir un domaine d'application selon les concepts agent. En utilisant ArchWare, cela reviendrait à combiner les notations mixfix spécifiées au niveau des styles contenus dans les deux métamodèles. Une combinaison de la syntaxe peut se faire en utilisant le langage ARL.

La figure 5.13 illustre l'agentification dune entité active. Ainsi, d'après ce patron de transformation, le flux d'entrée d'une entité active devient un flux d' évènements que les capteurs d'un agent réactif va intercepter. Le comportement de l'entitéActive qui permet d'interpréter ces évènements sera inclue dans le moteur de l'agent. Enfin, les flux en sortie seront envoyés par les effecteurs.

La règle de raffinement de cette figure fournit un patron d'agentification qui pourrait être utilisé par l'architecte. Un nouveau style décrivant l'entité formalisée sera généré à la suite de cette action. Le nouveau style conservera les propriétés définies dans le métamodèle du domaine mais aussi dans le métamodèle orienté agent.

Cependant, pour le moment cette opération bien que formalisée, sera effectuée manuellement par l'architecte. En effet, les outils implémentés dans le cadre du projet ArchWare (en particulier l'outil Refiner) ne permettent pas de prendre en compte le raffinement des styles. Nous soulèverons ce point dans les perspectives de cette thèse. Par ailleurs, notons que la formalisation de cette règle *"d'agentification"*, permet de conserver le savoir-faire.

10 Génération de l'architecture

Les outils fournis par ArchWare peuvent assister les utilisateurs lors de l'application de la démarche ArchMDE. Nous supposons à partir de ce point que les experts de métamodélisation ont fourni les métamodèles du domaine, et le métamodèle orienté agent. Le concepteur suite au processus d'agentification veut générer son architecture. Cette tâche se fait en utilisant ASL TOOLKIT (c.f. section 6). Cet outil prend en entrée une expression en ASL et génère l'architecture en ADL selon le processus décrit dans la figure 5.14.

Pour illustrer la génération de l'architecture, reprenons l'exemple de l'agent réactif représentant un service d'approvisionnement. Celui-ci reçoit un besoin et lance une commande. Le capteur de cet agent va recevoir un évènement indiquant qu'il y a besoin de se réapprovisionner. Dans la figure 5.15 nous nous concentrons sur le style du capteur. La première case contient la syntaxe qui va être utilisée par le concepteur. Cette syntaxe est produite suite au processus d'agentification. Selon le mécanisme du langage ASL, cette syntaxe renvoie à un constructeur de style, lequel décrit la manière dont l'architecture doit être décrite. Le compilateur de ASL Toolkit va interpréster le constructeur et générer l'architecture ADL correspondante.

11 Validation des propriétés

La validation des propriétés est effectuée suite à la génération de l'architecture. En effet, il n'existe pas encore d'outil supportant la validation des propriétés au niveau du style architectural. Les propriétés qui sont exprimées au niveau du style du domaine agentifié, sont regroupées dans un fichier *".aal"*. L'outil Analyzer prend en entrée ce fichier avec celui qui est généré par l'outil ASL TOOLKIT et lance le processus de vérification des propriétés (c.f. figure 5.16).

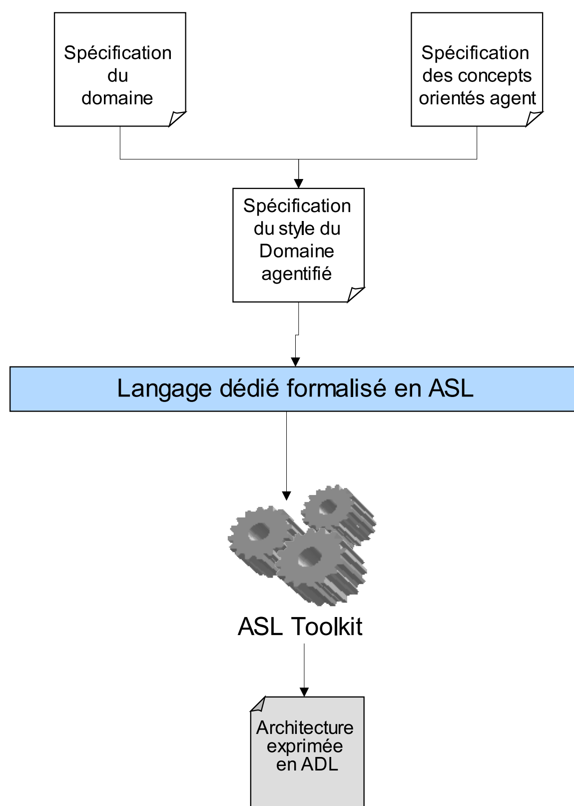


FIG. 5.14: Processus de génération d'architecture

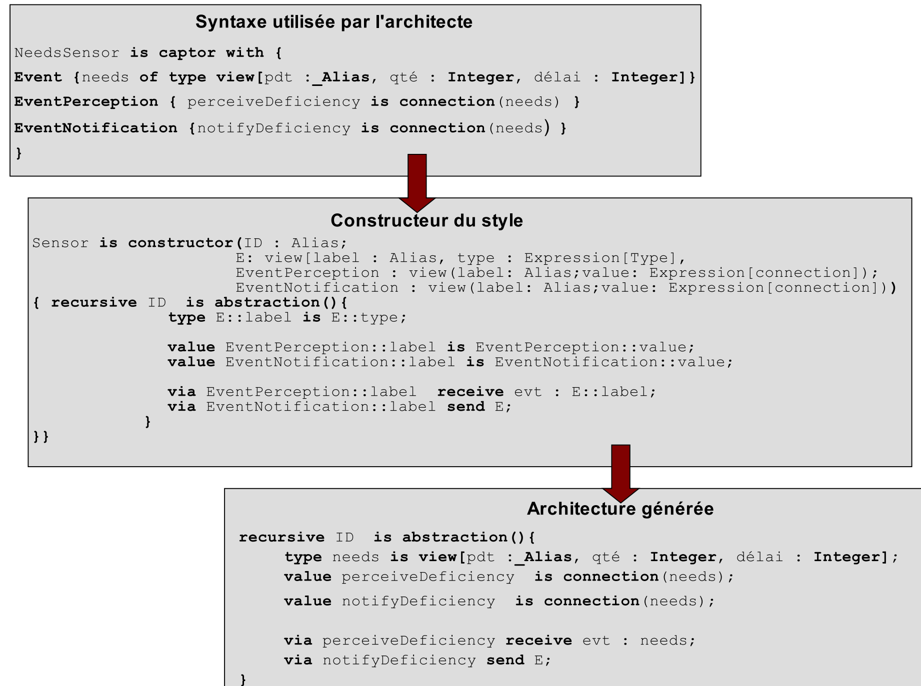


FIG. 5.15: Génération d'un capteur dans un agent réactif

fichier.adl correspond au fichier contenant la description architecturale générée par l'outil ASL TOOLKIT. Ce fichier va d'abord être compilé ensuite analysé sémantiquement. Suite à cette analyse, l'architecture est réifiée sous forme d'arbre sémantique représenté par des prédicats Prolog. Cet arbre va être par la suite parcouru pour vérifier les propriétés fournies dans *fichier-Prop.aal*. Celles-ci vont être aussi compilées afin de vérifier qu'elles respectent la syntaxe du langage AAL. Par la suite, les prédicats Prolog correspondant vont être activés afin de parcourir l'arbre de l'architecture. La figure 5.17 montre une capture d'écran de l'Analyseur. Dans cet exemple, nous vérifions que l'architecture du capteur décrit un comportement cyclique, ce qui revient à vérifier que dans cette architecture l'abstraction est récursive.

Nous donnons un autre exemple où nous vérifions cette fois-ci, si cette architecture contient un interblocage. Pour vérifier l'interblocage, plusieurs conditions doivent être validées :

- toutes les connexions déclarées et utilisées dans l'architecture doivent être unifiées,
- à chaque action d'envoi correspond une action de réception,
- les connexions unifiées transportent le même type de données.

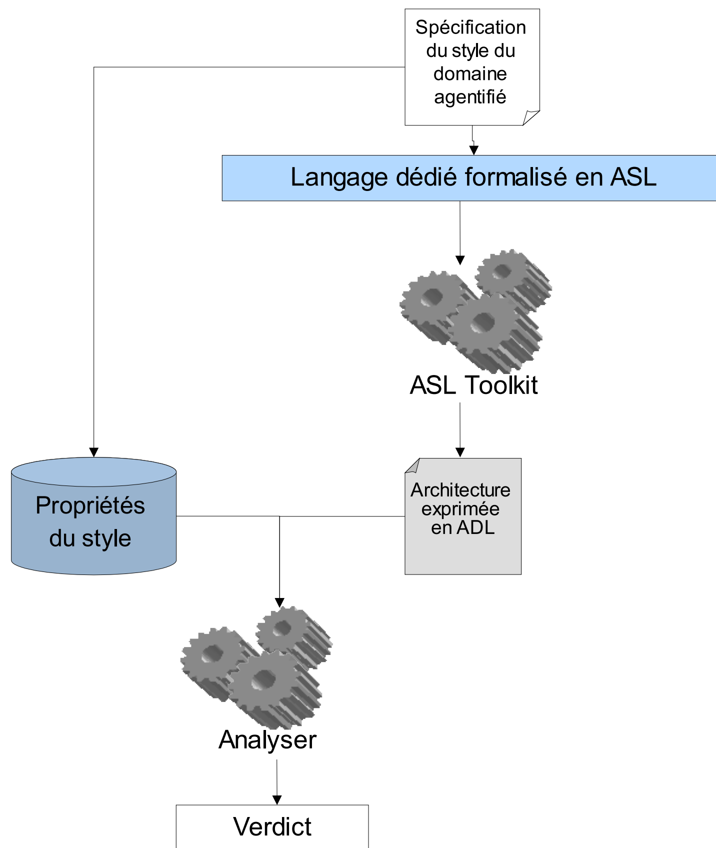


Fig. 5.16: Vérification des propriétés

```
C:\WINDOWS\system32\cmd.exe
E:\ArchWare\sauvegarde1\AnalyserProlog>..\XSB\config\x86-pc-windows\bin\xsb
[xsb_configuration loaded]
[sysinitrc loaded]
[packaging loaded]

XSB Version 2.6 (Duff) of June 24, 2003
[x86-pc-windows; mode: optimal; engine: slg-wam; gc: indirection; scheduling: lo
cal]

| ?- [checker].
[checker loaded]
[parser_dcg_adl loaded]
[parser_dcg_aal loaded]
[tools loaded]
[reification loaded]
[properties_analyser loaded]
[FunctionsLibrary loaded]

yes
| ?- checker('sensor.adl', 'sensor.aal').


deadlockChecking : property violation !!!
```

FIG. 5.17: Vérification de l'interblocage

Si nous prenons le capteur comme élément à part, cette propriétés ne va pas être respectée comme l'indique la figure 5.17. En effet, les connexions utilisées dans le capteur, ne sont pas encore unifiées. Cette situation va être réparée quand le capteur sera intégré dans l'agent (où sa connexion interne sera unifiée avec le moteur de l'agent) et que l'agent sera intégré dans le groupe d'agent correspondant (où la connexion externe de l'agent sera unifiée avec l'élément qui va envoyer l'évènement à l'agent).

12 Génération du code

Cet outil permet de générer un fichier XML interprétable à partir duquel des règles de génération du code peuvent être appliquées. Pour ce faire, le générateur de code s'appuie sur des règles de transformation « Langage ArchWare ADL - Langage cible » pour générer le fichier XML. Nous n'avons pu tester cet outil, étant donné qu'il était implanté par notre partenaire (University of St Andrews) et qu'il n'était pas en notre possession. Nous fournissons cepedant dans le tableau 12 les règles de mapping qui ont été utiliseés pour générer du code Java à partir d'une spécification en π-ADL [BFDP05].

13 Conclusion

Dans ce chapitre, nous avons présenté notre approche ArchMDE qui est basée sur un processus de développement en double Y. L'utilisation de l'approche ArchMDE se situe à deux niveaux. Le premier niveau concerne la

ADL	JAVA
file_identifier.adl	file_identifier.java
value identifier is abstraction(...) ;	public class file_identifier extends ProcessCreator public static void main(String[] args) new file_identifier() ; public void createProcess() throws Exception setProcessName("file_identifier") ; setAutoInstanciated() ; ...
value identifier is free connection(...) ; via identifier send ... ;	WfVertexDef identifier = setActivity(null, "identifier", outcome+"identifier.x true, null, prefill+"identifier.xml", false, 0, null) ;
value identifier is free connection(...) ; via identifier send ... ; via identifier receive ... ;	WfVertexDef identifier = setActivity(null, "identifier", outcome+"identifier.xsd", true, null, prefill+"identifier.xml", false, 0, null) ;

Tab. 5.2: Mapping entre architecture en π-ADL et code Java

construction du cadre de développement qui va être utilisé par les développeurs des applications multi-agents. Ce niveau est réalisé par des experts en métamodélisation. A ce niveau, durant la phase d'analyse, les méta-entités permettant de décrire le domaine doivent être formalisées et leurs propriétés exprimées indépendemment des concepts orientés agent. Durant la phase de conception, le métamodèle orienté agent est formalisé et les différentes propriétés auxquelles obéissent les concepts orientés agent exprimées. Par la suite, les règles de transformation sont exprimées afin de *tisser* les concepts du domaine aux concepts orientés agent.

Le deuxième niveau d'utilisation de ArchMDE concerne les développeurs des applications qui ne sont pas experts en métamodélisation. Ceux-ci vont utiliser le métamodèle du domaine pour spécifier le domaine d'application. Par la suite ils utilisent les règles de transformation fournies par le cadre de développement afin *d'agentifier* leur domaine. A la fin du processus d'agentification, nous obtenons une architecture abstraite du système que l'utilisateur va pouvoir configurer pour générer son architecture concrète. L'architecture du système doit respecter aussi bien les propriétés du domaine que les propriétés des concepts orientés agent.

La phase d'implémentation traite le mapping entre les plates-formes d'implémentation et l'architecture du système. Il s'agit durant cette phase d'énumérer les entités de la plate-forme et d'étudier leurs propriétés. Par la suite, des règles de mapping doivent être explicitées afin de générer automatiquement le code.

Afin de construire la cadre de développement basé sur l'approche ArchMDE, nous avons choisi d'utiliser l'environnement ArchWare. En utilisant le langage de style de ArchWare nous avons introduit un métamodèle de domaine basé sur quatre entités :

- dataEntity : qui représente les données
- taskEntity : qui représente des comportements atomiques devant être réalisés dans le système
- activeEntity : qui représente les entités actives du système
- structureEntity : qui représente les entité structurelles regroupant différentes données et différentes entités actives du système

Le langage de style ASL offre un mécanisme de mixfix qui permet de définir des DSL pouvant être utilisés par des personnes non spécialistes des langages ArchWare. Lors du processus d'agentification ces syntaxes vont être mixées avec celles qui ont été spécifiées pour les concepts orientés agent. Les règles de transformation du processus d'agentification sont exprimées en utilisant le langage ARL qui a été étendu afin de permettre le tissage des styles.

A partir de la syntaxe représentant le mix des deux syntaxes l'architec-

ture concrète va être générée en utilisant l'outil ASL TOOLKIT. Après la génération de l'architecture, les propriétés peuvent être vérifiées en utilisant l'outil Analyser. Dans le prochain chapitre, nous validons l'application de notre approche ArchMDE dans le cadre d'une agentification d'un système de production.

Chapitre 6

Validation de l'approche ArchMDE - Agentification d'un système de production

1 Présentation du projet

Nous avons appliqué notre approche dans le cadre d'un projet BQR regroupant différents membres du laboratoire LISTIC travaillant sur des thématiques différentes [HHP+07]. Ce projet concerne l'étude d'une nouvelle approche pour la modélisation, la simulation et le pilotage des systèmes de production (notés SdP). Nous avons appliqué dans ce projet l'approche orientée agent afin de procurer plus de flexibilité aux plates-formes de simulation actuelles [Bak96], [Ber00], [Hab01]. En effet, pour faire face aux changements brusques de leurs environnements et de la demande des clients, les entreprises manufacturières doivent piloter avec beaucoup d'agilité leur outil de production. Or, les SdPs tels qu'ils sont simulés actuellement se concentrent au niveau du flux physique en précisant le processus de fabrication et les spécificités des machines utilisées pour la réalisation des opérations. Cela a pour conséquence de rendre difficile la simulation car il est nécessaire pour l'utilisateur de trouver après plusieurs essais les bons réglages pour obtenir un résultat acceptable (nombre des machines, caractéristiques des machines, taille des lots, capacité des stocks, etc.). Il apparaît clairement que la simulation des SdPs manque d'automatisation et qu'il serait préférable de laisser le système évoluer de lui-même pour aboutir à une solution par émergence. Dans ce contexte, l'utilisation de l'approche orientée agent est bien appropriée. En effet, l'utilisation des systèmes multi-agents permettra au simulateur de faire évoluer la configuration du système de production en prenant en compte les changements survenus dans son environnement.

Au cours de ce projet, nous nous sommes basés sur des concepts utilisés

par l'équipe Simulation avec laquelle nous avons collaboré. Il a été question dans ce projet de concevoir ces concepts selon l'approche orientée agent afin d'apporter plus de flexibilité dans leur utilisation. Le but étant de créer un simulateur qui satisfasse des besoins de plusieurs ordres :

- créer un cadre logiciel modulaire et évolutif : le simulateur est conçu de manière à pouvoir être utilisé dans de nombreux cas, tout en ne nécessitant que des modifications mineures.
- créer un outil puissant pour le théoricien en lui facilitant la compréhension des phénomènes complexes. En effet, le programme permet par une simple modification, via une interface graphique, de tester de nombreuses hypothèses et, ainsi, de comprendre les influences des paramètres.
- permettre une analyse complète et détaillée : des fichiers comprenant les informations sur l'état des agents, les échanges réalisés, etc. Ces informations sont générées par le simulateur afin de permettre une étude ultérieure.

Pour développer ce simulateur, nous avons appliqué notre démarche basée sur l'approche ArchMDE. Dans ce contexte, nous avons utilisé le métamodèle du domaine que nous avons fourni dans le chapitre précédent (cf. section 7). Par la suite, nous avons appliqué des règles d'agentification afin de générer un style de systèmes de production agentifiés. Nous voulons focaliser sur ce point au cours de ce chapitre.

Ainsi, dans la section 2 de ce chapitre, nous décrivons de manière plus détaillée les systèmes de production. Par la suite, nous présentons les concepts du domaine que nous formalisons en utilisant l'environnement ArchWare (section 3). Nous expliquons ensuite l'agentification des concepts SdP (section 4).

2 Les systèmes de production

Un SdP peut être défini comme étant une association dynamique de trois parties :

- la première partie constituée par un flux de matières ou d'entités à transformer ou à assembler et qui représentent généralement matières premières, composants, produits en cours, pièces de rechange et produits finis. Le flux formé par ces éléments peut se trouver sous différentes formes : des pièces unitaires, des groupements temporaires, des produits assemblés, des lots, des séries, etc.
- la deuxième partie physique qui représente l'ensemble des moyens nécessaires à la réalisation des opérations sur le flux de matière. Elle

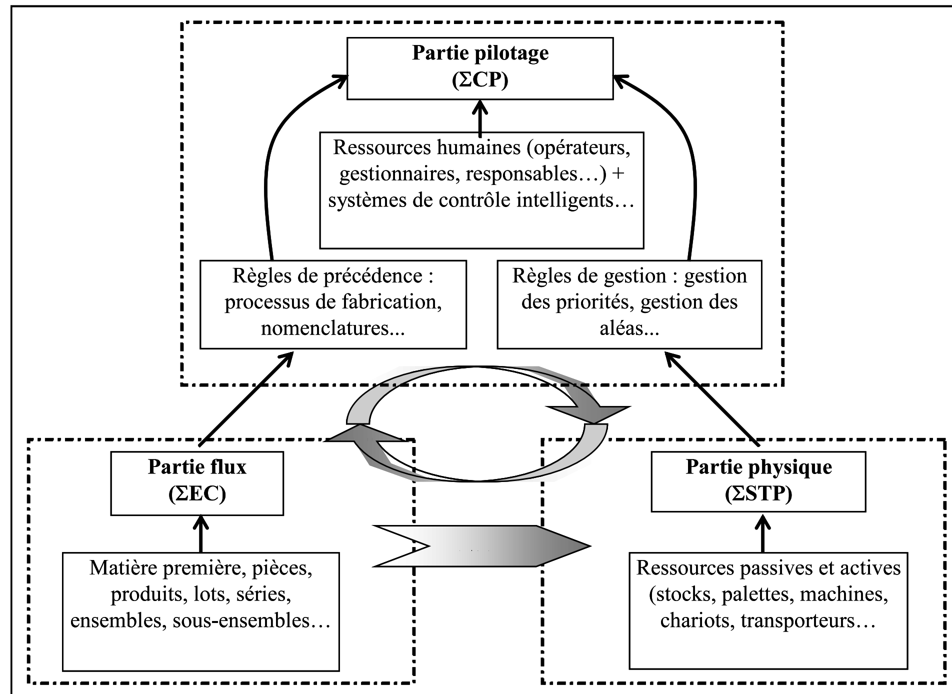


Fig. 6.1: Modèle de Système de Production

comprend les machines, les espaces de stockage, les moyens de transfert, les robots, les manipulateurs, etc.

– la partie contrôle ou pilotage qui représente la partie intelligente du système c'est-à-dire les systèmes de commande, les systèmes de supervision, les acteurs humains (opérateurs, responsables, gestionnaires, etc.). La partie pilotage associe des points d'acquisition et de collecte de l'information, des processus décisionnels (analyse, traitement de l'information et évaluation pour générer des décisions) et des actions (points de transfert des ordres ou des actions de la partie pilotage à la partie physique).

Ces trois parties sont illustrées dans la figure 6.1. Dans cette figure la partie physique reçoit des données à partir de la partie flux. La partie pilotage agit aussi bien sur la partie physique que la partie flux qui lui transmettent les informations nécessaires. A partir de ce modèle générique, plusieurs concepts ont été identifiés. Nous décrivons ces concepts dans la section suivante.

3 Métamodélisation d'un SdP

Une analyse plus approfondie des trois parties constituant un SdP a permis de constater que les éléments du système peuvent être divisés en trois types structurellement et fonctionnellement différents. Il s'agit de :

- l'Entité Circulante (EC),
- le Système de Transformation du Produit (STP),
- le Centre de Pilotage (CP).

Pour chacune de ces entités, nous décrivons leurs rôles dans le système ainsi que la manière dont elles ont été formalisées en utilisant l'environnement ArchWare.

3.1 L'Entité Circulante (EC)

Le Rôle de l'EC

L'EC est un concept générique modélisant le flux du produit. Il possède une trajectoire définie *a priori*, il est passif vis-à-vis de lui-même mais il active les STPs et les CPs à son passage et déclenche leurs comportements.

Formalisation du métamodèle de l'EC

Nous considérons l'EC comme une *dataEntity*. En effet, bien que ce concept représente le fait que ce soit une entité qui "circule" dans le SdP, elle n'a pour autant pas d'actions à exécuter. Ce sont les éléments du SdP chargés de sa transformation qui auront un rôle actif sur celle-ci. L'EC sert donc à regrouper les informations concernant le flux de production. Les informations principales qui sont contenues dans l'EC sont les suivantes :

- un identifiant unique,
- le processus : qui est un ensemble d'opérations devant être effectuées sur l'EC. Les opérations sont décrites par l'identifiant du poste qui va les exécuter et le temps opératoire.

En se basant sur ces informations, nous formalisons l'entité EC de la manière suivante :

```
EC is dataEntity with {
type is {
view[ID: Alias;
processus: set[view[numPoste: Alias;tpsOperatoire: Real]]
}
}
```

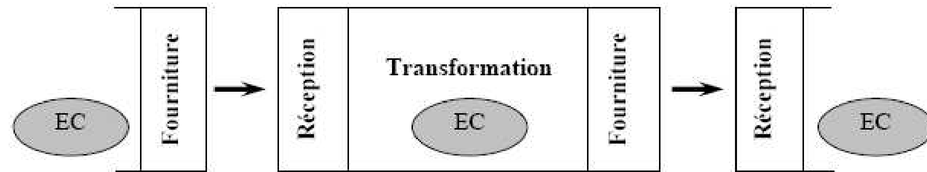


Fig. 6.2: Les trois fonctions fondamentales d'un STP

3.2 Le Système de Transformation du Produit (STP)

Le Rôle du STP

Le STP est un concept générique possédant toutes les caractéristiques structurelles et fonctionnelles d'une ressource de production. Le STP exécute trois opérations (cf. figure 6.2) :
- réception : obtention d'une ou plusieurs ECs à transformer,
- transformation : occupation du STP et blocage de l'EC pendant le temps de transformation,
- fourniture : libération du STP et fourniture de l'EC transformée à un autre STP consécutif.

Le STP est une structure récursive qui est principalement utilisé pour décrire une ressource de production ajoutant de la valeur au produit en cours de fabrication. Toutefois, le concept de STP peut aussi représenter d'autres ressources de production telles que : les stocks et les moyens de transfert. En effet, ceux-ci sont interprétés comme des STPs particuliers qui n'effectuent pas d'opérations de transformation, mais qui réceptionnent et fournissent les ECs après un temps d'attente souvent nécessaire pour réguler le flux. Parfois, il est nécessaire de représenter un groupe de machines fonctionnellement identiques par des STPs différents (différence entre les performances, les défaillances, les réglages, etc.). Dans, ce cas nous utilisons le concept de Centre de Charge (CC) qui est un groupement de plusieurs STPs différents mais réalisant la même fonction ou opération.

Formalisation du métamodèle du STP

L'EC est manipulée par les STPs que nous considérons comme des *activeEntities*. Celles-ci réalisent différentes opérations que nous considérons comme des *taskEntities* et qui sont la réception des EC, la mise à jour de leurs données, la mise à jour des indicateurs et la fourniture des ECs transformées. Nous avons choisi de différencier, dans notre métamodèle entre le STP-Machine qui va réaliser des transformations sur l'EC et le STP-Stock qui va recevoir puis fournir les ECs sans opérer de changements sur les don-

nées qu'elles contiennent. En effet, en modélisant le comportement de ces deux entités, nous avons relevé plusieurs disparités, ce qui nous a poussé à distinguer les deux types en modélisant chacun par un style particulier. Ainsi, le STP-Stock est représenté comme suit :

```
STP–Stock is activeEntity with {
 attributes { ID : Alias;
         capacité : Integer;
         niveauStock : Integer;
         ECStockés : set[EC.Instance]
                         }
 process {
         recursive Process{
         compose{
         new STPStockRecevoitEC
         and
         new stockerEC
         and
         new     STPStockFournitEC}

         where{
         STPStockRecevoitEC::transmettreECrecue unifies
         stockerEC::stockerECrecu
         and
         stockerEC::sendECstoquee unifies
         STPStockFournitEC::preparerECaFournir
                 }
 }}
 analysis{
 constraints{
         –– contrainte 1
         to STP–Stock.Instances apply
         {not(STP–Stock.capacité < STP–Stock.niveauStock)}
         and
         –– contrainte 2
         to STP–Stock.Instances apply
         {(successionDirecte(stockerEC,fournirEC)
          or
          (successionIndirecte(stockerEC,fournirEC)}
         –– contrainte 3
         to STP–Stock.Instances apply
         {hasCycle(STP–Stock)}
 ...
 }
```

Les trois *taskEntities* STPStockRecevoitEC, stoquerEC et STPStock-fournitEC qui sont composées par l'entité active *STP-Stock* sont des instanciations de style de type *taskEntity*. L'entité active *STP-Stock* permet alors

de spécifier comment ces tâches sont exécutées à son niveau et comment elles sont reliées à travers des connexions partagées qui ont été unifiées au niveau de *STP-Stock*. Ces taskEntities agissent alors comme des modules internes de *STP-Stock* qui sont indépendants mais qui partagent les attributs déclarés au niveau de *activeEntity*.

Dans ce métamodèle formalisé par un style, nous avons fournis quelques exemples de contraintes que la STP-Stock doit satisfaire. Ainsi, la contrainte 1 stipule que le niveauStock ne doit jamais dépasser la capacité. La contrainte 2 veut que le stockage d'un EC peut être directement suivi par sa fourniture (cas d'un flux tendu) ou que son stokage peut être suivi par la réception d'autres instances d'EC. Enfin, la contrainte 3 veut que le comportement du STP-Stock soit récursif.

Dans ce qui suit, nous focalisons sur la métamodélisation des *taskEntities* qui ont été utilisées au niveau du style STP-Stock. Une description détaillée des trois opérations STPStockRecevoitEC, stoquerEC et STPStockfournitEC est fournie. Le comportement de chacune de ces opérations est spécifiée au niveau de leur constructeur.

```
STPStockRecevoitEC is taskEntity{
input : { readCapacité : connection[capacité];
        readNiveauStock : connection[niveauStock]}
operation : {
recursive behaviour {

        value readCapacité is connection(capacité.Instance);
        value readNiveauStock is connection(niveauStock);

        -- récupérer les attributs capacité et niveauStock
        via readCapacité receive c : capacité;
        via readNiveauStock receive n : niveauStock;

        if capacité < niveauStock then

        -- la capacité du stock n'est pas atteinte
        -- STP-Stock reçoit l'EC
        via recevoirEC receive EC : EC.Instances;
        via transmettreECrecue send EC;
        new STPStockRecevoitEC

        -- la capacité du stock n'est pas atteinte
        -- STP-Stock refuse l'EC
        else
        via refusReception send;
```

```
        new STPStockRecevoitEC
}
output : {}
}


stoquerEC is taskEntity {
input : {lockECStockés : connection[ECStockés];
                 lockNiveauStock : connection[niveauStock]}
operation : {
        value lockECStockés is connection(ECStockés);
        value lockNiveauStock is connection(niveauStock);

        behavior{

        -- Le module STPStockRecevoitEC notifie
        -- l'arrivée d'une EC

                via stockerECrecu receive EC : EC.Instances;


        -- Accès à l'attribut ECStockés (formalise la liste)
        -- des ECs contenues dans le stock

        -- vérouiller ECStockés
                via lockECStockés receive ECS : ECStockés;

        -- modifier ECStockés en ajoutant l'EC reçue
                ECS includes EC;

        -- dévérouiller ECStockés
                via unlockECStockés send ECS : ECStockés;

        -- Accès à l'attribut niveauStock
        -- vérouiller niveauStock
                via lockNiveauStock receive niveauStock;

        -- modifier niveauStock en l'incrémentant
                niveauStock := niveauStock +1;

        -- dévérouiller niveauStock
                via unlockNiveauStock receive niveauStock;
                new stoquerEC
                }
                new stoquerEC
        }
}
output : { unlockECStockés : connection[ECStockés]
                unlockNiveauStock : connection[NiveauStock]}
```

```
}


STPStockfournitEC is taskEntity {
input : {lockECStockés : connection[ECStockés];
         recevoirParamNiveau receive n : niveauStock}
operation :{

        fournirEC is recursive behaviour{

          value lockECStockés is connection(ECStockés);
          value lockNiveauStock is connection(niveauStock);

      -- vérouiller les attributs ECStockés et NiveauStock
          via lockECStockés receive ECS : ECStockés;
          via lockNiveauStock receive n : NiveauStock;

          choose {
          envoyerEC is behaviour{
          -- Envoyer l'EC stoquée
          --(si le STP suivant est prêt à recevoir)
          via sendECaFournir send ECStockés::1;
          ECStockés excludes ECStockés::1;
          niveauStock := niveauStock - 1;
          via unlockECStockés send ECS;
          via unlockNiveauStock send n;
          }
          or
         -- débloquer attributs
         -- (si le STP suivant est prêt à recevoir)
          via unlockECStockés send ECS;
          via unlockNiveauStock send n;
          }

         STPStockfournitEC;
               or
         fournirEC
        }
        }
output{ unlockECStockés : connection[ECStockés]
        unlockNiveauStock : connection[NiveauStock]}
}
```

Ces trois opérations bien qu'elles tournent en parallèle au niveau du STP-Stock communiquent entre elles : quand le module réception perçoit l'arrivée de l'EC, il vérifie que le niveau du stock actuel n'est pas atteint. Si c'est le cas, il accepte l'EC et la transmet au module de stockage (à travers la connexion *transmettreECrecue*). Si la capacité est atteinte, il envoie un refus de réception.

L'opération *stockerEC* reçoit l'EC et l'inclue dans la liste des ECs reçus. Suite à cette opération, le niveau courant du stock est incrémenté.

Le module *fournirEC*prépare l'EC qui va être envoyée. Si le STP suivant est prêt à la recevoir, celui-ci la lui fournit et remet à jour la liste des EC stockées sinon il relache les valeurs vérouillées et reboucle sur lui-même.

Nous remarquons que ces trois modules ont un accès concurrent aux différents attributs fournis par l'entité active. Pour garantir la cohérence de ces attributs, nous avons spécifié des sentinelles sur ces attributs. Ainsi quand l'un des modules essaie d'accéder à une ressource pour la modifier, il l'acquiert d'abord en utilisant la connexion *lockNomAttribut*, une fois la ressource modifiée, le module la relache en utilisant la connexion *unlockNomAttribut*. Avant que l'opération ***via*** *unlockNomAttribut* ***send*** ne soit effectuée, l'attribut devient inaccessible. Cependant, notons que les opérations de lecture peuvent être effectuées en parallèle en employant la connexion *readNomAttribut*.

STP-Machine est formalisé selon le même principe. Cependant plusieurs différences existent avec le comportement STP-Stock. Le STP-Machine est responsable d'opérer des changements sur l'EC. Ainsi, STP-Stock et STP-Machine se différencient au niveau des *taskEntity* qu'ils vont inclure : stockerEC et transformerEC. Bien que les opérations de réception et de fourniture d'EC peuvent paraître identiques, celles-ci exhibent un comportement interne différent. Tandis que STP-Stock va vérifier si sa capacité est atteinte ou pas (ce qui lui permet ou pas de recevoir l'EC), STP-Machine va vérifier son état. En effet, divers états indiquent si la machine est disponible, en panne, en maintenance, prévue pour un changement de série etc. Les changements d'états peuvent être programmés en avance au niveau du STP-Machine. Par exemple, à n'importe quel moment une panne peut survenir. Si une panne intervient alors qu'EC est en train d'être transformée, celle-ci sera détruite et comptabilisée comme un rebus. Enfin, le STP-machine diffère du STP-Stock par les indicateurs qu'il manipule. Ces indicateurs peuvent être par exemple les taux de rebus, les taux de pannes etc. Suivant ces caractéristiques, STP-Machine est représenté comme suit :

```
STP–Machine is activeEntity with {
attributes {
        ID : Alias;
        Etats : union[panne, changementDeSérie,
                        diponible, maintenance];
        EtatCourant : quote[Etats];
        indicateurs : set[Indicateur]
                        }
```

```
process {
        recursive behaviour{
        compose{
        new stpMachineRecoitEC
        and
        new transformerEC
        and
        new detruireEC
        and
        new changerEtat
        and
        new stpMachinefournitEC}
...
}
```

Les styles modélisés au niveau des *taskEntities* de STP-Machine respectent le même principe que ceux modélisés au niveau du STP-Stock.

3.3 Le Centre de Pilotage

Le Rôle du CP

Le CP a été défini par [Ber00] comme suit :

> " une structure organisée et autonome, dépendante de la stratégie de l'entreprise, ayant un pouvoir décisionnel, associée à une entité à piloter et disposant d'un ensemble de moyens nécessaires à la mise en place d'actions pour atteindre un ou plusieurs objectifs définis dans le cadre global de l'entreprise ".

Le CP peut intervenir à différents niveaux de décision dans l'atelier de production : de celui qui dirige un STP jusqu'au CP ayant un pouvoir décisionnel et stratégique vis-à-vis du développement de l'entreprise. Les activités de supervision et de pilotage de l'agent-CP sont de trois natures :

– l'optimisation : d'abord, l'agent-CP a une activité d'optimisation où le pilotage défini a priori. Dès l'instant initial, avant exécution de la simulation, l'agent-CP doit être capable de construire d'une part ses propres objectifs ainsi que le domaine d'atteignabilité de ces objectifs, et d'autre part, les plans d'action à exécuter en cas de dérive du système,
– la prévention : pendant la simulation, l'agent-CP doit avoir une capacité de raisonnement lui permettant d'enclencher des activités de prévention proactives. En effet, il doit être capable d'agir sur le système, si nécessaire, avant même qu'une dérive puisse avoir lieu sur un élément du système,

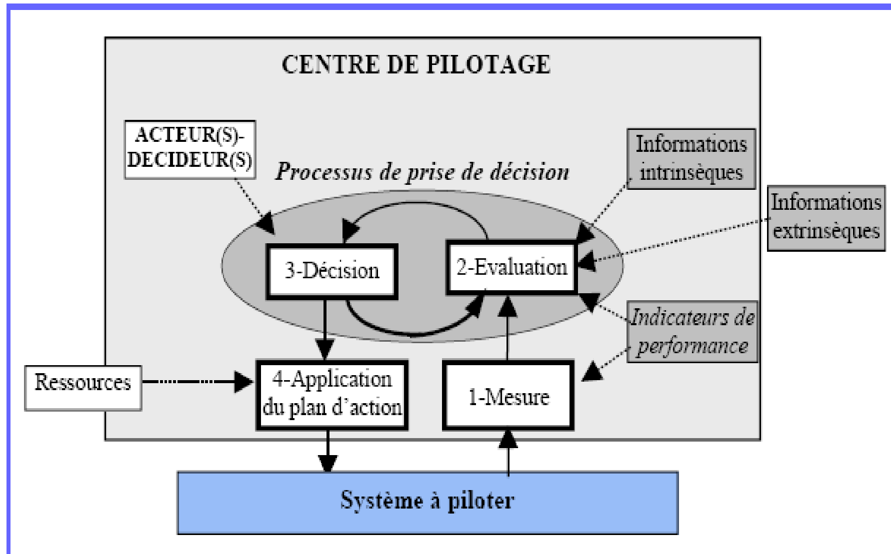


Fig. 6.3: Modèle du CP

- la correction : enfin, en cas de dérive d'un élément du système, l'agent-CP doit enclencher le plan d'actions nécessaire pour rétablir l'état du système.

Les principales étapes du processus de pilotage du CP sont les suivantes :

- acquisition de l'information d'un STP ou d'un CP,
- traitement et évaluation de l'information,
- prise de décision,
- choix d'un plan d'action adapté et transfert de ce dernier aux ressources concernées.

Le schéma de la figure 6.3 présente de manière simplifiée les 4 étapes du processus de pilotage du CP. Ces étapes concernent : la mesure, l'évaluation, la décision et l'application du plan. Le processus de décision se fait au niveau des étapes 2 et 3 (évaluation et décision). Il prend en compte :

- les informations intrinsèques (endogènes) qui sont propres au CP telles que les référents, les objectifs locaux et les règles de conduite du CP,
- les informations extrinsèques (exogènes) qui sont les informations externes au CP et qui sont liées à l'environnement, à la structure et aux autres CPs,
- les mesures liées au système à piloter et qui sont traduites sous forme d'indicateurs de performance.

La métamodélisation du CP

Pour métamodéliser le CP, nous nous sommes basés sur la figure 6.4. Cette figure décrit de manière détaillée les opérations à effectuer au niveau du CP. Ainsi, le processus de pilotage du CP comporte les 7 étapes suivantes :

- prise de mesure : lors de cette étape, il s'agit d'acquérir la mesure qui peut être soit évènementielle soit périodique. La mesure est fournie par un STP, une EC ou un CP de niveau inférieur. Cette mesure concerne l'indicateur principal et les indicateurs secondaires associés aux inducteurs qu'un CP doit suivre.
- évaluation de la performance du système piloté : l'évaluation de la performance du système piloté se fait en deux étapes : d'abord, le calcul de la performance (comparaison de la mesure avec l'objectif), et ensuite l'évaluation de cette performance (analyse de cette comparaison pour déduire un état de dérive ou de non dérive du système). Dans le cas d'une dérive, le processus de pilotage est poursuivi pour identifier l'inducteur interne responsable. Dans le cas de non dérive, le comportement du système étant celui qui est attendu, le processus de pilotage est stoppé.
- identification de l'inducteur interne : l'identification de l'inducteur est essentielle pour construire un système de pilotage. Il s'agit de déterminer les ressources critiques dans le système piloté et les principaux facteurs qui agissent sur la performance de ces ressources (leurs inducteurs).
- évaluation de l'inducteur interne : cette étape consiste à mettre en évidence, parmi la liste des inducteurs, ceux qui sont responsables de la dérive de l'indicateur principal suivi du CP. Les inducteurs sont évalués grâce aux indicateurs de performance secondaires.
- identification du plan d'action : une fois que l'inducteur interne est identifié et évalué, le CP cherchera à identifier la liste des actions correspondant à cet inducteur et pouvant améliorer la situation.
- simulation du plan d'action : l'objectif de cette étape est de tester les actions à l'aide d'un historique des actions mises en place dans le passé et de leur influence sur le système.
- application du plan d'action : si l'action évaluée a permis dans le passé de rétablir la situation de non dérive, elle sera appliquée sur le système piloté.

Sur la base de ces informations, nous formalisons le CP de la manière suivante :

```
CP is activeEntity with {
attributes {
```

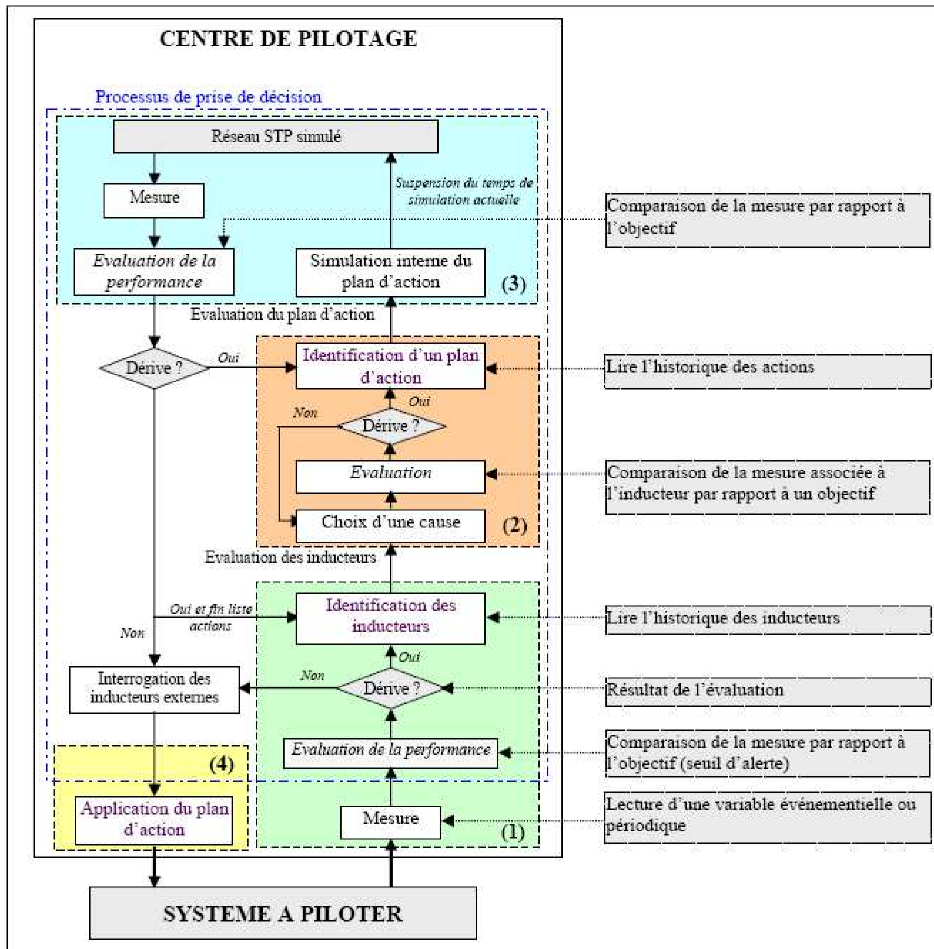


FIG. 6.4: Le comportement du CP

```
  ID : Alias;
  entitésPilotés : set[tuple[EC, STP-Stock, STP-Machine]]
  indicateursDePerformance : set[Indicateur];
  inducteursInternes : set[Inducteur];
  planActions : set [ProcessusCP];
  historique : set[view[actionPassées: set[Action],
                          résultatObtenu : Any]]
  objectifs : set[Any]
                          }
process {
        recursive behaviour{
        compose{
        new Mesure
        and
        new EvaluationDePerformance
        and
    new MesureDerive
        and
        new InterrogationInducteursExternes
        and
        new identificationPlanAction
        and
        new simulationPlanAction
        and
        new applicationPlanAction
        }}
}
}
```

Ces opérations sont mises en parallèle. C'est avec le mécanisme d'unification que les séquences d'action vont s'établir (même principe qu'avec les *taskEntities* de STP-Stock et STP-Machine)

4 Agentification du système de production

Dans cette section nous présentons la manière d'agentifier les concepts présentés dans la section précédente. Nous considérons que le concept EC peut être considéré comme une ressource de l'environnement. Les concepts STP-Stock et STP-Machine vont être considérés comme des agents hybrides étant donné que ceux-ci contiennent des attributs internes qui vont être considérés comme des états mentaux. Le CP doit avoir des capacités cognitives d'évaluation, de planification qui lui permettent de piloter le système en fonction des connaissances et des objectifs qui lui sont assignés. Il a été donc modélisé par un agent cognitif. Dans cette section, nous allons décrire l'agentification de chacun de ces concepts.

4.1 La ressource EC

La ressource EC a été représentée au niveau du métamodèle du domaine par une *dataEntity* n'ayant aucune action. Au niveau de la modélisation multi-agents du concept d'EC, notre choix s'est porté sur une représentation de l'EC par une ressource passive dans l'environnement.

A ce niveau, nous appliquons la règle de transformation dataEntity-EnvironmentRessource qui vérifie dans les préconditions que la *dataEntity* est visible par divers ressources dans le domaine. Cette règle est formalisée en ARL de la manière suivante :

```
on AgentifiedMM action dataEntityAgentification
is refinement(SdPMM: metamodel, AgentMM : metamodel)
{

pre-style is {SdPMM::EC
                              AgentMM::PassiveRessource}

-- vérifier que EC est visible par plus qu'une
-- activeEntity dans le système
pre {to EC apply
         { forall{ a : SdPMM::activeEntity |
         iterate a by i: Natural do
         {if isVisible(EC,a::i) do
          a::i includes activeEntitySet}
      and
      activeEntitySet.size > 1}
     }}

post-style is {ECPassiveRessource with{EC with{...} }}

transformation is{ECPassiveRessource includes AgentifiedMM}

}assuming {AgentMM::PassiveRessource::constraints }
```

Au niveau de post-style, nous introduisons la nouvelle syntaxe correspondant à l'entité EC modélisée en tant que ressource dans l'environnement. La transformation correspondant à cette action consiste dans l'intégration du *ECPassiveRessource* dans le métamodèle agentifié *AgentifiedMM*. Le style correspondant à cette entité combinera le style d'une ressource passive de l'environnement et de la ressouce passive *PassiveRessource*. Au niveau de *assuming*, nous pouvons intégrer les contraintes qui doivent être respectées lors de l'application du nouveau style. Ces propriétés peuvent correspondre à celles déjà déclarées au niveau de la *dataEntity* EC. Par exemple, nous incluons à ce niveau la propriété stipulant qu'un niveau de stock ne doit jamais dépasser sa capacité.

4.2 Les agents hybrides STP-Stock et STP-Machine

Nous avons modélisé précédemment les fonctionnements internes des STP-Stocks et STP-Machines. Ces entités ont des attributs qu'elles gèrent en interne. Ces attributs seront considérés comme des états mentaux des agents. De plus, nous savons que ces entités sont communiquantes puisque c'est à travers la communication qu'elles réussissent à échanger les ECs (cf. figure fournie par l'équipe Simulation 6.5). Nous considérons ces entités comme des agents hybrides. Pour réaliser cette transformation plusieurs étapes sont nécessaires. Nous décrivons le processus étape par étape dans ce qui suit.

Etape 1 : attributs ⟶ états mentaux

La première règle est AttributToMentalState. Cette règle permet de transformer les attributs en états mentaux internes de l'agent hybride. Pour que cette règle puisse être exécutée, la précondition suivante doit être vérifiée. Elle stipule que les attributs internes ne soient pas visibles de l'extérieur. Ce qui est le cas de ces entités actives. Nous exprimons ci-dessous la règle correspondant à la transformation des attributs du STP-Stock :

```
on AgentifiedMM action AttributToMentalState
is refinement
(SdPMM: metamodel, AgentMM : metamodel){


pre-style {to a : SdPMM::STPStock::attributes.instance apply
        { forall {activeEntityRef : SdPMM::activeEntity |
        not(isVisible(a, activeEntityRef) and
         activeEntityRef <> STPStock) }
    }

post-style is {MentalStateSTPStock
        with{SdPMM::STPStock::attributes}}

transformation is{MentalStateSTPStock includes AgentifiedMM}

}assuming {AgentMM::constraints}
```

Cette première étape doit être suivie par un autre raffinement au niveau du nouveau style. Afin de garantir la cohérence de ces états mentaux, nous avons mentionné au préalable que des sentinelles sont nécessaires. Il serait judicieux à ce niveau d'inclure ces états mentaux dans un module interne qui va implémenter ces sentinelles. Nous utilisons pour cela un concept introduit par Ingenias [PGS03] et qui consiste en un gestionnaire d'état mentaux. Nous décrivons ceci dans l'étape 2.

Etape 2 : Intégration du gestionnaire d'états mentaux

L'intégration du gestionnaire d'états mentaux est formalisée par la règle de transformation suivante :

```
on AgentifiedMM action IncludeMentalStateManager is refinement{

pre-style is {  AgentMM::MentalState with{}     }

post-style is {StockMentalStateManager with{
        MentalStates is {AgentMM::MentalState}
        LockProcess is {
        compose

        sentinelleECStockés is recursive behaviour{

        value lockECStockés
                is connection(MentalStates::ECStockés);
        choose {
        via lockECStockés send MentalStates::ECStockés;
        via unlockECStockés receive MentalStates::ECStockés;
        sentinelleECStockés}
        or
        via readECStockés send MentalStates::ECStockés;
        sentinelleECStockés
        }
        and behaviour {...}
}

transformation is{StockMentalStateManager
                      includes AgentifiedMM;
                               MentalStateSTPStock
                      excludes AgentifiedMM}

}assuming {
        -- prop1 tous les état mentaux ont leur sentinelle

        -- prop 2 cohérence de fonctionnement des sentinelles
        not(via lockECStockés send MentalStates::ECStockés;
        via readECStockés send MentalStates::ECStockés;)
        and
        ...
}
```

Ainsi, les différents états mentaux générés par l'étape précédente sont inclus au niveau du gestionnaire des états mentaux (*MentalStates*). Dans *lockProcess*, toutes les sentinelles nécessaires sont spécifiées sous forme de comportement qui s'il reçoit un évènement sur la connexion lock bloque la ressource et ne permet de la libérer que lorsque unlock est effectué. Sinon,

la lecture peut être lue de manière concurrentielle.

Au niveau de *assuming*, nous avons spécifié une propriété qui vérifie qu'il est impossible de lire une valeur si celle-ci est vérouillée en écriture.

Dans cet exemple, nous nous somme contentés de décrire la sentinelle correspondant à l'état les ECStockés. Le procédé est le même pour les autres états mentaux.

Notons par ailleurs que pour garder la cohérence du métamodèle agentifié, nous avons supprimé le style des états mentaux *MentalStateSdPStock*. Celui-ci est désormais inclu dans le style *StockMentalStateManager* et il serait désormais impossible d'inclure cet état mental dans un agent sans déclarer son gestionnaire qui spécifie les sentinelles.

Etape 3 : taskEntity $\longrightarrow$ actions internes de l'agent

A ce niveau, nous sommes en mesure de faire la correspondance entre les *taskEntities* définies au niveau de l'*activeEntity* STP-Stock. Pour qu'une *taskEntity* soit considérée comme une action interne, il faut qu'elle manipule des attributs dans l'état mental et qu'elle n'ait pas de connexion externe. Bien que les actions de réception et d'envoi d'EC nécessite la communication avec l'extérieur, on n'a pas spécifié de connexions libres au niveau de ces deux *taskEntities*. Ceci a été fait parce que nous étions conscients qu'au niveau des agents, les interfaces de communication (rôle) externe doivent être séparées des modules internes. Mais si l'utilisateur a spécifié des connexions libres au niveau des tâches de réception et d'envoi d'EC, une action supplémentaire devient nécessaire qui permet de transformer les connexions libres en connexions internes avant d'exécuter cette étape.

Dans notre cas, étant donné que les trois actions que nous avons spécifiées valident les préconditions nous appliquons directement l'action *activeEntity-ProcessToInternAction* suivante :

```
on AgentifiedMM action activeEntityProcessToInternAction
is refinement
(SdPMM: metamodel, AgentMM : metamodel)
{
pre { to SdPMM::STP-Stock apply{
            forall{ t : taskEntity.Instances |
                  -- le processus ne contient pas d'actions
                  -- ayant des connexions libres
                  not( exits{c : connection[Any]|
                         freeConnection(c)})
                  and
                  -- les actions manipulent les attributs
```

```
                exits{c1 : connection[Any]|
                        exists{ a: attribut |
                        c1.value = a}}
                }}
}

post-style is {InternProcessSTPStock with{STP-Stock::Process}}

transformation is{InternProcessSTPStock
                        includes AgentifiedMM }

}assuming {AgentMM::InternProcess::constraints}
```

Pour vérifier que les actions manipulent des états mentaux, il suffit de vérifier qu'au niveau de ce processus il existe au moins une connexion qui tranfert un attribut. Rappelons que pour manipuler un attribut la tache doit accéder à l'attribut à travers une connexion (lock).

Le résultat de cette action de transformation est la création du style *InternProcessSTPStock*, qui contient le processus interne décrit au niveau de l'entité active STP-Stock.

Etape 4 : activeEntity ⟶ Agent hybride

A ce niveau du processus, nous pouvons agentifier l'entité active STP-Stock. Les attributs de l'agent ont été redéfinis et inclus dans *StockMentalStateManager* et son processus en *InternProcessSTPStock*. Ces deux composants vont être intégrés au niveau de l'agent hybride *hybridAgentSTP-Stock*. Ceci est possible puisque le style agent hybride n'interdit pas d'avoir de tels composants. Si à ce niveau, le développeur se trompe et intègre ces composants dans un agent réactif, les propriétés seront violées puisque l'agent réactif n'a pas d'états mentaux. L'action correspondant à l'agentification du STP-Stock est la suivante :

```
on AgentifiedMM action activeEntityToHybridAgent
is refinement
(SdPMM: metamodel, AgentMM : metamodel)
{

pre-style is {AgentMM::HybridAgent}

pre { }

post-style is {HybridAgentSTP-Stock with{
        MentalStates is {StockMentalStateManager}
        InternProcess is {InternProcessSTPStock}
        bindings{--unification des connexions}}
```

```
transformation is{HybridAgentSTP–Stock
                      includes AgentifiedMM }

}assuming {–– au niveau de bindings
       ––toutes les connections internes sont unifiées}
```

Ainsi, dans cette action, une nouvelle syntaxe est introduite qui décrit exclusivement un agent hybride pour un STP-Stock.

Etape 5 :intégration des rôles

Au niveau de cette étape, le style défini dans l'étape 5 va être redéfini pour intégrer l'aspect communicatif de ces agents. Ceci va être réalisé par le concept de rôle. En effet, les entité STP-Stock et STP-Machine sont des entités communiquantes qui participent à des protocoles d'interaction. L'équipe avec laquelle nous avons travaillé, nous a transmis un exemple de protocole défini dans la figure 6.5

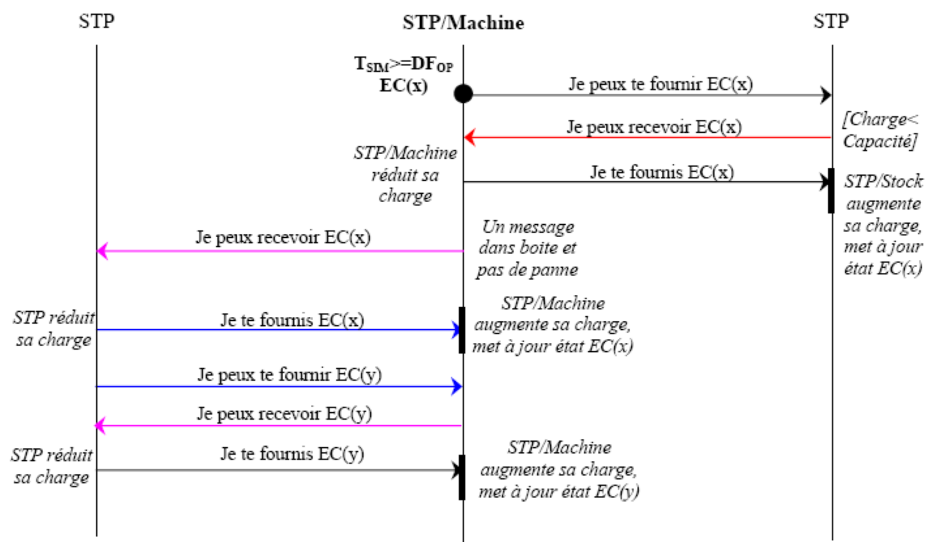


FIG. 6.5: Communication entre STP

Pour intégrer la communication externe, nous avons décidé d'intégrer un module *communicationAction* dans notre style d'agents hybrides. Cette transformation pourrait s'écrire comme suit :

```
on AgentifiedMM::HybridAgentSTP–Stock action
refineStyle is refinement
{
pre–style is {AgentMM::HybridAgent}
```

```
transformation is{HybridAgentSTP-Stock becomes
        HybridAgentSTP-Stock with{
        MentalStates is {StockMentalStateManager}
        InternProcess is {InternProcessSTPStock}
        roles{}
        bindings{--unification des connexions} }

}
```

Le processus d'agentification continue avec plusieurs autres étapes qui vont raffiner les rôles joués par l'agent pour communiquer. Nous avons ici, montré une partie d'un processus qui va être appliqué pour agentifier le STP-Stock. Le même principe va être appliqué aux autres entités du système.

L'agent cognitif CP

L'agentification du CP n'est pas encore achevée. Mais nous commençons à analyser la manière dont on va procéder. Le Centre de Pilotage (CP) joue à la fois le rôle de pilote et de superviseur. Il doit être capable de prendre des décisions pour permettre au système de production d'atteindre ses objectifs. Pour cette raison, nous nous sommes naturellement tournés vers un agent cognitif pour modéliser le CP, plus précisément un agent BDI ayant :

- La faculté de mémoriser l'historique d'évolution du système et des actions qu'il a réalisées sur ce dernier,
- La faculté d'apprendre de son passé pour éviter les erreurs qu'il a pu commettre mais aussi de minimiser son temps de réponse,
- La faculté de choisir la bonne action à entreprendre en se rattachant à des critères (coûts, impact sur l'ensemble du système, etc.),
- L'autonomie dans le comportement.

Afin de mieux expliciter l'agentification de l'Agent-CP, nous proposons dans le tableau 4.2 une correspondance entre les concepts BDI et les composantes du CP.

Dans ce tableau, nous avons défini la base de connaissances de l'agent-CP à travers : les désirs, les croyances et les intentions. Nous avons ainsi pu déterminer ce qu'on peut mettre derrière chaque module à partir de l'analyse du centre de pilotage.

4.3 Agentification des autres éléments du SdP

D'autres concepts appartiennent aux systèmes de production. Nous avons abordé par exemple le concept de centre de charge (CC). Celui-ci a été agentifié par un groupe d'agents constitué d'agents hybrides Stock et agents hy-

Agentification de l'Agent-CP		
Architecture BDI	Architecture interne CP	Explication
Perceptions (P)	Mesures + informations des autres CPs	Ce sont les informations perçues au niveau des éléments du système (STPs, ECs, etc.) rattachés ou des CPs de niveau inférieur.
Croyances (B)	Valeurs des indicateurs et inducteurs + ressources + règles de gestion	C'est la base de connaissances de l'agent contenant les informations nécessaires sur son environnement et sur lui-même.
Désirs (D)	Objectifs et sous-objectifs du CP + Seuil d'alerte	Ce sont les objectifs que l'agent voudrait voir se réaliser. Ils seront en liaison avec les croyances pour déterminer s'il y a dérive ou non sur l'une des perceptions à un instant donné.
Intentions (I)	Les plans d'action mis en place par le CP en cas de dérive d'un inducteur.	Les intentions sont reliées à une bibliothèque de plans où on trouve des plans d'action prédéfinis et stockés à partir des situations passées.

TAB. 6.1: Agentification de l'Agent-CP

brides Machine. Les concepts de "lot", "commande", "processus", "opération" (un processus est composé d'opérations), "calendrier d'ouverture" et "atelier" ne jouent aucun rôle actif : ces éléments n'ont pas d'action à mener. Au même titre que l'EC, ces différentes entités sont représentées par des ressources passives de l'environnement.

5 Conclusion

Dans ce chapitre, nous avons exposé un travail que nous avons mené en parallèle avec nos recherches. A travers ce travail, nous validons notre approche sur un cas réel fourni par nos collègues de l'équipe Simulation. Au cours de ce chapitre nous avons focalisé sur les aspects métamodélisation du domaine et déroulement d'un processus d'agentification en employant l'environnement ArchWare. Nous avons employé dans ce contexte le métamodèle du domaine fourni dans le chapitre 5. Nous avons aussi utilisé des règles de transformation que nous avons formalisées en ArchWare ARL. Ce travail nous a prouvé qu'il était tout à fait envisageable de construire un processus d'agentification en définissant des règles de transformation qui vont générer le style agentifié pas à pas.

Chapitre 7

Conclusions et Perspectives

1 Rappel de la problématique

Au cours de cette thèse, nous nous sommes intéressés à l'ingénierie des systèmes multi-agents. Le point de départ de nos travaux a été de parcourir l'état de l'art relatif aux techniques d'ingénierie des systèmes multi-agents. Nous avons classé ces travaux en trois grandes familles :

- langages spécifiques au paradigme agent qui facilitent la spécification des systèmes multi-agents,
- méthodologies orientées agent qui guident les développeurs durant les phases de développement,
- plates-formes multi-agents qui facilitent l'implémentation et le déploiement des systèmes multi-agents.

L'étude de ces travaux nous a révélé de grandes disparités relatives à l'application des théories orientées agent. Ainsi, nous avons constaté que :

- différents concepts ont été définis et utilisés au niveau des méthodologies, langages de modélisation et plates-formes d'implémentation. La sémantique de ces concepts est souvent abordée de manière informelle, ce qui a suscité différentes interprétations au niveau des différentes techniques d'ingénierie,
- différentes vues ont été spécifiées pour définir les systèmes multi-agents : ces vues regroupent plusieurs concepts et sont fortement interreliées (exemple vue Agent, vue Organisation, etc.). Nous avons relevé que la mise en valeur d'une vue plutôt qu'une autre peut dépendre entière-

ment du domaine d'application,
- le cycle de développement est souvent incomplet : un large fossé sépare les phases d'analyse/conception des phases d'implémentation. En effet, rares sont les méthodologies qui prennent en compte la phase d'implémentation et la génération de code vers une plate-forme spécifique. De plus, les plates-formes ne couvrent pas tous les concepts pris en compte au niveau des modèles produits par la méthodologie. C'est donc au développeur que revient la charge de créer les entités correspondantes.

Dans ce contexte, notre problématique s'est concentrée sur deux aspects :

- proposition d'un cadre de développement flexible : qui devrait permettre aux développeurs de choisir au cours de la conception les concepts et vues orientées agent qui s'accordent le mieux à leurs domaines d'application. Ce cadre de développement devrait permettre par ailleurs de fixer (très tôt dans le processus de développement) la sémantique des concepts multi-agents qui vont être utilisés dans le cadre de l'application.
- intégration de phases de vérification et validation des modèles : les concepts orientés agent ainsi que les modèles qui les appliquent sont généralement abordés de manière informelle ce qui ne favorise pas la vérification et la validation des modèles durant le cycle de développement. Cependant, dans le contexte du développement orienté agent, il est nécessaire d'avoir une conception sûre afin de produire des logiciels de qualité. En effet, les systèmes orientés agent sont composés d'entités jouissant d'une grande autonomie, qui s'exécutent en parallèle tout en partageant plusieurs ressources dans leur environnement. Dans ce contexte, il est nécessaire de s'assurer que les modèles garantissent certaines propriétés dans l'application future telles que la vivacité, l'absence d'interblocage, l'accès sécurisé aux ressources, etc.

Pour garantir la flexibilité du cadre de développement, nous avons adhéré à l'approche adoptée par Ingenias qui consiste à diviser le problème en plusieurs aspects qui seront modélisés par des métamodèles, les modèles et notations nécessaires seront alors générés à partir de ces métamodèles. Le cadre de développement est ainsi construit en fonction des besoins du développeur. A cette approche, nous ajoutons l'utilisation de modèles basés sur des notations formelles qui permettent l'expression et la validation des propriétés. Pour proposer un tel cadre de développement nous avons combiné l'approche dirigée par les modèles et l'approche centrée architecture.

2 Contributions

Notre principale contribution consiste en la proposition d'une démarche de développement qui aidera les développeurs à construire leur cadre de développement. Cette démarche est basée sur la combinaison de l'approche IDM et de l'approche centrée architecture. Dans cette section, nous récapitulons d'abord les apports de chacune de ces approches, ensuite ceux de notre approche ArchMDE.

2.1 Apports de l'approche IDM

L'ingénierie dirigée par les modèles nous a permis d'identifier les niveaux d'abstraction ainsi que les modèles qui sont définis dans chaque niveau. En se basant sur l'architecture à quatre niveaux proposée par l'OMG [OMG03], nous adoptons les niveaux d'abstraction constitués du méta-métamodèle, métamodèle, modèle et système. L'application de cette architecture procure une grande flexibilité dans la construction des cadres de développement. En effet, la couche métamodèle permet de définir de manière explicite les concepts qui sont utilisés pour représenter les différentes vues d'un système multi-agents. Au cours de cette thèse, nous nous sommes principalement basés sur le métamodèle du domaine, le métamodèle des concepts orientés agent et le métamodèle de la plate-forme d'implémentation. Au niveau de la couche modèle, nous avons identifié les deux modèles PIM et PSM qui sont générés par composition des concepts associés aux différents métamodèles.

IDM conçoit l'intégralité du cycle de vie comme un processus de production, de raffinement itératif et d'intégration de modèles ; ce qui rejoint notre objectif. Rappelons qu'un des points auxquels nous nous sommes interessés au niveau de notre problématique est l'absence d'un processus de développment cohérent qui couvre les différents aspects du système ainsi que les différentes phases du cycle de vie. Nous avons vu que dans la plupart des travaux que nous avons étudiés, plusieurs aspects au niveau des systèmes multi-agents devraient être traités de manière *ad hoc* par les développeurs. L'approche IDM permet d'éviter cette situation en exploitant les modèles de manière productive. Ceci veut dire qu'ils sont assemblés, modifiés ou transformés en exécutant des règles de transformation explicitement exprimées. Les modèles produits à chaque niveau d'abstraction deviennent alors des artefacts pouvant être pleinement exploités au niveau du cycle de développement, perdant ainsi leur caractéristique contemplative dans laquelle ils étaient cantonnés dans les approches de développement classiques (où les modèles permettaient uniquement de documenter les applications en cours de développement).

2.2 Apports de l'approche centrée architecture

Le développement orienté architecture se penche sur les techniques de description des modèles architecturaux ainsi que leur intégration. Nous avons choisi de considérer les modèles appartenant à chaque couche d'abstraction comme des constructions architecturales. Ce choix est motivé par le fait que les constructions architecturales permettent de définir des vues du système de manière lisible et qui expose les propriétés les plus cruciales. De plus, cette approche datant d'une dizaine d'années commence à offrir aux développeurs des environnements de développement exploitables. Parmi ces environnements, certains se basent sur des langages formels que nous pouvons utiliser afin de rendre possible l'expression et la validation de diverses propriétés structurelles et comportementales.

En se basant sur cette approche, nous avons choisi de représenter les différents métamodèles par des styles architecturaux. Les styles offrent des mécanismes puissants permettant de définir :

- un vocabulaire pour spécifier les concepts utilisés (exemple : Agent, Groupe, Client, Serveur, etc.)
- des règles de configuration, ou contraintes, qui déterminent comment les concepts peuvent être apppliqués.
- une interprétation sémantique par laquelle les compositions des éléments architecturaux ont une signification bien définie.
- des analyses qui sont associées au style et qui identifient des caractéristiques de différentes sortes (par exemple vérifier une mesure telle que le nombre d'éléments ou vérifier la satisfaction d'une propriété, etc.). Ces analyses forment des propriétés que les architectures générées à partir du style devraient satisfaire.

Ainsi, le PIM appartenant à la couche modèle va être considéré comme une architecture globale du système obtenue par intégration d'architectures représentant différents aspects (ou vues) du système. Cette intégration d'architecture est réalisée grâce à l'application de règles de transformation. Celles-ci sont exprimées par des langages de raffinement que certains environnements centrés architecture offrent. Dans les environnements centrés architecture basés sur des langages formels, les règles de transformation sont exprimées en terme de pré-conditions et post-conditions, ce qui permet de garantir la cohérence de l'architecture.

2.3 Apports de la démarche ArchMDE

Notre démarche que nous avons baptisé ArchMDE (Architecture Model Driven Engineering) applique la combinaison des approches IDM et approche

centrée architecture. Elle permet de construire un cadre de développement flexible et sûr basé sur les besoins du domaine étudié. En effet, au niveau de notre approche, nous avons adopté un cycle de développement modélisé par un double Y. Ce cycle de développment se base sur le métamodèle du domaine, le métamodèle des concepts orientés agent et le métamodèle de la plate-forme d'implémentation. Deux principaux processus de transformation sont définis : le processus d'agentification et le processus d'implémentation que nous avons appelé *mapping*.

Nous avons identifié deux principales manières complémentaires d'utiliser l'approche Arch-MDE, chacune réalisée par des acteurs différents. Ainsi nous distinguons :

1. les spécialistes en métamodélisation : ceux-ci sont responsables de la construction du cadre de développement qui va être utilisé par les développeurs des applications multi-agents. Les spécialistes de métamodélisation fournissent les méta-entités (décrites sous forme de style architectural) permettant de décrire le domaine. Chaque méta-entité définit les propriétés qui lui correpondent et qui doivent être respectées tout au long du cycle de développement. Selon notre approche, les concepts du domaine sont formalisés indépendemment des concepts orientés agent. Parallèlement, les spécialistes de métamodélisation fournissent aussi les métamodèles contenant le métamodèle orienté agent. Par la suite, les règles de transformation entre le métamodèle du domaine et le métamodèle orienté agent sont exprimées. Elles expriment les pré-conditions qu'une entité du domaine doit valider afin qu'elle soit considérée comme une entité agent, ressource, rôle, etc.

2. les développeurs d'applications orientées agents : ceux-ci utilisent le cadre de développement fourni par les spécialistes de métamodélisation. Leur rôle consiste à spécifier le domaine d'application en instanciant les méta-entités définies dans le métamodèle du domaine. Par la suite, ils utilisent les règles de transformation fournies par le cadre de développement afin *d'agentifier* leur domaine. A la fin du processus d'agentification, nous obtenons une architecture abstraite du système que l'utilisateur va pouvoir configurer pour générer son architecture concrète. Enfin, l'architecture du système doit respecter aussi bien les propriétés du domaine que les propriétés des concepts orientés agent.

Dans le cadre de cette thèse, nous avons utilisé l'environnement ArchWare, qui est un environnement formel basé sur le langage π-calcul pour construire l'environnement de développement. Grâce aux langages ArchWare, nous avons fourni les métamodèles nécessaires à l'application de ArchMDE.

Nous avons identifié dans chaque métamodèle les propriétés correspondantes qui sont utilisées au niveau des règles de transformation lors du processus d'agentification. Les outils d'ArchWare nous ont aidé à l'application de ce processus. Ces outils sont principalement composés de :

- l'ASL TOOLKIT qui permet de compiler le style du domaine agentifié. Cet outil comprend aussi un module de configuration permettant la génération de l'architecture de l'application,
- l'Analyser : à la suite de la génération de l'architecture, les propriétés déclarées au niveau du style peuvent être vérifiées.

D'autres outils existent dans cet environnement qui peuvent être exploités tel que l'animateur, le refiner (étendu pour prendre en compte la transformation de styles) et le synthétiseur (pour la génération de code).

Ces travaux ont été appliqués avec succès au niveau du projet BQR dont l'objet était d'agentifier une plate-forme de simulation des système de production afin de lui procurer plus de flexibilité et de faciliter l'application des règles de pilotage.

2.4 Bilan

L'approche ArchMDE traite les deux points que nous avons identifié au niveau de la problématique. La métamodélisation permet d'obtenir un cadre de développement flexible et personnalisable et l'utilisation des langages formels permet de mettre en valeur les propriétés de chaque vue du système. Cependant, notre approche présente plusieurs difficultés et ce pour les deux types d'utilisateur. Ainsi, pour les experts de métamodélisation, les difficultés se situent au niveau de :

- l'identification des méta-entités nécessaires ainsi que leurs relations pour la représentation d'un domaine d'application quelconque,
- l'identification des règles de transformation génériques ainsi que les propriétés sur lesquelles elles se basent (pré et post conditions).

Pour les développeurs d'applications orientées agents, les difficultés se situent dans l'application des règles de transformation. Tout d'abord notre démarche modifie considérablement la manière de développer des systèmes. Dans ce contexte, les développeurs devraient acquérir de nouvelles habitudes de développement basées sur la construction de processus de transformation (dans notre cas ce sont les processus d'agentification et de mapping). Par ailleurs, les développeurs doivent identifier le bon ordre d'exécution de ces règles. La transformation se fait pas à pas, et à chaque pas les modèles doivent être validés.

Cependant, malgré ces difficultés nous pensons que notre approche présente des avantages majeurs. Si les développeurs se familiarisent avec l'utilisation des règles de transformation, cette démarche permettra d'augmenter la productivité et la qualité des systèmes produits. L'application des règles de transformation pour agentifier le systèmes devient alors un moyen de prototypage rapide alors que les règles de transformations vers le code évitera la fastidieuse tâche de codage. La validation des propriétés à chaque application des règles de transformation assurera la qualité de l'application finale. De plus, durant notre expérience, il nous a paru plus facile d'identifier les propriétés relatives au système à un niveau élevé d'abstraction. Par ailleurs, les métamodèles ainsi que la spécification des règles de transformation permettront de conserver le savoir-faire. En terme de réutilisabilité, l'expérience est prescrite sous forme de métamodèles et de règles de transformation.

3 Perspectives

Plusieurs perspectives ont été identifiées nous permettant d'envisager la suite de nos travaux. Nous énumérons ces perspectives dans cette section :

1. Adaptation de l'outil refiner : comme nous l'avons mentionné dans la section 9.2 du chapitre 5, l'outil Refiner a été conçu dans le cadre d'ArchWare pour supporter le raffinement des architectures exprimées selon un style bien particulier qui est les composants/connecteurs. Nous avons voulu au niveau de notre approche éviter la transformation du domaine vers ce style afin de l'agentifier. En effet, nous pensons que cela risque d'alourdir l'approche. Ainsi, nous avons décidé d'étendre ARL afin de prendre en compte la transformation du style. De ce fait il faudrait aussi faire évoluer l'outil Refiner afin de prendre en compte l'extension du langage. Ceci nous permettra d'avoir un outil assistant les développeurs dans l'application des règles de transformation.

2. Définition d'un langage graphique : celui-ci doit être basé sur la sémantique d'ArchWare. En effet, l'utilisation de notre approche dans le cadre du projet BQR, nous a révélé les difficultés que peuvent avoir les non informaticiens dans la pratique des langages textuels. Nous nous sommes alors fixés l'objectif de définir une syntaxe graphique basée sur la sémantique des langages ArchWare. Un profil UML appliqué avec beaucoup de rigueur peut être utilisé à ce niveau. Ces travaux sont en cours actuellement. Nous préconisons par exemple, l'utilisation des diagrammes pour définir des scénarios qui seront considérés comme des structureEntity contenant des acteurs *activeEntities* effectuant des tâches *taskEntities*. Chaque tâche peut être exprimée par un

diagramme d'état/transition dont les transitions entrantes modélisent des actions *receive* et les transactions sortantes, des actions *send*. Les noeuds peuvent modéliser les actions internes (par exemple affectation d'une nouvelle valeur à un attribut).

3. Métamodélisation des plates-formes d'implémentation : la partie implémentation a été abordée de manière superficielle au cours de cette thèse. A moyen terme nous préconisons de métamodéliser les concepts appliqués au niveau des plates-formes d'implémentation. Cette tâche risque d'être délicate. Il faudrait trouver le moyen d'exprimer clairement ces métamodèles ainsi que les règles de transformation permettant le passage du style du domaine agentifié vers le code. Nous explorerons à ce niveau deux approches. La première consiste à exprimer le style du domaine agentifié par un langage exploitable au niveau d'un langage QVT. La deuxième est de faciliter la production de générateur de code à partir de l'environnement ArchWare.

4. Phases de test et maintenance : nous pouvons envisager d'étendre le processus de ArchMDE pour intégrer les phases de test et de maintenance. Pour la phase de test, nous pouvons envisager que les propriétés exprimées au niveau des styles soient transformées en scénarios de test. Pour la phase de maintenance, nous pouvons utiliser un cadre d'exécution fourni par l'environnement de développement centré architecture tel que celui proposé par ArchWare (chapitre 4). Ce cadre d'exécution est composé d'une machine virtuelle qui pourrait exécuter l'architecture modélisée. Celle-ci peut être connectée à la plate-forme d'exécution réelle et agir comme un système de monitoring. Au niveau de ce cadre d'exécution, un langage (l'ArchWare Hyper-code ADL) est conçu pour supporter l'évolution d'une architecture en cours d'exécution en s'appuyant sur la notion de "réflexion structurelle". Cette notion est définie comme la capacité d'une spécification en cours d'exécution à générer de nouveaux fragments de spécification et à les intégrer au sein de sa propre exécution.

5. Système d'aide à la décision : à long terme, nous pouvons envisager de proposer un système d'aide à la décision, qui à partir de l'analyse des propriétés exprimées au niveau du métamodèle du domaine et du métamodèle agent puisse suggérer des règles de transformation que les développeurs puissent utiliser lors du processus d'agentification. Ceci implique que des propriétés telles que l'intelligence ou l'autonomie des agents puissent être explicitement exprimées. En effet, à ce jour ces propriétés sont encore considérées de manière vague. Cette perspective pourrait encore plus faciliter l'application des systèmes multi-agents

et la rendre plus accessible à des non-informaticiens. Ceux-ci n'auront qu'à décrire leur domaine. A partir des propriétés de ce domaine, le système pourra identifier les règles de transformation à appliquer au niveau du processus d'agentification. Le même processus pourrait être envisagé pour effectuer le processus de mapping.

Chapitre A

Description des langages ArchWare

Afin de simplifier la compréhension de ce manuscrit de thèse, nous fournissons dans cette annexe une description détaillée des langage ArchWare. Rappelons que l'environnement ArchWare fournit un ensemble de langages qui sont :

- ArchWare Architecture Description Language (π-ADL) : c'est un langage pour la description d'architectures ayant des caractéristiques dynamiques et évolutives. En effet, le langage permet de capturer les mécanismes permettant à l'architecture de changer de topologie et de configuration durant l'exécution,
- ArchWare Architecture Analysis Language (AAL) : c'est un langage pour la spécification de divers propriétés architecturales,
- ArchWare Architecture Refinement Language (ARL) : c'est un langage pour la description de raffinements d'architectures.
- ArchWare Architecture Style Language (ASL) : c'est un langage pour la description de styles architecturaux.

1 Le langage ArchWare ADL

En π-ADL, une architecture est un ensemble d'éléments, appelés éléments architecturaux, qui sont reliés par des liens de communication. Ces éléments sont définis en terme de comportement (***behaviour***, ***abstraction***). Ce dernier est défini par un ensemble d'actions ordonnancées qui spécifient le traitement interne de l'élément (actions internes) et les interactions avec son environnement (actions de communication). Un élément architectural communique avec les autres par une interface caractérisée par un ensemble de connexions (***connections***) qui permet de faire transiter des données. Un

mécanisme de composition et un mécanisme de liaison, appelé unification (c'est une substitution au sens du π-calcul) permettent la mise en relation des éléments architecturaux. Ces éléments peuvent interagir lorsqu'ils sont composés et liés. Un élément architectural peut être défini comme une composition d'autres éléments (composite). En d'autres termes, un comportement peut être défini comme un ensemble d'autres comportements interconnectés. π-ADL permet la description d'architectures dynamiques. En effet, les éléments architecturaux peuvent, d'une part, être composés ou décomposés à la volée et, d'autre part, des éléments architecturaux peuvent être créés et liés dynamiquement. Enfin, des éléments architecturaux peuvent transiter comme des données à travers les connexions. Ce qui fournit un mécanisme adéquat pour décrire des architectures dynamiques dont la topologie peut changer au cours du temps.

Dans les paragraphes suivants, nous donnons plus de détails sur la description des comportements et des structures d'une architecture. Nous exposerons les types de données pouvant être définis ainsi que leurs mécanismes de manipulation.

1.1 Comportement

Tout comportement est construit autour d'actions et d'opérateurs spécifiques. Ces actions et ces opérateurs sont similaires à ceux du π-calcul. D'ailleurs, un comportement est l'équivalent d'un processus en π-calcul. Il existe différents types d'actions possibles qui sont décrites ci-dessous. Le texte qui est en caractère gras correspond aux mots clé du langage π-ADL :

- l'action d'envoi : qui permet à un élément architectural d'envoyer une donnée (ou d'autres éléments architecturaux en cas d'architecture dynamique). Cette action est décrite de la manière suivante :

```
via nomConnexion send données;
```

- l'action de réception : celle-ci décrit la réception d'un message par un élément architectural. Elle est décrite de la manière suivante :

```
via nomConnexion receive données : DéfinitionDuType;
```

- l'action inobservable : représente une action interne qui ne participe pas à une communication ; elle est inobservable depuis l'environnement du comportement.

```
unobservable
```

- l'inaction : celle-ci correspond à l'action *nil* du π-calcul. Elle peut être utilisé pour marquer la terminaison d'un processus en exécution. L'inaction est décrite par le mot clé **done**.

```
done
```

Un comportement dans sa globalité, inclut un ensemble ordonné d'actions. Plusieurs opérateurs sont utilisés pour ordonnancer les actions, ce sont les opérateurs de succession, de condition, de choix, de composition et de réplication. Nous présentons dans ce qui suit chacun des opérateurs.

L'opérateur de succession

- L'opérateur de succession : il est noté par un point virgule. L'action précédent la virgule doit s'exécuter avant l'action qui suit la virgule.

L'expression suivante est la spécification d'un comportement nommé *b* qui correspond à l'envoi du message "Hello". A la suite de cet envoi, le processus s'arrête (action *done*).

```
value b is behaviour{ via a send "Hello"; done}
```

L'action done peut être omise lorsqu'elle est préfixée. L'expression précédente devient équivalente à :

```
value b is behaviour { via a send "Hello"}
```

L'opérateur de condition

- L'opérateur de condition est noté par **if(** condition **) then** $comportement_1$ **else** $comportement_2$ définit un comportement qui se comporte comme comportement1 si la condition est vérifiée et comme comportement2 sinon.

L'expression π-ADL suivante est la définition d'un comportement qui définit la réception de deux chaînes de caractères. Si celles-ci sont équivalentes, il renvoie la première, sinon il renvoie un message d'erreur.

```
behaviour {
        value x, y, z is connection(String);

        via x receive c1: String;
        via x receive c2: String;

        if c1 == c2 then via z send y;
        else via z send "Error"; }
```

L'opérateur de choix

– L'opérateur de choix correspond à l'expression **choice** { *comportement*$_1$ **or ... or** *comportement*$_n$ } : Il traduit un choix indéterministe qui doit être fait entre des comportements possibles, le choix correspond à un OU exclusif, si l'une des actions est choisie, l'autre est abandonnée. Ceci peut être très utile dans la description des architectures distribuées, qui peuvent être confrontées à des troubles liés au réseau. Le comportement d'un élément architectural peut être bloqué s'il n'a pas de correspondant immédiat. Dans ce cas, une autre action peut être effectuée.

L'expression π-ADL suivante est un comportement qui peut se comporter de deux manières différentes : soit recevoir un message via une connexion x, soit envoyer un *"timeout"* via la connexion z.

```
behaviour {
        value x is free connection(String);
        value z is free connection(String);
        choose{
        {via x receive y : String; unobservable}
 or
        {via z send "timeout"; unobservable}
}}
```

L'opérateur de composition

– L'opérateur de composition est défini par l'expression suivante **compose** { *comportement*$_1$ **and ... and** *comportement*$_n$ } : elle traduit la mise en composition parallèle de plusieurs comportements c'est-à-dire que ceux-ci vont s'exécuter en parallèle et vont être concurrents. La synchronisation de ces comportements permet à deux éléments mis en parallèle de communiquer. Celle-ci se fait par l'unification des connexions libres (qui sont préfixées par le mot clé *free*). Après unification les connexions deviennent partagées.

L'expression liée à l'unification des connections se fait est la suivante :

```
nomComportement::nomConnection
unifies
nomComportement::nomConnection
```

L'exemple suivant présente la composition de deux comportements, p et q. Ces deux comportements possèdent des connexions libres (free) qui sont unifiées de telle sorte que p puisse envoyer un message sous forme de String à q.

```
        compose {
                p is behaviour {
                        value y is free connection(String);
                        value message is "Hello";
                        via y send message }
        and
                q is behaviour {
                        value z is free connection(String);
                        via z receive message : String;
                        unobservable}
        where {p::y unifies q::z}
}
```

L'opérateur de réplication

- l'opérateur de réplication **replicate**{ comportement } est équivalent à une composition infinie du même comportement. Nous pouvons l'assimiler à un opérateur de clonage d'un même comportement plusieurs fois.

L'expression suivante est un comportement qui peut être déclenché une infinité de fois par la réception d'une requête par la connexion *requestConnection.*

```
behaviour {
        replicate {
        value requestConnection is connection(String);
        via requestConnection receive y : String;
        unobservable;
                        }
}
```

La notion d'abstraction

Afin de réutiliser des définitions de comportement, ArchWare ADL fournit un mécanisme de réutilisation appelé abstraction. L'abstraction permet d'encapsuler des définitions paramétrables de comportement. Nous pouvons comparer une abstraction à une procédure. On obtient un comportement d'une abstraction par l'application de cette dernière, en fournissant éventuellement une liste de paramètres.

L'exemple suivant définit l'abstraction d'une architecture décrivant un groupe de deux agents. Ces agents communiquent entre eux : l'un envoie une requête et l'autre y répond. Le comportement de chaque agent est décrit dans une abstraction. Ces abstractions sont ensuite appliquées au niveau du groupe qui est lui même décrit par l'abstraction *Groupe2Agents*. Au niveau de cette abstraction les connexions sont unifiées afin que les deux agents puissent communiquer. L'Agent1 effectue un appel à l'Agent2, puis il attend une réponse. L'Agent2 est en attente de l'appel de l'Agent1. Lorsqu'il reçoit cet appel celui-ci lui répond. *Groupe2Agents* est alors considéré comme un élément composite.

```
value Agent1 is abstraction ()
{
value call, wait is connection();
via call send; via wait receive;
unobservable
};

value Agent2 is abstraction ()
{
value request, reply is connection();
via request receive; unobservable;
via reply send
};

value 2AgentGroup is abstraction ()
{
compose { A1 is Agent1()
        and
        A2 is Agent2() } where {
                        A1::call unifies A1::request
                        and
                        A2::reply unifies A2::wait}
}
```

La notion de récursivité

En plus de permettre la réutilisation, les abstractions permettent de définir des comportements récursifs, c'est-à-dire des comportements qui retournent à leur état initial après un certain nombre d'étapes. Nous reprenons l'exemple précédent de deux agents communiquant dans un groupe. Après avoir reçu une réponse, l'Agent1 peut renvoyer une requête alors que l'Agent2 peut se remettre en attente d'une requête. Pour décrire un tel comportement récursif,le mot clé **recursive** est nécessaire. Ceci va être décrit comme suit :

```
recursive value Agent1 is abstraction ()
{
via call send; via wait receive; unobservable;
Agent1()
};

recursive value Agent2 is abstraction ()
{via request receive; unobservable; via reply send;
 Agent2()
};
```

Nous avons vu que les comportements communiquent par des envois de données, nous étudions les aspects concernant ces données dans ce qui suit.

1.2 Les valeurs et les types

ArchWare ADL est un langage fortement typé. Il fournit un système de types de données complet et des opérateurs pour manipuler les données. Nous présentons ici ces types et ces opérateurs. ArchWare ADL fournit des types qui sont classés en types de base, types d'architectures, types construits et types collections :

- Les types de base sont Natural, Integer, Real, Boolean, et String.
- Les types d'architectures sont les suivants : connection[Type de donnée], Behaviour, abstraction[Types des paramètres]. Ils représentent les valeurs architecturales présentées dans les sections précédentes.
- Les types construits représentent des valeurs composites. Ces types sont les suivants : tuple[Liste de types], view[Liste de types étiquetés], union[Liste de types], variant[Liste de types étiquetés],
- Les types collections représentent des valeurs composées de plusieurs autres valeurs de même type. Ces types sont bag[Type], set[Type] et sequence[Type].

Prenons comme exemple, le type vue, qui représente une structure de valeurs de types différents. Au niveau d'une vue, ces valeurs sont étiquetées. Dans l'exemple suivant, une valeur v de type view est définie. Elle est com-

posée d'une valeur true étiquetée b, et d'une valeur *"Chaîne de caractère"* étiquetée s.

```
value v is view(b is true, s is "Chaîne de caractère")
```

Cette valeur a comme type, MaVue sui est défini de la manière suivante :

```
type MaVue is view[b : Boolean, s : String]
```

Comme nous l'avons mentionné précédemment, des mécanismes sont fournis pour pouvoir manipuler les valeurs construites.

Projection

Des mécanismes de projection permettent d'accéder aux composantes d'une valeur composée (vue, tuple, variant ou union). La ligne suivante exprime que les composantes de v sont associées aux identifiants b et s qui sont utilisés par la suite (si b vaut true alors on envoie s par la connexion c).

```
project v as b, s; if b do via c send s
```

Le mécanisme de projection peut être décrit en utilisant du sucre syntaxique noté " : :".

```
if v::b do via c send v::s
```

iterate

Ce mécanisme permet de manipuler les valeurs ayant un type collection, en permettant de parcourir une collection. L'exemple suivant permet de compter le nombre d'éléments contenus dans un ensemble de type set. Ceci est réalisé par un mécanisme d'itération : pour tout élément i de l'ensemble, la valeur somme (initialisée à 0) est incrémentée de la valeur de l'élément i ; le résultat final est stocké dans la valeur resultat.

```
value ensemble is set(1,2,3);
iterate ensemble by i : Integer;
from value somme is location(0)
accumulate {somme:=somme' + i}
as resultat;
```

Pour accéder aux éléments des collections, le mécanisme de projection est utilisé. Dans l'exemple suivant, le deuxième élément est récupéré et stocké dans la valeur $resultat_1$.

```
    value seq is sequence(2,4,1);
    project seq at 2 as resultat1;
```

Includes et Excludes

D'autres mécanismes permettent d'ajouter ou de retirer des éléments d'un ensemble. Dans l'exemple suivant, la valeur 2 est ajoutée à ensemble. Ensuite une valeur 3 est retirée.

```
value ensemble is set(1,2,3);
ensemble includes 2;
ensemble excludes 3;
```

2 Le langage ArchWare AAL

En AAL, les propriétés sont définies comme des formules prédicatives. Ce langage fournit des prédicats prédéfinis et des mécanismes (opérateurs et quantificateurs) pour construire de nouveaux prédicats. Un support de vérification des propriétés est fourni à travers les outils Analyser [AO05] et Model Checker [BCD+04].

2.1 Les opérateurs et les quantifieurs

ArchWare AAL fournit des opérateurs booléens ainsi que des quantificateurs. Ceux-ci permettent de construire de nouveaux prédicats à partir de prédicats définis.
Le vocabulaire utilisé pour les opérateurs booléens est le suivant : **not, or, and, xor, implies, equivalent.**

Les quantificateurs sont le quantificateur existentiel, **exists**, et le quantificateur universel **forall**.
Un prédicat peut être appliqué à une collection via l'opérateur **to ... apply**.

```
to collection apply prédicat
```

Ce dernier est généralement utilisé avec les quantificateurs **exist** et **forall** pour appliquer un prédicat à tous les éléments d'une collection. L'expression

suivante est vraie si le prédicat *p* est vrai pour chacun des éléments de la collection.

```
to collection apply forall{element | p(element)}
```

2.2 Prédicats et fonctions sur les données

AAL fournit un éventail de prédicats et de fonctions prédéfinis concernant les données en général. Nous présentons ci-dessous un sous-ensemble des prédicats prédéfinis.

isValue(name, type **)** est vrai si la valeur name est de type *type*.
isType(alias, type **)** est vrai si l'identifiant alias réfère le type *type*.
isAbstraction(name **)** est vrai si la valeur name est une abstraction.
isInstance(name, abstractionName **)** est vrai si le comportement name est une instance de l'abstraction *abstractionName*.
isConnection(name, type **)** est vrai si **isValue(**name, connection[type] **)** est vrai.

Concernant les fonctions prédéfinies, nous présentons également un sous-ensemble.

parameters(name **)** - cette fonction s'applique à une abstraction et retourne l'ensemble de ses paramètres.
name(data) - cette fonction retourne le nom d'une donnée, elle existe pour tout type de donnée ainsi que pour les paramètres.
value(name) - cette fonction retourne la valeur d'une donnée.

2.3 Prédicats sur les comportements

ArchWare AAL permet de décrire des patrons comportementaux. Ce sont des prédicats qui sont vrais si l'ordonnancement des actions d'un comportement vérifie certaines règles qu'on exprime. Ces règles expriment l'agencement possible entre des actions de communication. Elles sont décrites à l'aide d'opérateurs spécifiques :

- la concaténation, '.',
- le choix, |,
- la fermeture transitive réflexive, '$*$' : elle correspond à la concaténation de zéro ou plusieurs actions.
- la fermeture transitive, '$+$' : elle correspond à la concaténation d'une ou plusieurs actions.

L'expression suivante dénote un patron pour lequel une réception via la connexion c2 ne peut être précédée directement d'une ou plusieurs émissions via la connexion c1.

```
(not(via c1 send any))*.(via c2 receive any)
```

De plus, AAL fournit deux autres opérateurs en se basant sur les opérateurs modaux de nécessité et de possibilité du μ-calcul. Le premier permet de spécifier que toute séquence d'actions vérifiant un patron comportemental donné doit aboutir à un état donné. Le second permet de spécifier qu'il doit exister au moins une séquence d'actions vérifiant un patron comportemental donné et aboutissant à un état donné. La propriété suivante exprime qu'il ne peut y avoir d'envoi par la connexion c2 que s'il y a eu réception par la connexion c1 juste avant. Dit autrement, chaque séquence dans laquelle un envoi par la connexion c2 n'est pas précédé par une réception par la connexion c1 conduit vers un état faux.

```
every sequence {not via c1 receive any. via c2 send any}
               leads to state {false}
```

La propriété suivante exprime qu'il doit exister au moins une séquence dans laquelle une action de réception par la connexion c1 précède un envoi par la connexion c2.

```
some sequence { via c1 receive any }
        leads to state { via c2 send any }
```

Enfin, AAL fournit deux derniers opérateurs se basant sur les quantificateurs de points fixes maximaux et minimaux du μ-calcul :
- **finite tree** Y (x :T) **given by** s (Y) (v)
- **infinite tree** Y (x :T) **given by** s (Y) (v)

Le premier opérateur est basé sur les points fixes minimaux. Cet opérateur spécifie qu'en partant d'un état satisfaisant Y, on arrive après un nombre fini de pas (un "pas" = un dépliage récursif de la formule s) à un état satisfaisant s (false). L'intuition est que les points fixes minimaux se comportent comme des fonctions récursives en langage de programmation, sauf qu'ils sont interprétés sur des systèmes de transitions. En d'autres termes, un patron d'actions peut être répété un nombre fini de fois avant que le système atteigne un état donné.

Par exemple, la formule suivante exprime l'existence d'une succession finie d'actions *via x receive any*.

```
finite tree Y given by
{some sequence{via x receive any}
        leads to state {true} or Y}
```

Après un nombre fini de pas (transitions franchies en dépliant le corps "{ some sequence { via x receive any } leads to state { true } or Y" de la formule), on doit arriver à un état qui satisfait {some sequence { via x receive any } leads to state { true } or { some sequence { via x receive any } leads to state { true }. Le second opérateur est basé sur les points fixes maximaux. Par exemple, la formule suivante signifie l'existence d'une séquence infinie de réception par la connexion x.

```
infinite tree Y given by
        {some sequence {via x receive any} leads to {Y}}
```

3 Le langage ArchWare ARL

Le noyau du langage est un ensemble d'opérations appelées actions de raffinement qui supportent le raffinement d'une architecture abstraite vers une architecture concrète en vue de son implémentation [Meg04]. Dans ARL, les actions de raffinement sont exprimées par des pré-conditions, des transformations et des post-conditions. Les pré-conditions sont des conditions qui doivent être satisfaites dans une architecture avant l'application d'une action de raffinement. Les post-conditions doivent être satisfaites suite à l'application d'une action de raffinement. La transformation décrit l'opération à réaliser.
Une étape de raffinement s'effectue par l'application d'une action de raffinement que l'on peut exprimer en utilisant la syntaxe suivante :

```
on architectureDefId action refinementActionId
        is refinement (actionParameters){
        pre is { actionPreconditions }
        post is { actionPostconditions }
        transformation is { refinementActions }
} assuming { properties } as { mixfix }
```

où :

- actionPreconditions sont les conditions nécessaires à satisfaire avant d'appliquer l'action de raffinement,
- actionPostconditions sont les conditions à satisfaire après l'application de l'action de raffinement,
- refinementActions sont les actions de raffinement à appliquer pour exécuter une transformation depuis l'architectureDefId vers une nouvelle architecture raffinée,
- properties sont les conditions que l'on suppose vraies et qui doivent être vérifiées (des obligations de preuve) afin d'appliquer les actions de raffinement,
- mixfix définit la notation d'une action dans une forme mixfix.

Notons que actionPreconditions, actionPostconditions et properties sont des expressions logiques.

Différentes actions de raffinement peuvent être envisagée au niveau des architectures. Nous les résumons ci-dessous.

Les opérations de raffinement similaires sur tous les éléments architecturaux

Elles consistent principalement à l'ajout, suppression et la modification des éléments architecturaux ou au remplacement d'un élément architctural par un autre. Elles sont exprimées par les expressions suivantes :
- Includes : ajoute un élément ou un sous-ensemble d'élément à un ensemble,
- Exludes : supprime un élément ou un sous-ensemble d'éléments d'un ensemble,
- Replaces : remplace un élément par un élément,
- Becomes : remplace un comportement par un autre comportement.

Voici un exemple de règle de raffinement permettant de supprimer plusieurs comportements dans une architecture.

```
Archtype NouvelleArchitecture refines AncienneArchitecture
using
{
behaviour  excludes  {beh1, ..., behN}
}
```

Les opérations de raffinement spécifiques aux connections

Celles-ci permettent de raffiner les relations entre les différents éléments architecturaux. Elles sont exprimées comme suit :
- Unifies : unifie des connections
- Separates : sépare des connections

L'exemple suivant permet de séparer deux connections.

```
Archtype NouvelleArchitecture refines AncienneArchitecture
using (Ancienne_Connections: tuple[connection])
{
separates
        Element1::port1::connection1
with
        Element2::port2::connection2
}
```

4 Le langage ArchWare ASL

Le langage ASL a été construit comme une couche supplémentaire sur les langages ADL et AAL. Nous présentons dans ce qui suit les mécanismes qu'il offre pour la formalisation de nouveaux styles.

4.1 description générale d'un style

En ASL, un style est formalisé selon trois entités :

- les constructeurs (qui fournissent un support à la description architecturale),
- les contraintes,
- les analyses.

La structure générale de la description d'un style est la suivante :

```
Nom_du_style is style extending Style_parent where {
        types { Définitions de types }
        styles { Définitions de styles }
        constructors { Définitions de constructeurs }
        constraints { Définitions de contraintes }
        analyses { Définitions d'analyses }
}
```

Nom du style

Un style est nommé. Son nom est l'unique référence vers ce style. Il est nécessaire de faire référence à un style à plusieurs occasions : lors de son instanciation, lors de la vérification de la satisfaction d'une architecture par rapport à ce style, lors de l'héritage par un autre style.

Style parent

Comme nous l'avons mentionné précédemment, ASL fournit un mécanisme de spécialisation : l'héritage. Celui-ci permet de construite un nouveau style S sur la définition d'un autre style S'. Le style S' est alors appelé style parent du style S et le style S un sous-style de S'. Les éléments définis au

niveau du style S' (les constructeurs, les contraintes et les analyses) sont des acquis pour le style S.

On spécifie qu'un style en spécialise un autre par le mot clé extending. Dans l'exemple ci-dessous, le style Style hérite du style StyleParent.

```
Style is style extending StyleParent where { ... }
```

4.2 Types

La définition d'un style peut englober la définition d'un ensemble de types. Les types sont construits par rapport aux types prédéfinis dans π-ADL. Trois sortes de types génériques sont cependant rajouté :

- **Type** : c' est un type dont les valeurs associées sont des types,
- **Expression[T]** : permet de capturer des définitions de valeurs de type T. Par exemple, un paramètre de type Expression[Behaviour] est la définition d'un comportement et non pas une instance de comportement qui s'exécute,
- **Alias** : c'est un type défini pour pouvoir passer un alias en paramètre (et non l'instance qu'il référence),

Ces types sont utilisables pour paramétrer les constructeurs. Nous clarifions cette notion ci-dessus.

4.3 Styles

Styles permet de structurer un style en terme d'autres styles dits agrégats. Les styles agrégats encapsulent des constructeurs, de contraintes et des analyses qui alimentent le style contenant.

La structure d'un style en termes d'agrégats peut être construite parallèlement à celle des architectures qu'il représente. Reprenons l'exemple des deux agents interagissant au sein d'un groupe. Cette architecture peut être représentée par un style qui sera structuré avec un style *Agent1* et un style *Agent2*. Un style agrégat *Groupe2Agents* donne un support pour des sous-parties des architectures.

```
Groupe2Agents is style {
 styles {
                Agent1 is style where{...}
                Agent2 is style where{...}
               }
...
 }
```

4.4 Les constructeurs

Les constructeurs fournissent le support à la description architecturale. Ils permettent en effet de fournir des patrons architecturaux, respectant un style particulier. La valeur définie par un constructeur peut représenter aussi bien une architecture, qu'un élément architectural, qu'un comportement, qu'une donnée ou une connexion. Ainsi, différents niveau de granularité peuvent être traités, allant d'une simple donnée à une architecture complexe.

Un constructeur est un mécanisme de réutilisation. L'ensemble des constructeurs d'un style constitue une bibliothèque de patrons architecturaux réutilisables.

Un constructeur est défini avec un nom (nomDuConstructeur), un ensemble de paramètres typés (p1, p2, ..., pn), un corps (corpsDuConstructeur) et une notation mixfix (notationMixfix). La structure syntaxique correspondant à une style est la suivante :

```
nomDuConstructeur is constructor(p1:TypeP1, ..., pn:TypePn);
{ corps_du_constructeur}
as { notationMixfix }
```

Le corps d'un constructeur contient la description d'une valeur construite relativement aux paramètres fournis au niveau du constructeur. Pour décrire ces valeurs, le langage ArchWare ADL (π-ADL) est utilisé.

La notation mixfix permet de créer une syntaxe adéquate aux concepts définis par le style. Cette syntaxe sera utilisée lors de l'élaboration des architectures au niveau de la couche M1. Cette notation permet d'abstraire les concepts basiques de ASL et de ADL et facilite ainsi l'utilisation de l'environnement par des non spécialistes d'ArchWare.

Pour illustrer l'utilisation du constructeur, nous reprenons notre exemple. Nous créons ici un constructeur pour l'Agent1, d'un constructeur de valeurs de type abstraction. Ces abstractions représentent des agents qui envoient des requêtes aux autres agents.

```
Agent1 is constructor(nom: String, requete : String)
{
abstraction () {
        value requete is tuple(nom, requete)
        via call send requete;
        via wait receive reponse: String;
        unobservable
        }
}as {agent named $nom sends $requete}
```

4.5 Les contraintes

Les contraintes de style caractérisent une famille d'architecture qui présentent des qualités spécifiques en termes de structure ou de comportement. On s'assure qu'une architecture présente l'ensemble des qualités entraînées par un style lorsqu'elle satisfait ses contraintes. Pour cette raison, il est important de formaliser les contraintes pour pouvoir par la suite vérifier automatiquement qu'une architecture satisfait un style.

Plusieurs types de contraintes peuvent être différenciées :
- les contraintes structurelles : par exemple, une architecture ne contient pas de cycles ;
- les contraintes comportementales : par exemple, la vivacité, l'abscence d'interblocage,
- les contraintes d'attributs : par exemple, la valeur d'un attribut ne doit pas excéder 100.

La structure syntaxique d'une description de contrainte est la suivante :

```
nom_de_la_contrainte is constraint {
        corps_de_la_contrainte
}
```

Une contrainte peut aussi être définie par référence à une contrainte d'un autre style.

```
constraints {
contrainte is autre_style@contrainte1
}
```

où, *autre_style@contrainte1* désigne la contrainte nommée *contrainte1* dans *autre_style*.

Le corps de la contrainte est exprimé par des propriétés en AAL. La contrainte concerne l'instance d'un style. Généralement, il s'agit d'une architecture représentée sous forme d'abstraction, mais il peut s'agir de n'importe quel autre type de donnée. Au sein de la description cette instance est représentée par le mot-clé **styleInstance**.

Nous illustrons la définition des contraintes par l'exemple suivant qui décrit une formalisation de la contrainte *deadLockFreedom*. Cette contrainte vérifie que pour toute séquence d'actions on peut trouver une action prochaine telle que celle-ci débouche elle-même sur une action. Ainsi, cette propriété signifie qu'une action peut toujours être effectuée, donc que le système ne peut pas bloquer.

```
deadLockFreedom is constraint {
        to styleInstance.behaviour apply{
          every sequence {true*}leads to state
          {some sequence { true } leads to state {true}}
        }
}
```

4.6 Les analyses

Afin d'automatiser l'étude d'une architecture suivant un style particulier, il est nécessaire de définir des outils d'analyse spécifiques à ce style.

Les caractéristiques identifiées peuvent être de différentes sortes. Il peut s'agir d'une vérification ou d'une mesure. Une analyse peut vérifier la satisfaction d'une propriété par l'architecture ; par exemple, une analyse permet de vérifier que la structure de l'architecture est cyclique. Une analyse peut retourner une mesure ; par exemple, l'analyse retourne le nombre d'éléments d'une architecture.

La performance d'un système s'évalue différemment selon la nature de son architecture. Par exemple, s'il s'agit d'une architecture décrivant un agent réactif, on évaluera son temps de réponse. Un agent cognitif sera plus évalué sur le nombre de cas qu'il pourrait résoudre.

La structure syntaxique permettant de créer une analyse est la suivante :

```
nom_de_l_analyse is analysis {
        kind type_d_analyse
        input { parametres_d_entrée }
        output { type_de_la_valeur_retournée }
        body{ corps_ de_l_analyse }
}
```

Ainsi, une analyse est définie avec :

- un nom : il est utilisé pour référencer l'analyse ;
- le type d'analyse : celui-ci référence l'outil d'analyse capable de traiter la description contenue au niveau du corps.
- des paramètres d'entrée (*input*) : ils sont constitués par un ensemble d'identifiants typés. Les valeurs qui sont passées en paramètre représentent à la fois des éléments sur lesquels l'analyse porte et des éléments pour configurer (régler) l'analyse. Ils sont utilisés dans la description du corps de l'analyse comme s'il s'agissait de valeurs définies.
- un type de la valeur retournée (*output*) : ici sont spécifiés les types de données qu'une analyse va retourner.
- un corps : il contient la description de l'analyse dans un langage associé au type de l'analyse.

Bibliographie

[AA96] A. Abd-Allah. *Composing Heterogeneous Software Architecture.* PhD thesis, Center for Software Engineering, University of Southern California, 1996.

[AAG95] G. Abowd, R. Allen, and D. Garlan. Formalizing style to understand descriptions of software architecture. *ACM transaction on Software Engineering and Methodology*, pages 319–364, 1995.

[Abr91] J.R. Abrial. The B Method for Large Software Specification, Design and Coding. *S. Prehn and W.J. Toetenel. Eds. VDM'91 : Formal Software Development Methods*, 1991.

[AGMO02] I. Alloui, H. Garavel, R. Mateescue, and F. Oquendo. The ArchWare Architecture Analysis Language. ArchWare Europeen RTD Project IST-2001-32360. Delivrable D 3.1b, 2002.

[AHO06] S. Azaiez, M.P. Huget, and F. Oquendo. Approach for Multi-Agent Metamodelling. *Multi-Agent and Grid Systems, Special Issue on Agent-Oriented Software Development Methodology*, 2006.

[All97] Robert Allen. *A Formal Approach to Software Architecture.* PhD thesis, Carnegie Mellon University, USA, 1997.

[AO05] S. Azaiez and F. Oquendo. Final ArchWare Architecture Analysis Tool by Theorem Proving. ARCHWARE European RTD Project IST-2001-32360. Deliverable D1.2b, 2005.

[APVO03] S. Azaiez, F. Pourraz, H. Verjus, and F. Oquendo. Final ArchWare Architecture Animator - Release 1. ArchWare European RTD Project IST-2001-32360, Deliverable D2.2b, 2003.

[Aus62] J.L. Austin. How to do Things with Words. *Oxford University Press*, 1962.

[Bak96] M. Bakalem. *Modélisation et simulation orientées objet des systèmes manufacturiers.* PhD thesis, Thèse de Doctorat en Electronique - Electrotechnique - Automatique, Université de Savoie, 1996.

[BAP02] M. Le Bars, J.-M. Attonaty, and S. Pinson. An agent-based simulation for water sharing between different users. In *Interna-*

tional Joint Conference on Autonomous Agents and Multi Agent Systems : Bringing people and agents together, pages 211–213, Bologne, 2002. ACM.

[BBB+05] J. Bézivin, M. Blay, M. Bouzhegoub, J. Estublier, J.M. Favre, S. Gérard, and J.M. Jezequel. Rapport de Synthèse de l'AS CNRS sur le MDA (Model Driven Architecture). Rapport CNRS, janvier 2005.

[BBF06] A. Belangour, J. Bézivin, and M. Fredj. Towards a new software development process for MDA. In *Proceedings of the European Workshop on Milestones, Models and Mappings for Model-Driven Architecture*, Bilbao, Spain, 2006.

[BCD+04] D. Bergamini, D. Champelovier, N. Descoubes, H. Garavel, R. Mateescu, and W. Serwe. ArchWare Architecture Analysis Tool by Model-Checking. ARCHWARE European RTD Project IST-2001-32360. Deliverable D3.6b, 2004.

[BCG+04] C. Bernon, M. Cossentino, M.P. Gleizes, P. Turci, and F. Zambonelli. A study of some Multi-Agent Meta-Models. In *Proceedings of the fourth international workshop on software engineering for large-scale multi-agent systems*, 2004.

[BCK99] L. Bass, P. Clements, and R. Kazman. Software architecture in practice. *Addison Wesley, ISBN 0-201-19930-0*, 1999.

[BCP05] C. Bernon, M. Cossentino, and J. Pavon. An Overview of Current Trends in European AOSE Research. *Informatica (Slovenia)*, pages 379–390, 2005.

[BD94] O. Boissier and Y. Demazeau. ASIC : An Architecture for Social and Individual Control and its Application to Computer Vision. *MAAMAW*, pages 135–149, 1994.

[BDD+02] P. Boulet, J.-L. Dekeyser, C. Dumoulin, Ph. Kajfasz, Ph. Marquet, and D. Ragot. Sophocles : Cyber-Enterprise for System-On-Chip Distributed Simulation. LIFL02, June 2002.

[Ber00] C. Berchet. *Modélisation pour la simulation d'un système d'aide au pilotage industriel.* PhD thesis, Thèse de Doctorat en Génie Industriel, INP de Grenoble, 2000.

[BFDP05] D. Balasubramaniam, P. Fabriani, R. Dindeleux, and F. Pourraz. Archware code synthesiser. ARCHWARE European RTD Project IST-2001-32360.Project Deliverable D6.4b, June 2005.

[BG01] J. Bézivin and O. Gerbé. Towards a precise definition of the omg/mda framework. In *ASE'01*, Novembre 2001.

[BGG04a] O. Boissier, S. Gitton, and P. Glize. *Caractéristiques des systèmes et des applications.* Observatoire Français des Techniques Avancées, Rapport de synthèse du Groupe Systèmes Multi-Agents, Février 2004.

[BGG+04b] P. Bresciani, P. Giorgini, F. Giunchiglia, J. Mylopoulos, and A. Perini. Tropos : An Agent-Oriented Software Development Methodology. . *Journal of Autonomous Agent and Multi-Agent Systems*, 2004.

[BGPG02] C. BERNON, M.-P. GLEIZES, G. PICARD, and P. GLIZE. The ADELFE Methodology For an Intranet System Design. In *Fourth International Bi-Conference Workshop on Agent-Oriented Information Systems (AOIS-2002)*, volume 57, Toronto, Canada, 2002. CEUR Workshop Proceedings.

[BIP88] M.E. Bratman, D.J. Israel, and Martha E. Pollack. Plans and resource-bounded practical reasoning. *Computational Intelligence*, 4(4) :349–355, 1988.

[BKJT97] F. M. T. Brazier, B. Dunin Keplicz, N. Jennings, and J. Treur. DESIRE : Modelling multi-agent systems in a compositional formal framework. *Int. J. Coop. Inform. Syst. 9(1)*, 1997.

[BKPPT00] P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer. Consistency checking and visualization of OCL constraints. In *The Unified Modeling Language Advancing the strandard, third international conference*, pages 294–308, 2000.

[BMO01] B. Bauer, J.P. Muller, and J. Odell. Agent UML : A Formalism for Specifying Multiagent Software Systems. *The International Journal of Software Engineering and Knowledge Engineering*, 2001.

[Boa95] M. Boasson. The Artistry of Software Architecture. *Guest editor's introduction,IEEE Software*, 1995.

[Boe76] B. W. Boehm. Software Engineering. *IEEE Transaction Computers*, 1976.

[Boe88] B. W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 1988.

[Boo92] G. Booch. *Conception orientée objets et applications.* Addison-Wesley, 1992.

[BPR99] F. Bellifemmine, A. Poggi, and G. Rimassa. JADE - A FIPA compliant agent framework. In *4th International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agents*, 1999.

[BPSM05] D. Bertolini, A. Perini, A. Susi, and H. Mouratidis. The Tropos visual modeling language. A MOF 1.4 compliant meta-model. Contribution for the AOSE TFG meeting, 2005.

[BRA87] M. E. BRATMAN. Intention, Plans, and Practical Reason. 1987.

[BRHL99] P. Busetta, R. Ronnquist, A. Hodgson, and A. Lucas. Jack intelligent agents - components for intelligent agents in java. AgentLink News Letter, 1999.

[BRJ00] G. Booch, J. Rumbaugh, and I. Jacobson. *Le guide de l'utilisateur UML*. 2000.

[Bro86] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 1986.

[BSE+03] F. Budinsky, D. Steinberg, R. Ellersick, E. Merkes, S.A. Brodsky, and T.J. Grose. Eclipse Modeling Framework. Addison Wesley, 2003.

[Béz04] J. Bézivin. In Search of a Basic Principle for Model-Driven Engineering. *Novatica Journal. Special Issue.*, 2004.

[Cam94] Valérie Camps. Une évaluation de la méthode de la relaxation restreinte pour l'auto-organisation dans les systèmes multi-agents. Master's thesis, IRIT -Toulouse, 1994.

[CBP05] M. Cossentino, C. Bernon, and J. Pavón. Modelling and Meta-modelling Issues in Agent Oriented Software Engineering : The AgentLink AOSE TFG Approach. Report of the AOSE TFG Ljubljana meeting, 2005.

[CCS04] A. Chella, M. Cossentino, and L. Sabatucci. Tools and patterns in designing multiagent systems with PASSI. In *6th WSEAS International Conference on Telecommunications and Informatics*, Cancun, Mexico, 2004.

[CCZ05] L. Cernuzzi, M. Cossentino, and F. Zambonelli. Process Models for Agent-based Development. *Journal of Engineering Applications of Artificial Intelligence*, 18, 2005.

[CDS+03] M. Clavel, F. Duran, S.Eker, P. Lincoln, N.Marti-Oliet, José Meseguer, and C.Talcott. The Maude 2.0 System. In *In Proc. Rewriting Techniques and Applications, 2003, Springer-Verlag LNCS 2706*, pages 76–87, June 2003.

[CGGS07] M. Cossentino, S. Gaglio, A. Garro, and V. Seidita. Method fragments for agent design methodologies : from standardization to research. *International Journal on Agent Oriented Software Engineering (IJAOSE)*, 2007.

[CGSS05] M. Cossentino, S. Gaglio, L. Sabatucci, and V. Seidita. The PASSI and Agile PASSI MAS Meta-Models Compared with a Unifying Proposal. In *Proceeding of the CEEMAS' 05 Conference*, pages 183–192., Budapest, Hungary, sept 2005.

[CH03] K. Czarnecki and S. Helsen. Classification of model transformation approaches. In *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[Che76] P. Chen. The entity relationship model - towards a unified view of data. *ACM Transactions on Database Systems*, pages 9–36, 1976.

[Cia96] P. Ciancarini. Coordination models and languages as software integrators. *ACM Computing Surveys*, 1996.

[CLC+01] G. Caire, F. Leal, P. Chainho, R. Evans, F. Garijo, J. Gomez, J. Pavon, P. Kearney, J. Stark, and P. Massonet. Agent-oriented analysis using message/uml, 2001.

[CNNL98] J. C. Collis, D. T. Ndumu, H. S. Nwana, and L.C. Lee. The ZEUS agent building toolkit. *BT Technology Journal, 16(3)*, 1998.

[COB+02] S. Cimpan, F. Oquendo, D. Balasubramaniam, G. Kirby, and R. Morrison. The ArchWare ADL : Definition of the Textual Concrete Syntax. ArchWare European RTD Project IST-2001-32360, Deliverable D1.2b, 2002.

[CP01] M. Cossentino and C. Potts. PASSI : a process for specifying and implementing Multi-Agent Systems Using UML, 2001.

[CSB05] L.R. Coutinho, J.S. Sichman, and O. Boissier. Modeling Organization in MAS : A comparison of Models. 2005.

[CT04] R. Cervenka and I. Trencansky. Agent Modeling Language : Language Specification. Technical report, Version 0.9. Technical report, Whitestein Technologies, 2004.

[CTCG05] R. Cervenka, I. Trencansky, M. Calisti, and D. Greenwood. AML : Agent Modeling Language. Toward Industry-Grade Agent-Based Modeling. In *In Odell, J., Giorgini, P., Muller, J., eds. : Agent-Oriented Software Engineering V : 5th International Workshop, AOSE 2004, Springer-Verlag*, 2005.

[CZ04] L. Cernuzzi and F. Zambonelli. Experiencing AUML with the Gaia Methodology. In *6th International Conference on Enterprise Information Systems*, Porto, April 2004.

[DCF95] A. Drogoul, B. Corbara., and D. Fresneau. Manta : New experimental results on the emergence of (artificial) ant societies. *Artificial Societies : the computer simulation of social life, N. Gilbert and R. Conte, eds., UCL Press*, 1995.

[Dem01] Y. Demazeau. Voyelles. Mémoire d'habilitation à diriger des recherches, INP Grenoble, 2001.

[Des00] P. Desfray. UML Profiles Versus Metamodel Extensions : An Ongoing Debate. In *OMG's UMLWorkshops : UML in the .com Enterprise : Modeling CORBA, Components, XML/XMI and Metadata Workshop*, 2000.

[DGS06] T.-L.-A. Dinh, O. Gerbé, and H. Sahraoui. Un méta-métamodèle pour la gestion de modèles. In *IDM 06 Actes des 2èmes Journées sur l'Ingénierie Dirigée par les Modèles*, Lille, France, 2006.

[DK75] F. DeRemer and H. Kron. Programming-in-the large versus programming-in-the-small. In ACM Press, editor, *In Proceeding of the International Conference on Reliable software*, pages 114–121, New York, NY, USA, 1975.

[dKLW97] M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge. A formal specification of dMARS. In *Agent Theories, Architectures, and Languages*, pages 155–176, 1997.

[DKW99] J.-C. Derniame, B. A. Kaba, and D.G. Wastell, editors. *Software Process : Principles, Methodology, Technology*, volume 1500 of *Lecture Notes in Computer Science*. Springer, 1999.

[Dro93] A. Drogoul. *De la simulation multi-agents à la résolution collective de problèmes. Une étude de l' émergence de structures d'organisation dans les systèmes multi-agents.* PhD thesis, University of Pierre et Marie Curie (Paris VI), 1993.

[DW03] K.H. Dam and M. Winikoff. Comparing agent-oriented methodologies. In AAMAS'03, editor, *Fifth International Bi-Conference Workshop on Agent-Oriented Information Systems (AOIS-2003)*, volume 3030, pages 78–93, Melbourne, Australia, 2003. Springer-Verlag.

[EQ96] B. Esfandiari and D. Quinqueton. An interface agent for network supervision. In *Proceedings of ECAI*, Budapest, 1996.

[EVI05] J. Estublier, G. Vega, and A. Ionita. Composing Domain-Specific Languages for Wide-scope Software Engineering Applications. In *MODELS*, 2005.

[Fav04] Jean-Marie Favre. Towards a Basic Theory to Model Driven Engineering. In *Workshop on Software Model Engineering, WISME*, 2004.

[Fer95] Jacques Ferber. *Les systèmes multi-agents : vers une intelligence collective.* Informatique, Intelligence Artificielle. InterÉditions, 1995.

[FG98] J. Ferber and O. Gutknecht. A meta-model for the analysis and design of organizations in multi-agent systems. In *In Proceedings of the third international conference on multi-agent systems.*, 1998.

[FG00] J. Ferber and O. Gutknecht. Madkit : A generic multi-agent platform. In *4 th International Conference on Autonomous Agents.*, 2000.

[FGM03] J. Ferber, O. Gutknecht, and F. Michel. From agents to organizations : An organizational view of multi-agent systems. In *AOSE*, pages 214–230, 2003.

[FGSP04] R. Fuentes, J. Gómez-Sanz, and J. Pavón. Towards requirements elicitation in multi-agent systems. In *Proc. Of the 17th European Meeting on Cybernetics and Systems Research*, 2004.

[FIP99] FIPA. FIPA Specification. rapport, 1999, FIPA Association, 1999.

[FLM95] T. Finin, Y. Labrou, and J. Mayfield. KQML as an agent communication language. *Jeff Bradshaw (Ed.), "Software Agents", MIT Press,Cambridge*, 1995.

[Fox81] M. Fox. An organizational view of distributed systems. *In IEEE Transactions on Systems*, 11 :70–80, 1981.

[Gar89] Hubert Garavel. *Compilation et vérification de programmes LOTOS.* PhD thesis, Université Joseph Fourier (Grenoble), 1989.

[Gar95] D. Garlan. What is style ? In *Proceedings of Dagshtul Workshop on Software Architecture*, 1995.

[GGG05] J-P. Georgé, M-P. Gleizes, and P. Glize. Basic approach to emergent programming - feasibility study for engineering adaptive systems using self-organizing instruction-agents. In LNCS 3910 Springer Verlag, editor, *The Third International Workshop on Engineering Self-Organising Applications (ESOA'05) at the Fourth International Joint Conference on Autonomous Agents & Multi-Agent Systems (AAMAS05)*, pages 16–30, Utrecht, Netherlands, July 2005.

[GHB00] M. Greaves, H. Holmback, and J. Bradshaw. What is a conversation policy ? *Issues in Agent Communication, F. Dignum, M. Greaves editors, LNAI 1916, Springer Verlag*, pages 118–131, 2000.

[GJM03] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering.* Second Edition, ISBN 0-13-305699-6, 2003.

[GKMP04] P. Giorgini, M. Kolp, J. Mylopoulos, and M. Pistore. The Tropos Methodology : An Overview. In *Methodologies and Software Engineering for Agent Systems, Kluwer*, 2004.

[GL87] M. P. Georgeff and A. L. Lansky. Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, pages 677–682, Seattle, 1987.

[GMMZ04] P. Giorgini, F. Massacci, J. Mylopoulos, and N. Zannone. Filling the gap between Requirements Engineering and Public Key/-Trust Management Infrastructures. In *Proceedings of the 1st European PKI Workshop : Research and Applications (1st EuroPKI), LNCS 3093*, 2004.

[GMP02] F. Giunchiglia, J. Mylopoulos, and A. Perini. The tropos software development methodology : Processes, models and diagrams. In *AOSE 02*, pages 162–173, 2002.

[GMP03] M.-P. Gleizes, T. Millan, and G. Picard. ADELFE : Using SPEM Notation to Unify Agent Engineering Processes and Methodology. Rapport interne IRIT/2003-10-R, Institut de Recherche en Informatique de Toulouse (IRIT), 2003.

[GMW00] D. Garlan, R.T. Monroe, and D. Wile. Acme : Architectural description of component-based systems. *In Gary T. Leavens and Murali Sitaraman, editors, Foundations of Component-Based Systems, Cambridge University Press*, pages 47–68, 2000.

[Gro00] IEEE Architecture Working Group. Recommended practice for architectural description of software-intensive systems. Technical report, IEEE, 2000.

[Gro02] Object Management Group. OMG : Software Process Engineering Metamodel Specification Version 1.0. Rapport technique Version 1.0, formal/02-11-14, 2002.

[GS93] D. Garlan and M. Shaw. Introduction to software architecture. In World Scientific Publishing Company, editor, *Advances in Software Engineering and Knowledge Engineering*, 1993.

[Hab68] J. Habermas. The Idea of the Theory of Knowledge as Social Theory. Knowledge & Human Interest, 1968.

[Hab01] G. Habchi. Conceptualisation et modélisation pour la simulation des systèmes de production. HDR, ESIA, Université de Savoie., 2001.

[HHP+07] G. Habchi, M.-P. Huget, M. Pralus, S. Azaiez, J. Tounsi, and K. Tamani. Modélisation système multi-agents (sma) du processus de pilotage d'un système de fusion d'informations appliqué à un système de production. Technical report, laboratoire LISTIC, 2007.

[HOB04] M.-P. Huget, J. Odell, and B. Bauer. The AUML Approach. *volume 11, chapter 1. Kluwer*, 2004.

[HSB02] J.F Hübner, J.S Sichman, and O. Boissier. Spécification structurelle, fonctionnelle et déontique d'organisations dans les systèmes multi-agents. JFIADSMA 02, 2002.

[HSE06] J. Höbler, M. Soden, and H. Eichler. Coevolution of Models, Metamodels and Transformations. http ://www.ikv.de/index.php, 2006.

[Hug05] M.-P. Huget. Agent Communication. In Intelligent Decision Support Systems in Agent-Mediated Environments. *Frontiers in Artificial Intelligence and Applications. Gloria Phillips-Wren and Lakhmi Jain (eds.), IOS Press*, 115, 2005.

[IEV05] A. D. Ionita, J. Estublier, and G. Vega. Domaines réutilisables dirigés par les modèles. In *Proc. Of Ingénierie dirigée par les modèles, IDM05*, Paris, 2005.

[IGG99] C.A. Iglesias, M. Garijo, and J.C. Gonzalez. A survey of agent-oriented methodologies. In *Proceeding of the Fifth International Workshop on Agent Theories Architectures, and Languages*, "Budapest, Hungary", sept 1999.

[Jac93] I. Jacobson. Object-Oriented Software Engineering - a Use Case Driven Approach. *TOOLS*, 1993.

[JFL+01] N. R. Jennings, P. Faratin, A.R. Lomuscio, S. Parsons, C. Sierra, and M. Wooldridge. Automated negotiation : prospects, methods and challenge. *International Journal of Group Decision and Negotiation (GDN)*, 10 :99–215, 2001.

[JGB06] J.-M. Jézéquel, S. Gérard, and B. Baudry. Le génie logiciel et l'idm : une approche unificatrice par les modèles. *L'ingénierie dirigée par les modèles, Lavoisier, Hermes-science*, 2006.

[JK06] F. Jouault and I. Kurtev. On the architectural alignment of atl and qvt. In *Proceedings of ACM Symposium on Applied Computing (SAC 06), Model Transformation Track*, Dijon (Bourgogne, FRA), April 2006.

[JPSF00] N. R. Jennings, S. Parsons, C. Sierra, and P. Faratin. Automated negotiation. In *5th International Conference on The Practical Application of Intelligent Agent and Multi-Agent Systems(PAAM-2000)*, Manchester, UK, 2000.

[JSCK04] G. Jack, K. Short, S. Cook, and S. Kent. *Software Factories : Assembling Applications with Patterns, Models, Frameworks, and Tools*. West Sussex, England : John Wiley & Sons, Ltd., 2004.

[JSHL07] H. Jonkers, M. Steen, L. Heerink, and D. Van Leeuwen. From Business Process Design to Code : Model-Driven Development of Enterprise Applications. https ://doc.freeband.nl/dsweb/Get/Document-77839/, 2007.

[JSW98] N. R. Jennings, K. Sycara, and M. Wooldridge. A Roadmap of Agent Research and Development. *Journal of Autonomous Agents and Multi-Agent Systems*, 1998.

[Kad05] H. Kadima. *MDA - conception orientée objet guidée par les modèles*. NFE114, 2005.

[KK99] M. Klein and R. Kazman. Attribute-based architectural styles. Technical report, CMU/SEI-99-TR-022. ESC-TR-99-022, 1999.

[Koz83] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 1983.

[KPR03] R. Kazhamiakin, M. Pistore, and M. Roveri. T-tool tutorial. Technical report, University of Trento, 2003.

[Kru95] P. Kruchten. The 4+1 view model of architecture. *IEEE Softw*, 1995.

[KWB03] A. Kleppe, J. Wagner, and W. Bast. *MDA Explained : The Model Driven Architecture : Practise and Promise.* Addison-Wesley., 2003.

[Lar94] P. Larvet. Analyse des systèmes : de l'approche fonctionnelle à lf'approche objet. *InterEditions.*, 1994.

[LBM+01] A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, and J. Sprinkle. Composing domain-specific design environments. *Computer Networks and ISDN Systems*, pages 44–51, 2001.

[Ley04] F. Leymonerie. *ASL : un langage et des outils pour les styles architecturaux. Contribution à la description d'architectures dynamiques.* PhD thesis, LISTIC, Université de Savoie, Annecy, 2004.

[LL92] M. Ljunberg and A. Lucas. The OASIS air traffic management system. In *In Proceedings of the Second Pacific Rim International Conference on AI (PRICAI-92)*, Seoul, Korea, 1992.

[LL93] S. E. Lander and R. Lesser. Understanding the role of negotiation in distributed search among heterogeneous agents. In *International Joint Conference on Artificial Intelligence (IJCAI-93)*, 1993.

[LO98] D.B. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets.* Addison Wesley, 1998.

[Mal87] T. Malone. Modeling coordination in organizations and markets. *In Management science*, 33, 1987.

[MBWH03] P. Marrow, E. Bonsma, F. Wang, and C. Hoile. DIET - A Scalable, Robust and Adaptable Multi-Agent Platform for Information Management. *BT Technology Journal*, 21(4) :130–137, 2003.

[MC97] P. Marcenac and S. Calderoni. Self-organisation in agent-based simulation. In *Proceedings of MAAMW' 97*, Ronnedy, Sweden, 1997.

[MCG05] T. Mens, K. Czarnecki, and P. Van Gorp. A taxonomy of model transformations. In *Dagstuhl Seminar Proceedings 04101*, 2005.

[MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. 1995.

[Meg04] K. Megzari. REFINER : environnement logiciel pour le raffinement d'architectures logicielles fondé sur une logique de réécriture. Thèse de Doctorat de l'Université de Savoie, 2004.

[Mil99] R. Milner. Communicating and mobile systems : The π-calculus. *Cambridge University Press.*, 1999.

[MKMG97] R.T. Monroe, A. Kompanek, R. Melton, and D. Garlan. Architectural styles, design patterns, and objects. In *Proceedings of the IEEE Transactions on Software Enginering*, 1997.

[Mül02] J.P. Müller. *Des systèmes autonomes aux systèmes multi-agents : Interaction, émergence et systèmes complexes.* PhD thesis, Université Montpellier II, 2002.

[MP05] P. Mathieu and S. Picault. Towards an interaction-based design of behaviors. In *The Third European Workshop on Multi-Agent Systems*, 2005.

[MPS02] P. Moraitis, E. Petraki, and N.I. Spanoudakis. Engineering jade agents with gaia methodology. In *Agent Technologies, Infrastructures, Tools, and Applications for E-Services*, pages 77–92, 2002.

[MRRR02] Nenad Medvidovic, David S. Rosenblum, David F. Redmiles, and Jason E. Robbins. Modeling software architectures in the unified modeling language. In *ACM Transactions On Software Engineering and Methodology*, 2002.

[MRT99] N. Medvidovic, D.S Rosemblum, and R. Taylor. A language and environment for architecture-based software development and evolution. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, Los Angeles, Mai 1999.

[MT00] N. Medvidovic and N R. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, pages 70–93, 2000.

[MT01] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 2001.

[NR99] E. Di Nitto and D. Rosenblum. Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructures. *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, 1999.

[OACV02] F. Oquendo, I. Alloui, S. Cîmpan, and H. Verjus. The ArchWare ADL : Definition of the Abstract Syntax and Formal Semantics. ARCHWARE European RTD Project IST-2001-32360. Deliverable D1.1b, décembre 2002.

[OBDK01] M. Occello, C. Baeijs, Y. Demazeau, and J.L. Koning. Mask : An aeio toolbox to develop multi-agent systems. *Knowledge Engineering and Agent Technology, Cuena, Demazeau, Garcia, Treur eds, IOS Series on Frontiers in Artificiel Intelligence and Applications*, 2001.

[Ode02] J. Odell. Objects and Agents Compared. *Journal of Object Technology*, 2002.

[OJ96] G. M. P. O'Hare and N. R. Jennings. Foundations of Distributed Artificial Intelligence. *Wiley-Interscience*, 1996.

[OMG03] OMG. MDA GUIDE Version 1.0.1. document number omg/2003-06-01., 2003.

[OMG04] OMG. Uml 2 metamodel. ptc/04-10-05 (UML 2.0 Superstructure FTF Rose model containing the UML 2 metamodel), 2004.

[OMG05] OMG. MOF QVT Final Adopted Specification. OMG document, november 2005.

[OMG06] OMG. Meta Object Facility Core Specification version 2.0. OMG Document, january 2006.

[Oqu03] F. Oquendo. The ArchWare Refinement Language. Deliverable D6.1b. ArchWare European RTD Project. IST-2001-32360, 2003.

[Pav06] Juan Pavón. Ingenias : Développement dirigé par modèles des systèmes multi-agents. HDR Université Paris 6, 2006.

[PBL05] A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex : A bdi reasoning engine, chapter of multi-agent programming. *Kluwer Book, Editors : R. Bordini, M. Dastani, J. Dix and A. Seghrouchni*, 2005.

[PGS03] J. Pavon and J. Gomez-Sanz. Agent-oriented Software Engineering with INGENIAS. In *3rd Central and Eastern European Conference on Multi-Agent Systems (CEEMAS'03)*, Prague, Czech Republic., 2003.

[Pic04] G. Picard. *Méthodologie de développement de systèmes multi-agents adaptatifs et conception de logiciels à fonctionnalité émergente.* Institut de Recherche en Informatique de Toulouse(IRIT), décembre 2004.

[PPRS03] A. Perini, M. Pistore, M. Roveri, and A. Susi. Agent-oriented modeling by interleaving formal and informal specification. *Agent Oriented Software Engineering - AOSE*, 2003.

[PVA+05] F. Pourraz, H. Verjus, S. Azaiez, F. Oquendo, and C. Zavattari. Extended ArchWare Architecture Animator - Release 2.1. ARCHWARE European RTD Project IST-2001-32360. Deliverable D2.2d, avril 2005.

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes*, 17(4) :40–52, 1992.

[RB98] J. Rouchier and F. Bousquet. Non-merchant economy and multi-agent system : an analysis of structuring exchanges. Number 111–12. Springer Verlag, 1998.

[RBE+95] J. Rumbaugh, M. Blaha, F. Eddy, W. Premerlani, and W. Lorensen. OMT. Modélisation et conception orientées objet. *Masson Paris and Prentice Hall London*, 1995.

[RM01] J.-C. Routier and P. Mathieu. Magique : Agents, compétences et hiérarchies. *Revue de Technique et Science Informatiques (TSI)*, 2001.

[Rod94] M. Rodriguez. *Modélisation d'un agent autonome : Approche constructiviste de l'architecture de contrôle et de la représentation de connaissances*. PhD thesis, Université de Neufchâtel., 1994.

[SBPL06] J. Sudeikat, L. Braubach, A. Pokahr, and W. Lamersdorf. Validation of BDI Agents. In *The Fifth International Workshop on Programming Multiagent Systems (PROMAS-2006)*, 2006.

[SDS03] A. Sturm, D. Dori, and O. Shehory. Single-model method for specifying multi-agent systems. In *Proceedings of AAMAS'03*, 2003.

[Sea69] J.R. Searle. Speech acts. *Cambridge University Press, UK*, 1969.

[Sei03] E. Seidewitz. What models mean. *IEEE Software*, 2003.

[SMA06] http ://smartqvt.elibel.tm.fr/, 2006.

[Smi88] R.G. Smith. The contract net protocol : High-level communication and control in a distributed problem solver. *Readings in Distributed Artificial Intelligence, Bond A. H. and Gasser L.(Eds.)*, pages 357–366, 1988.

[Spi92] J. M. Spivey. The Z Notation : A Reference Manual. Prentice Hall International Series in Computer Science., 1992.

[SPM04] R. Sebastiani, P.Giorgini, and J. Mylopoulos. Simple and minimum-cost satisfiabilityfor goal models. In *Proceedings of the 16th Conference On AdvancedInformation Systems Engineering (CAiSE 04), LNCS Springer*, 2004.

[SRB06] D.R. Dos Santos, M. Blois Ribeiro, and R. Melo Bastos. A comparative study of multi-agent systems development methodologies. In *Second Workshop on Software Engineering for Agent-Oriented Systems*, pages 37–48, Florianopolis, 2006.

[SS03] A. Sturm and O. Shehory. A framework for evaluating agent-oriented methodologies. In *Fifth International Bi-Conference Workshop on Agent-Oriented Information Systems (AOIS-2003)*, volume 3030, pages 94–109, Melbourne, Australia, 2003. Springer-Verlag.

[Str95] E. Le Strugeon. Une méthodologie d'auto-adaptation d'un système multi-agents cognitifs. *Thèse de doctorat, Université de Valanciennes*, 1995.

[TBH07] J. Tounsi, J. Boissière, and G. Habchi. *Modélisation pour la simulation des fonctions de management de la chaine logistique globale dans un environnement de production PME Mécatronique.* PhD thesis, Université de Savoie - thèse en cours, 2007.

[Ves92] S. Vestal. Metah reference manual. Technical report, Honeywell Technology Center Minneapolis MN, 1992.

[Ves94] S. Vestal. Mode changes in real-time architecture description language. In *Proceedings of the Second International Workshop on Configurable Distributed Systems*, 1994.

[VSDD04] J. Vazquez-Salceda, V. Dignum, , and F. Dignum. Organizing multiagent systems. Technical Report UU-CS-2004-015, Institute of Information and Computing Sciences, Utrecht University, 2004.

[WH05] E. Woods and R. Hilliard. Architecture Description Languages in Practice Session Report. *Software Architecture, 2005. WICSA 2005. 5th Working IEEE/IFIP Conference*, pages 243–246, 2005.

[WJK00] M. J. Wooldridge, N.R. Jennings, and D. Kinny. The Gaia Methodology for Agent-Oriented Analysis and Design. In *Autonomous Agents and Multi-Agent systems*, volume volume 3, pages 285–312, "The Netherlands", 2000.

[WT03] G. Wagner and F. Tulba. Agent-Oriented Modeling and Agent-Based Simulation. In *Proceedings of 5th Int. Workshop on Agent-Oriented Information Systems (AOIS-2003), ER2003 Workshops, Springer-Verlag, LNCS.* In P. Giorgini and B. Henderson-Sellers (Eds.), 2003.

[YM94] E. Yu and J. Mylopoulos. Towards Modelling Strategic Actor Relationships for Information Systems Development - With Examples from Business Process Reengineering. In *Proceedings of the 4th Workshop on Information Technologies and Systems*, pages 21–28, 1994.

[ZJW03] F. Zambonelli, N. Jennings, and M. Wooldridge. Developing multiagent systems : The Gaia methodology. *ACM Trans. Software Eng. Meth. 12(3)*, pages 417 – 470, 2003.

[ZMZY05] W. Zhang, H. Meiand, H. Zhao, and J. Yang. Transformation from CIM to PIM : A Feature-Oriented Component-Based Approach. In *Model Driven Engineering Languages and Systems, 8th Intenational Conference MoDELS*, Montego Bay, Jamaica, 2005.

Résumé

Les systèmes multi-agents s'attaquent aux nombreuses problématiques posées actuellement dans le monde informatique telles que la distribution, l'évolution, l'adaptabilité et l'interopérabilité des systèmes. Les solutions proposées par ces systèmes sont prometteuses et permettent d'obtenir des systèmes flexibles et évolutifs. Cependant, leur mise en oeuvre reste difficile. Ceci est dû au manque de techniques d'ingénierie adaptées à ce genre de système et qui permettent un développement fiable et cohérent.

Bien qu'il existe plusieurs propositions intéressantes au niveau des méthodologies, des langages de spécification et des plates-formes d'implémentation orientés agent, celles-ci manquent de cohésion et font ressortir plusieurs différences aussi bien au niveau de la sémantique des concepts utilisés mais aussi au niveau des démarches de développement.

Notre but durant cette thèse a été de proposer une approche flexible et cohérente supportant le développement des systèmes multiagents. Cette approche que nous baptisons ArchMDE se base sur une combinaison de l'approche centrée architecture et de l'approche dirigée par les modèles. L'approche centrée architecture nous permet de raisonner sur les éléments qui structurent le système multi-agents ainsi que leurs interactions. Elle permet d'identifier les patrons architecturaux nécessaires au développement des systèmes multi-agents en prenant en compte les différentes vues du système (vue organisationnelle, vue environnementale, etc.). L'approche orientée modèles nous permet d'exprimer de façon explicite la manière de combiner ces patrons architecturaux afin d'avoir une représentation globale du système multi-agents. D'autre part, IDM permet de couvrir les différentes phases du cycle de développement en adoptant une démarche basée sur la transformation de modèles. Cette démarche permet de garantir la cohérence du système durant les différentes phases du cycle de vie. Par ailleurs, celle-ci offre l'avantage de préserver le savoir-faire des développeurs en exprimant explicitement les opérations d'intégration (entre les patrons architecturaux) et de mapping (entre les modèles de conception et les modèles d'implémentation).

Pour implanter ArchMDE, nous utilisons le cadre de développement ArchWare qui est basé sur le π-calcul typé, polyadique et d'ordre supérieur, ce qui permet de supporter les aspects communicatifs et évolutifs des systèmes multi-agents. Le choix d'un cadre formel vise à réduire l'ambiguïté liée aux concepts multi-agents mais aussi à garantir une conception sûre. En effet, l'utilisation d'un langage formel donne la possibilité d'exprimer explicitement différentes propriétés structurelles et comportementales. Le cadre de développement ArchWare offre divers langages accompagnés de différents outils qui nous seront utiles pour mettre en oeuvre notre approche.