REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITE 20 AOUT 1955 − SKIKDA

Faculté des Sciences

Département d'Informatique

Ecole Doctorale de l'Est – Pôle ANNABA

MEMOIRE

Présenté en vue de l'obtention du diplôme de MAGISTER en INFORMATIQUE

Ecole Doctorale en Informatique de l'Est – EDI Est

Option : GENIE LOGICIEL

Approche agent pour l'architecture logicielle

Par

Hanene BOURRICH

Composition du jury

Président	Dr S. MAAZOUZI	MC A, Université 20 Août 1955 − Skikda
Rapporteur	Dr R. MAAMRI	MC A, Université Mentouri − Constantine
Examinateurs	Dr B. BOUCHEHAM	MC A, Université 20 Août 1955 − Skikda
	Dr M. REDJIMI	MC A, Université 20 Août 1955 − Skikda

Je dédie ce travail
À mes parents pour leur amour et leur soutien
A mes frères et mes sœurs
A mes nièces

Remerciement

J’adresse mes profonds remerciements à tous ceux qui ont contribué de près ou de loin à la réalisation de ce mémoire.

Je tiens tout d’abord à remercier Dr MAAMRI RAMDANE, d’avoir accepté d’encadrer ce travail ainsi que pour sa patience et pour les multiples chances qu’il m’a offert pour accomplir ce travail.

Je remercie Dr Smain Maazouzi, Dr Bachir Boucheham et Dr Mohamed Redjimi d’avoir bien voulu accepter d’être membres du Jury de ce mémoire et de l’intérêt qu’ils portent pour ce travail.

Ma reconnaissance s’adresse spécialement à M^{lle} Boukerma Hanene pour son amour, aide et soutien.

J’aimerais remercier particulièrement mes chers parents et mes chers frères et sœurs, ainsi que toute ma famille pour leur amour, soutien et précieuse aide. Merci également à toutes mes amies. Je tiens aussi à remercier mes responsables de travail ainsi que mes collègues pour leur compréhension et encouragement.

Table des matières

Liste des figures

Liste des tables

الملخص

في عالم هندسة البرمجيات، تعتبر النظم متعددة الوكلاء عموما نهجا جديدا وغير واسع الانتشار لهندسة النظم المعقدة، من جهة أخرى تعد معمارية البرامج أداة للتحكم في التعقيد وتحقيق الجودة المطلوبة من قبل النظام مقبولة على نطاق واسع في المجال الصناعي. إزدواجية الوكيل باعتباره كيان مستقل و قابل للتكيف و معمارية البرامج كوسيلة للتنظيم و الولوج للمجال الصناعي يقدم في الواقع إطارا مميزا جدا لمعالجة قضايا المعلوماتية متزايدة الحركية والموزعة للتطبيقات المستقبلية. ونحن مهتمون في هذا العمل باستخدام شبكات بتري كمنهج للتوصيف الرياضي للمعماريات البرمجية المبنية باستخدام وكلاء.

كلمات دلالية: *المعمارية البرمجية، الأنظمة متعددة الوكلاء، شبكات بيتري.*

Résumé

Dans le monde du génie logiciel, les systèmes multi-agents sont généralement considérés comme une nouvelle approche pour l'ingénierie des systèmes complexes qui n'est pas encore très répandue, d'autre part l'architecture logicielle est considérée comme un véhicule primaire pour la maîtrise de la complexité et l'atteinte des qualités exigées par le système largement acceptée dans l'industrie. La dualité entre agent comme entité autonome et adaptative et l'architecture logicielle comme organisation et moyen pour l'adoption industrielle, offre en effet un cadre tout à fait privilégié pour aborder les enjeux de l'informatique de plus en plus dynamique et distribuée des applications du futurs. On s'intéresse dans ce travail à l'utilisation des réseaux de Petri pour la description formelle des architectures logicielles à base d'agents.

Mot clés : Architecture logicielle, Systèmes multi-agents, Réseaux de Petri.

Abstract

In the realm of software engineering, multi-agent systems are generally considered as a new approach for engineering complex systems that is not yet widespread; on the other hand software architecture is a widely accepted practice in the industry that is considered as primary vehicle for mastering complexity and achieving required system qualities. Duality between agent as an autonomous and adaptive entity and software architecture as an organization and a means for industrial adoption, indeed offers a quite privileged framework to address the issues of increasingly dynamic and distributed informatics of future applications. We are interested in this work in the use of Petri nets for the formal description of agent-based software architectures.

Keywords: Software architecture, Multiagents systems, Petri nets

Chapitre 1

Introduction

L'architecture logicielle comprend les structures du système, qui comportent les éléments du logiciel, les propriétés extérieurement visibles de ces éléments, et les rapports entre eux. Les éléments de logiciel fournissent la fonctionnalité du système, alors que les qualités requises du système sont principalement atteintes par les structures de l'architecture logicielle .Les systèmes multi-agents fournissent une approche pour résoudre un problème en décomposant le système en un certain nombre d'entités autonomes dans un environnement, agissant sur ce dernier et interagissant entre eux afin de réaliser les besoins fonctionnels et de qualité du système. Donc il y a une connexion claire entre les systèmes multi-agents et l'architecture logicielle. Plusieurs travaux ont tenté d'explorer cette connexion. Partant de la convergence ainsi que la complémentarité qui existe entre les deux domaines, dans notre travail nous proposons une description architecturale formelle des systèmes multi-agents.

Bien que le concept d'agent n'est pas récent et date des années 5Os, le premier workshop consacré aux méthodologies orientées agent a été organisé en 2000 et leur utilisation n'est pas très répondue, c'est la raison pour laquelle la majorité d'architectures à base d'agents sont trop dépendantes à la connaissance du domaine et leur généralité est très limitée. Donc en accentuant le perspective d'architecture logicielle pour les systèmes multi-agents nous allons permettre un cadre pour une description générale des architectures d'applications à base d'agents ce qui devra favoriser leur utilisation et par conséquent leur maturité et développement.

Les méthodes formelles sont connues par leur capacité à fournir des descriptions rigoureuses et précises, dans ce travail nous proposons une description et une validation formelle des architectures des systèmes multi-agents. Les réseaux de Petri sont des outils à la fois graphiques et mathématiques permettant de modéliser le comportement dynamique des systèmes à évènements discrets. Leur représentation graphique permet de visualiser d'une manière naturelle le parallélisme, la synchronisation, le partage de ressources, les choix (conflits), …etc. Leur représentation mathématique permet d'analyser le modèle pour étudier ses propriétés et de les comparer avec le comportement du système réel.

Dans ce mémoire nous allons tout d'abord dans le deuxième chapitre présenter le domaine de l'architecture logicielle en s'intéressant à une activité importante dans l'architecture logicielle qui est la description de l'architecture.

Le troisième chapitre à été consacré à l'agent et les systèmes multi-agents ainsi que toutes les notions qui découlent de ce domaine.

Dans le quatrième chapitre nous avons essayé d'investiguer la connexion qui existe entre le domaine d'architecture logicielle et les systèmes multi-agents en présentant les travaux considérant les systèmes multi-agents du point de vue de l'architecture logicielle ainsi que les formalismes pour la description de cette dernière.

Le cinquième chapitre introduit notre contribution à ce nouveau domaine en proposant notre approche basée réseaux de Petri pour la description architecturale des systèmes multi-agents.

Finalement, dans le sixième chapitre nous présentons nos conclusions et nos perspectives pour la poursuite de ce travail.

Chapitre 2

L’architecture logicielle

2.1 Introduction

L'architecture logicielle est un domaine assez récent du génie logiciel qui est devenu un champ à part entière, dont le but est de trouver les meilleures solutions pour garantir la stabilité et la performance des logiciels. L'accent est particulièrement mis sur des phases de tests dès les premières étapes. Des workshops et des conférences spécialisées tels que EWSA (European Conference on Software Architectures) et CAL (Conférence francophone sur les Architectures Logicielles) font maintenant le point sur ce domaine. Ce premier chapitre va introduire le domaine de l’architecture logicielle en présentant ses différentes définitions ainsi que les formalismes utilisés pour la description architecturales.

2.2 Historique

C’est l’accroissement de la complexité des logiciels qui a amené progressivement à repenser la façon de construire les systèmes logiciels. Les concepts qui ont fait naître l’architecture logicielle remontent aux débuts de la discipline du génie logiciel.

- Une première étude de la structure d’un logiciel a été faite par E. Dijkstra , où il a montré qu’il été avantageux de s’intéresser à la structure d’un programme plutôt qu’à sa programmation [Dij68], de même les travaux de cet auteur mettent en évidence le besoin de la notion de « niveaux d’abstraction » dans la conception des systèmes à grande taille.

- Les travaux de Parnas [Par72] aux débuts des années 70, ont introduit le concept de « Modularité », qui permet d'améliorer la flexibilité et la compréhension.

C'est incontestablement dans les années 90s que les architectures logicielles commencent à avoir un essor important. Les idées de base telles que donner de l'importance aux décisions prises dans les premières étapes du développement du système ainsi que de l'importance d'une définition correcte de la structure du système ont joué un rôle déterminant dans l'évolution de la discipline.

2.3. Définitions

Le terme d'"Architecture Logicielle" est probablement l'un des plus surexploité et des plus surchargé en génie logiciel à titre d'exemple, le SEI (Software Engineering Institute) [SEI] a documenté et catalogué des centaines de définitions, malgré ça, jusqu'à maintenant li n'y a pas une définition de l'architecture logicielle qui soit acceptée par tous et aucune des définitions proposées ne s'est vraiment imposée. Cependant les chercheurs sont d'accord pour dire qu'une architecture logicielle décrit les structures de haut niveau d'un système.

Nous allons maintenant parcourir les principales définitions dans le domaine et les présentant dans l'ordre chronologique de leur apparition:

2.3.1 Perry et Wolf 1992

L'architecture logicielle est un domaine qui a été formalisé au début des années 90, avec l'apparition de systèmes d'information toujours plus gros et plus complexes à développer. Perry & Wolf, dans cet article considéré comme un article fondateur de l'architecture logicielle en tant qu'un domaine à part entière du génie logiciel, ont proposé les notions suivantes :

Un modèle de l'architecture logicielle : par analogie à l'architecture des bâtiments les auteurs ont définit leur modèle par un triplet :

"Software Architecture= {Elements, Form, Rationale}" [PW92]

- **Les éléments architecturaux:** sont divisés en trois catégories: les éléments de données contenant les données à utiliser et à transformer, les éléments de traitement qui manipulent les données stoker dans les éléments de données et les éléments de connexion qui relient les éléments précédents.

- **La forme architecturale :** correspond à des propriétés et des relations pondérées. Les propriétés définissent des contraintes sur les éléments pris individuellement. Les relations définissent des contraintes sur la disposition des éléments entre eux. La pondération des contraintes permet soit d'indiquer l'importance de la propriété ou la relation, soit la nécessité de choisir entre plusieurs alternatives.

- **Les critères de choix :** capturent les motivations de l'architecte pour le choix d'une architecture plutôt qu'une autre, ces critères varient des aspects fonctionnels de base jusqu'aux différents aspects non fonctionnels qui peuvent être de nature économique (temps…), technique (performance…), liées à la conception (évolution…)…

La notion de style architectural : il définissent un style architectural comme "ce qui abstrait les éléments et les aspects formels de diverses architectures spécifiques". Le style architectural représente un regroupement des caractéristiques et des décisions de conception communes pour des architectures similaires. Les auteurs ont cité les différences entre une architecture logicielle et un style architectural, une différence parmi autres est qu'un style architectural et moins contraint et moins complet qu'une architecture, leur but était de montrer qu'un style architectural n'est pas une architecture bien que la distinction entre les deux n'est pas évidente.

Des vues architecturales : il permettent de représenter différents aspects de l'architecture, ces vues sont interdépendantes et les auteurs proposent trois vues correspondant aux trois types d'éléments qui sont la vue des données, la vue transformation et la vue de connexion.

L'analyse architecturale: en plus de fournir une documentation précise et claire, les spécifications ont pour objectif premier de fournir des analyses automatiques de ces documents et de faire apparaître divers problèmes. Les auteurs listent deux catégories d'analyse : les analyses de cohésion et les analyses de couplage.

2.3.2 Garlan et Perry 1995

"The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time." [GP95]

Cette définition a été formulée en 1994 par un groupe de travail de discussion au Software Engineering Institute de l'université de Carniege Mellon. Les auteurs pensaient que c'étaient un bon point de départ en notant que cette définiion et les définitions similaires ne traitent qu'une partie de l'architecture logicielle. Les auteurs ont montré des spécifités de la'rchitecture logicielle qui sont les suivants :

Les préoccupations : le développement et la maintenance des gros systèmes ont faits émerger des nouvelles préoccupations autres que celles des méthodes trditionnelles tel que le choix des structures de données ou des algorithmes. L'architecture logicielle tente de fournir des notations et des outils pour l'étude de l'organisation des structures de haut niveau d'un système.

La description du système : la description du système en terme de définition/ utilisation ou chaque module définit un ensemble de services disponibles pour les autres modules et de services qu' ils utilisent et qui sont fournis par les autres modules, tant que la description architecturale décrit le système comme une configuration de composants et de connecteurs où les composants représentent les unités de calcul et de stokage de données du système et les connecteurs définissent l'interaction entre les différents composants.

Méthodes de conception vs Architecture logicielle : bien que les méthodes de conception et l'architecture logicielle tentent de réduire l'écart entre les besoins et

l'implémentation, il existe une différence importante (cf. figure 2.1). Sans une méthode de conception ou une discipline de l'architecture logicielle, l'implémentation est laissée au developpeur, une solution peut être obtenue en utilisant des techniques ad-hoc qui sont à la portée (cf. figure 2.1(a)). Les méthodes de conception améliorent la situation en fournissant un chemein entre quelques classes de besoins du système et quelques classes d'implémentation (cf. figure2.1(b)). L'architecture est concernée par les compromis entre différents choix architecturaux pour leur capacité à résoudre certains types de problèmes (cf. figure 2.1(c)). L'architecture logicielle et les méthodes de conception sont des concepts différents mais ils sont complémentaires.

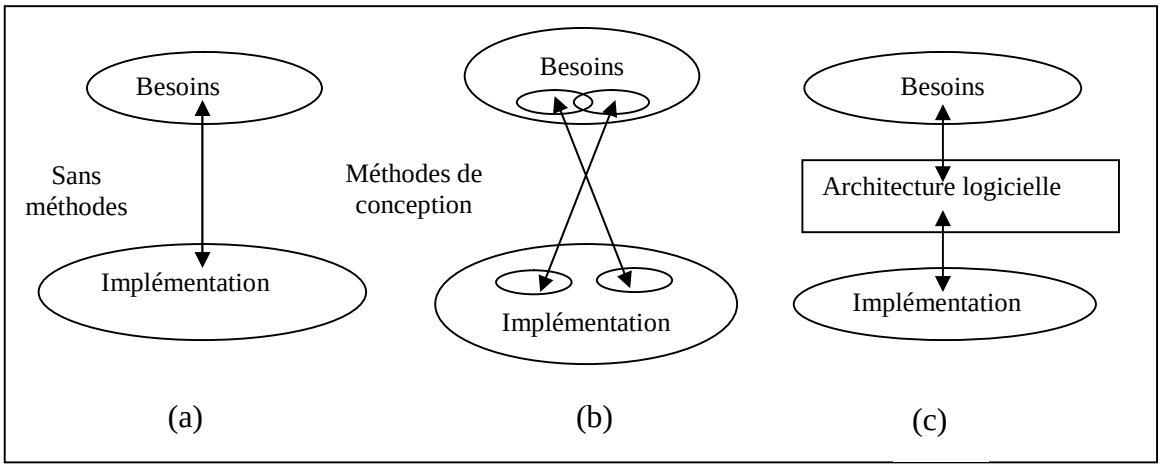


FIG. 2.1 – Méthodes de conception et architecture logicielle [GP95]

2.3.3 Gacek, Abdallah, Clark et Boehm 1995

"*A software architecture comprises :*

- *A collection of software and system components, connections and constraints.*
- *A collection of system stakeholders' need statements.*
- *A rationale which demonstrates that the components, connections, and constraints define a system that, if implemented, would satisfy the collection of system stakeholders' need statements*" [GACB95]

Dans cet article les auteurs ont présenté une nouvelle définition de l'architecture logicielle. Ils considèrent que les définitions précédentes se focalisent sur les représentations architecturales et ne permettent pas de déterminer si un ensemble de diagrammes est une architecture ou non ? Donc ils proposent une définition de l'architecture logicielle qui fournit un ensemble de critères qui permettent de faire cette détermination. Pour cela ils préconisent de prendre en compte les critères de choix d'une architecture au même titre que la structure d'une architecture. Ces critères de choix doivent assurer que les composants, connexions et contraintes définissent un système qui satisfait les besoins des différents acteurs.

Les auteurs considèrent que l'architecture logicielle est au centre du processus du cycle de vie. Ils proposent que l'architecture logicielle puisse servir de référence tout au long du cycle de vie (cf. figure 2.2). Pour cela elle doit pouvoir satisfaire l'ensemble les besoins des différents acteurs. Donc la description de l'architecture doit comprendre des vues multiples correspondant aux besoins des différents auteurs. Concernant les notations utilisées, les auteures préconisent l'utilisation des méthodes formelles non seulement pour la description de la structure statique mais aussi pour la description d'autres aspects du système.

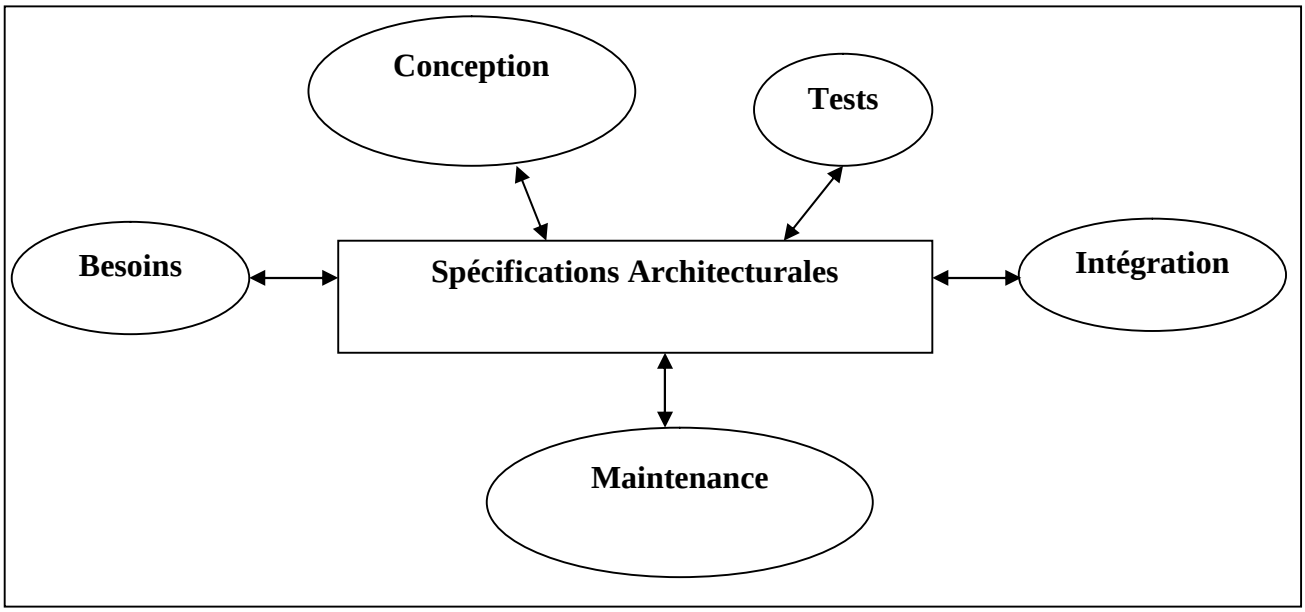


FIG. 2.2 – Description architecturale et cycle de vie [GACB95]

2.3.4 Bass, Clements et Kazman 1998

"A software architecture of a program or a computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them" [BCK98]

Ce livre fut le deuxième après celui de Garlan et Shaw [SG96], consacré à l'architecture logicielle.

Les auteurs indiquent que l'architecture logicielle n'est qu'une abstraction de la réalité qui se focalise plus particulièrement sur l'interaction entre les composants. De plus, la définition proposée met également en lumière le fait qu'un système peut avoir plusieurs structures correspondant chacune à un point de vue et aucune d'entre elles ne peut être considérée comme celle définissant l'architecture. Comme une architecture du bâtiment, un logiciel est représenté par plusieurs plans et schémas destinés chacun à un corps de métier et dépendant de l'étape du processus de développement.

Est-ce que tous les systèmes ont une architecture ? Les auteurs pensent que oui puisque chaque système peut être vue comme une décomposition d'éléments interagissant entre eux. Dans le cas le plus trivial, le système peut n'être qu'un élément. Ils remarquent que l'architecture associée au système peut ne pas être connue (les architectes initiaux sont partis, nous ne possédons que le code source…), ce qui clarifie la différence entre une architecture logicielle et une description architecturale.

Les auteurs notent que le comportement de chaque élément fait partie de l'architecture. Le comportement d'un élément permet de définir comment un autre élément peut interagir avec lui. Ils jugent ainsi que les diagrammes informels sous forme de boites et de lignes ne sont pas des architectures puisqu'ils ne décrivent pas concrètement le comportement des élément.

Les auteurs pensent qu'il est intéressant de pouvoir évaluer une architecture par rapport à ces besoins, ils ont élaboré deux méthodes pour analyser une architecture

logicielle. La première appelée ATAM (Architecture Tradeoff Analysis Method) qui fournit aux architectes les moyens pour évaluer les compromis techniques connus durant la conception et la maintenance du logiciel. La deuxième appelée CBAM (Cost Benefit Analysis Method) construite sur la première méthode, elle permet de faire une estimation des enjeux techniques et économiques et par conséquent faire les décisions architecturales.

2.3.5 Garlan 2000 [Gar00]

"*While there are numerous defenitions of software architecture, at the core of all them is the notion that the architecture of a system describes its gross structure. This structure illuminates the top level design decisions, including things such as how the system is composed of interacting parts, where the main pathways of interaction, and what are are the key properties of the parts. Additionally, an architectural description includes sufficient information to allow high level analysis and critical appraisal*".[Gar00]

Dans cet article l'auteur n'a pas donné une nouvelle définition, mais plutôt a examiné quelques tendances importantes de l'architecture logicielle dans la recherche et la pratique. L'ensemble des définitions qui ont été formulées ont au centre de chacune la notion que l'architecture d'un système décrit les élément essentiels des sa structure.

Dans cet article Garlan fournit une vision de l'évolution de l'architecture logicielle qui est résumée dans la figure 2.3

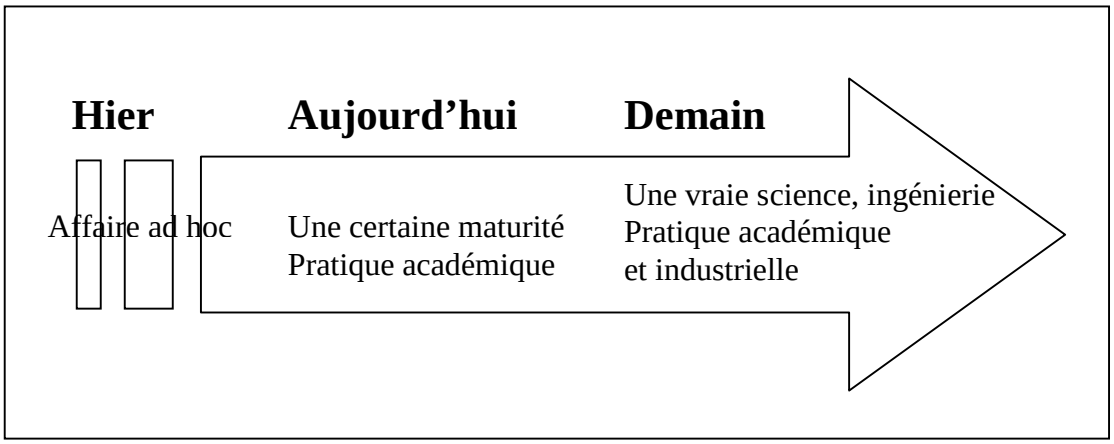


FIG. 2.3 – Evolution de la pratique architecturale[San02]

2.3.6 La norme ANSI/IEEE Std 1471-2000

"The fundamental organization of a system embodied in its components, their relationships to each other, and the environment, and the principles guiding it's design and evolution " [IEEE00]

L'IEEE a défini une norme pour la description architecturale des systèmes à prédominance logicielle « software-intensive systems ». La norme IEEE 1471 est une pratique recommandée qui adresse les activités de la création, de l'analyse, et du soutien des architectures des systèmes dépendants du logiciel, et de l'enregistrement de telles architectures en termes de descriptions architecturales. Un cadre conceptuel pour la description architecturale est établi, elle comprend l'utilisation de plusieurs vues, de modèles réutilisables au sein de vues, et la relation qu'entretient l'architecture avec le contexte du système.

La figure suivante (cf. figure 2.4) montre le modèle conceptuel de description architecturale tel qu'il a été défini dans l'IEEE std 1471-2000.

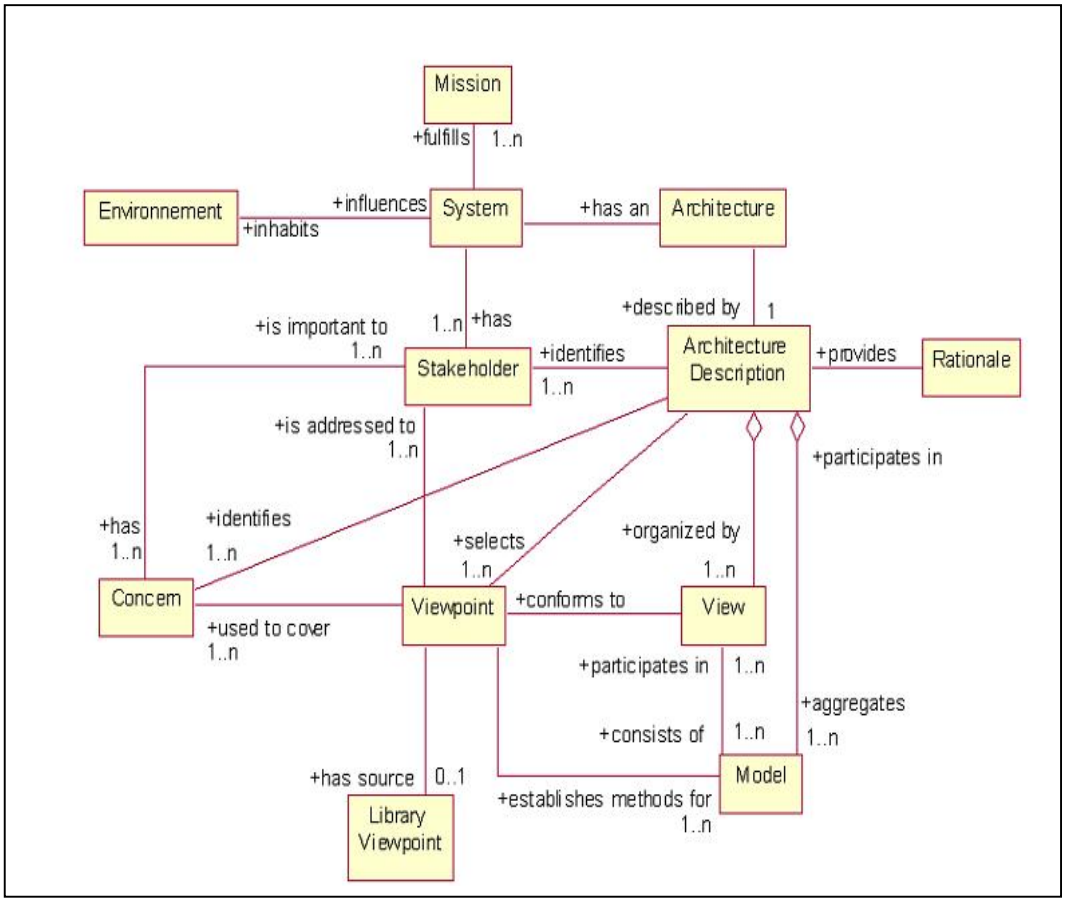


FIG. 2.4 – Modèle conceptuel IEEE std 1471-2000pour la description architecturale

2.4. La description des architectures logicielles

Il existe plusieurs formalismes pour la description de l'architecture logicielle, les travaux réalisés dans ce domaine peuvent être classés selon les axes suivants :

2.4.1 Les ADLs

Les travaux sur les ADLs ont été un axe de recherche important et de nombreuses propositions ont rapidement émergé [Gar00]. Chaque ADL est spécialisé dans un thème particulier fournissant des fonctionnalités complémentaires pour la description et l'analyse architecturale. Pour la description d'une architecture logicielle, les ADLs reposent sur les concepts suivants : Composant, connecteur et configuration.

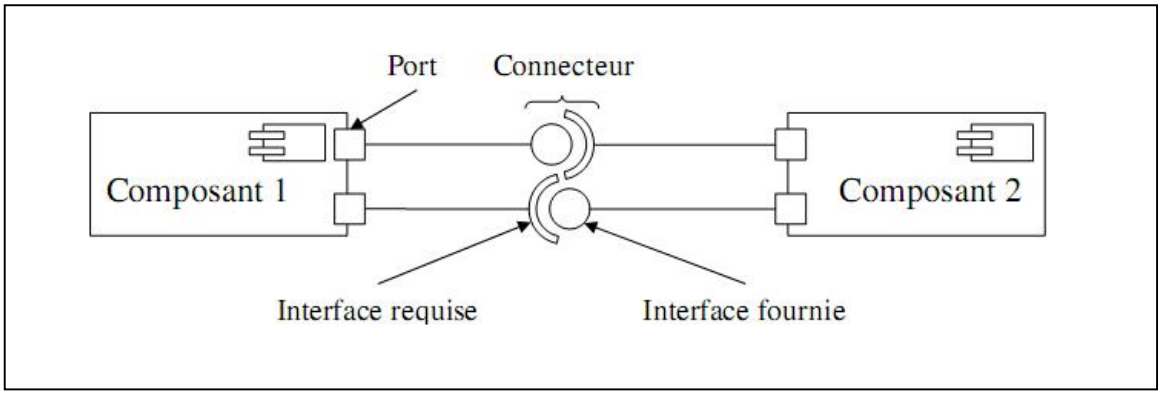


FIG. 2.5 – Les éléments de l'architecture logicielle

- **Le composant**

  Un composant est une entité fournissant des fonctionnalités (principalement des unités de calcul et de données). Il possède un certain nombre d'interfaces (souvent appelées ports). Chacune d'entre elles décrit un sous ensemble de comportement du composant et du comportement attendu du système dans lequel il va interagir. Une interface d'un composant joue en quelque sorte le rôle d'un filtre sur le comportement global du composant. Une description de ce dernier est aussi est aussi fournie qui permet entre autre

de décrire comment les différentes interfaces du comportement interagissent ensemble.

- **Le connecteur**

  Un connecteur est une entité définissant l'interaction (protocole de communication) entre plusieurs composants. Comme pour un composant un connecteur possède certain nombre d'interfaces (souvent appelées rôles) et une description globale de son comportement. Un des apports des ADLs a été de remonter le concept de connecteur au même niveau que celui du composant.

- **La configuration**

  La configuration de l'architecture définit les propriétés topologiques de l'application. Les composants et les connecteurs sont considérés comme des types qui peuvent être instanciés et assemblés à partir de leurs interfaces (ports et rôles) pour former une configuration particulière. Un langage de contraintes est souvent disponible afin de formuler certaines règles de composition à vérifier.

Plusieurs ADLs ont été développés que ce soit dans la recherche ou dans l'industrie. Parmi les plus connus nous présentons:

2.4.1.1 Darwin

Le langage Darwin [MDK93] [MDK94] a été développé par "Distributed Software Engineering Group" de l'Imperial College. Darwin propose un modèle de composant pour la construction d'applications distribuées. Ce langage se centre sur la description de la configuration et sur l'expression du comportement d'une application plutôt que sur la description structurelle de l'architecture d'un système [ACC02]. La particularité de Darwin est de permettre la spécification d'une partie de

la dynamique d'une application en terme de schéma de création de composants logiciels avant ou pendant son exécution.

Comme la plupart des ADLs, Darwin reprend les trois concepts de base à savoir le composant, le connecteur et la configuration.

2.4.1.2 Rapide

Rapide [Luc96] est un langage de description d'architecture dont le but principal est de vérifier, par simulation, la validité d'une architecture logicielle. Il fut proposé à l'origine au projet ARPA (Advanced Research Projects Agency) en 1990. Il est basé sur des évènements concurrents. Il est conçu essentiellement pour prototyper des architectures dans des systèmes distribués.

Le langage Rapide ne fournit pas d'outils de génération de code, il permet cependant la description du comportement dynamique et il offre des méthodes de validation (simulation) des architectures qu'il décrit. Il permet aussi de vérifier certaines propriétés comme l'inter blocage lors de l'exécution de l'application. Les concepts de base de Rapide sont le composant, l'évènement et l'architecture.

2.4.1.3 Wright

Wright [AG97], et un langage de description d'architectures logicielles créé par Allen et Garlan, il fournit des bases formelles pour spécifier les interactions entre les composants (via des connecteurs). Wright est un langage de description orienté plus vers la vérification des protocoles entre les composants, que vers la correction fonctionnelle de l'architecture globale. Il permet de décrire formellement une architecture à l'aide de l'algèbre de processus CSP.

Comme les autres ADLs, Wright reprend les trois concepts de l'architecture logicielle à savoir le composant, le connecteur et la configuration.

Dans ce qui suit nous allons présenter un exemple de la description d'une architecture en utilisant le langage Wright.

L'exemple proposé concerne un système filtre de type PipeFilter permettant de lire un flot de caractères pour le transformer en flot contenant les mêmes caractères en lettres capitales. Le schéma représente ce système sous forme de diagramme :

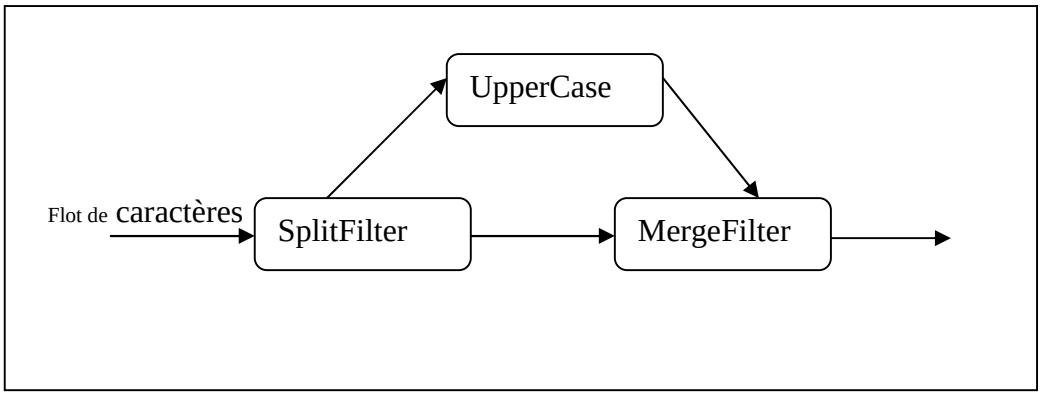


FIG. 2.6 – Exemple de filtres

Le système comprend trois composants :

− Le composant SplitFilter permettant de récupérer le flot de caractères en entrée et de créer deux flots :

- un flot identique à celui en entrée pour le composant UpperCase pour la conversion en lettres capitales,

- un flot identique à celui en entrée pour le composant Merge.

− Le composant UpperCase permettant de transformer un flot de caractères en flot contenant les mêmes caractères en lettres capitales,

− Le composant MergeFilter permettant de fusionner les deux flots.

```
Configuration Capitalize
  component Split =
     port input = getchar?x --> input
     port Left = putchar!x --> Left
     port Right = putchar!x --> Right
     computation = input.getchar?y --> Left.putchar!y
                          --> input.getchar?y --> Right.putchar!y -->
             computation
component Filter =
    port input = get?x --> input
    port output = put!x --> output
    computation = input.get?x --> output!x -->  computation
component Merger
    port Left = get?c --> Left
    port Right = get?c --> Right
    port output = put!c --> output
    computation = Left.get?c --> output.put!c -->
            right.get?d --> output.put!d --> computation
connector pipe =
   port source = in!x --> source
   port sink = out?y --> sink
   glue = source.in?x --> sink.out!x --> glue
  Instances
  Split : SplitFilter
  Upper : UpperCase
  Merge : MergeFilter
  P1, P2, P3 : Pipe
  Attachments
  Split.Left as P1.Source
  Upper.Input as P1.Sink
  Split.Right as P2.Source
  Merge.Right as P2.Sink
  Upper.Output as P3.Source
  Merge.Left as P3.Sink
End Configuration
```

FIG. 2.7 – Description de l'architecture des filtres dans Wright

2.4.2 Les vues architecturales

La notion de vue est à l'heure actuelle largement utilisée, que ce soit au niveau du cycle de développement du logiciel ou bien simplement d'une seule phase du développement. Concernant la notion de vue dans le domaine de l'architecture logicielle ont trouve:

2.4.2.1 Les 4+1 vues Kruchten

Dans ce travail [Kru95] l'auteur donne une description de l'architecture logicielle en cinq vues. Ces cinq vues sont : « la vue logique » qui sert à décrire les fonctionnalités du système (quels sont les services que le système doit fournir aux utilisateurs), « la vue processus » qui prend en charge des aspects non fonctionnels tel que la performance et la disponibilité, « la vue développement» qui à son tour focalise sur l'organisation du code en différent modules et leurs dépendances , « la vue physique » la vue physique même chose que la vue processus sauf que cette vue manipule des entités matérielles, et finalement la vue « scénarios » considérée comme redondante et qui sert à illustrer des exemples de coopération entre les vues précédentes .

2.4.2.2 Les 4 vues de Hofmeister et al.

Dans [HNS00] les auteurs utilisent quatre vues pour la description de l'architecture qui sont : « la vue conceptuelle » qui permet de modéliser le produit en termes de composants et connecteurs conceptuels, « la vue module » dont l'objectif principal est de prendre en compte les besoins liés à l'implémentation et l'organisation du développement , « la vue code » organise le code sous forme de fichiers sources, librairies, …etc, finalement « la vue exécution » qui prend en compte les aspects dynamiques du logiciels en décrivant le système en termes d'élément exécutable de la plateforme.

2.4.3 UML

UML (Unified Modeling Language)[Fow04] est un langage qui offre une notation graphique unifiée connue par la majorité des architectes et apportant une solution pour décrire l'architecture logicielle.

Les travaux qui ont étudié l'utilisation d'UML pour la description des architectures logicielles peuvent être classés selon deux approches. La première comme dans [KSLB01], consiste à utiliser la notation existante d'UML telle qu'elle est sans aucune extension pour représenter l'architecture logicielle. La deuxième, consiste à étendre la syntaxe et la sémantique de la notation UML afin de décrire l'architecture logicielle. La deuxième approche elle-même peut être divisée en deux classes selon la nature des extensions apportées à UML, il y a l'extension « heavyweight » [PER03] qui adapte UML à la description des architectures par la modification de son métamodele en ajoutant de nouveaux modèles ou remplaçant la sémantique existante. L'extension « lightweight » [KCSS02], quand à elle définit de nouveaux éléments permettant d'étendre la notation UML sans modification du métamodèle en définissant des profils UML.

La version 2.0 de UML comporte 13 diagrammes. 6 diagrammes décrivent la structure du système (diagramme de classes, diagramme structure composite, diagramme de composants, diagramme de déploiement, diagramme d'objets et diagramme de paquetage). Les 7 autres diagrammes permettent de décrire le comportement du système (diagrammes d'interactions : diagramme de séquence, diagramme de vue d'ensemble des interactions, diagramme de communication et diagramme de temps), diagramme d'activité, diagramme de machine à états et diagramme de cas d'utilisation).

UML 2.0 apporte des améliorations pour la représentation des architectures logicielles. Le diagramme de composants a été révisé et le concept de classificateur structuré (structured classifier) a été intégré. UML 2 .0 introduit la notion de composant (component) ainsi que la notion de connecteur (connector : assembly connector et delegation connector). L'introduction des ces concepts, ainsi que ceux

de port ou encore la distinction entre interfaces offertes et requises fournissent une palette d'éléments et de notations intéressantes pour l'architecture logicielle.

2.4.4 Les Descriptions formelles :

L'architecture logicielle offre une représentation de la structure globale du système pour permettre sa compréhension et la maîtrise de sa complexité. Bien que les techniques informelles peuvent répondre à cet objectif, il y a un autre objectif très important qui doit être pris en considération : c'est permettre la validation et la vérifications du système très tôt dans le processus logiciel. C'est la raison pour laquelle plusieurs travaux [Met98][LHJD04] visent à tirer profit des techniques formelles pour la description architecturale. Une autre alternative est de combiner les avantages de plusieurs techniques comme dans [RMRR98][BHTV04].

2.5 Conclusion

Dans ce chapitre nous avons présenté le domaine de l'architecture logicielle en donnant les principales définitions de l'architecture logicielle dans la littérature. La description de l'architecture est l'une des activités importantes qui permet une représentation globale du système, dans ce chapitre nous avons aussi présenté les différents formalismes pour la réalisation de cette activité. Nous avons constaté dans ce chapitre que ce domaine s'appuie sur le concept de composant comme entité de base pour la description des architectures, dans le prochain chapitre nous allons introduire le concept agent qui se présente comme une nouvelle alternative qui permet de concevoir et de décrire les architectures logicielle des systèmes modernes.

Chapitre 3

Les systèmes multi-agents

3.1 Introduction

Les systèmes multi-agents sont conçus comme un ensemble d'entités logicielles autonomes (agents), dans une structure partagée (environnement). Les agents peuvent atteindre leurs objectifs d'une façon flexible en interagissant les uns avec les autres. Cette nature des agents les a attribué des propriétés de qualité telle que l'adaptabilité, la robustesse et l'évolutivité. Dans ce chapitre nous allons présenter les concepts qui découlent du domaine de recherche que constituent les systèmes multi-agents.

3.2 L'Agent

3.2.1 Définitions

Actuellement il y a plusieurs définitions de l'agent qui se ressemblent mais différent suivant le type d'application pour laquelle l'agent est conçu [Cha99]. Donc, il n'y a encore aucun consensus, entre les différents chercheurs, quant à la définition de l'agent. Dans ce sens, des organismes internationaux tels que FIPA(Foundation for Intelligent Physical Agent) et OMG (Object Management Group) sont entrain de faire des efforts pour la standardisation du concept « agent », mai jusqu'à maintenant ils n'ont pas encore atteint ce but en raison de l'ampleur de ce problème. L'une des définitions les plus connues dans la littérature est celle de Ferber [Feb95] : «Un agent est une entité autonome, réelle ou abstraite, qui est capable d'agir sur elle-même et sur son environnement, qui dans un univers multi agent, peut

communiquer avec d'autres agents, et dont le comportement est la conséquence de ses observations, de ses connaissances et des interactions avec les autres agents ».

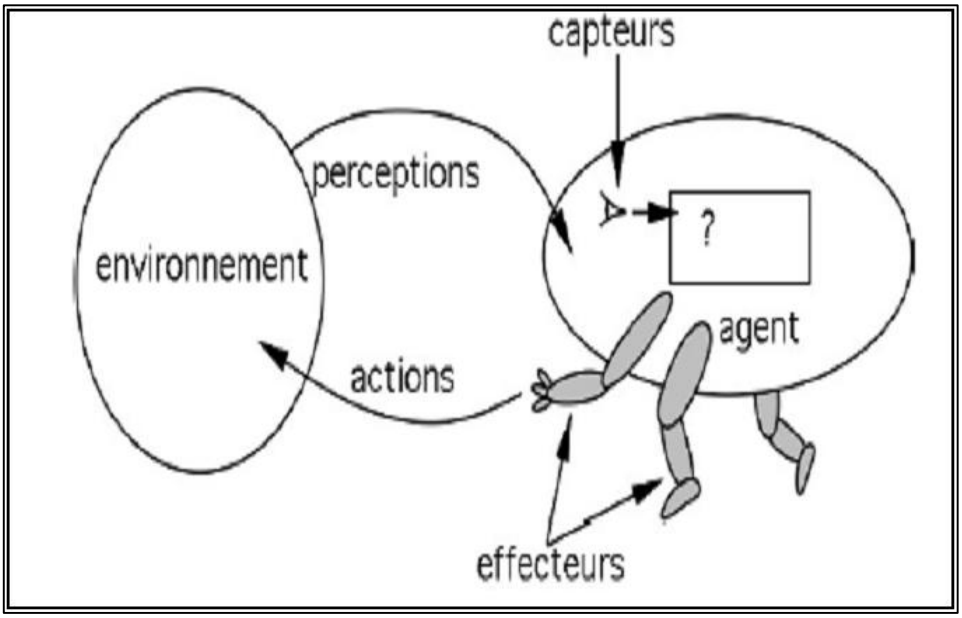


FIG. 3.1 – Schématisation d'un agent

Une autre définition a été proposée par [Syc98] :

« *If a problem domain is particularly complex, large or unpredictable, then the only way it can reasonably be addressed is to develop a number of functionally specific and (nearly) modular components (agents) that are specialized at solving a particular problem aspect*»

Cette dernière définition donne une forte raison pour la programmation orientée agent, la modularité et la décomposition. Mais en fait les agents sont sensés être plus que ça. Les agents sont considérés aussi comme un nouveau paradigme conceptuel pour l'analyse des besoins, la conception et l'implémentation des systèmes complexes et distribués.

Pour qu'une entité logicielle peut être dite 'Agent' elle doit avoir des caractéristiques qui la délimite des autres entités (ex : objet), Jennings, Wooldridge et

Sycara dans [JWS98] ont essayé de répondre a ce problème par la définition suivante:

« *An agent is a computer system, situated in some environment, that is capable of flexible autonomous action in order to meet its design objectives*».

Les notions ‘‘situé’’, ‘‘autonomie’’ et ‘‘flexible’’ qui constituent les caractéristiques que l’agent doit avoir sont définies comme suit :

- *Situe* : l’agent est capable d’agir sur son environnement à partir des entrées sensorielles qu’il reçoit de ce même environnement. Exemple : systèmes de contrôle de processus, système 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* : l’agent dans ce cas est

  J*capable de répondre à temps* : l’agent doit être capable de percevoir son environnement et élaborer une réponse dans les temps requis,

  J *proactif* : l’agent doit exhiber un comportement proactif et opportuniste, tout en étant capable de répondre l’initiative au ‘‘bon’’ moment ;

  J *social* :l’agent doit être capable d’interagir avec les autres agents (logiciels et humains) quand la situation l’exige afin de compléter ses tâches ou aider ces agents à accomplir les leurs.

Dans ce mémoire nous intéressons particulièrement à la description des architectures logicielles à base d’agents c’est à dire la structure globale du système et les interactions entre les différents agents composant ce dernier, donc en va pas trop nous soucier de l’intelligence de l’agent. Pour nous un agent peut être vu comme

une entité logicielle autonome, réactive ou proactive qui peut interagir et coopérer pour atteindre son objectif.

3.2.2 Architectures des Agents

L'architecture d'un agent est une description de son organisation interne et son fonctionnement. L'architecture d'un agent découle de son aspect réactif ou cognitif ou les deux ensembles. Ainsi, il existe trois types d'architectures:

3.2.2.1 Architectures réactives

Comme son nom l'indique, un agent réactif ne fait que réagir aux changements qui surviennent dans l'environnement. Autrement dit, un tel agent se contente simplement d'acquérir des perceptions et de réagir à celles-ci en appliquant certaines règles pré définies. Étant donné qu'il n'y a pratiquement pas de raisonnement, ces agents peuvent agir et réagir très rapidement

Les architectures réactives représentent le fonctionnement de l'agent au moyen de composantes avec une structure de contrôle simple, et sans représentation évoluée des connaissances de l'agent.

L'architecture réactive la plus connue est celle proposée par Brooks [Bro86], appelée architecture de subsomption. Cette architecture consiste d'un ensemble de comportements pour l'accomplissement d'une tâche, où chaque comportement est une machine à états finis qui fait correspondre les entrées des capteurs aux actions des effecteurs. Ces comportements sont organisés en couche, allant du plus simple (éviter les obstacles dans le cas d'un robot,…) au plus sophistiqués. Si à un instant donné plusieurs actions sont générées à partir de différents comportements activés, la résolution de conflits sera effectuée par interaction entre les différentes actions (par exemple une action inhibe l'autre).

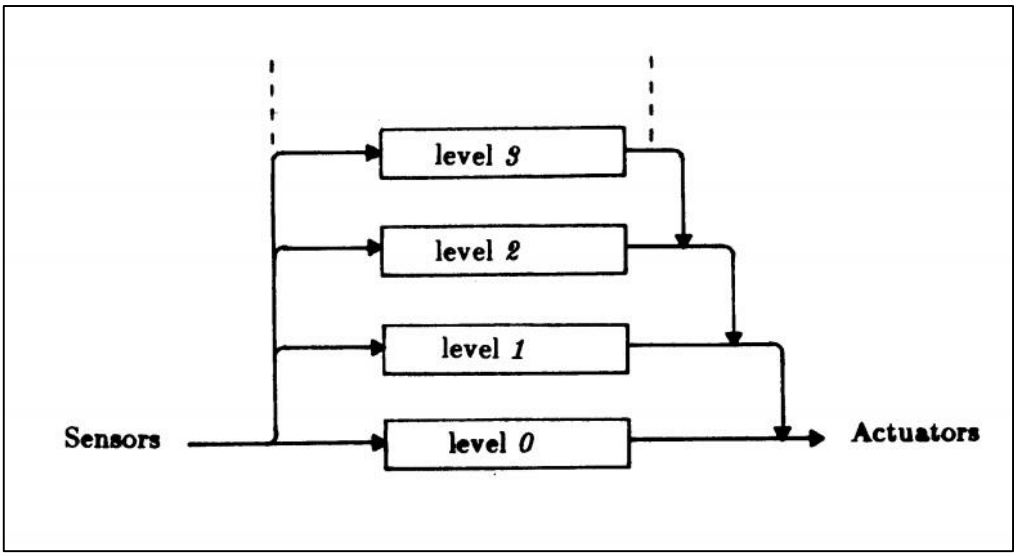


FIG. 3.2 – Architecture de subsomption [Bro86]

Les architectures réactives ont l'avantage de la simplicité et de l'efficacité du calcul, mais elles deviennent insuffisantes pour les applications où la prise de décision nécessite une modélisation de l'environnement [JWS98] et une prise en considération de l'historique et des expériences passées.

3.2.2.2 Architectures Cognitives

Un agent cognitif possède un état mental qui l'incite à agir dans un sens particulier et donc une représentation symbolique, de plus ou moins haut niveau en fonction de la complexité de la tâche à accomplir, de son environnement et de ses buts. Cette représentation symbolique lui permet de prendre conscience de l'état d'accomplissement de ses buts et des contraintes restant à soulever, mais aussi de ses motivations propres.

L'architecture BDI (Beliefs-Desires-Intentions) [Bra87] [RG91] a été développée pour la prise en compte des états mentaux. Généralement les attitudes mentales représentées sont les suivantes :

1. «Beliefs» les croyances d'un agent sont les informations que l'agent possède sur l'environnement et sur d'autres agents qui existent dans le même environnement. Les

croyances peuvent être incorrectes, incomplètes ou incertaines. Les croyances peuvent changer par perception ou par l'interaction avec d'autres agents.

2. «Desires» ou désirs représentent les préférences sur l'état futur de l'environnement d'un agent. Une caractéristique importante des désirs est qu'un agent peut avoir des désirs contradictoires, d'où la nécessité du choix d'un sous-ensemble consistant. Les désirs consistants sont parfois identifiés avec les buts de l'agent.

3. «Goals» ou buts représentent les engagements d'un agent pour atteindre un ensemble d'états de l'environnement. Ainsi un désir est une étape dans le processus de création d'un but. La notion d'engagement d'atteindre un but décrit la transition des buts aux intentions.

4. « Intentions» représentent les actions que l'agent s'engage à exécuter. A cause des ressources limitées, un agent ne peut poursuivre tous ses buts; ainsi il doit choisir un certain nombre de buts pour lesquels il s'engage. C'est ce processus qui est appelé la formation des intentions.

5. « Plans» plans représentent les intentions que l'agent s'engage à exécuter pour atteindre ses buts. Par conséquent, il est possible de structurer les intentions en plans plus étendus, et de définir les intentions courantes d'un agent comme les plans partiels qui sont couramment adoptés et qui seront raffinés par la suite.

Basée sur les notions précédentes, une architecture BDI ayant sept composants a été proposée par Wooldrige [WOO99] comme dans la figure 3.3 suivante:

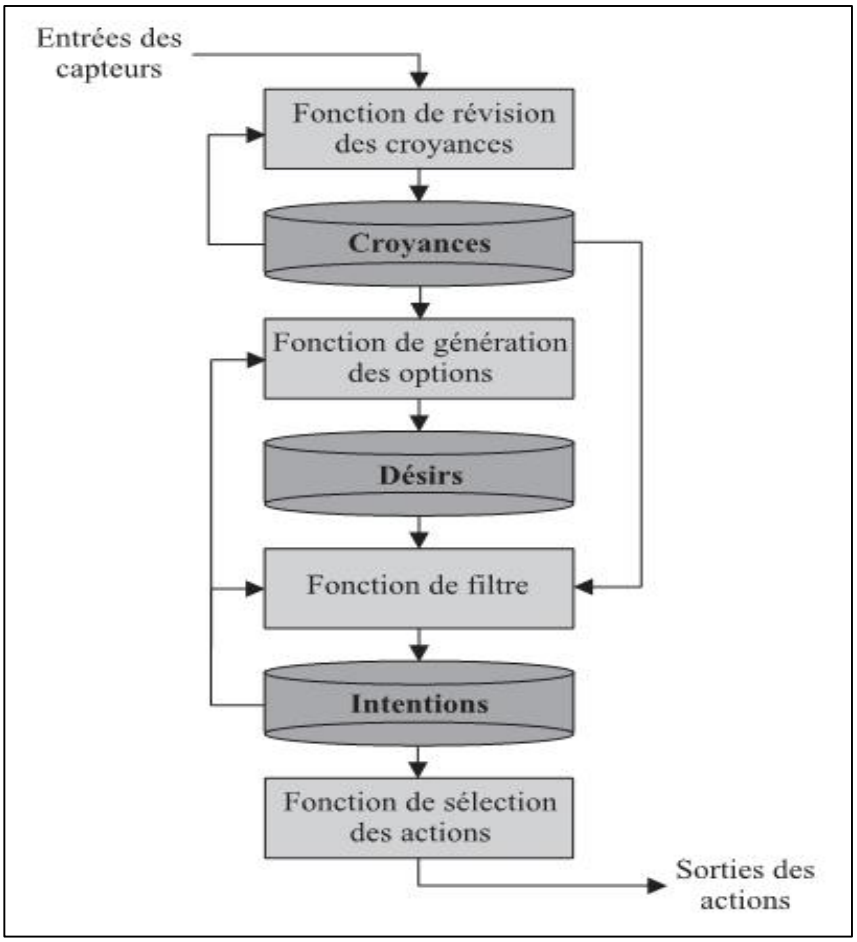


FIG. 3.3 – Diagramme d'une architecture BDI [WOO99]

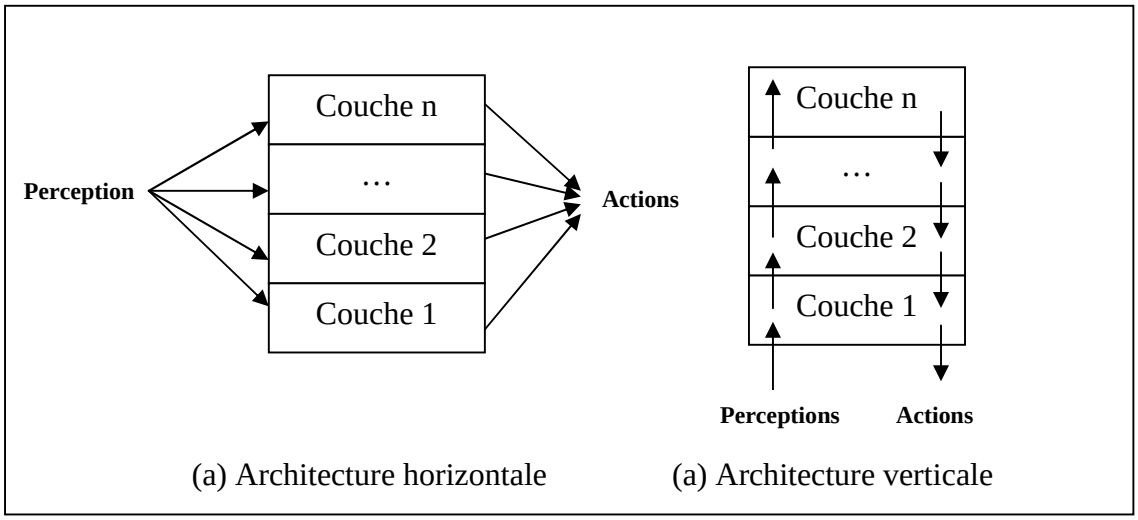
Comme illustrés dans la figure 3.3, les formes ovales correspondant à des structures de connaissances ou de données. Pour la résolution d'un problème, l'agent doit produire des plans, par conséquent une séquence d'actions qu'il va exécuter. Les plans sont crées à partir des attitudes mentales : croyances, désirs et intentions, en se basant sur les composants: Révision, Génération d'options, le Filtre et Exécution. La fonction Révision, permet de réviser les croyances de l'agent selon les entrées perçues. La fonction de Génération d'options détermine les options disponibles à l'agent, donc ses désirs. Le Filtre est la partie de l'architecture qui a la responsabilité de bâtir des plans pour réaliser les intentions de l'agent. Cette fonction représente le processus de délibération de l'agent, d'où la nomination d'agent délibératif. Le Filtre produit des plans partiels, dits intentions, puis les raffines en plans étendus en fonction des croyances, désirs et intentions, afin d'obtenir un plan exécutable qui

est un sous ensemble des intentions. Les actions d'un plan étendu seront ensuite exécutées.

L'architecture BDI, reste la plus référencée pour les architectures cognitives. Elle permet de modéliser les connaissances de l'agent et ses croyances sur l'environnement, ainsi que les changements qui peuvent affectés ce dernier.

2.2.2.3 Architectures Hybrides

Les architectures hybrides sont une solution aux problèmes nécessitant des agents hybrides, donc intégrant des comportements réactifs et cognitifs. Une architecture hybride d'un agent intelligent est une architecture composée d'un ensemble de modules organisés dans une hiérarchie, chaque module étant soit une composante cognitive avec représentation symbolique des connaissances et capacités de raisonnement, soit une composante réactive



(a) Architecture horizontale (a) Architecture verticale

FIG. 3.4 – Architecture d'agent en couches [JWS98]

Dans ce type d'architecture chaque couche se comporte comme un agent de fait quelle est liée aux entrés des capteurs et aux sorties des actionneurs et elle produit des propositions sur les actions à exécuter. Le problème avec cette architecture réside

du fait que lorsque chaque couche propose une action il faut choisir laquel doit être exécuter.

Dans cette architecture seulement une couche est liée au entrées des capteurs et une autre seule couche est liées aux sortie des actionneurs, donc le problème de conflit entre les couches lors du choix des comportements ne se pose pas. On trouve deux types d'architectures en couches verticales: architecture en couches à un seul passe et architecture en couches à deux passes.

Dans une architecture en couches à deux passes les informations circulent dans un sens et le flux de contrôle dans le sens inverse.

L'architecture hybride de Müller [Mül96] est la plus étudiée et la plus référencée. Connue sous le nom de système InteRRaP (Integration of Reactive Behavior and Rational Planning), l'architecture InteRRaP est une architecture en couches avec des couches verticales où les données d'entrée, notamment les perceptions, passent d'une couche à l'autre. En cela, elle est un peu différente de l'architecture de subsomption qui est une architecture en couches horizontales, où les perceptions sont transmises directement à toutes les couches à la fois.

3.3 Les systèmes multi-agents

Les systèmes multi-agent (MAS) sont des systèmes composés de plusieurs agents autonomes qui peuvent fonctionner collectivement pour atteindre des objectifs individuels ou communs qui sont difficiles voir impossibles de réaliser par un seul agent, en suivant l'idée d'organisation humaine. Les systèmes complexes et distribués peuvent être modélisés comme MAS pour réaliser une meilleure décomposition du problème et pour une modélisation plus naturelle des sociétés d'entités collaboratives.

3.3.1 Définitions

Comme pour les agents, plusieurs définitions de systèmes multi agents ont été proposées, nous retenons celle introduite par Ferber[Fer95], qui le définit comme un système composé des éléments suivant (figure 3.5) :

- Un environnement E identifié et muni d'un système de réparage dans l'espace (souvent Euclidien)
- Un ensemble d'objets O passifs pouvant être perçus, créés, modifiés ou détruits par des agents ;
- Un ensemble d'agents A actif($A \subseteq O$) ;
- Un ensemble de relations R qui unissent des objets entre eux ;
- Un ensemble d'opération Op offrant la possibilité aux agents de A de percevoir, produire, consommer, transformer et manipuler des objets de O ;
- Un ensemble de loi universelles qui sont des opérateurs chargés de représenter l'application des actions des agents sur le monde et la réaction du monde à ces actions

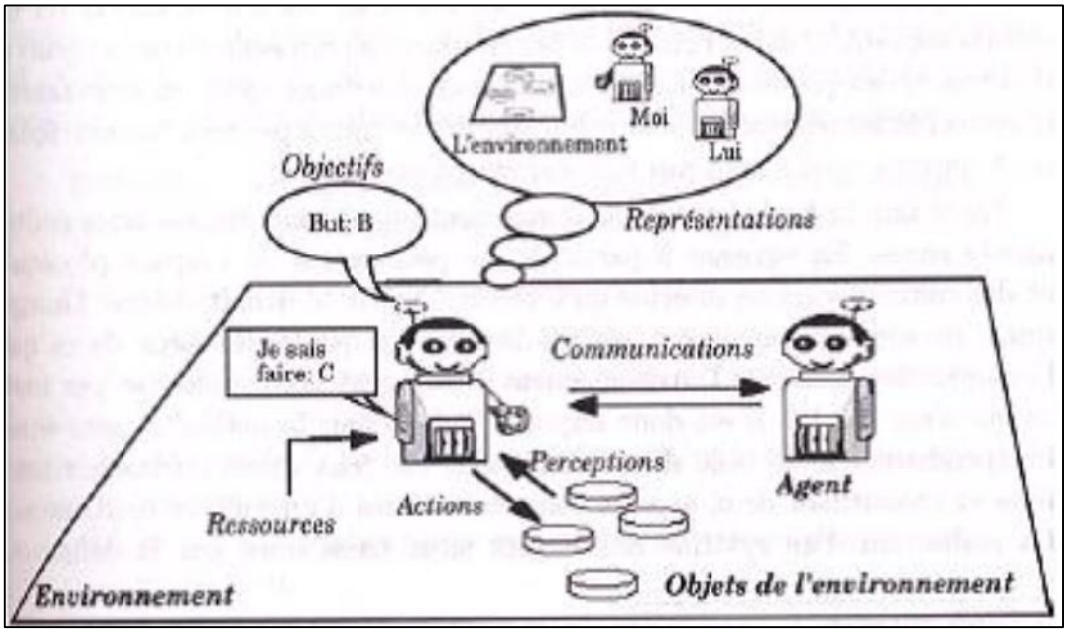


FIG. 3.5 – Interaction de l'agent avec son environnement [Fer95]

Les systèmes multi-agents possèdent les caractéristiques principales suivantes [JSW98] :

- Chaque agent a des informations ou des capacités de résolution de problèmes incomplètes, donc chaque agent a un point de vue limité;
- Il n'y a pas de contrôle global du système;
- Les données sont décentralisées;
- Les calculs sont asynchrones.

3.3.2 Interaction entre les agents

Les systèmes multi-agents ont surtout l'avantage de faire intervenir des schémas d'interaction sophistiqués. Ils peuvent ainsi coexister, être en compétition ou coopérer. L'interaction entre l'agent et son environnement peut être définie comme étant l'influence mutuelle entre ces deux entités, qui est établie via le processus de perception de l'agent de son environnement et de son action sur ce dernier. L'interaction peut être directe, comme dans le cas de l'action de l'agent sur son environnement ; c'est une modification explicite de l'état de ce dernier. Elle peut aussi être indirecte, comme dans le cas de la perception de l'environnement par l'agent, l'environnement ne peut pas modifier lui-même l'état de l'agent, mais ce dernier peut changer d'état en fonction des percepts récupérés à partir de l'environnement. En ce qui concerne l'interaction entre les agents plusieurs situations imposent l'interaction entre agents ; l'échange de données, la mise en commun des compétences pour résoudre un problème global, le partage de ressources limitées, l'aide d'un agent par un autre,…etc. L'interaction est une caractéristique importante des systèmes multi-agents où Ferber [Feb95] confirme qu'un agent sans interaction avec d'autres agents n'est plus qu'un corps isolé, qu'un système de traitement d'information, dépourvu de caractéristiques adaptatives.

3.3.3 Communication entre les agents

Dans un système multi-agents les agents communiquent pour pouvoir atteindre leurs objectifs. Autrement dit le but visé n'est toute fois pas de communiquer avec l'autre, mais c'est un moyen pour échanger des inforamtions et coordoner leurs activités.

Dans les SMA, deux stratégies principales ont été utilisées pour supporter la communication entre agents : les agents peuvent échanger des messages directement, ou ils peuvent accéder à une base de données partagée (appelée tableau noir ou "blackboard") dans laquelle les informations sont postées. Les communications sont à la base des interactions et de l'organisation sociale d'un SMA [Cha99].

Il existe plusieurs langages de communication, qui se basent sur les actes de langage, parmi lesquels on trouve le KQML, « Knowledge Query Manipulation Language » [FF94] et FIPA-ACL (« Foundation for Intelligent Physical Agent-Agent Communication Language ») [FIP94]. Ces deux langages de communication entre agents, ont émergé des efforts de standardisation de la communauté des systèmes multi agents.

3.4 Les méthodologies orientées agent

La conception des systèmes orientés agent nécessite une méthodologie appropriée de génie logiciel qui supporte les nouveaux concepts et la nouvelle vue aux systèmes logiciels propagés par ce nouveau paradigme de programmation. Le développeur doit être guidé pendant toutes les phases de développement du logiciel, de l'analyse des besoins à l'implémentation, avec un système orienté agent à l'esprit, pour pouvoir profiter de tous ses avantages.

Les méthodologies orientées objets traditionnelles ne sont pas adaptées pour développer les systèmes orienté agent, ce qui a mené à l'élaboration de plusieurs méthodologies orientées agent. MaSE [DW01] et Gaia [WJK00] étendent des approches orientées objets existantes dans le but de tirer profit de l'expérience et du

succès liés à l'utilisation des méthodes OO, Promotheus [PW00] a été construite en se basant sur l'expérience avec les plateformes agent, MASCommonKADS [Igl98], a ces racines dans l'ingénierie des connaissances et Tropos[GKMP04] est une méthodologie fondée sur l'ingénierie des besoins et elle est influencée par *i** [Yu95].

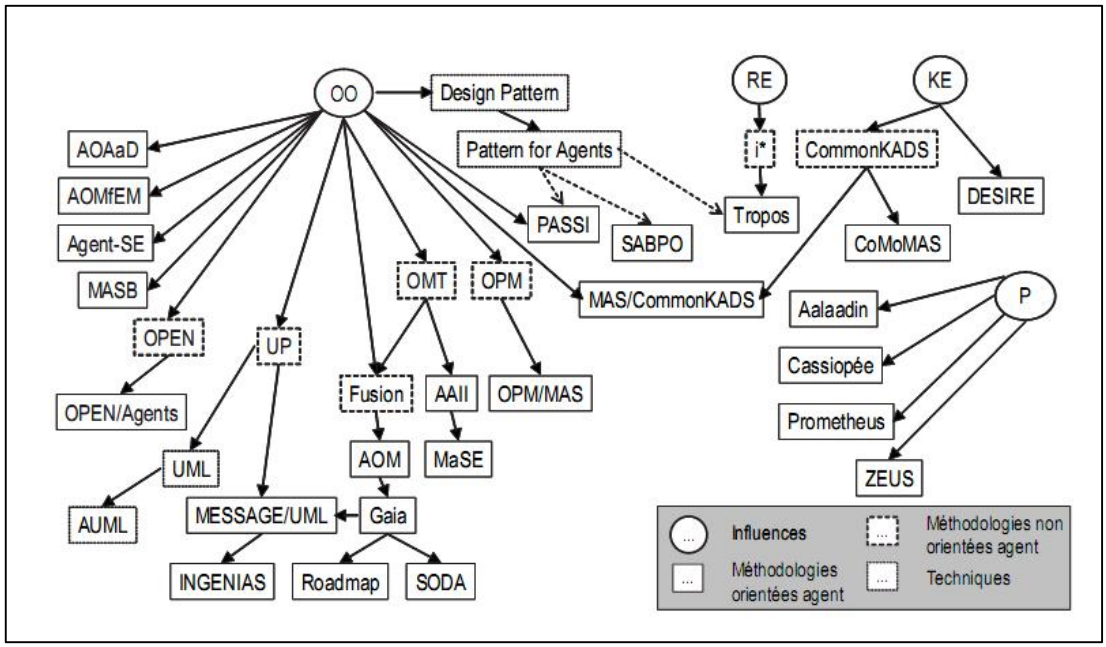


FIG. 3.6 – Généalogie des méthodes orientées agent et les différentes influences [Pic04]

Dans ce qui suit nous allons présenter des exemples de quelques méthodologies, d'autres détails et méthodologies peuvent être trouvés dans [HG05].

3.4.1 MaSE

La méthodologie MaSE [DW00] se présente comme une approche complète pour le développement des systèmes multi-agents de l'analyse jusqu'au déploiement. La méthode à été construite en adaptant les méthodologies orientées objets pour répondre aux caractéristiques des agents car les auteurs considèrent l'agent comme une spécialisation de l'objet.

Le processus de développement de MaSE comporte deux phases :

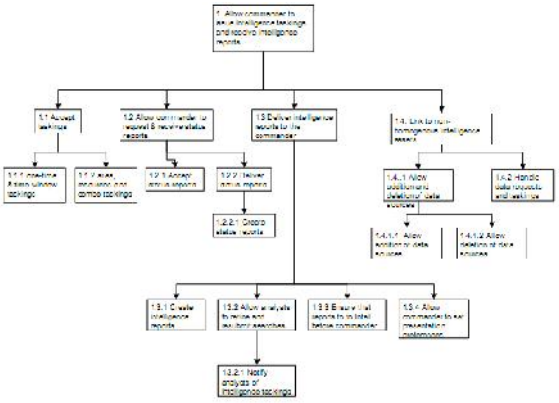
La phase d'analyse : est composée de trois étapes :

1. Identifier les buts en prenant les spécifications initiales du système et les transformer en ensembles structurés de buts. Un but est défini comme objectif au niveau système. Après l'identification, les buts sont alors analysés, structurés et organisés par ordre d'importance.
2. Identifier les cas d'utilisation et les diagrammes de séquence : Les diagrammes de cas d'utilisation sont tirés des conditions de système. Ils présentent des descriptions narratives d'une séquence d'opérations. Ces descriptions définissent le comportement désiré du système. Le diagramme de séquence est employé pour déterminer l'ensemble de messages envoyés entre les rôles.
3. Transformer les buts en rôles : Les rôles sont les modules employés pour définir les classes d'agent et pour identifier les buts de système pendant la phase de conception. Chaque but est associé à un rôle et chaque rôle est joué par une classe d'agent.

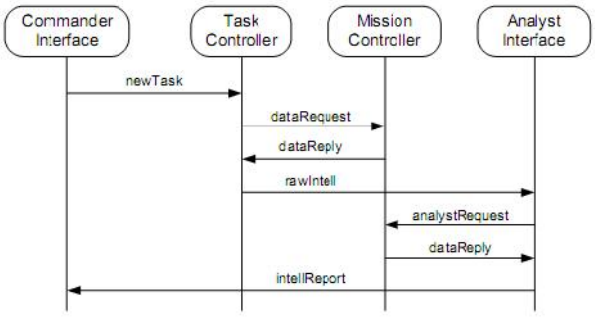
La phase de conception : est composée de trois étapes :

1. Créer des classes d'agents: Les classes d'agents sont identifiées à partir des rôles. Le produit de cette phase est un diagramme de classe d'agent, qui présente les classes d'agent et les conversations entre elles. Cette étape présente deux composants de classes: rôles et conversations.
2. Construire la conversation: Une conversation définit un protocole de coordination entre deux agents. Elle se compose de deux diagrammes de classe et de communication, un pour l'initiateur et un autre pour le répondeur. Cette étape est étroitement liée à la suivante (Assemblage des classes d'agents) et il est possible d'alterner entre les deux.
3. Assembler les classes d'agents: Cette étape s'intéresse à la création des classes internes des agents selon le modèle architectural choisi (BDI ou autres).
4. Définir la structure du système: La structure du système est présentée par les diagrammes de déploiement qui sont employés pour définir un système basé

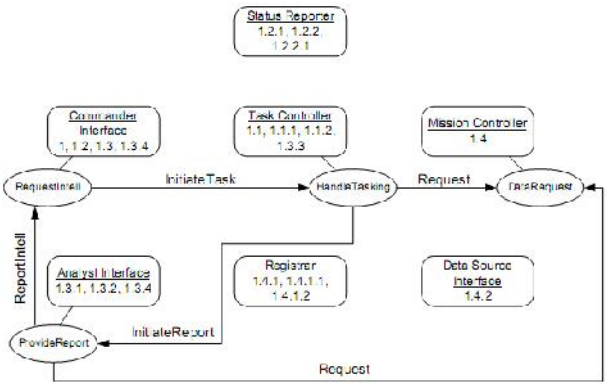
sur des classes d'agent. Ces diagrammes définissent les paramètres du système tels que le nombre et les types d'agents.



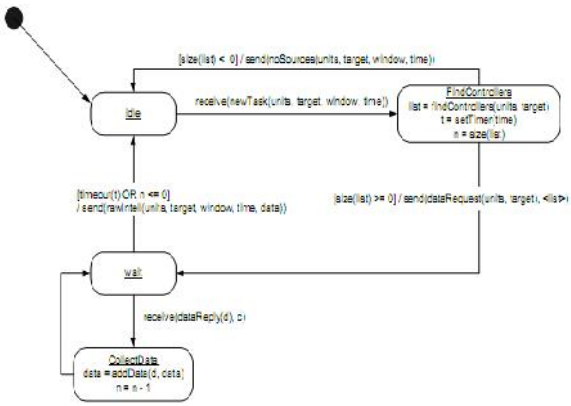
(a) Diagramme d'hiérarchie de buts



(b) Diagramme de séquences



(c) Modèle de rôle MaSE



(d) Diagramme de tâches MaSE

FIG. 3.7 – Exemples des diagrammes MaSE pour la phase de la conception architecturale [WD00]

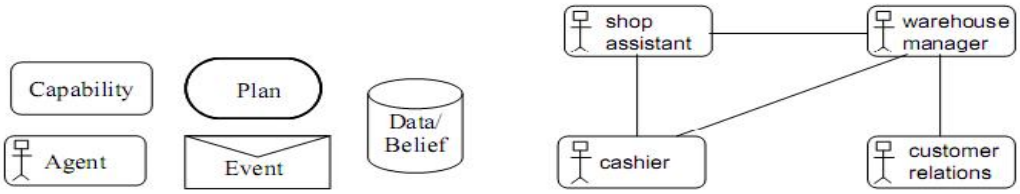
La hiérarchie de buts (a) est obtenue en décomposant le but principal du système en sous-buts; Les diagrammes de séquences (b) sont définis pour chaque cas d'utilisation identifié; les rôles (c) qui sont vus comme le moyen d'atteindre les buts fixés au système; les tâches concurrentes (d) qui décrivent les moyens pour les rôles d'atteindre les buts par des automates à états finis (ou bien des réseaux dePetri);

3.4.2 Promotheus

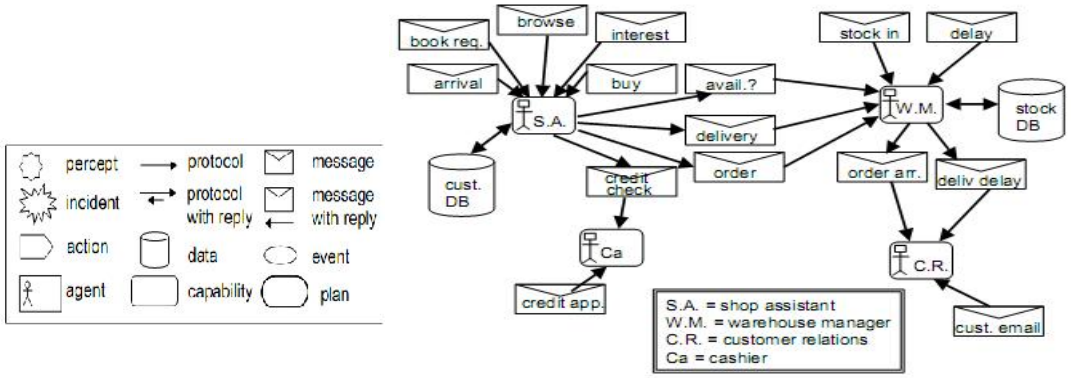
Prometheus [PW02] est une méthodologie complète, elle présente des méthodes de développement des systèmes d'agents intelligents avec des agents BDI, cette méthodologie est améliorée pour être indépendante de tous types d'architectures. Les agents dans Prometheus sont spécifiés dans des groupes fixés et interagissent entre eux selon des protocoles spécifiés. Ces agents présentent des croyances des autres agents, qui se communiquent par transfert de message et planifient alors leurs actions en fonction de leurs buts.

Le processus de développement de la méthodologie Prometheus se compose de trois phases:

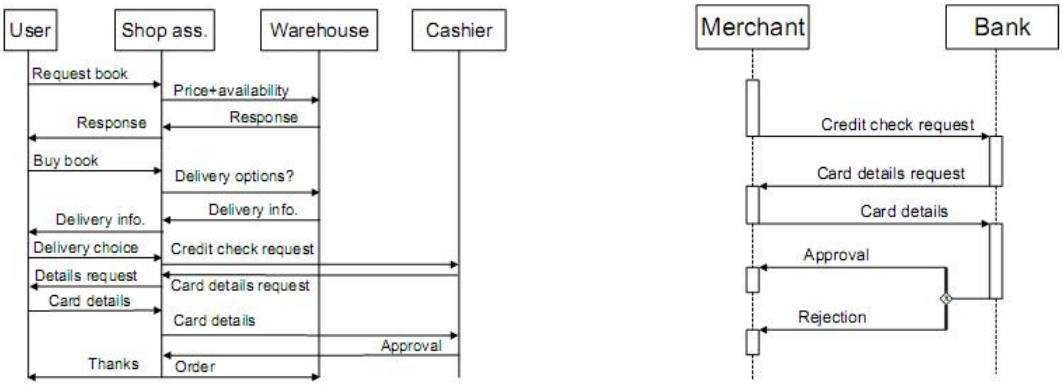
1. La phase de spécifications de système se concentre sur l'identification des fonctionnalités de base du système, avec des entrées (perceptions), des sorties (actions). Les actions et les perceptions définissent l'interface entre les agents et leur environnement. Des scénarios de cas d'utilisation sont générés pour identifier les buts et les sous-buts du système.
2. La phase de conception architecturale emploie les sorties de la phase précédente pour déterminer quels agents le système contiendra et comment ils agissent l'un sur l'autre. Dans cette phase, Les diagrammes d'accointances d'agent décrivent les interconnexions entre agents de différents types, il est considéré comme un diagramme de classes UML simplifié où les agents sont représentés par des classes (sans compartiment) et les connexions sont représentées par des associations binaires avec multiplicité. Les diagrammes de vue du système relient les agents, les événements et les objets partagés dans un même modèle en utilisant une notation dédiée. Les diagrammes d'interaction montrent les interactions entre agents. Prometheus exploite les diagrammes de séquences UML pour modéliser les interactions et AUML pour modéliser les protocoles d'interaction.



(a) Diagramme d'accointances



(b) Diagramme de vue du système



(c) Digrammes d'interactions

FIG. 3.8 – Exemples des diagrammes Prometheus pour phase de la conception architecturale [PW02]

3. La phase conception détaillée se préoccupe de la structure interne de chaque agent et de l'accomplissement de leurs tâches dans le système. Les

diagrammes présentés dans les phases de conception peuvent être tracés dans la plate-forme JACK qui génère un squelette de code source correspondant.

3.4.3 Mas-CommonKADS

MAS-CommonKADS [IGGV97] est une amélioration de la méthodologie CommonKADS [SWHV94] d'ingénierie cognitive avec des techniques orientées objet et des méthodologies d'ingénierie de protocole afin de modéliser les systèmes multi-agents. Le processus de développement comprend différents modèles, chaque modèle se compose de constituants (les entités à modeler) et de rapports entre eux.

Le processus de développement est constitué de trois phases:

1. La phase conceptualisation pendant cette phase, on effectue une description préliminaire du problème. Ceci est effectué en déterminant les cas d'utilisation (scénarios) qui peuvent aider à comprendre les conditions et à examiner le système. Les cas d'utilisation sont décrits en utilisant la notation OOSE (Object Oriented Software Engineering) et les interactions sont présentées avec diagrammes de séquence des messages.
2. La phase d'analyse, cette phase comprend le développement des modèles suivants:
    - Modèle d'agent: indique les caractéristiques d'agent, possibilités de raisonnement, services, groupes d'agent et hiérarchies.
    - Modèle de tâche: décrit les tâches que les agents peuvent effectuer: buts, décompositions et les méthodes de résolution des problèmes.
    - Modèle d'expertise: décrit la connaissance requise par les agents pour réaliser leurs buts.
    - Modèle d'organisation: décrit l'organisation dans laquelle le système multi-agents va être présenté et l'organisation sociale de la société d'agent.
    - Modèle de coordination: décrit les conversations entre les agents, leurs interactions, protocoles et possibilités requises.
    - Modèle de communication: décrit la relation dynamique entre les agents humains et les agents logiciels.

3. La phase de conception, cette phase est prolongée pour les SMA et s'intéresse au modèle de conception qui est constitué de trois sous-modèles:
   - Conception de réseau pour concevoir les aspects appropriés de l'infrastructure de réseau d'agent.
   - Conception d'agent: déterminer l'architecture la plus appropriée pour les agents.
   - Chaque agent est subdivisé dans des modules pour tous les modèles détaillés dans la phase d'analyse.
   - Conception de plateforme: choix de l'environnement de développement multi-agents et du matériel nécessaire ou disponible pour le système.

3.4.4 Tropos

La méthodologie de développement orienté agent TROPOS [GKMP04] a été développé par un travail commun entre les universités de Toronto (Canada), Louvain (Belgique), Pernambuco (Brésil), et Trento (Italie) et ITC-ITC-Irst (Italie).TROPOS couvre toutes les phases de développement de l'analyse des besoins à la conception détaillée et donne un rôle essentiel à l'analyse des besoins, elle adopte i*[Yu95], Ce langage manipule les différents concepts du méta modèle.

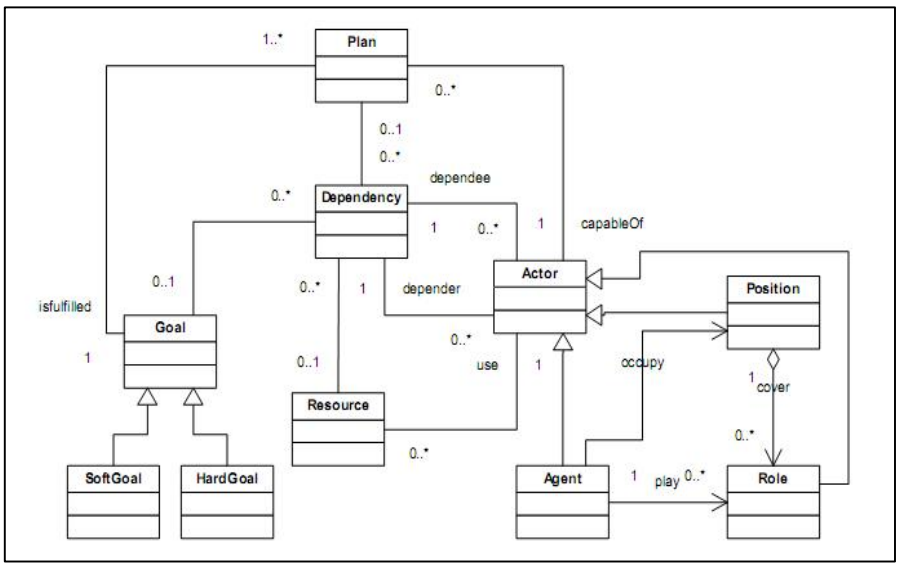
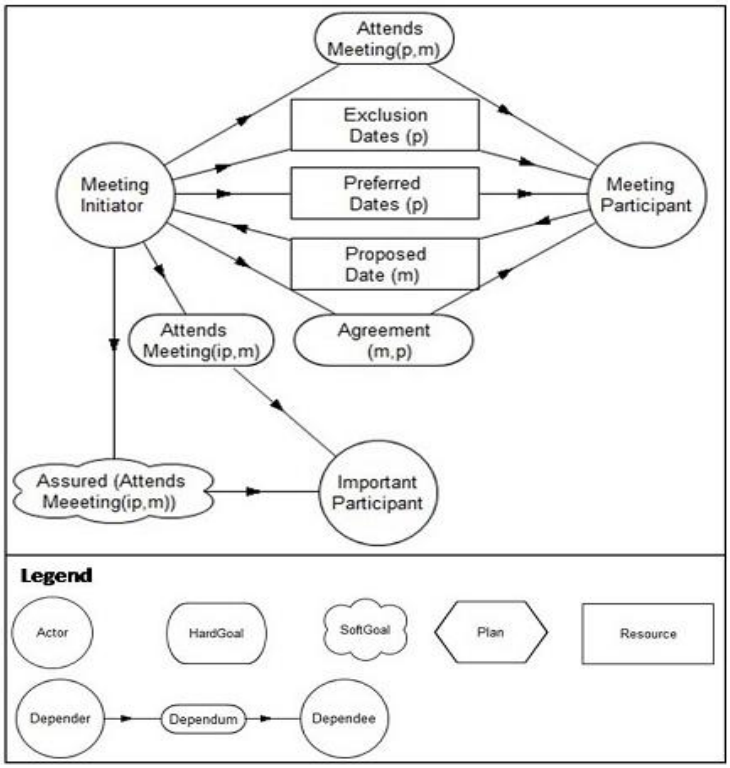


FIG. 3.9 – Meta modèle de la méthodologie Tropos [CBP05]

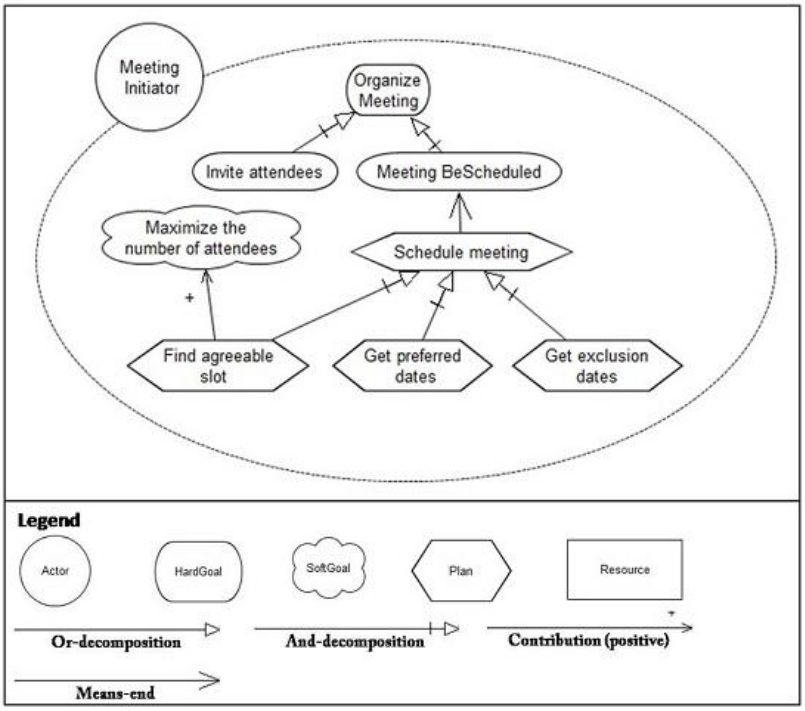
Cinq phases sont clairement définies:

1. L'analyse des besoins initiaux qui produit un modèle de dépendances stratégiques (acteurs, buts et dépendances) et un modèle de raisonnement stratégique (moyen d'atteindre les buts collectivement);
2. L'analyse des besoins finaux fournit une description du système étudié dans son environnement opérationnel et modélise le système en tant qu'ensemble d'acteurs possédant des dépendances;
3. Définir l'architecture globale du système selon des acteurs (sous-systèmes) interconnectés par leurs dépendances (flots de contrôle et de données). De nouveaux acteurs sont introduits incluant les sous-acteurs résultants de la décomposition d'un acteur :
   - Inclusion de nouveaux acteurs et délégation des sous-buts aux sous acteurs d'après une modélisation des buts
   - Inclusion de nouveaux acteurs selon un choix de style d'architecture spécifique
   - Inclusion d'acteurs contribuant positivement à la satisfaction d'exigences

4. la conception détaillée qui définit chaque composant architectural en termes d'entrées, de sorties, de contrôle et qui permet de détailler la communication entre acteurs et le comportement des acteurs en différents niveaux;
5. L'implémentation: à partir des résultats de la phase de conception, un squelette de code source sera généré dans la plateforme JACK à base d'une architecture BDI.



(a) Diagramme d'acteurs



(a) Diagramme de buts

FIG. 3.10 – Exemples des diagrammes Tropos pour phase de la conception architecturale [BGGMP02]

3.4.5 Discussion

Nous remarquons que malgré l'importance de la phase de la conception architecturale et le rôle critique de l'architecture logicielle du système dans le processus de développement, certaines méthodologies telle que MASCommonKADS ne traitent pas explicitement cette phase et d'autres comme MaSE et Prometheus modélisent la structure de l'architecture en utilisant des diagrammes dédiés informels (diagramme de vue Prometheus et le diagramme de hiérarchie de buts) et adopte les diagrammes des séquences UML et AUML pour la description des interactions entre les éléments de l'architecture. Concernant Tropos l'architecture du système est décrite en utilisant le langage i∗.

Une description architecturale formelle des systèmes, intégrée dans le processus de développement des méthodologies orienté agent sera d'un grand intérêt pour ces méthodologies dans le but de découvrir les erreurs très tôt dans le processus de développement et construire des systèmes fiables et robustes. D'une autre part le fait que la description architecturale ne soit plus une activité isolée va contribuer à l'évolution et la maturité de ce domaine.

Un exemple de l'importance attribuée à la description architecturale dans le cadre des méthodologies de développement orienté agent, les travaux réalisés dans le contexte de la méthodologie orienté agent Tropos en utilisant UML-RT [SCTAM06] (cf. section 4.3.1.4) pour l'amélioration la description architecturale des systèmes ainsi que la définition de l'ADL SKwyRL-ADL [MFKG05] (cf. section 4.3.2.4).

3.5 Les plateformes multi-agents

Les plate-formes multi-agents permettent de faciliter l'implémentation et l'exploitation des systèmes multi-agents et de réduire le coût de développement. Leur but étant de développer la plupart des composants de la technologie d'agent (communication, coopération etc.) pour permettre au développeur de fournir la résolution des problèmes de son application. Plusieurs plateformes ont été développées par des groupes de recherche universitaire et d'autres par l'industrie.

3.5.1 La plate forme JADE

JADE (Java Agent Development Framework) est une plate-forme multi-agents développée en Java par CSELT (Groupe de recherche de Gruppo Telecom, Italie). JADE traite tous les aspects qui ne sont pas particuliers aux comportements internes des agents et qui sont indépendants des applications comme le codage, l'analyse, le cycle de vie d'agent et la communication par transport de message en utilisant le langage FIPA-ACL. JADE est distribuée à travers les machines (qui doivent partager le même système d'exploitation) et la configuration peut être commandée par l'intermédiaire d'un GUI à distance et peut être même changée au cours de l'exécution en déplaçant des agents d'une machine à une autre. JADE fournit des mécanismes et des interfaces de transport des messages entre les agents présentés dans différentes plateformes.

3.5.2 La plate forme ZEUS

La plateforme ZEUS [FGL01] fournit une bibliothèque des composants et des outils de logiciel qui facilitent la conception, le développement et le déploiement rapides des systèmes de collaboration d'agent. ZEUS est mis en application comme collection de classes Java et peut être divisé en trois composants principaux: la bibliothèque composante d'agent qui présente une collection de composants de logiciel mettant en application la fonctionnalité nécessaire pour des systèmes multi-agents, le logiciel de construction d'agent qui fournit une suite intégrée d'éditeurs d'ontologie, de définition d'agent, de description, de tâche et d'organisation et les outils de visualisation de société d'agent qui peuvent être utilisés au temps d'exécution pour surveiller et contrôler le comportement d'une société d'agent.

3.5.3 La plate forme MaDKiT

MadKit [GFM00] est une plate-forme de conception et de développement des systèmes multi-agents développée en Java (source ouverte). Elle est basée sur un modèle organisationnel AGR (Agent, Groupe et Role) faisant partie d'un ensemble

plus général qui est le projet Aalaadin [FG98] permettant d'analyser et d'exprimer divers systèmes multi-agents en utilisant des concepts organisationnels. MadKit est aussi une plate-forme distribuée. Il est possible de faire communiquer, différentes plate-formes MadKit situées dans un même réseau, et donc de faire communiquer des agents de plate-formes différentes. MadKit présente quelques limites, son absence de modèle d'agent, la nécessité d'une certaine connaissance du langage de programmation Java et le manque de bibliothèque qui facilite le développement des agents complexes. Ces problèmes sont résolus par le développement des nouvelles librairies pour les agents cognitifs.

3.5.4 La plate forme JADEX

Jadex [PBL03] est une plateforme d'agents qui est basée sur JADE, respecte les standards FIPA et permet de créer des agents qui sont orientés vers les buts conformément au modèle BDI. Jadex offre un cadre et des outils de développement pour simplifier la création et le test des agents. Le moteur du raisonnement de Jadex combine les deux approches : il respecte les normes FIPA et il est basé sur le modèle BDI.

Jadex incorpore ce modèle dans les agents JADE, en introduisant des croyances, des buts et des plans comme des objets de première classe qui peuvent être créés et manipulés à l'intérieur de l'agent.

En Jadex, les agents ont des croyances, qui peuvent être n'importe quel genre d'objets Java et sont stockées dans une base des croyances. Les buts sont des descriptions implicites ou explicites d'états qui doivent être atteints. Pour atteindre ses buts, l'agent exécute des plans, qui sont des procédures programmés en Java.

3.6 Conclusion

Dans ce chapitre, nous avons représenté les concepts agent et système multiagents ainsi que leurs différentes caractéristiques. La particularité de ce nouveaux paradigme à mené à la définition de nouvelle méthodologies de développement dont nous avons détaillé quelques unes dans ce chapitre en mettant

l'accent sur la phase de conception architecturale. L'architecture logicielle du système joue un rôle crucial dans le processus de développement des systèmes et l'utilisation d'une approche formelle pour la description de cette architecture permettra sa vérification et validation et l'existence des plateformes agent peut faciliter la transformation automatique de la description de l'architecture vers une implémentation correcte. Dans le prochain chapitre, nous allons présenter les formalismes de description des architectures logicielles à base d'agents.

Chapitre 4

Les architectures logicielles à base d'agents

4.1 Introduction

Une nouvelle tendance dans l'ingénierie des systèmes multi-agents est de les considérer du point de vue d'architecture logicielle. Dans ce chapitre nous allons tout d'abord explorer les travaux qui essayent de converger les deux disciplines, puis présenter les différents formalismes qui ont été proposés pour la description des architectures logicielles à base d'agents.

4.2 Les SMA et l'architecture logicielle

Plusieurs travaux tentent de rapprocher entre les systèmes multi-agents et l'architecture logicielle. Dans [WHS06], les auteurs pensent qu'il ne faut pas considérer les systèmes multi-agents comme une approche radicale du génie logiciel, mais essayer plutôt de les intégrer dans le processus général du génie logiciel. Partant de ce point de vue ils présentent l'architecture logicielle comme un cadre adéquat pour cette intégration car, ils considèrent que les systèmes multi-agents fournissent une approche pour résoudre un problème en le décomposant en entités autonomes embarquées dans un environnement, afin de répondre aux besoins fonctionnels et non fonctionnels du système. Cette perspective indique que les systèmes mult-iagents constituent une approche pour résoudre des problèmes logiciels. En particulier, un système multi-agents est une décomposition spécifique du système en éléments en interaction destinés à atteindre les besoins du système, et

c'est bien ça l'architecture logicielle. Les éléments de cette architecture sont les agents, les ressources, les services et l'environnement. L'interaction entre ces éléments varie des simples traces de phéromones jusqu'aux protocoles sophistiqués de négociation. Les auteurs mentionnent que les systèmes multi-agents sont une famille d'architectures logicielle ayant certains attributs de qualité tels que la robustesse et la flexibilité, ces attributs peuvent former des critères pour choisir les systèmes multiagents parmi d'autres architectures. L'intégration des systèmes multi-agents dans le processus général du génie logiciel – en le reliant à l'architecture logicielle- peut amplifier leur adoption industrielle [WHH09]. Yu et Cai [YC06], à leur tour trouvent que le fait de considérer les systèmes multi-agents comme une architecture logicielle va permettre de réduire l'écart entre la modélisation formelle des systèmes multi-agents et la pratique industrielle. Mouratidis et *al* [MFKG05] justifient l'adoption de ce point de vue par le fait que l'architecture logicielle permet aux concepteurs de maîtriser la complexité des systèmes multiagents et de s'assurer que le système satisferait ses besoins clés.

Selon Azaiez [Aza07] l'approche orientée architecture et plus précisément les ADLs ont plusieurs apports pour l'ingénierie des systèmes multi-agents, qui sont résumés dans le tableau suivant (tableau 4.1)

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. 4.1 – Apports de l'approche orientée architecture dans le contexte multi-agents [Aza07]

4.3 La description architecturale des systèmes à base d'agents

Comme cité précédemment (section 4.2), il existe une communauté qui considère les systèmes multi-agents du point de vue d'architecture logicielle. Bien que les systèmes multi-agents représentent un nouveau paradigme dans le génie logiciel, et que la majorité des formalismes de description de l'architecture logicielle ont été

développés '*en pensant composant*', logiquement une première tentative pour la description des architectures logicielle à base d'agent était d'essayer d'adapter les formalismes existants. Dans la suite nous allons présenter les formalismes utilisés pour la description architecturale des systèmes multi-agents.

4. 3.1 UML

Nous avons présenté dans le deuxième chapitre (section 2.2.2) l'utilisation d'UML pour la description des architectures logicielles à base de composants, la communauté agent à son tour a essayé d'exploiter UML pour la description des architectures à base d'agents.

4 .3 .1.1 Les langages de modélisation

Plusieurs langages ont été proposés dans le cadre de l'ingénierie orienté agent, qui sont basés sur UML et qui ont été définis par l'extension d'UML comme le cas pour AUML (Agent -UML) [BMO01] [HOB04] et AML (Agent Modeling Language) [CT04] [CT07].

Le langage AUML a été proposé en 1999 comme étant une extension de la notation UML pour la modélisation des agents. L'idée des auteurs et de réutiliser le plus que possible les diagrammes UML quand ces derniers répondent aux besoins des concepteurs des systèmes à base d'agents et les faire étendre avec les mécanismes d'extension pour les adapter à la particularité des agents. Cependant les auteurs indiquent qu'AUML n'est pas limité aux diagrammes UML, d'autres notations peuvent être utilisées ou carrément créer des nouvelles notations quand jugé nécessaire [HOB04]. L'extension d'UML concerne les diagrammes de séquence pour modéliser les protocoles d'interaction entre agents ainsi que les diagrammes de classe pour modéliser les agents.

AUML modélise les protocoles d'interaction entre agents (ou AIP:Agent Interaction Protocol) sur trois niveaux de définition :

- Le premier niveau concerne un protocole dans son ensemble par l'utilisation de templates génériques produisant des paquetages pour des protocoles spécifiques.
- Le deuxième niveau représente les interactions entre agents grâce à des diagrammes d'interactions : diagrammes de séquences modifiés, diagrammes de collaborations, diagrammes d'activités ou bien diagrammes d'états/transitions. La modification des diagrammes de séquence concerne l'ajout de nouveaux types de connecteurs afin de prendre en compte l'indéterminisme du comportement d'un agent.
- Le troisième niveau de représentation correspond aux processus internes aux agents qui sont principalement modélisés par des diagrammes d'états/transitions.

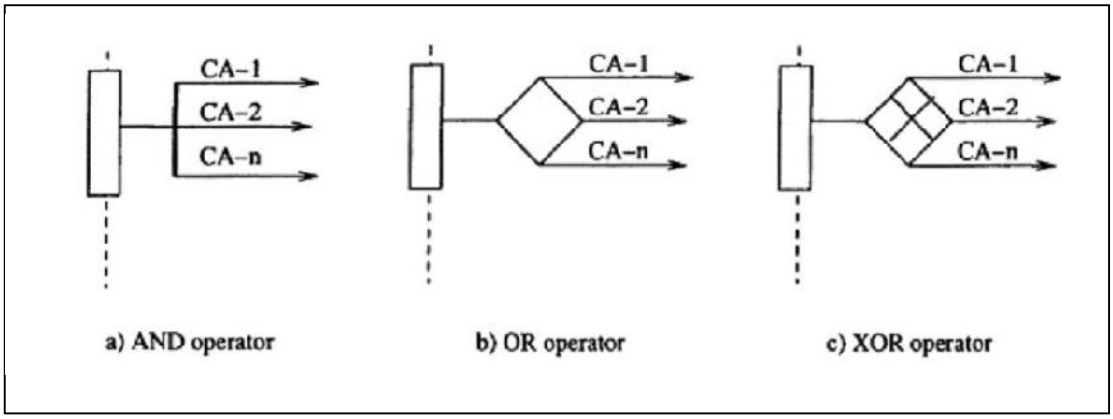


FIG. 4.1 – Les branchements définis par AUML [HOB04]

Pour modéliser les types d'agents et leurs propriétés, une extension aux diagrammes de classe UML a été faite. Comme définit dans [Bau01] un diagramme de classe Agent contient plusieurs éléments :

- Un nom d'agent ;
- Une description d'état ;
- Des actions ;
- Des méthodes ;
- Des capacités, descriptions de service, protocoles supportés ;
- Une référence vers un agent-head automata.

Les auteurs trouvent que les concepteurs peuvent fournir une vue globale du système en utilisant les différentes classes d'agents et leurs relations sans tenir compte de ce que pourraient être les éléments au sein des classes. Ce qui convient particulièrement à une description de l'architecture du système [HOB04].

AML [CT04] est aussi un langage de modélisation des systèmes multi agents basés sur une extension d'UML 2.0. AML fournit un certain nombre d'éléments conçus pour décrire différents aspects statiques et comportementaux suivants:

- Les entités d'un SMA : une entité désigne un agent, une ressource ou un environnement
- l'aspect social : cet aspect concerne les concepts de rôle et d'organisation.
- l'abstraction et la décomposition du comportement
- les interactions de communication (communicative interactions)
- les services
- les interactions d'observation et de modification de l'environnement
- l'aspect mental utilisé pour modéliser les attitudes mentales des entités traitant les concepts de but, croyance, plan,...
- le déploiement des SMA
- la mobilité d'agents
- les ontologies importez un document à traduire.

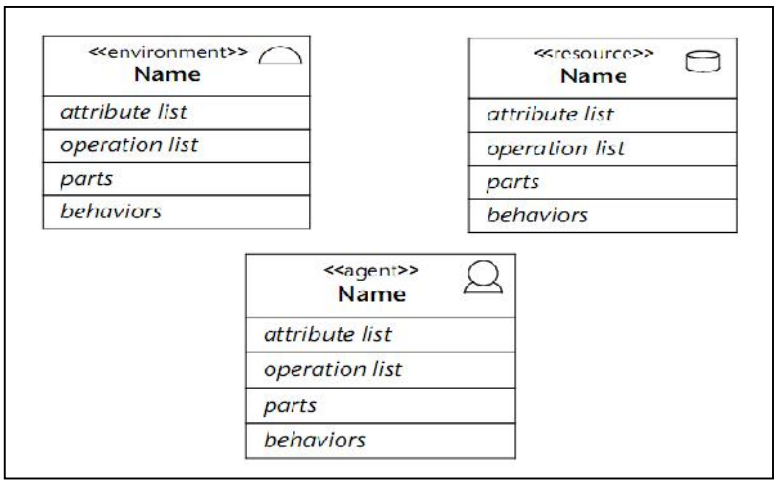


FIG. 4.2 – Notations des types d'entité avec AML [CT07]

Bien que les deux langages de modélisation AUML et AML fournissent des notations dédiés pour décrire les systèmes multiagents, ils souffrent d'un manque de support pour différents éléments architecturaux comme étant des entités de première classe [Wey10].

4 .3 .1.2 Yim et al 2000

Dans cet article [YCJP00], les auteurs ont introduit une méthode pour la description de l'architecture des systèmes multi-agents basée sur l'extension d'UML. Les extensions prennent en compte les caractéristiques des SMA et les concepts de l'architecture logicielle. Les auteurs trouvent que l'utilisation d'UML va permettre aux concepteurs des systèmes multi-agents l'exploitation des outils UML ainsi que la large expérience et connaissance acquises par les utilisateurs d'UML

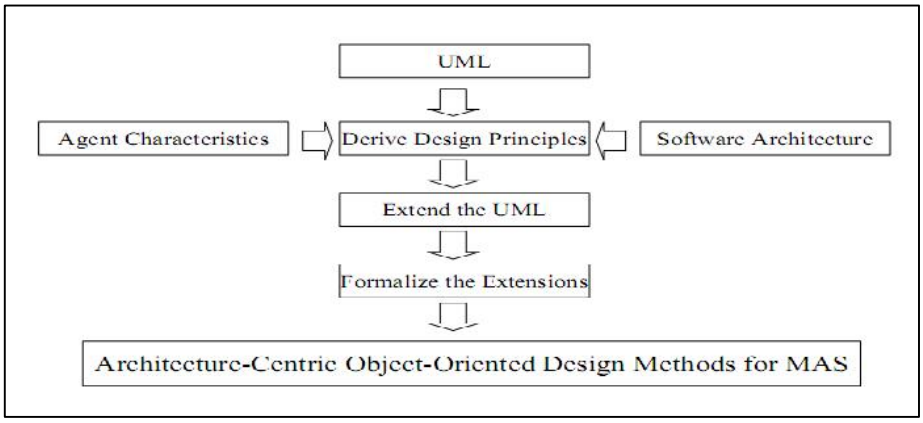


FIG. 4.3 – Cadre conceptuel de l'approche proposée [YCJP00]

Les principes de conception tirés sont :

- La structure de l'architecture se compose d'agents ayant des rôles, de composants et de connecteurs.

- Un agent est un bloc de construction primitif pour la conception de systèmes multi-agents.

- La conception centrée architecture favorise le mécanisme basé patrons sur celui de l'héritage.

Les auteures fournissent des extensions d'UML sur la base des principes cités ci-dessus, puis les extensions sont formalisées par OCL (Object Constraint Language). Les formalisations ont le rôle de contraintes syntaxiques ou sémantiques pour restreindre le modèle ou les éléments de modélisation. Un exemple de ces formalisations est montré dans la figure suivante (figure 4.4) :

Le stéréotype ArMessage est une instance de la méta-classe Message
[1] ArMessages sont étiquetés pour identifier les types de protocoles arMsgType : enum {advertise, unadvertise, ask, ask-all, reply, … }
Le stéréotype ArOperation est une instance de la méta-class Operation
[1] ArOperatons sont étiquetés pour identifier le ArMessages correspondants arOprType : enum {advertise, unadvertise, ask, ask-all, reply, recruit, …
[2] ArOperations ne renvoie aucune valeur
self.parameter ! not exists(p | p.kind = return)
Le stéréotype ArInterface est une instance de la méta-class Interface
[1] Toutes les opérations de ArInterface correspondent au stéréotype ArOperation.
self.oclType.operation ! forall (o | o.stereotype = ArOperation)

FIG. 4.4 – Restrictions sur l'interface d'un agent [YCJP00]

4.3 .1.3 MASIF-DESIGN

MASIF-DESIGN profile [GM01] est un ADL définit comme un profile UML qui permet la description d'un système multi-agents en prenant en considération l'environnement dans lequel les agents vont s'exécuter. Les environnements en question sont ceux qui sont en conformité avec le standard de l'OMG-MASIF.

MASIF platform elements	**Type level** UML Meta-model Class	**Instance level**	Stereotype Name
Region	Stereotyped Subsystem	Stereotyped Subsystem instance	Region
Agent System	Stereotyped Node	Stereotyped Node instance	Agent System
Core Agency	Stereotyped Subsystem	Stereotyped Subsystem instance	CoreAgency
Place	Stereotyped Package	Stereotyped Component instance	Place
Agent	Stereotyped Component	Stereotyped Component instance	Agent

TAB. 4.2 –Les stéréotypes modélisant les éléments de la plateforme MASIF [GM01]

L'idée est d'utiliser les mécanismes d'extension d'UML pour définir un profile pour les systèmes à base d'agents, ce dernier permettra aux concepteurs de détailler leur architecture et l'enrichir avec des informations qui concernent les aspects de distribution relatifs à l'environnement d'exécution. Les éléments de la plateforme MASIF sont :

- *Region* relie tous les agents du système et permet le transfert point-à-point de l'information entre eux. La Région dispose d'un registre région avec toutes les informations sur les *Places* et les *Agents*

- *Agent System* c'est la plate-forme qui permet de créer, d'interpréter, d'exécuter, de transférer et de mettre fin aux agents.

- *CoreAgency* il prend en charge les services de gestion fournis à des agents dans un système d'agent.

- *Place* est un contexte au sein *Agent System* dans lequel un agent peut fonctionner. Il peut fournir des fonctions telles que le contrôle d'accès.

- *Agent* c'est l'unité exécutable de base.

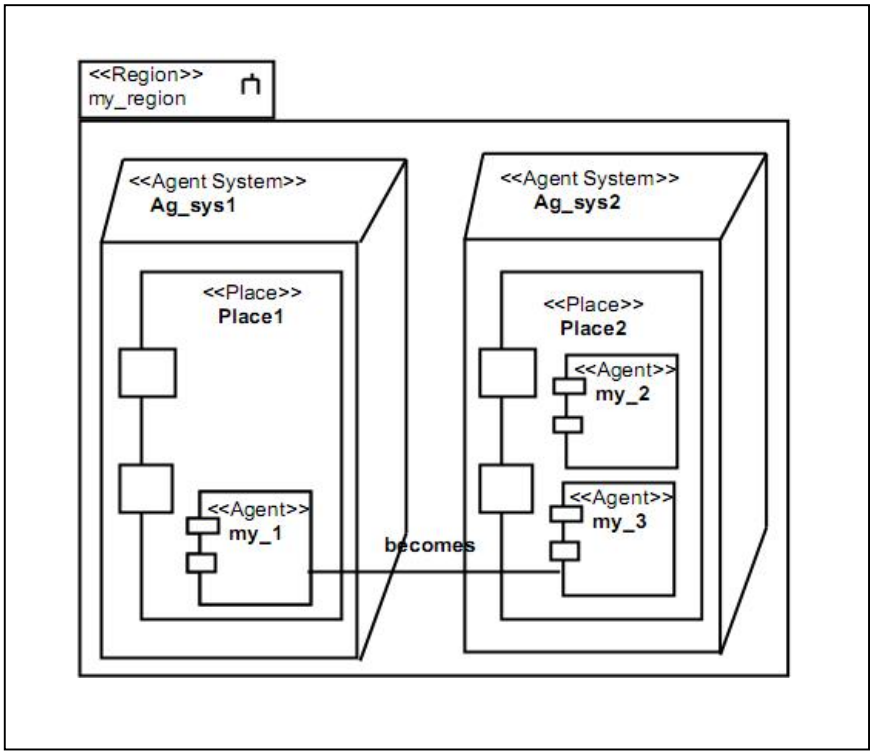


FIG. 4.5 – Architecture d'un système en utilisant MASIF-DESIGN

4 .3 .1.4 Silva et al 2006

Dans [SCTAM06], les auteurs proposent une approche pour la conception architecturale des systèmes multi-agents dans le contexte de la méthodologie Tropos. Pour la modélisation et l'analyse du système durant la phase de la conception architecturale, Tropos adopte les concepts de la framework i*, cependant l'utilisation

de i* en tant qu'ADL n'est pas appropriée du fait qu'elle présente certaines limitations pour décrire le comportement et la structure du système au niveau de détail requis pour la conception architecturale. Une approche utilisant UML-RT (UML- Real Time) comme un ADL dans Tropos a été proposée.

Les concepts utilisés par Tropos sont :

- Acteur (actor) : modélise une entité ayant des buts, il représente un agent (physique ou logiciel), un rôle ou une position. le rôle est une caractérisation d'un comportement d'un acteur dans un contexte précis tant que la position est un ensemble de rôles d'un acteur

- But : modélise l'intérêt d'un acteur, il existe deux types de but : le but flou (soft goal), il n'a pas de définition exacte et/ou des critères permettant de savoir s'il a été satisfait ou non (il représente généralement un attribut non fonctionnel du système).Le but exacte (hard goal) le contraire d'un but flou (représente une fonctionnalité du système).

- Plan : modélise une façon de faire pour satisfaire un but

- Ressource : modélise une entité physique ou informationnelle

- Dépendance : entre deux acteurs où un acteur (Depender) dépend d'un autre (Dependee) afin d'atteindre un but, d'exécuter un plan ou de fournir une ressource (Dependum).

Le diagramme d'acteurs (Actor Diagram) définit l'architecture globale du système selon des acteurs (sous-systèmes) interconnectés par leurs dépendances (flots de contrôle et de données)

Une transformation entre les concepts de Tropos et ceux de UML-RT est montré par la figure suivante (figure 4.7) :

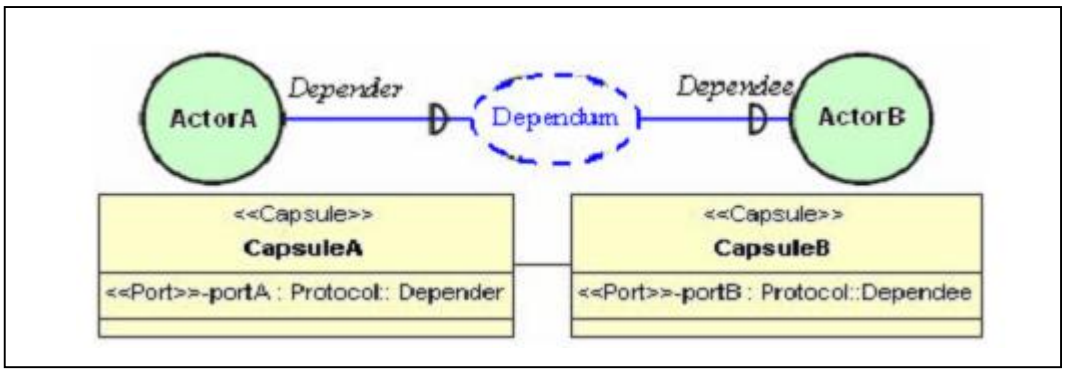


FIG. 4.6 – Transformation d'une dépendance entre acteurs vers UML-RT [SCTAM06]

Un modèle conceptuel contant les notions relatives à la spécification des systèmes multi-agents a été proposé où les auteurs considèrent que l'agent rôle (Agent role) est l'unité de base pour la description de l'architecture donc le rôle agent ainsi que la façon de construire une organisation à partir de ces rôle agents doivent être défini.

Une extension du diagramme de classe UML par les stéréotypes a été utilisée pour représenter la structure de l'agent rôle et l'organisation comme suit:

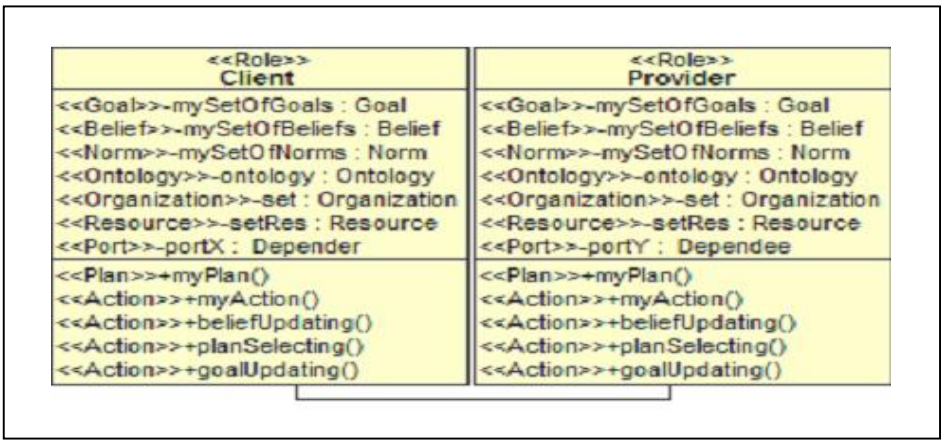


FIG. 4.7 – Diagramme d'agent rôle [SCTAM06]

- <<Organisation>>, <<Belief>>, <<Goal>>, <<Norm>> , <<Ontology>>, <<Resource>>, <<Port>>, <<Ontology>>: attributs stéréotypés

- <<Plan>>+myPlan(), <<Action>>+MyAction, <<Action>>+beliefApdating, <<Action>>+planSelecting, <<Action>>+goalUpdating : operations stéréotypées.

Selon la classe rôle de la figure 4.7, un rôle appartient au moins à une *organisation*. Un agent qui joue un rôle a au moins une *croyance* et un *but*. Les agents jouant un rôle peuvent s'échanger des messages selon au moins un *protocole* dont ses messages sont conforme à une certaine *ontologie*. Un *plan* se compose de plusieurs actions et afin de satisfaire un but, l'agent qui joue un rôle choisit un plan parmi autres en exécutant *planSelecting()*. Pour exécuter une action l'agent peut avoir besoin d'une *ressource* dans l'environnement. Chaque agent est capable de mettre à jour ces croyances et ces buts en utilisant *beliefUpdating()* et *goalUpdating()* respectivement chaque fois que l'environnement change, les buts atteints ou des messages arrivent .

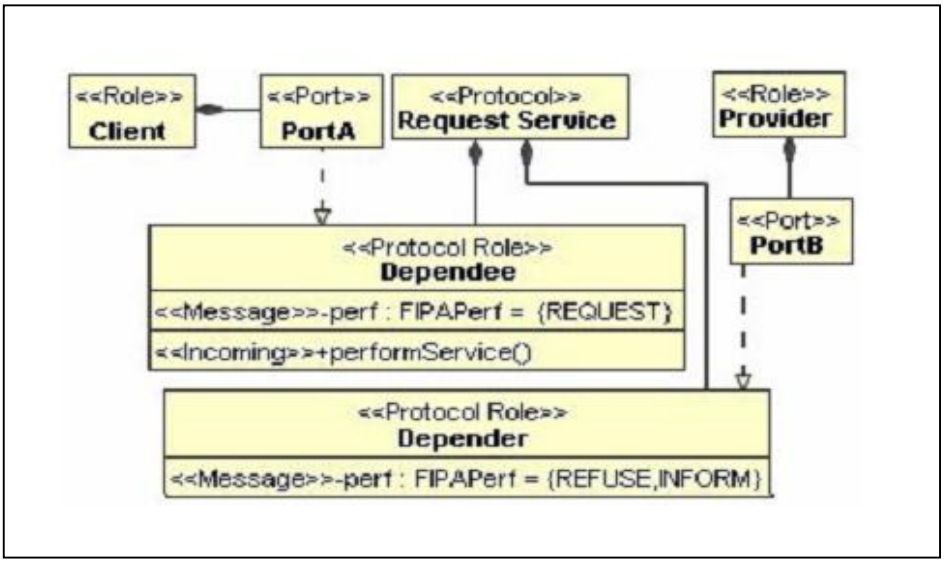


FIG. 4.8 – La spécification du protocole [SCTAM06]

La spécification du protocole à son tour a été étendue pour prendre en charge les performatives FIPA. Le stéréotype <<Message>> dans la classe <<Protocol Role>> indique les performatives FIPA qu'un agent exécutant ce rôle peut utiliser.

Dans Kavi et *al* [KJB03], les auteurs proposent des extensions d'UML pour la modélisation des systèmes multi-agents. Les extensions ne concernent pas exactement la description de l'architecture, mais une palette d'extensions a été proposée permettant la modélisation des systèmes à base d'agents BDI comme les concepts : d'agent, but, croyance ….ect et qui peuvent servir à la description architecturale du système.

4.3.2 ADLs pour les architectures à base d'agents

4.3.2.1 MAS-ADL

MAS-ADL [CG99], est un langage simple et personnalisé pour la description des architectures logicielles. Il a été définit pour un système de prototypage de gestion des trains de marchandise dans le cadre du projet EuROPE-TRIS. Ce langage permet la définition des classes d'agent, des instances de ces classes (i.e les agents constituant le système) et les liens entre ces instances, qui relient un service fournis par un agent à un service requis par un autre agent.

Dans le cas du système proposé par les auteurs, l'architecture a été décrite en utilisant MAS-ADL comme suit :

- la définition des classes d'agent, cinq classes ont été définies :

```
agentclass Terminal {
      kind: logical;
      architecture: reactive;
      requires: loading_authorization,
                dispatching;
      provides: nil;}

agentclass ProdDep {
      kind: logical;
      architecture: reactive;
      requires: nil;
      provides: loading_authorization;}
```

```
agentclass Coim {
      kind: logical;
      architecture: reactive;
      requires: loading_authorization;
      provides: dispatching,
                loading_authorization;}

 agentclass Section{
       kind: logical;
       architecture: reactive;
       requires: dispatching,
                 time_band_update;
       provides: dispatching;}
```

```
agentclass Timer {
      kind: logical;
      architecture: proactive;
      requires: nil;
      provides: time_band_update;}
```

- la définition des instances de classes, huit instances ont été définit

```
Agent Instances {
      4 terminal Terminal;
      1 coim Coim;
      1 pr_deputy ProdDep;
      3 section Section;
      1 timer Timer;}
```

- la définition des liens entre les différentes instances (agents) du système

```
link{
terminal*.loading_authorization<-
coim.loading_authorization;
terminal1.dispatching<-section1.dispatching;
terminal2.dispatching<-section1.dispatching;
terminal3.dispatching<-section2.dispatching;
terminal4.dispatching<-section3.dispatching;
coim.loading_authorization<-
pr_deputy.loading_authorization;
section*.dispatching<-coim.dispatching;
section*.time_band_update<-timer.time_band_update;}
```

4.3.2.2 ADLMAS

Le langage ADLMAS [YC06], a été proposé comme un moyen de réduire l'écart entre la modélisation formelle des systèmes multi-agents et la pratique industrielle, la raison pour laquelle ils ont considérer les systèmes multi-agents du point de vue d'architecture logicielle. Jugeant les ADLs existants d'inadéquats pour la description des architectures à base d'agents BDI, les auteurs ont proposé leur propre ADL. ADLMAS est fondé sur une théorie formelle qui est les réseaux de Petri orientés objet (OPN : Object Oriented Petri nets) .

ADLMAS = {Computing agents, Connecting agents, Configurations}

- Agents de calcul: un ensemble d'agents de calcul, chaque agent définit par un 2-uplet {ID, SA} ou ID est un identificateur de l'agent et SA sa structure (les structures des différents agents représentent les objets du OPN).

- Agent de connexion : un ensemble d'agents de connexion, chaque agent définit par un 5-uplet {MPR, KBF, T, F, Role} ou MPR représente des relations de passage de message (un tuple du OPN),

Les agents dans ADLMAS sont basés sur le modèle BDI(Belief, Desire, Intention) , la structure interne des deux types d'agent (agent calcul et agent de connexion) est également définie.

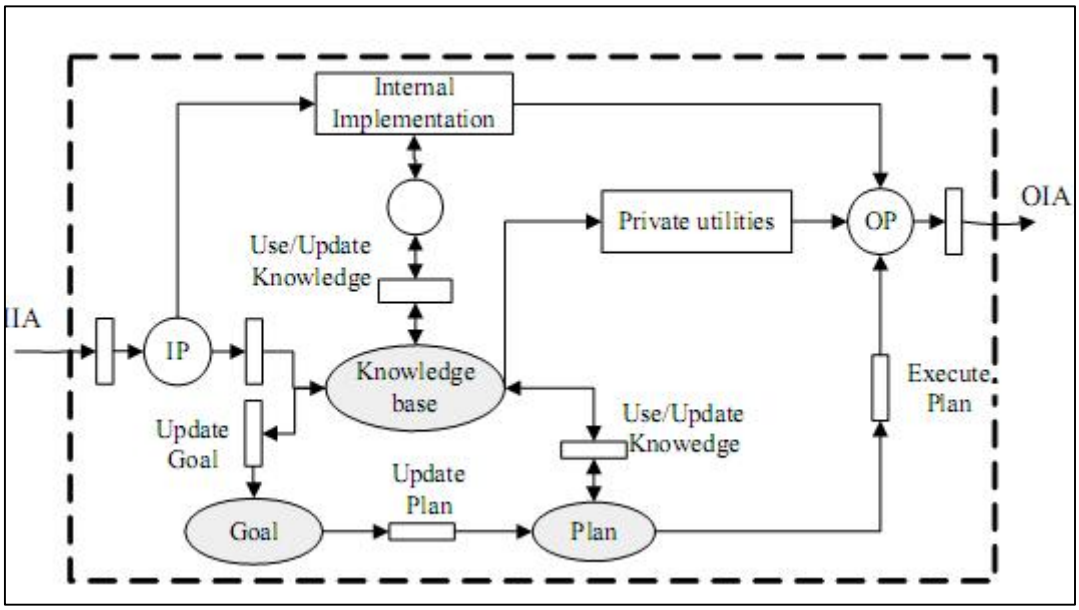


FIG. 4.9 – Méta modèle de l'agent de calcul [YC06]

Les agents de calcul interagissent entre eux et avec l'environnement pour assurer les fonctionnalités du système, alors que les agents de connexion jouent le

rôle d'un agent facilitateur. La communication entre les agents de l'architecture s'effectue via passage de messages FIPA ACL.

L'ADL définit le concept de groupe qui est un ensemble d'agents de calcul avec un agent de connexion. La configuration de l'architecture est obtenue en reliant les différents groupes.

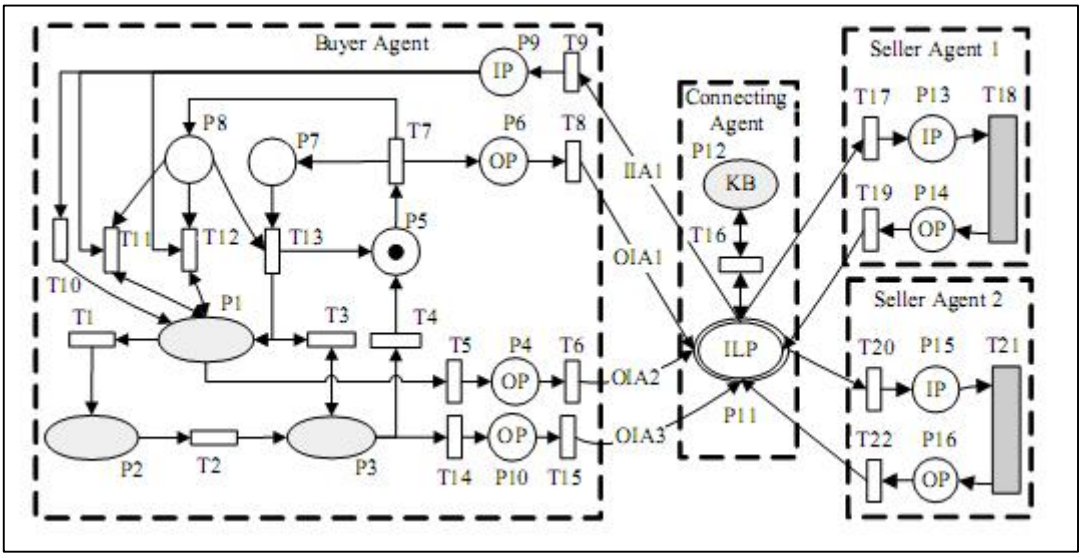


FIG. 4.10 – Architecture d'un système de commerce électronique en ADLMAS [YC06]

Un apport essentiel de ADLMAS basé réseaux de Petri objet est la vérification et la validation de l'architecture (ex : vivacité , abscence d'interblocage) avant implémentation ce qui va permettre le développement de systèmes multi-agents fiables et sûr.

4 .3 .2.3 L'ADL π-net

Le langage de description d'architecture π-net ADL [YCX06] [YCWH06] est un langage basé sur les réseaux de Petri orienté agent (AOPN :Agent Oriented Petri Nets) et le π-calculus [MPW92]. Bien que les réseaux de Petri sont simples et permettent la vérification de l'architecture, leur structure est statique, rendant difficile la représentation des architectures dynamiques des systèmes multi-agents.

Le π-calculus se présente comme une méthode appropriée pour la description des systèmes où les composants sont dynamiquement crées et supprimés et les interactions entre les composants sont dynamiquement établies et modifiées. C'est la raison pour laquelle les auteurs ont opté pour la combinaison du π-calculus et AOPN pour tirer profit de cette complémentarité.

Les systèmes multi-agents sont des systèmes dynamiques, ou les agents peuvent rejoindre ou quitter l'application à l'exécution. Le changement de la structure et du comportement du SMA (création, suppression, reconfiguration, etc) peut être décrit par le π-calculus puis analyser et vérifier par des outils de π-calculus. Dans π-net ADL les trois éléments constituant l'architecture a savoir : les agents de calcul, les agents de connexion et la configuration ont été redéfinit en leur ajoutant des descriptions en π-calculus.

Exemple :

$$AcquireKnowledge(nk) = x(nk).UpdateKb(nk).$$

Cela signifie que l'agent de calcul reçoit de nouvelles connaissances nk sur le cannal *x*, et ensuite mettre à jour sa base de connaissances.

$$QueryService(i,\ r,\ p) = i(y).([y = a]\overline{r}\langle p\rangle + [y \neq a]\overline{Subscribe}(y))$$

Cela signifie qu'un agent de calcul envoie une requête *y* et l'agent de connexion reçoit la requête sur le canal *i* puis vérifie s'il existe un service *a* qui satisfait cette requête, si ce service existe l'agent de connexion va envoyer à l'agent de calcul demandant le service l'identifiant de l'agent de calcul offrant ce service, sinon il va inscrire le service.

Dans la configuration de l'architecture, les différents agents sont considérés comme des processus communicant via des canals.

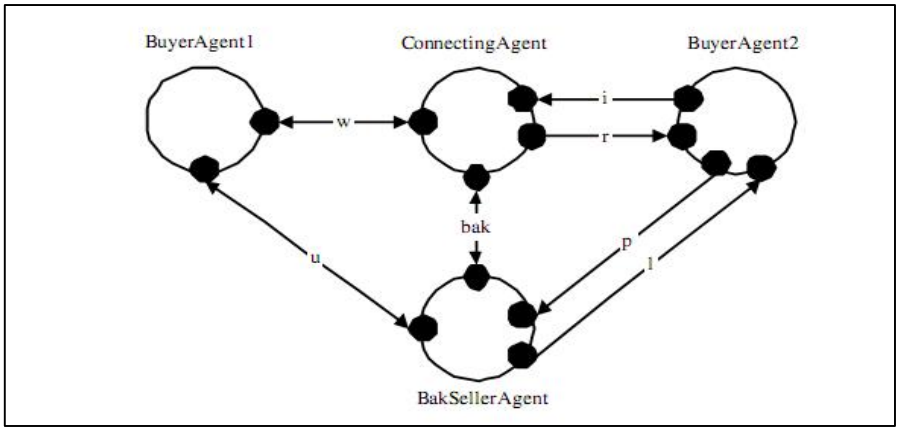


FIG. 4.11 L'architecture du système de commerce électronique sous forme d'un graph de flux en π-calculus [YCX06]

La vérification et la validation de l'architecture s'effectue sur deux étapes :

- La première consiste à utiliser des outils pour les réseaux de Petri afin de vérifier des propriétés tel que : la vivacité, l'absence d'interblocage et la bornitude.

- La deuxième se fait après l'ajout de la description de la dynamique du système en π-calculus, dan ce cas des outils supportant le π-calculus, seront utilisés pour vérifier la vivacité après que le système évolue.

4 .3 .2.4 L'ADL SKwyRL-ADL

Les systèmes modernes sont distribués et ouverts ce qui a changé profondément l'ingénierie des systèmes d'information. L'ADL SKwyRL-ADL (Socio-Intentional ArChitecture for Knowledge Systems & Requirements ELicitation) [MFKG05] tente de répondre à ses nouveaux besoins, en proposant une architecture à base d'agents BDI et tenant compte des aspects de sécurité. En se basant sur l'architecture BDI pour les agents, et en explorant les ADLs existants et la littérature sur la sécurité, les auteurs ont proposé un ensemble de concepts pour leur ADL. La figure 4.12 représente ces différents concepts ainsi que les relations entre eux.

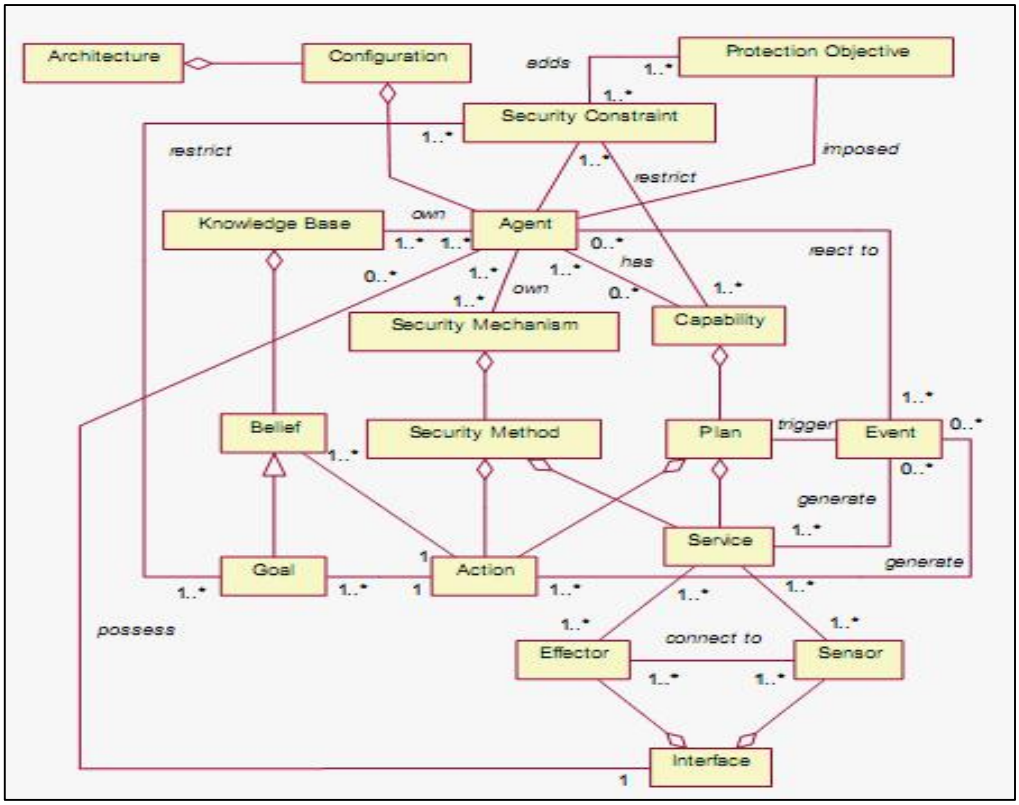


FIG. 4.12 – Méta modèle de SKwyRL-ADL [MFKG05]

Le langage de spécification Z [Spi92] a été utilisé pour la description formelle des différents concepts de l'ADL SKwyRL-ADL la figure 4.13 montre la spécification du concept but :

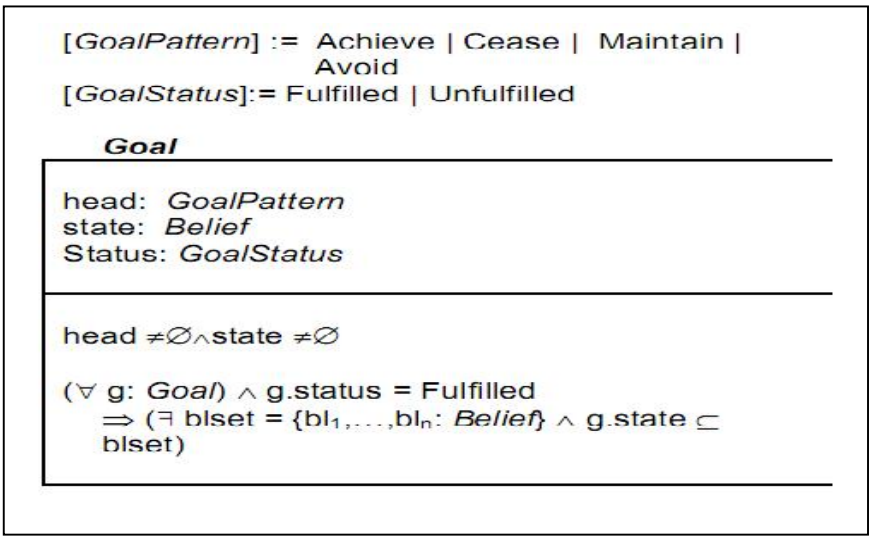


FIG. 4.13 – Spécification en Z du concept But [MFKG05]

L’ADL SKwyRL-ADL est composé de trois sous modèles :

Le modèl de l’agent il identifie les états d'un agent et son comportement potentiel. L'agent doit avoir des connaissances sur l'environnement pour pouvoir faire des décisions. Les connaissances sont contenues dans les agents sous la forme de plusieurs bases de connaissances. Une base de connaissances se compose d'un ensemble de croyances que l'agent a sur l'environnement et d’un ensemble de buts qu'il poursuit.

Le modèl de sécurité : qui est composé de :

- *Objectif de protection* indique un attribut de sécurité désiré que l'agent peut avoir, un agent peut ne pas avoir des objectifs de sécurités comme il peut avoir plusieurs objectifs.

- *Contrainte de sécurité* une contrainte de sécurité définit un ensemble de restrictions aux buts et aux capacités de l'agent. Ces restrictions sont liées à la sécurité et sont imposées par l'environnement de l'agent

- *Mécanisme de sécurité* représente un ensemble de méthodes de sécurité standards que l'agent peut avoir et ils aident à la satisfaction des objectifs de protection de l'agent.

- *Méthode de sécurité* Une méthode de sécurité définit une séquence d'actions et / ou de services tels que les algorithmes de cryptographie et les protocoles sécurisés utilisés pour réaliser les objectifs de protection de l'agent**.**

Le modèl architectural décrit les interactions entre les agents qui composent le système. Les configurations sont le concept central de la conception architecturale, ils permettent de définir la topologie d'un système multi-agents. La topologie est définie par un ensemble de liaisons entre les services fournis et requis. Un agent interagit avec son environnement grâce à une interface composée de capteurs et d’effecteurs. Un effecteur fournit un ensemble de services à l'environnement tandis qu’un capteur nécessite un ensemble de services de l'environnement. Un service est

une opération effectuée par un agent qui interagit en dialoguant avec un ou plusieurs agents. Enfin, le système multi-agents est spécifié avec une architecture qui est composé d'un ensemble de configurations.

Agent: {Billing-Processor

Interface

Effector[provide(*shopping_cart*)]

Effector[provide(*billing*)]

Effector[provide(*stock_orders*)]

Effector[provide(*finance_security*)]

Sensor[require(*strategic_behavior*)]

Sensor[require(*statistical_info*)]

KnowledgeBase:

Stock_KB Pricing_Kb

BP_Customer_KB Providers_KB

BP_System_KB Statistical_KB

Protection Objectives:

Confidentiality_PO Integrity_PO

Availability_PO Non_Repudiation_PO

Authentication_PO AccessControl_PO

Security mechanisms:

Encipherment_SM DIgitalSignature_SM

AccessControl_SM DataIntegirty_SM

AuthenticationExchange_SM

TrafficPadding_SM RoutingControl_SM

Notarization_SM

Capabilities:

Shopping_Cart_Management_CP

Billing_CP Stock_Management_CP

Statistic_CP

}

FIG. 4.14 – Description en SKwyRL-ADL d’un agent processeur de facturation [MFKG05]

4.3.2.5 WRIGHT*

Les auteurs proposent leur ADL WRIGHT* [LPH06] comme étant un nouveau langage pour la description des architectures orientées coopération. Ce langage se base sur l'extension de l'ADL WRIGHT [AG97].

Les définitions des principaux concepts de l'architecture orientée coopération ont été proposées comme suit:

- **Un agent** : est un triplet

$$\begin{cases} Agent := <R,\Omega,\Phi> \\ R := \{<role><responsability>\}^* \\ \Omega := \{<Computation><Reasoning><Cooperation>\}^* \\ \Phi := \{<Belif><Desire><Intention>\}^* \end{cases}$$

Dan cette définition, R est un ensemble de responsabilité, Ω un ensemble de capacités y inclut le raisonnement et la coopération, Φ l'ensemble des propriétés mentales de l'agent par exemple dans le cas d'un agent BDI ces propriétés sont les croyances, désires et intentions.

- **Coopération** : est un quadruplet

$$\begin{cases} Cooperation := <T,A,B,E> \\ T := \{<task><goal>\}^* \\ A := \{<Agent>\}^* \\ B := \{<Behaivor>\}^* \\ E := \{<constraint><situation>\}^* constraint := \{<rule><specification>\}^* \end{cases}$$

Dans cette définition, T est un ensemble des buts et taches, A est l'ensemble des agents, B un ensemble de comportements et E les contraintes de l'environnement.

T peut être considéré comme les prés conditions de coopération, suivant lesquelles un ensemble d'agents A est formé. La coopération doit tenir en compte un mécanisme de support de coopération pour la prise en charge des problèmes liés à résolution des conflits, la maintenance des connaissances et données partagées, ….ect

- **Support de coopération** : est de fournir des mécanismes ou services tel que le support de communication et de coordination.

Cooperation_support :=<service,customers,constraints>

*service := {<port><content>}**

*customers := {<Agent>}**

*constraints := {<rule><specification>}**

- **Architecture orientée coopération** : le modèle de l'architecture logicielle d'un system orienté coopération est basé sur la coopération :

SA_CO={*cooperation*, *supporters*, *configurations*}

Pour la description de leur architecture orienté coopération les auteurs préfèrent l'utilisation des méthodes formelles, les ADLs se présentent comme un candidat mais ces derniers sont basé sur le concept de composant et il s'avère difficile de les utiliser pour la description des systèmes multi-agents ou autres systèmes coopératifs. La solution été l'extension des ADLs pour supporter les concepts des architectures orientées coopération.

WRIGHT a été choisi comme L'ADL de base auquel les extensions suivantes ont été effectuées :

- Ajout des notations de *Cooperation* et *Supporter* à l'ensemble des éléments de base.

- Dans un style architectural le concept *Interface Type* a été étendu pour supporter des nouvelles propriétés telle que : *Agent*, *Task* et *Service* et des nouveaux opérateurs liés aux concepts de *Cooperation* et *Supporter* ont été ajoutés comme : *Tasks()*, *Agents()*, *Services()* et *Customers()*.

- La définition dans CSP de l'événement *cooperation* qui est un événement composite qui représente les processus (agents) effectuant les comportements liés pour atteindre les objectifs communs du système. Le symbole « Θ » représente l'opérateur de la coopération.

  Soit P un ensemble de processus :

  $Event_{coop} = \forall pi, pj \in P | i, j = 1 \ldots n, i \neq j, (pi \Theta pj)$

Il existe d'autres travaux qui n'ont pas proposé des ADLs mais plutôt des cadres formels pour la spécification des architectures à base d'agents.

L'idée de Reza et Grant [RG04] consiste à faire une correspondance entre les éléments des systèmes multi-agents et ceux de architecture logicielle, puis modéliser la structure du système en utilisant la DST (diagrammatic syntactic theory) proposée par Cordy et Dean [CD95], qui offre une description abstraite du système. La structure en DST sera ensuite transformée en un HPrTNs (Hierarchical Predicate Transition Nets) [He96], dont les éléments seront raffinés. Les HPrTNs donnent une vue plus concrète de l'architecture et sont considérés comme un modèle formel exécutable de cette dernière permettant sa vérification.

Algar et Zheng [AZ05], réclament que le fait d'assurer des caractéristiques spécifiques du système au niveau architectural est de peu de valeur sauf si ces caractéristiques sont aussi assurées au niveau de l'implémentation. Pour cette raison ils proposent une architecture à base d'agent BDI où chaque agent est modélisé par un ESM (Extended state machine), puis la structure globale du système est obtenue par la composition parallèle des différents ESMs (agents) constituant le système. L'implémentation du système dans un langage de programmation de haut niveau peut être testée pour sa conformité à l'architecture en générant des cas de test à partir de l'ESMs.

4.4 Conclusion

Nous avons présenté dans ce chapitre la convergence qui existe entre les systèmes multi-agents et l'architecture logicielle. Donc les systèmes multi-agents sont considérés comme un style architectural ayant certaines caractéristiques telles que la robustesse et la flexibilité pouvant être des critères pour les développeurs pour choisir les systèmes multi-agents parmi d'autres styles pour résoudre certains types de problèmes. Les architectures à base d'agents ont leur particularité qui rend leur description par les formalismes basés composant inappropriée. Deux axes ont émergé : le premier préconise l'utilisation d'UML avec certaines extensions du fait qu'il est le langage le plus utilisé dans l'industrie et l'autre préfèrent l'utilisation des ADL comme étant des méthodes formelles permettant la vérification et la validation de l'architecture.

Chapitre 5

Une approche pour la description des architectures à base d'agents

5.1 Introduction

Nous avons présenté dans le chapitre précédent le lien qui existe entre les systèmes multi-agents et l'architecture logicielle (cf. chapitre 4), ainsi que les formalismes qui ont été utilisés pour la description d'architectures logicielles à base d'agents. La description formelle des architectures logicielles permet d'effectuer des analyses et de vérifier le système très tôt dans le processus logiciel. Dans ce chapitre nous allons présenter notre approche pour la description des architectures logicielles basées agents en utilisant les réseaux de Petri.

5.2 Architecture logicielle des systèmes multi-agents

Avant de passer à la description d'une architecture à base d'agents et à la recherche d'un langage approprié pour le faire, il faut d'abord savoir quoi décrire ? C'est-à-dire quels sont les éléments que nous considérons des éléments architecturaux pour un système à base d'agents ? Et à part ces éléments quoi encore décrire ? Les uns peuvent considérer les architectures à base d'agents comme de nouvelles architectures, alors il faut répondre aux questions précédentes. Les autres peuvent garder les concepts des architectures à base de composants, mais cette orientation a aussi ces propres problèmes par exemple : est-ce qu'on va simplement remplacer le composant par l'agent ? Un connecteur est-il juste un type particulier d'agent ou autre chose ? … etc.

L'architecture d'un système logiciel définit ce système en termes de composants informatiques et des interactions entre ces composantes [SG96]. Dans notre travail nous allons reprendre cette définition dans le contexte des systèmes multi-agents et dire que l'architecture d'un système multi-agents définit ce système en termes d'agents et des interactions entre ces agents.

Après avoir précisé ce que nous allons décrire dans l'architecture d'un système multi-agents, il faut passer à la description de l'architecture qui se fait par l'utilisation d'un certain langage de description. La première étape à la recherche d'un langage approprié consiste à bien définir les besoins aux quels doit répondre ce dernier.

Selon [YC06] un langage formel pour la modélisation des systèmes multi-agents doit répondre aux besoins suivants :

a. Le langage devrait précisément et sans ambiguïté décrire la structure et les comportements des systèmes multi-agents d'une manière lisible et compréhensible;

b. Le langage devrait permettre la spécification des agents en utilisant une combinaison de graphiques et de texte;

c. Le langage doit cacher certains détails quand nécessaire, et décrire les systèmes multi-agents à différents niveaux afin que les développeurs peuvent les bien comprendre;

d. Le langage devrait représenter la sémantique statique et dynamique, et fournir des outils pour la modélisation, l'analyse et la vérification;

e. Le langage doit être orienté ingénierie de logiciels et facile à implémenter.

La deuxième étape concerne la sélection d'un langage qui répond à ces besoins. À cette étape nous avons trois choix :

- Utiliser un langage existant tel qu'il est ;
- Modifier ou faire adapter un langage existant ;
- Définir un nouveau langage.

Parmi ces choix, l'exploitation d'un langage existant offre les avantages les plus potentiels, qui incluent la documentation, l'utilisation, l'acceptation est l'existence d'outils. Si aucun langage ne peut satisfaire toutes les exigences, on va faire recoure au multi langages et voir si plusieurs langages peuvent être utilisés pour répondre aux exigences. Souvent, différents aspects du même problème peuvent avoir besoin des langages différents pour satisfaire toutes les exigences [Rob00].

5.3 Les réseaux de Petri

Comme cité dans la section 5.2.3 le langage choisi doit répondre aux différentes exigences. Concernant notre travail qui consiste à la description de l'architecture des systèmes multi-agents, nous avons précisé que l'architecture logicielle à base d'agents pour nous, c'est l'ensemble des agents et des interactions entre ces agents. Donc le langage choisi doit permettre la description des agents et leurs interactions.

Vu les caractéristiques des systèmes multi-agents (distribution, parallélisme, communication,…etc.) et l'adéquation des réseaux de Petri pour la modélisation des protocoles de communication et des comportements parallèles et concurrents ainsi que la possibilité d'analyser le modèle obtenu, les réseaux de Petri constituent un bon candidat pour la description des architectures des systèmes multi-agents.

Dans ce sens, de nombreux travaux ont été réalisés autour de l'utilisation des réseaux de Petri pour la modélisation des systèmes multi-agents, soit au niveau de l'agent c à d sa structure et son comportement soit au niveau organisation c.à.d. les interactions entre les différents agents constituant le système [Maz01][XY00]. Concernant l'utilisation des réseaux de Petri pour la description des architectures logicielles à base d'agents plusieurs travaux ont été aussi réalisés [XY06] [YCX06][AZ05] (cf. chapitre 4).

Dans notre architecture un agent est modélisé par un réseau de Petri . Une action d'un tel agent est modélisée par une transition. Nous distinguons deux types de places, à savoir les places internes et les places d'ïnterface. Une place d'interface est soit une place d'entrée ou une place de sortie pour l'agent. Une place d'entrée n'a pas de transitions en entrée et une place de sortie n'a pas de transitions en sortie. Toutes les autres places de l'agent sont considérées comme des places internes parmi les quelles on trouve deux places spéciales : initiale n'ayant aucune place en entrée et finale n'ayant aucune place en sortie. Les jetons dans les places d'interface représentent des messages que l'agent s'échange avec les autres agents. Dans les places internes les jetons marquent un certain état interne de l'agent. Les transitions sont aussi soit des transitions internes ou des transitions d'interface. Une transition interne n'est reliée qu'aux places internes, alors qu'une transition d'interface à comme place en sortie soit une place d'interface d'entrée, donc on l'appelle transition de réception ou une place d'interface de sortie, alors elle est appelée une transition d'envoi.

Le modèle choisi comme modèle de base pour la description de l'architecture des systèmes multi-agents est celui des réseau workflow ouverts (OWN : Open Workflow Net)[ALMSW08], qui sont une sous-classe des réseaux de Petri où chaque agent de l'architecture est modélisé par un OWN et l'architecture globale est obtenue par compositions de l'ensemble des OWNs. Nous justifions tout d'abord ce choix, puis nous présentons les notions de réseaux de Petri utiles pour notre travail.

Un système modélisé par OWNs est généralement conçu sous forme d'un ensemble de composants interagissant via les places d'interfaces. Donc partant de notre définition de l'architecture d'un système multi-agents, les OWNs répondent bien à nos besoins, car ils permettent la modélisation du comportement interne de l'agent par un réseau de Petri workflow et aussi ces interactions via les places d'interface. A l'origine les OWNs ont été proposés et utilisés pour la modélisation des architectures orientées services ainsi que les services web et leur composition [AHM09] [ALMSW08] [Mul08]. Mais en réalité les architectures des systèmes multi-agents en plusieurs aspects en commun avec les services web. Dans [Wal05], C. Walton a proposé une représentation des techniques d'agents principaux et leurs équivalents approximatifs dans le domaine des services web. D'une façon générale les caractéristiques principales en commun sont:

- Les deux architectures sont considérés comme des systèmes de communication asynchrones ;
- Faible couplage entre les composants du système ;
- Un couplage dynamique où des liens de communication peuvent être établis durant le temps d'exécution ;

Définitions :

Définition 1 : Réseau de Petri (RdP) [Mur89]

Formellement un réseau de Petri est un quintuplet, $R= (P, T, F, W, M_0)$ tel que :

- $\boldsymbol{P} = \{p_1, p_2,\ldots, p_m\}$ *ensemble fini de places ;*
- $\boldsymbol{T} = \{T_1, T_2,\ldots, T_m\}$ *ensemble fini de transitions ;*
- $\boldsymbol{F} \subseteq (P \times T) \cup (T \times P)$ *ensemble d'arcs;*

- $\boldsymbol{W} : F \rightarrow$ * *fonction de poids ;*
- $\boldsymbol{M_0} : P \rightarrow$ *marquage initial ;*
- $P \bigcap T = \varnothing$ et $P \bigcap T \neq \varnothing$

Notations

– Soit $Q= (P, T, F, W)$, un réseau de Petri,

- $R=(Q,M_0)$ est le réseau de Petri Q marqué avec M_0 le marquage initiale.
- $°t$: l'ensemble des places d'entrée de la transition t.
- $t°$: l'ensemble des places de sortie de la transition t.
- $°p$: l'ensemble des transitions d'entrée de la place p.
- $P°$: l'ensemble de transitions de sortie de la place p.

Marquage

On appelle marquage une distribution de jetons sur les places. Le marquage initial noté M_0 est la distribution initiale de jetons dans le réseau à l'instant initial. Un marquage définit l'état du système. Le marquage d'un réseau de Petri est une application $M :\rightarrow$ donnant pour chaque place le nombre de jetons qu'elle contient. Le marquage de la place pi est noté par *M (pi)* qui est un nombre entier.

Franchissement

Une transition $t \in T$ est franchissable dans un marquage M si et seulement si $\forall p \in \circ t : M(p) \geq W(p,t)$.

Son franchissement conduit au marquage M' défini par : $\forall p \in P : M'(p) = M(p) + C(p,t)$. C étant la matrice P × T définie par $C(p,t) = W(t,p) - W(p,t)$ et appelée matrice d'incidence de N.

Le franchissement est noté par $M \xrightarrow{t} M'$. S est la séquence de transitions *t1, t2,…, tn* et le passage du marquage M au M' par franchissement de la séquence de transitions s est noté par $M \xrightarrow{s} M'$

L'atteignabilité

Soit P un réseau de Petri. Un marquage M' de P est atteignable ou accessible depuis le marquage M s'il existe une séquence de transitions franchissables S menant de M à M'. L'ensemble des marquages accessibles de M est noté par $R(P,M)$

$M' \in R(P,M) \Leftrightarrow \exists S \in T^* : M \xrightarrow{S} M'$. Où T^* : l'ensemble des séquences de transition.

L'ensemble des marquages de P est noté *R(P).*

Définition 2 :Réseau de Petri ouvert (OPN :Open Petri net) [AHM09]

Un réseau de Petri ouvert (OPN) est un 7-uplet $N=(P_N, I_N, O_N, T_N, F_N, i_N, f_N)$ oú

- $N=(P_N \cup I_N \cup O_N, T_N, F_N)$est un réseau de Petri;
- P_N un ensemble de places internes ;
- I_N un ensemble de places d'entrée, et $^\circ I_N = \emptyset$;
- O_N un ensemble de places de sortie, et $O_N^\circ = \emptyset$;

- P_N, I_N, O_N, et T_N sont deux à deux disjoints ;
- $i_N \in B(P_N)$ est le marquage initial ;
- $f_N \in B(P_N)$ est le marquage finale et
- f_N est un blocage.

On appelle l'ensemble $I_N \cup O_N$ les places d'interface de *N*. Deux OPNs *N* et *M* sont disjoints si $P_N, P_M, I_N, I_M, O_N, O_M, T_N$ et T_M sont deux à deux disjoints.

Définition 3 : Squelette (Skeleton) [AHM09]

Soit $N=(P_N, I_N, O_N, T_N, F_N, i_N, f_N)$ an OPN. Le squelette de N est défini comme un réseau de Petri $S(N)=(P_N, T_N, F)$ avec $F= F_N \cap ((P_N \times T_N) \cup (T_N \times P_N))$.

Définition 4 : Réseau workflow ouvert (OWN : Open Workflow net) [AHM09]

Soit $N=(P_N, I_N, O_N, T_N, F_N, i_N, f_N)$ an OPN. *N* est réseau workflow ouvert si son squelette est un réseau workflow avec une place initiale $i \in P$ est une place finale $f \in P$, tel que $^\circ i = \emptyset, f^\circ = \emptyset, i_N = [i], f_N = [f]$.

5.4 Approche proposée

Dans notre approche on considère un système multi-agents comme un système de communication asynchrone, fondé sur l'échange de messages entre les entités de ce système qui sont les agents. Dans [OPB00] Odell et *al*. présentent AUML pour la représentation des systèmes multi-agents. Ils proposent les diagrammes d'interactions pour la représentation des interactions entre les agents du système et les diagrammes d'activité pour la représentation de la dynamique interne de l'agent. L'utilisation des ces deux diagrammes nous permet de représenter les

agents et leurs interactions séparément, en plus elles sont des notations informelles que nous ne pouvons pas vérifier.

Les diagrammes d'interaction AUML représentent les interactions entre les agents du système, ces interactions suivent généralement certains protocoles de communication.

Un protocole d'interaction est en fait une série de messages échangés entre des participants. La spécification du protocole stipule l'ordre des messages que les participants doivent respecter. Les protocoles d'interaction caractérisent différents scénarios de la communication. En se basant sur chaque situation concrète, le modélisateur emploie le protocole approprié. Pour chaque agent on distingue un état initial qui représente le début de l'interaction, cet état qui précède la réception ou l'envoie du premier message du protocole et un état final qui suit la réception ou l'envoi des messages que nous considérons terminaux c à d ceux qui représente la fin du protocole.

Les diagrammes d'activité sont utilisés pour une représentation plus clarifiée du processus d'interaction, ils expriment les opérations et les événements qui les déclenchent, ils sont particulièrement utiles pour les protocoles d'interaction avec des traitements simultanés complexes (Le pourquoi). Un diagramme d'activité est une variante des diagrammes d'états-transitions composé d'un ensemble de nœuds et un ensemble d'arcs. On distingue un nœud initial qui est un nœud de contrôle à partir du quel le flot débute et un nœud final qui est un nœud de contrôle dans lequel le flux d'activité s'arrête.

L'exploitation de ces deux diagrammes nous permet d'obtenir une description de la dynamique interne de l'agent ainsi que ces interactions sous forme d'un workflow qui peut ensuite être modélisé par OWNs. Donc partant de notre point de vue de l'architecture des systèmes multi-agents, la composition de l'ensemble des OWNs des différents agents nous permet d'obtenir une description architecturale formelle représentant les agents et leurs interactions au même temps. Dans le cadre

de notre travail nous nous basons sur les diagrammes d'interactions et les diagrammes d'activité pour l'obtention des différentes activités de l'agent et la construction des OWNs.

L'approche consiste à :

- Décrire l'ensemble des agents ;
- Décrire les interactions ;
- Construire l'architecture globale ;
- Analyse de l'architecture.

5.4.1 Description des agents

Dans l'architecture d'un système multi-agents, l'agent est considéré comme l'unité de base pour la construction de telles architectures. Dans notre travail les agents constituant le système peuvent être modélisés comme suit :

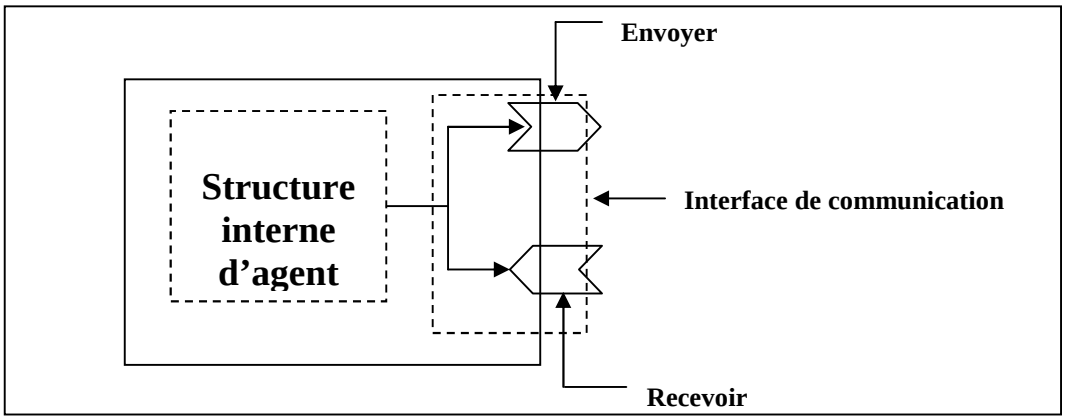


FIG. 5.1 – Modèle d'un agent

L'agent dans notre architecture est un est un réseau workflow ouvert $N=(P_N, I_N, O_N, T_N, F_N, i_N, f_N)$ composé des éléments suivants :

Les places :

- **P_N** : représente l'ensemble des places représentant avec leur marquage les différents états internes de l'agent y compris la place initial i et la place finale f

- **I_N**: représente l'ensemble des places pour les messages recrues où pour chaque message reçu l'agent est doté d'une place pou recevoir ce dernier.

- **O_N** : représente l'ensemble des places pour les messages envoyés où pour chaque message envoyé l'agent est doté d'une place pour envoyer ce dernier.

Les transitions: $T_N = \{T_{Ni}, T_{No}, T_{Np}\}$

- **T_{Ni}** : représente l'ensemble des transitions servant à recevoir des messages.

- **T_{No}** : représente l'ensemble des transitions servant à envoyer des messages.

- **T_{Np}** : représente l'ensemble des transitions les internes de l'agent représentant des activités qui n'inclut pas les actions d'envoie et de réception de messages.

Les arcs

- $F \subseteq (I_N \times T_{Ni}) \cup Fi \cup (T_{No} \times O_N)$

- $Fi \subseteq (T_{Ni} \times P_N) \cup (T_{Np} \times P_N) \cup (P_N \times T_{Ni}) \cup (P_N \times T_{No})$

Le marquage

- Un marquage initial : un jeton dans la place i, ce marquage indique qu'un agent peut commencer son exécution en franchissant une transition et aussi s'engager dans une interaction avec les autres agents

- Un marquage final : un jeton dans la place f, ce marque indique que le l'exécution de l'agent ainsi que son interaction avec les autre agents s'est terminée de façon valide.

La description des agents se fait selon les étapes suivantes :

1. La première étape consiste à la définition des différentes activités de chaque agent et qui revient à :

   - Définir les actions d'interaction (envoie et réception de messages). Ces actions et leur ordre seront définies à partir des diagrammes d'interaction AUML ainsi que les restrictions sur les interactions qui peuvent réduire le nombre des scénarios possibles;

   - Définir les activités internes de l'agent, autres que les actions d'interaction à partir de son diagramme d'activité et préciser leur ordre par rapport aux actions d'interaction.

2. Construction du réseau workflow (squelette du OWN) de chaque agent à partir des résultats de l'étape 1 comme suit :

   - La place i comme place initiale du réseau workflow avec un jeton (marquage initial), cette place sera une place en entrée pour la première transition;

   - Mettre l'ensemble des transitions $T_N = \{T_{Ni}, T_{No}, T_{Np}\}$ dans leur ordre.

   - Ajouter l'ensemble des places internes P_N entre la différentes transitions, pour chaque transition ayant des successeurs on ajoute une place de sortie qui sera une place en entrée pour ces successeurs dans le cas ou les successeurs représentent activités à choix exclusif. Si les activités sont parallèles on ajoute autant de places que de successeurs.

Les transitions n'ayant pas de successeurs auront la place finale comme place en sortie ;

- Relier les l'ensemble des nœuds (places et transitions) par les arcs F et Fi

- La place f comme place finale du réseau workflow

3. Construction des OWNs de chaque agent. Dans cette étape on définit les ensembles I_N et O_N, où pour chaque transition d'envoie de message on joute une place de sortie pour contenir le message envoyé et pour chaque transition de réception on ajoute une place d'entrée pour contenir le message reçu. Puis on relie chaque transition à la place d'interface correspondante pour obtenir l'OWN de l'agent.

Exemple :

Etap1 :

Soit deux agents A et B en interaction en utilisant le protocole FIPA-request :

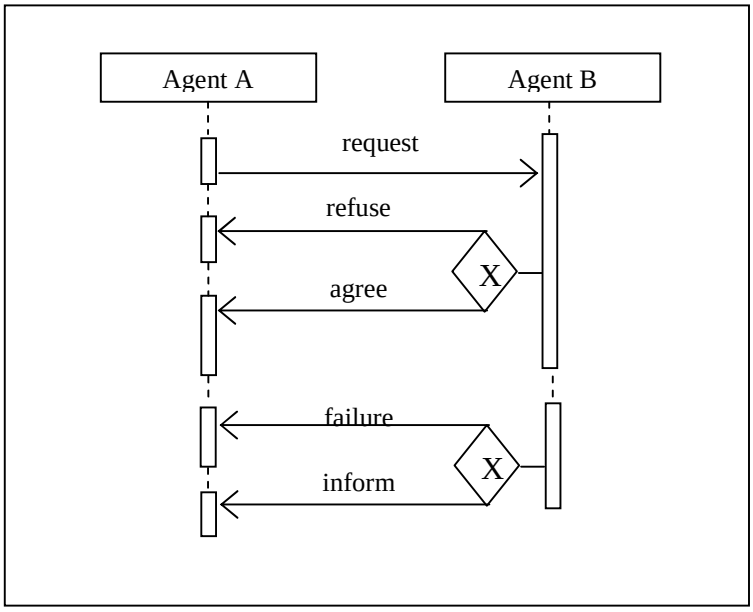


FIG. 5.2 – Exemple du protocole FIPA-request

Nous considérons les scénarios suivants :

1. L'agent A demande à l'agent B de faire quelque chose. L'agent B accepte la demande et informe l'agent A que la demande est exécutée.

2. L'agent A demande à l'agent B de faire quelque chose mais l'agent B refuse la demande.

3. L'agent A demande à l'agent B de faire quelque chose. L'agent B accepte la demande mais l'exécution de la demande a échoué.

A partir du diagramme précédent (FIG 5.2) nous avons la liste des transitions d'interaction de l'agent A et B comme suit:

Agent A

transition	description	Type transition	Transitions successeurs
1	request	envoie	2,3
2	refuse	réception	null
3	agree	réception	4,5
4	failure	réception	null
5	inform	réception	null

TAB. 5.1 –Transitions de communication Agent A

Agent B

transition	desciption	Type transition	Taransitions successeurs
1	request	réception	2,3
2	refuse	envoie	null
3	agree	envoie	4,5
4	failure	envoie	null
5	inform	envoie	null

TAB. 5.2 – Transitions de communication Agent B

- Pour l'agent A nous n'allons pas ajouter des transitions internes ;

- Pour l'agent B le comportement est simple le diagramme d'activité n'est pas utilisé et nous allons simplement ajouter une transition interne après la réception du message request qui représente une certaine activité de sélection d'action (accepter ou refuser la demande). Donc l'ensemble des transitions de l'agent B et leur ordre devient comme suit :

Agent B

transition	desciption	Type transition	Taransitions successeurs
1	request	réception	2
2	sélection d'action	interne	3,4
3	refuse	envoie	null
4	agree	envoie	5,6
5	failure	envoie	null
6	inform	envoie	null

TAB. 5.3 – Transitions de l'Agent B

Etap2 :

Pour l'agent A :

- Place initiale i ;
- T_{Ni} ={2, 3, 4, 5}, T_{No} ={1}, T_{Np} =∅ ;
- On ajoute les places interne entre les transitions, P1(1, (2,3)), P2(3, (4,5)).
- Les transitions sont : Fi ⊆ (T_{Ni} x P_N) ⋃ (T_{Np} x P_N) ⋃ (P_N x T_{Ni}) ⋃ (P_N x T_{No}) ;
- La place final *f* donc P_N={i,P1, P2,f} ;

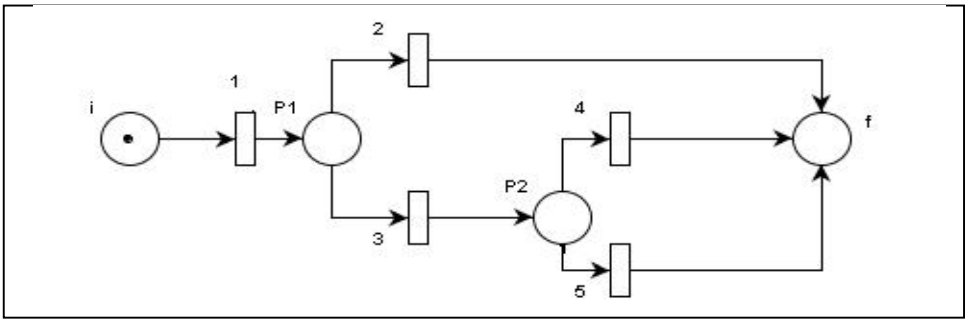


FIG. 5.3 – Réseau workflow de l'agent A

– Pour l'agent A nous avons une transition d'envoie donc on a une place d'interface de sortie, P_O = {P3}, et quatre transitions de réception, donc quatre places d'interface d'entrée, P_I = {P4, P5, P6, P7}. On va ajouter les places d'interface au réseau workflow de l'agent A en les reliant aux transitions d'interface par l'ensemble des arcs (I_N x T_{Ni}) ⋃ (T_{No} x O_N). l'OWN représentant l'agent À est le suivant :

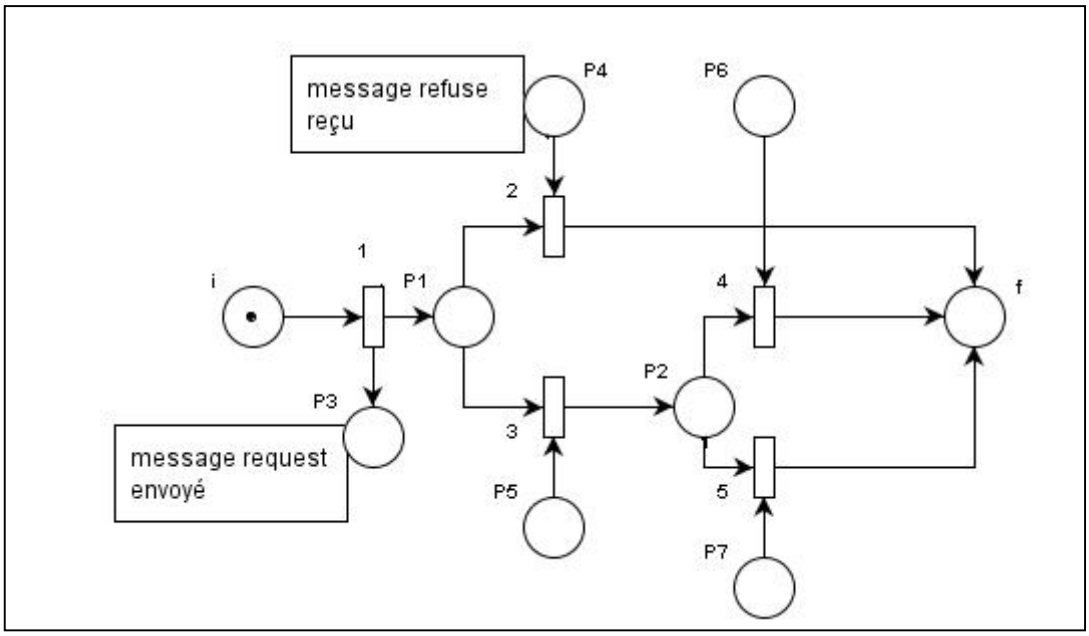


FIG. 5.4 – OWN de l'agent A

5.4.2 Description des interactions

Dans notre architecture les agents constituant le système multi-agents communiquent par envoie de message (communication asynchrone) et cette communication est la seule forme d'interaction qui existe entre les agents.

Dans un système, l'agent a une certaine fonctionnalité qu'il doit accomplir. Donc il a un ensemble de services requis qu'il demande des autres agents pour atteindre ces objectifs et par conséquent accomplir sa fonctionnalité dans le système. Un agent a aussi un ensemble de services fournis qu'il offre aux autres agents pour

qu'ils puissent à leur tour accomplir leurs fonctionnalités. Les agents dans notre architecture communiquent en utilisant le langage FIPA ACL.

En réalité la description des interactions n'est pas indépendante de la description de l'agent car au niveau de l'agent, la description des interactions revient à définir l'interface de communication de chaque agent dans l'architecture. Au niveau de l'architecture globale la description des interactions, consiste à décrire les différents connecteurs entre les différents agents, dans notre cas les agents sont relies seulement par fusion des places d'interface correspondantes. Donc nous allons :

- Vérifier que les OWNs des différents agents sont composable (définition 6).

- Composer les OWNs par l'opération de composition (définition 5)

Le OWN de l'agent B peut être obtenu en appliquant les trois étapes de construction :

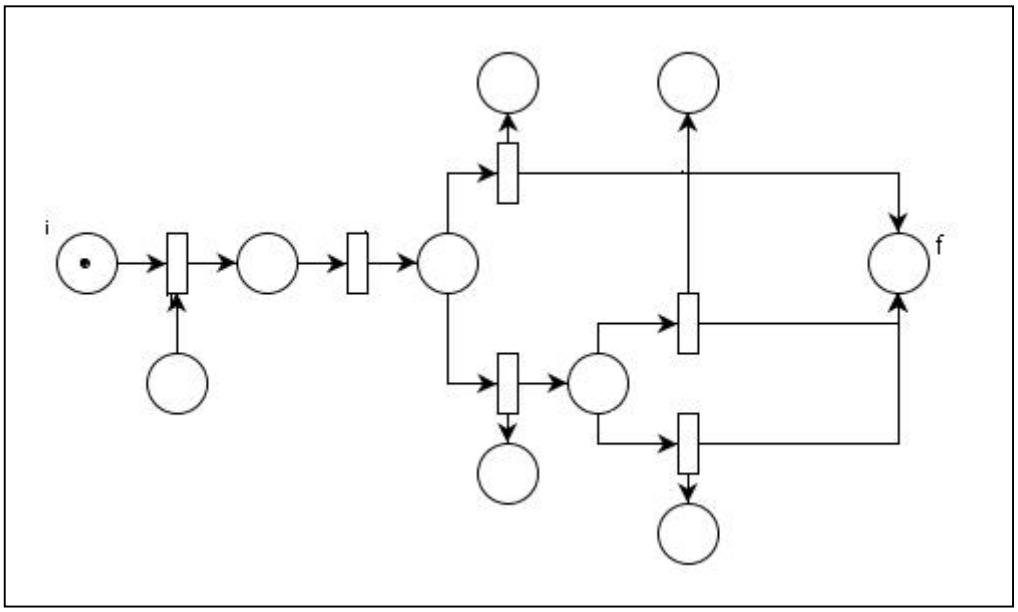


FIG. 5.5 – OWN de l'agent B

L'architecture globale est obtenu en connectant les différents agents de l'architecture via les places d'interface, cette connexion consiste à fusionner les

places d'interface correspondante des agents en interaction, c à d la place consacré à l'envoi d'un message X par l'agent émetteur qui est une place d'interface de sortie sera fusionner avec la place consacrée à la réception du message X pare l'agent récepteur qui est une place d'interface d'entrée de cet agent. La figure suivante montre la composition des OWNs de l'agent A et l'agent B

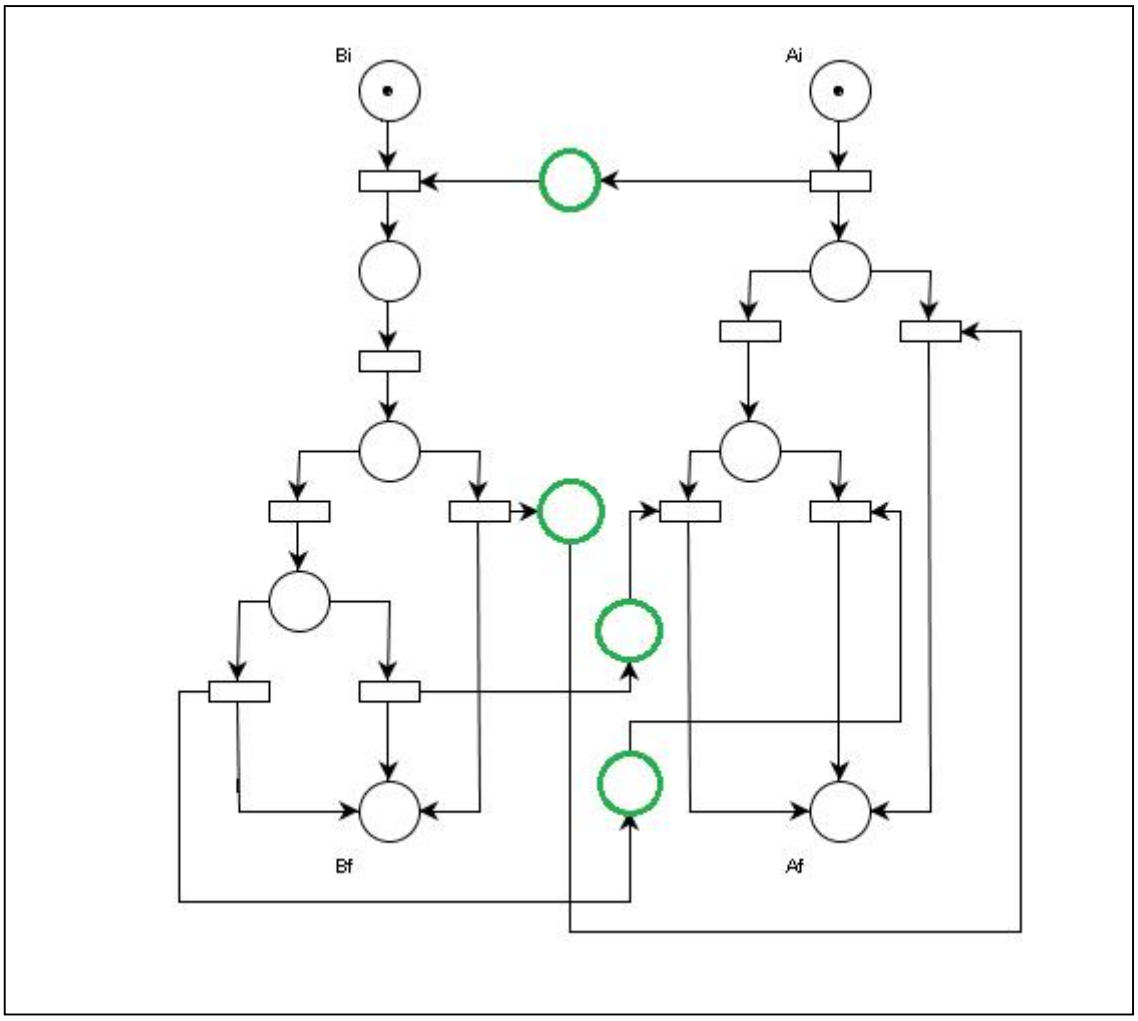


FIG. 5.6 – Composition des OWNs des agents A et B

5.5 Analyse de l'architecture

Dans notre architecture nous avons modélisé chaque agent de l'architecture par un OWN, l'architecture est obtenue par composition des OWNs des agents constituant le système par fusion des places d'interface. Dans leur travail [AHM09] Aalt *et al*.ont proposé une méthode pour la composition des services qui sont

modélisés par des réseaux de Petri ouvert (OPN : Open petri net), la composition d'un nombre de composants forme un arbre de service (Service tree) . Pour la vérification, la propriété considérée est la cohérence (soundness) qui signifie qu'à partir de tout marquage atteignable on peut atteindre le marquage final. Pour vérifie cette propriété au niveau de la composition, les auteurs donnent une condition suffisante qui ne nécessite que la vérification deux à deux des composants formant l'arbre de service (composition). Cette méthode paraît très intéressante pour nous pour la vérification de notre architecture qui est en fait une composition d'OWNs qui sont une classe plus restrictive des OPNs. Vérifier que l'architecture est cohérente revient à dire que le comportement des agents y compris leurs interactions mène toujours à un état final pour chaque agent, cet état qui représente une terminaison valide de l'exécution de chaque agent ainsi que les interactions entre les agents.

En plus que la méthode de vérification soit applicable sur notre description des architectures à base d'agents, elle accentue l'apport de la conception modulaire des systèmes multi-agents sous forme d'une architecture logicielle du fait que la vérification de la composition des agents deux à deux permet de conclure des propriétés de l'architecture globale. D'autre part la construction de l'architecture globale résulte en un réseau de Petri d'une taille importante. La vérification de la cohérence s'effectue de façon comportementale puisqu'elle fait recours au calcul de l'espace d'états. Donc, cette vérification est de complexité exponentielle qui augmente en fonction de la taille du réseau et la méthode proposée permet de maîtriser cette complexité.

Définition 5 : Composition (Composition)

Soit A et B deux OPNs. Leur composition est un OPN $A \oplus B$ défini par :

- $P_{A \oplus B} = P_A \bigcup P_B \bigcup (I_A \bigcap O_B) \bigcup (I_B \bigcap O_A)$;
- $I_{A \oplus B} = (I_A \setminus O_B) \bigcup (I_B \setminus O_A)$;

- $O_{A \oplus B} = (O_A \setminus I_B) \bigcup (O_B \setminus I_A)$;
- $T_{A \oplus B} = T_A \bigcup T_B$;
- $F_{A \oplus B} = F_A \bigcup F_B$;
- $i_{A \oplus B} = i_A \bigcup i_B$;
- $f_{A \oplus B} = f_A \bigcup f_B$.

Définition 6 : Composable (Composable)

Deux OPNs A et B sont composable si seulement si. $(P_A \bigcup I_A \bigcup O_A U T_A) \bigcap (P_B \bigcup I_B \bigcup O_B U T_B) = (I_A \bigcap O_B) \bigcup (O_A \bigcap I_B)$

Définition 7 : Arbre de service (Service tree) :

Soit $A_1, ..., A_n$ des OPNs deux à deux composables.

Soit $c : \{2, ..., n\} \to \{1, ...n-1\}$ tel que :

- $\forall i \in \{2, ..., n\} : c(i) < i$,
- $\forall 1 \le i \le j \le n : i \ne c(j) \Rightarrow I_{A_i} \bigcap O_{A_j} = \varnothing \wedge O_{A_i} \bigcap I_{A_j} = \varnothing$,
- $\forall 1 \le i \le j \le n : i = c(j) \Rightarrow I_{A_i} \bigcap O_{A_j} \ne \varnothing \vee O_{A_i} \bigcap I_{A_j} \ne \varnothing$.

On appelle $A_1 \oplus ... \oplus A_n$ arbre de service avec racine A_1 et fonction d'arbre *c*.

Définition 8 : Cohérence des réseaux de Petri ouverts (OPNs Soundness) :

Un OPN N est dit cohérent si pour tout marquage $m \in R(N)$ on a $S(N) : m \xrightarrow{*} f_N$.

Définition 9 : Condition $\Omega_{A,B}$:

Soit A et B deux OPNs composables la condition $\Omega_{A,B}$ est vraie si seulement si $\forall m \in R(A \oplus B)$:

$\pm \in (T_A)^* : (A : m_{|P_A} \xrightarrow{\pm} f_A) \Rightarrow (\exists \pounds \in (T_A \bigcup T_B)^* : (A \oplus B : m \xrightarrow{\pounds} f_A + f_B) \wedge \pounds_{|T_A} = \pm)$

Théorème1 : Cohérence des arbres de service (Soundness of service trees) :

Soit $A_1,...,A_n$ un arbre de service avec racine A_1 et fonction d'arbre c. Soit A_1 cohérent et pour $2 \le i \le n$, $\Omega_{A_i, A_{c(i)}}$ est vrai. Alors $A_1 \oplus ... \oplus A_n$ est cohérent.

5.6 Etude de cas

Le système considéré est un système de commerce électronique composé d'un agent Buyer et deux agents Seller1 et Seller2. L'interaction entre l'agent Buyer et les deux agents Seller1 et Seller2 se fait selon le protocole Contract net. On ajoute une restriction sur cette interaction et que les produit en question est toujours disponible chez les agents Seller, donc les agents toujours accepte de faire un offre.

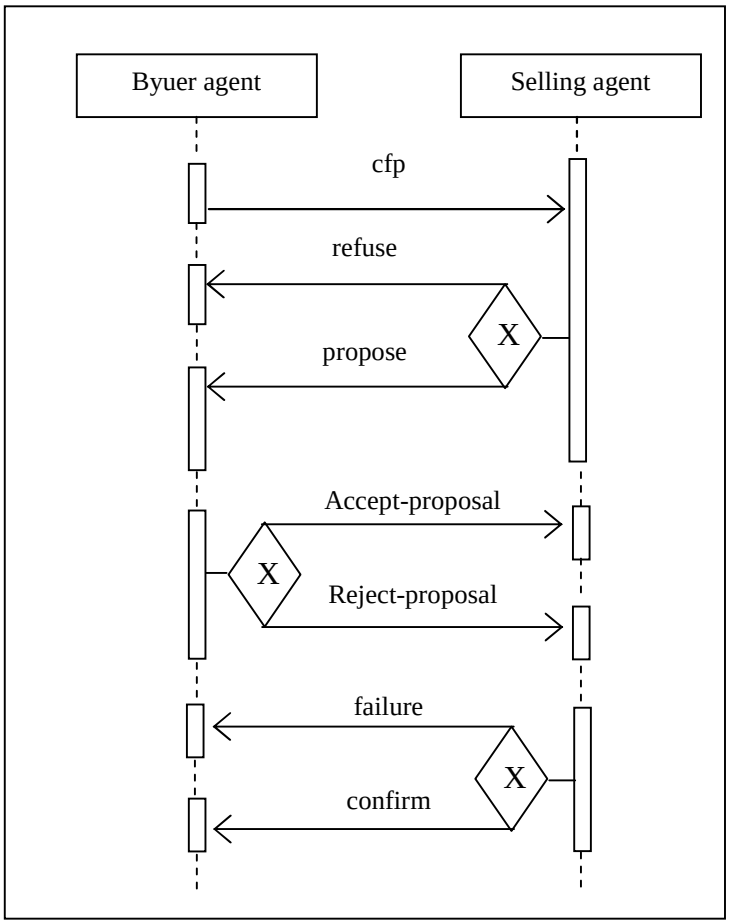


FIG. 5.7 – Protocole contract net

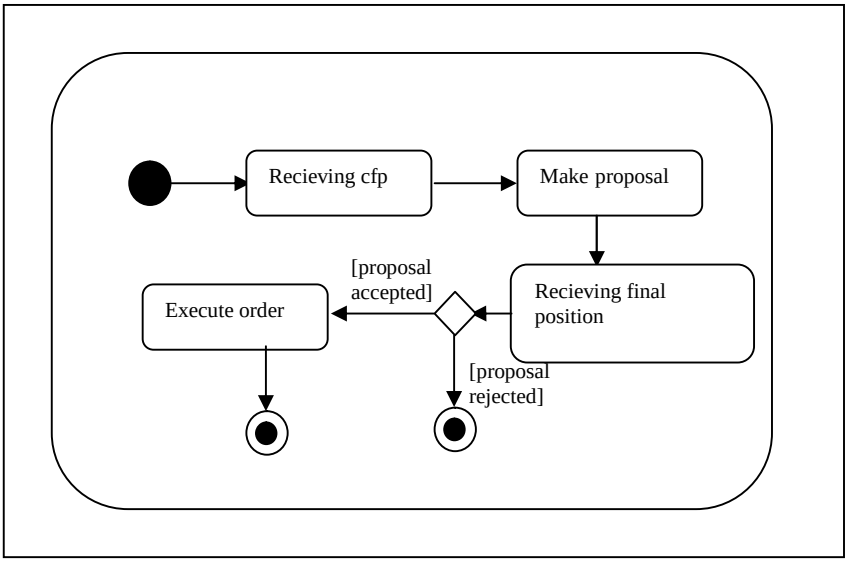


FIG. 5.8 – Diagramme d'activité de l'agent Seller

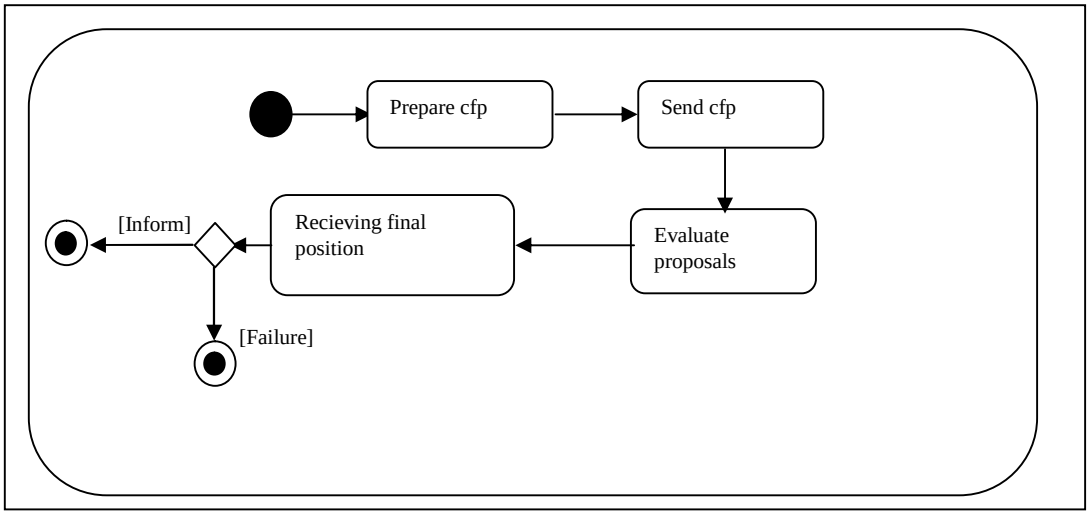


FIG. 5.9 – Diagramme d'activité de l'agent Buyer

Construction des des OWNs des trois agents :

Etape1 :

A partir du diagramme d'interaction et du fait que les agents Seller toujours acceptent de faire des propositions, nous considérons les scénarios suivants :

1 L'agent Buyer lance un appel d'offre les deux agents lui envoient des propositions, l'agent Buyer accepte la proposition de l'agent Seller1 et

refuse celle de l'agent Seller2 puis l'achat de l'article demandé s'effectue avec succès.

2 L'agent Buyer lance un appel d'offre les deux agents lui envoient des propositions, l'agent Buyer accepte la proposition de l'agent Seller1 et refuse celle de l'agent Seller2 puis l'achat de l'article demandé échoue.

3 L'agent Buyer lance un appel d'offre les deux agents lui envoient des propositions, l'agent Buyer accepte la proposition de l'agent Seller2 et refuse celle de l'agent Seller1 puis l'achat de l'article demandé s'effectue avec succès.

4 L'agent Buyer lance un appel d'offre les deux agents lui envoient des propositions, l'agent Buyer accepte la proposition de l'agent Seller2 et refuse celle de l'agent Seller1 puis l'achat de l'article demandé échoue.

Pour les agents Seller1 et Seller2 on obtient le tableau suivant :

transition	desciption	Type transition	Taransitions successeurs
1	cfp	réception	2,3
2	propose	envoie	3,4
3	refuse	envoie	null
4	accept-proposal	réception	6,7
5	reject-proposal	réception	null
6	inform	envoie	null
7	failure	envoie	null

TAB. 5.4 – Transitions de communication de l'agent Seller

– Pour les agents Seller et à partir de leur diagramme d'activité nous allons ajouter deux transitions internes : l'agent avant d'envoyer une proposition il prépare cette proposition et après avoir reçu l'acceptation de son offre il effectue l'opération de vente ;

Donc le tableau précédent devient comme suit :

transition	desciption	Type transition	Taransitions successeurs
1	cfp	réception	2
2	make proposal	interne	3
3	propose	envoie	4,5
4	accept-proposal	réception	6
5	reject-proposal	réception	null
6	execute order	interne	7,8
7	inform	envoie	null
8	failure	envoie	null

TAB. 5.5 – Transitions de l'agent Seller 1

Pour l'agent Seller2 :

transition	desciption	Type transition	Taransitions successeurs
9	cfp	réception	10
10	make proposal	interne	11
11	propose	envoie	12,13
12	accept-proposal	réception	14 ,16
13	reject-proposal	réception	null
14	execute order	interne	15 ,16
15	inform	envoie	null
16	failure	envoie	null

TAB. 5.6 – Transitions de l'agent Seller2

Pour l'agent Buyer on obtient le tableau suivant :

transition	desciption	Type transition	Taransitions successeurs
17	cfp Seller1	envoie	3
18	cfp Seller2	envoie	4
19	propose Seller1	réception	5 ,10
20	propose Seller2	réception	7,8
21	accept-proposal Seller1	envoie	6
22	inform Seller1	réception	null
23	reject-proposal Seller2	envoie	null
24	accept-proposal Seller2	envoie	9
25	inform Seller2	réception	null
26	reject-proposal Seller1	envoie	null

TAB. 5.7 – Transitions de communication l'agent Buyer

Pour l'agent Buyer et à partir de son diagramme d'activité nous avons aussi deux transitions internes : avant d'envoyer l'appel d'offre il le prépare et après la réception des propositions l'agent effectue une certaine d'évaluation pour choisir une proposition parmi les deux reçues. Donc le tableau des transitions de l'agent Buyer devient comme suit

transition	desciption	Type transition	Taransitions successeurs
17	prepare cfp	interne	18 ,19
18	cfp Seller1	envoie	20
19	cfp Seller2	envoie	21
20	propose Seller1	réception	22
21	propose Seller2	réception	22
22	evaluate proposal	interne	23,27
23	accept-proposal Seller1	envoie	24
24	reject-proposal Seller2	envoie	25,26
25	inform Seller1	réception	null
26	failure Seller1	réception	null
27	accept-proposal Seller2	envoie	28
28	reject-proposal Seller1	envoie	29,30
29	inform Seller2	réception	null
30	failure Seller2	réception	null

TAB. 5.8 – Transitions de l'agent Buyer

Etape2 :

Pour l'agent Seller1

- Place initiale i P1 ;
- T_{Ni} ={1, 4, 5}, T_{No} ={3, 7,8}, T_{Np} ={2, 6} ;
- On ajoute les places interne entre les transitions, P2(1,2), P3(2,3), P4(3,(4,5)), P5(4,6), P6(6,(7,8)).
- Les transitions sont : Fi $\subseteq (T_{Ni} \times P_N) \cup (T_{Np} \times P_N) \cup (P_N \times T_{Ni}) \cup (P_N \times T_{No})$;

- La place finale f P7, donc P_N={P1, P2,P3,P4,P5,P6,P7};

Les réseaux des agents Seller1 et Seller2 sont identiques.

Pour l'agent Buyer

- Place initiale i P1 ;

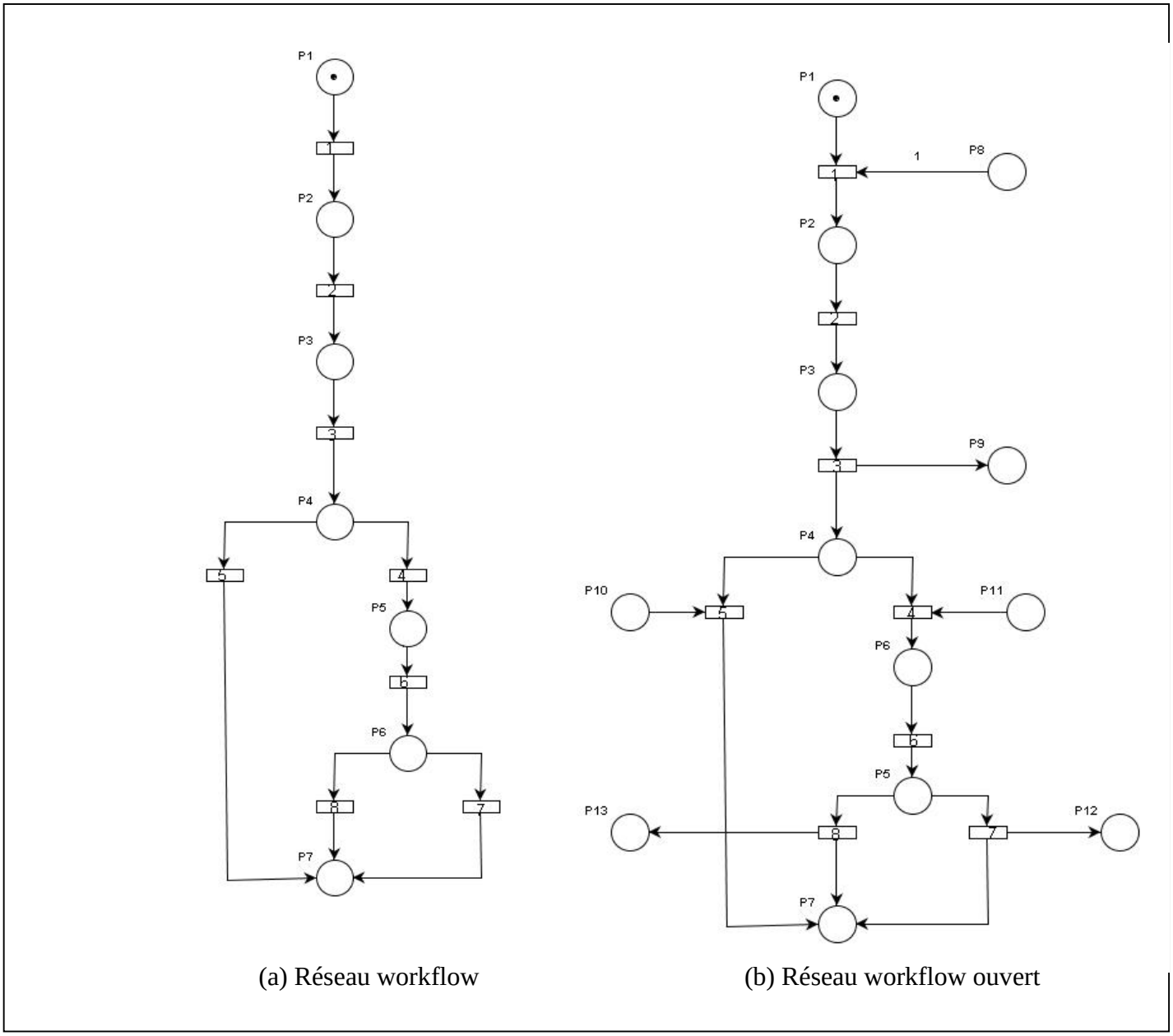
- T_{Ni} ={20, 21, 25, 26, 29, 30}, T_{No} ={18, 19, 23, 24, 27, 28}, T_{Np} ={17, 22} ;

- On ajoute les places interne entre les transitions, P2(17,18), P3(17,19), P4(19,21), P5(18,20), P6(21,22), P7(20,22) P8(22,(23,27)P9(27,28), P10(23,24), P11(28,(29,30)), P12(24,(25,26))

- Les transitions sont : Fi $\subseteq$ (T_{Ni} x P_N) $\cup$ (T_{Np} x P_N) $\cup$ (P_N x T_{Ni}) $\cup$ (P_N x T_{No}) ;

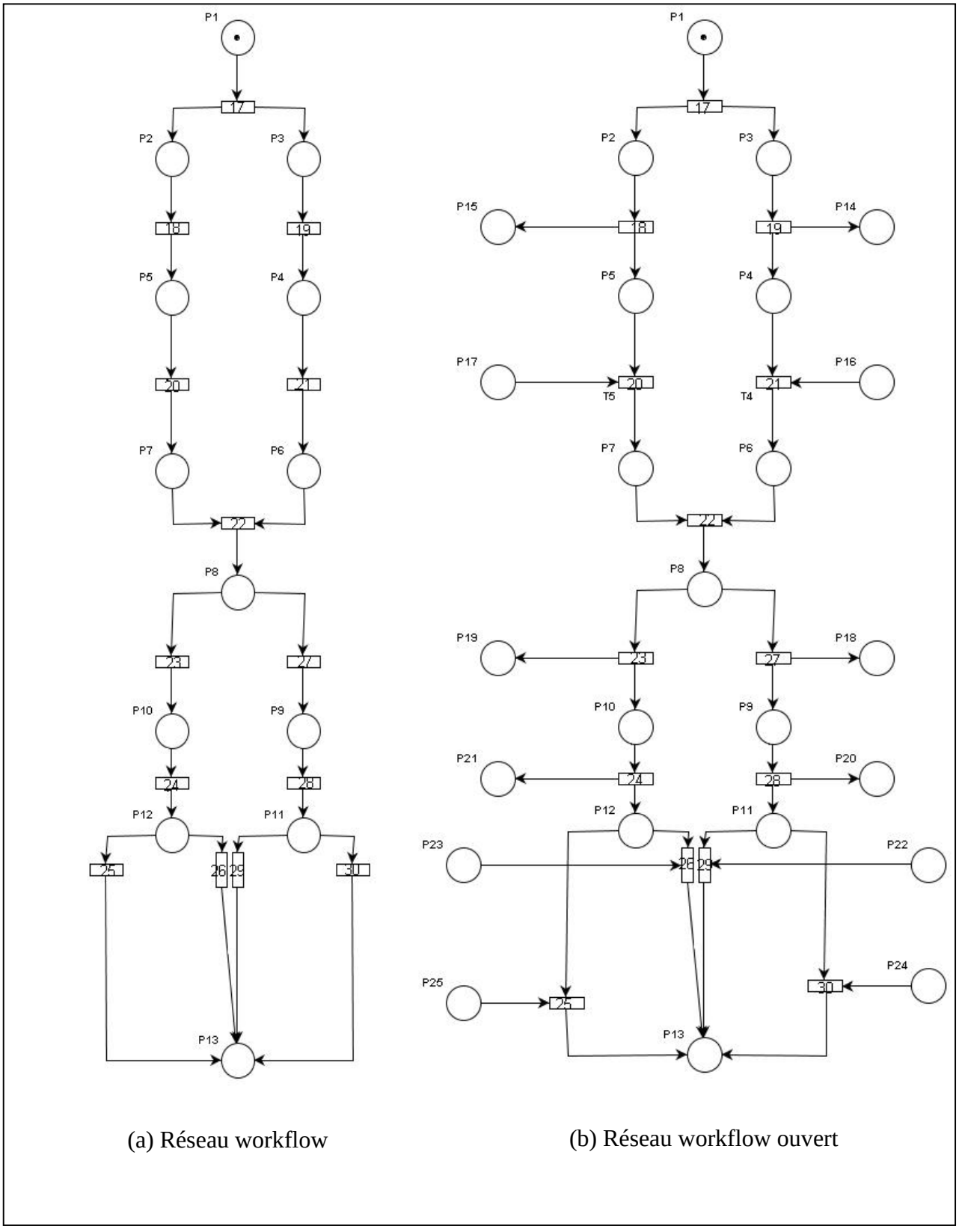
- La place finale f P13, donc P_N={P1, P2, P3, P4, P5, P6, P7, P8, P9, P10, P11, P12, P13};

- Pour l'agent Seller1 nous avons trois transitions d'envoie donc on aura trois places d'interface de sortie, P_O = {P9, P12, P13}, et trois transitions de réception, donc on aura quatre places d'interface d'entrée, P_I = {P8, P10, P11}. On va ajouter les palces d'iterface au réseau workflow de l'agent Seller en les reliant aux transitions d'interface par l'ensemble des arcs (I_N x T_{Ni}) $\cup$ (T_{No} x O_N). Pour l'agent Buyer P_O = {P14, P15, P18, P19, P20, P21}, P_I = {P16, P17, P22,P23, P24, P25}.



(a) Réseau workflow (b) Réseau workflow ouvert

FIG. 5.10 –Agent Seller1



(a) Réseau workflow

(b) Réseau workflow ouvert

FIG. 5.11 – Agent Buyer

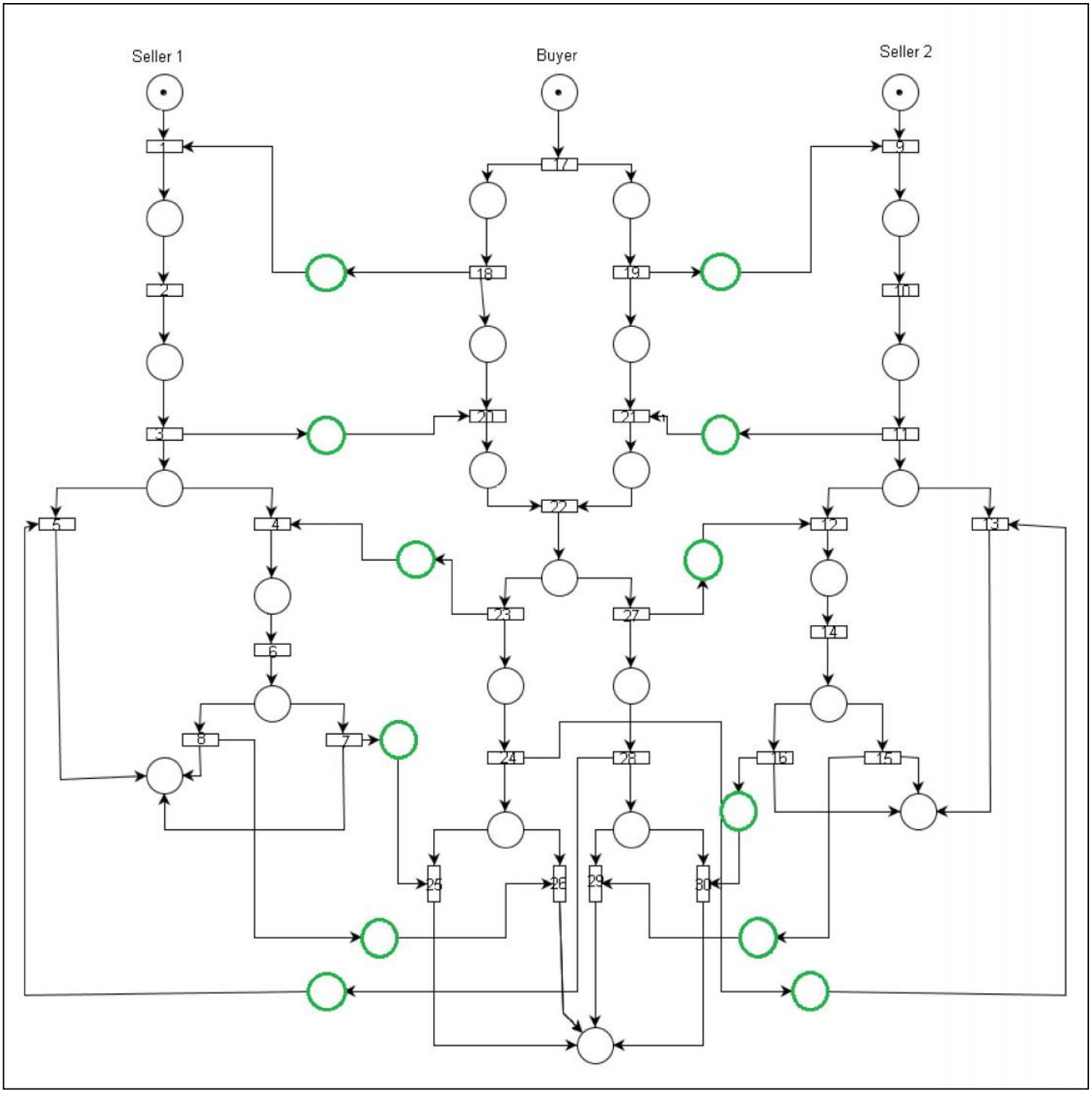
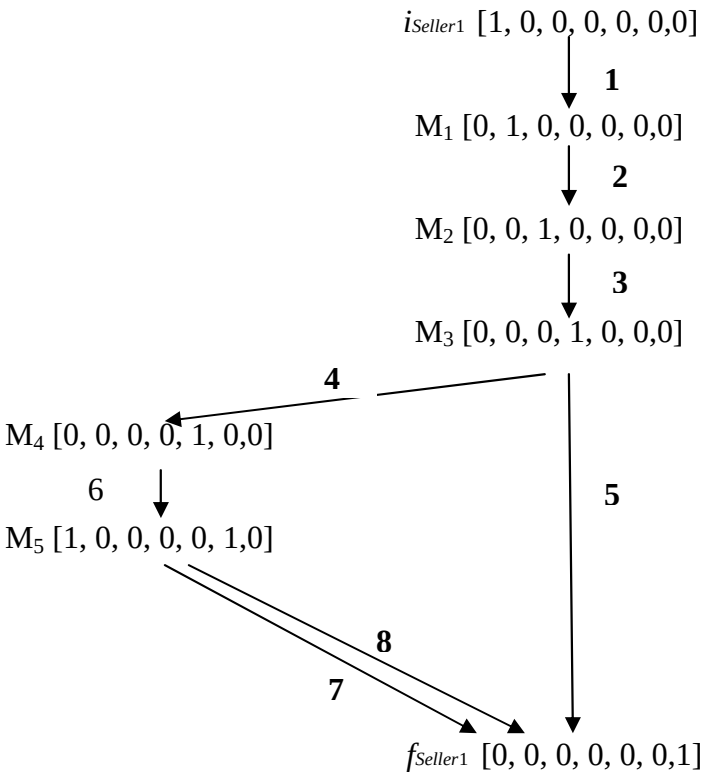


FIG. 5.12 – Architecture du système de commerce électronique

Pour la vérification de notre architecture on a :

1. Selon la définition 6, les agents Seller 1, Buyer et Seller 2 sont deux à deux composables.

2. Seller 1 est cohérent. Nous nous basons sur le graphe de marquage pour la vérification de la cohérence de l'agent Seller1. Le graphe de marquage de l'agent Seller1 est comme suit :



On remarque qu'à partir de chaque marquage il y a un chemin vers le marquage final. Donc selon la définition 8 Seller1 est cohérent.

3. On remplace seller 1 par A1, Buyer par A2 et Seller 2 par A3. A1, A2, A3 forment un arbre de service parce que $c:\{2,3\}\rightarrow\{1,2\}$ existe (Définition 7).

4. Montrer que $\Omega_{Seller1,\ Buyer}$ et $\Omega_{Buyer,\ Seller2}$ sont vérifiés. Nous composons $Seller1 \oplus Buyer$ et $Buyer \oplus Seller2$ et nous aurons l'ensemble de marquages atteignables des deux compositions $R(Seller1 \oplus Buyer)$ et $R(Buyer \oplus Seller2)$

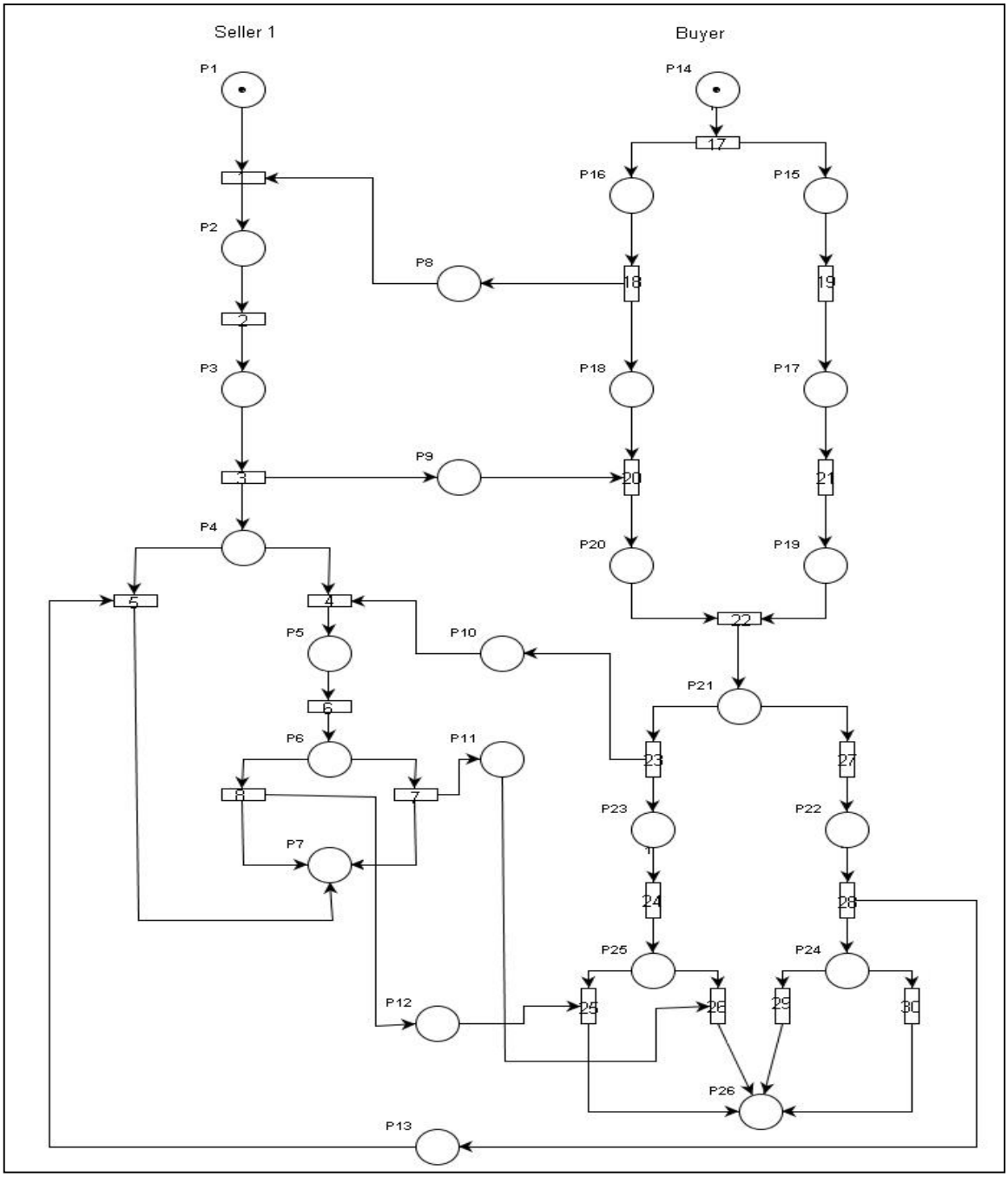


FIG. 5.13 – $Seller1 \oplus Buyer$

$R(Seller1 \oplus Buyer)$:

M_0\{	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	}
M_1\{	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	}
M_2\{	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	}
M_3\{	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	}
M_4\{	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	}
M_5\{	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	1	0	0	0	0	0	0	}
M_6\{	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	}
M_7\{	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	}
M_8\{	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	}
M_9\{	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	}
M_{10}\{	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	}
M_{11}\{	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	}
M_{12}\{	0	0	0	1	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	}
M_{13}\{	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	}
M_{14}\{	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	}
M_{15}\{	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	}
M_{16}\{	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	}
M_{17}\{	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	}
M_{18}\{	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	}
M_{19}\{	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	}
M_{20}\{	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	}
M_{21}\{	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	}
M_{22}\{	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	}
M_{23}\{	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0		}
M_{24}\{	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0		}
M_{25}\{	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	}
M_{26}\{	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0		}
M_{27}\{	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0		}
M_{28}\{	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1		}
M_{29}\{	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	}
M_{30}\{	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	}
M_{31}\{	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0		}
M_{32}\{	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1		}
M_{33}\{	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0		}
M_{34}\{	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0		}

Selon la définition 9, les marquages M_{22}, M_{25}, M_{26} et M_{31} satisfont la condition $(Seller1 : m_{\mid P_{Seller1}} \xrightarrow{\pm} f_{Seller1})$.

Pour M_{22} ± =6 7,£ = 6 7 24 26 ou ± =6 8, £ = 6 8 24 25

Pour M_{25} ± =8,£ = 8 24 25 26

Pour M_{31} ± =8,£ = 8 25 ou ± =7, £ = 7 26

Donc $\Omega_{Seller1,\ Buyer}$ est vérifiée

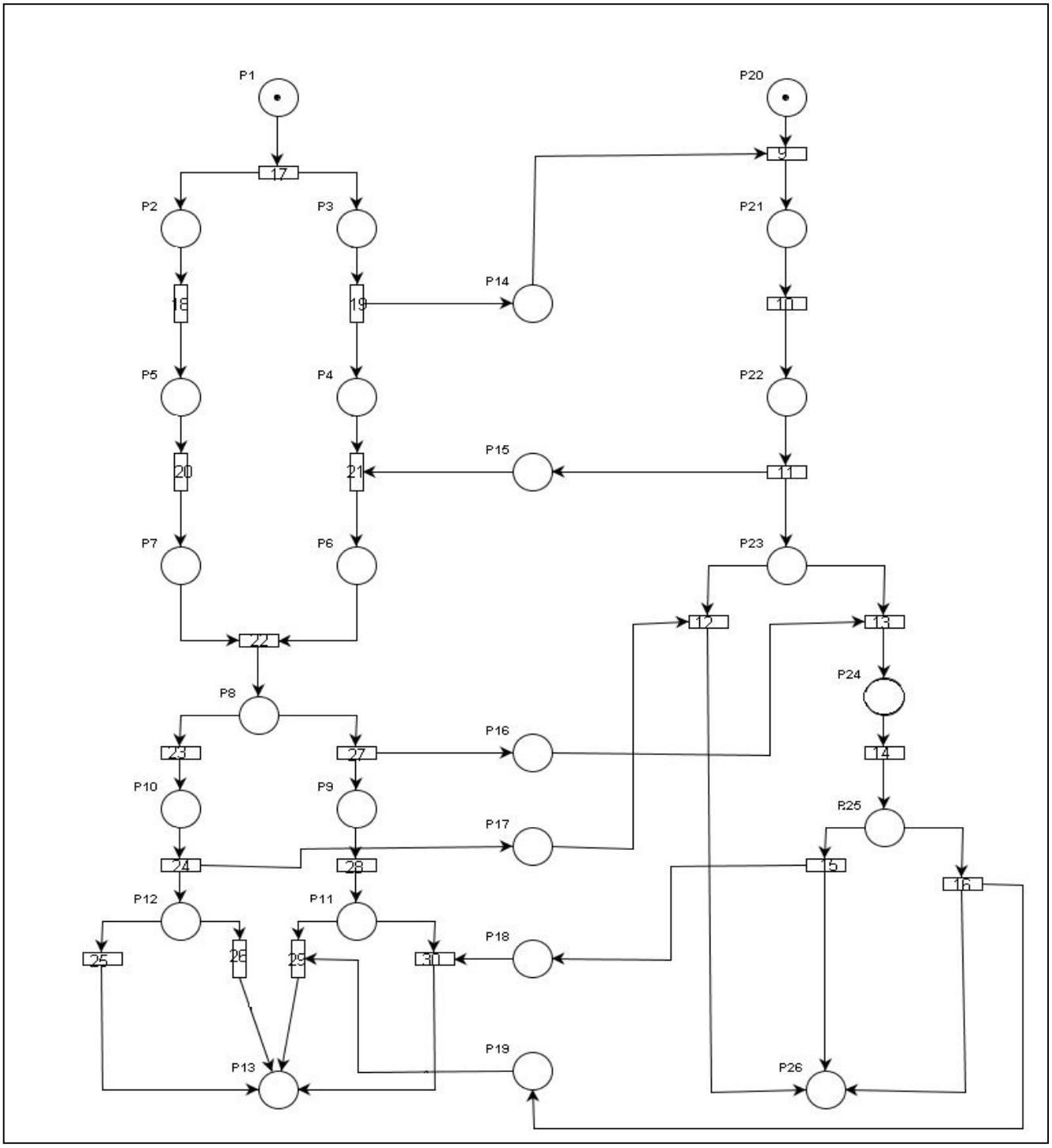


FIG. 5.14 – *Buyer* ⊕ *Seller*2

M_0\{	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0 }
M_1\{	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0 }
M_2\{	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0 }
M_3\{	0	1	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0 }
M_4\{	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0 }
M_5\{	0	0	0	1	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0 }
M_6\{	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0 }
M_7\{	0	0	0	1	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0 }
M_8\{	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0 }
M_9\{	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0 }
M_{10}\{	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0 }
M_{11}\{	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0 }
M_{12}\{	0	1	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0 }
M_{13}\{	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0 }
M_{14}\{	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0 }
M_{15}\{	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0 }
M_{16}\{	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0 }
M_{17}\{	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0 }
M_{18}\{	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0 }
M_{19}\{	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0 }
M_{20}\{	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0 }
M_{21}\{	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0 }
M_{22}\{	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	1	0	0	0 }
M_{23}\{	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0 }
M_{24}\{	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0 }
M_{25}\{	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	1	0	0	0 }
M_{26}\{	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1 }
M_{27}\{	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0 }
M_{28}\{	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	1	0	0	0 }
M_{29}\{	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	1	0	0	0	0	1	0	0	0 }
M_{30}\{	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0 }
M_{31}\{	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1 }
M_{32}\{	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0 }
M_{33}\{	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0 }
M_{34}\{	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	1	0	0 }

Pour $Buyer \oplus Seller 2$ les marquages M_{15}, M_{17} M_{18} M_{19} M_{20} M_{22} et M_{26} satisfont la condition $(Buyer : m_{\mid P_{Buyer}} \xrightarrow{\pm} f_{Buyer})$.

Pour M_{15} ± =18 20 22 23 24 25,£ = 18 20 22 23 24 12 25ou ± =18 20 22 23 24 26, £ = 18 20 22 23 24 12 26

Pour M_{17} ± =20 22 23 24 25,£ = 20 22 23 2412 25 ou ± = 20 22 23 24 26, £ = 20 22 23 24 12 26

Pour M_{18} ± = 22 23 24 25, £ = 22 23 24 12 25ou ± =22 23 24 26, £ = 22 23 24 12 26

Pour M_{19} ± =23 24 25,£ = 23 24 12 25 ou ± =23 24 26, £ = 23 24 12 26

Pour M_{20} ± =24 25,£ = 24 12 25 ou ± =24 26, £ = 24 12 26

Pour M_{22} ± =25 ,£ = 25 18 ou ± =26, £ = 26 18

Pour M_{26} ± =25,£ = 25 ou ± =26, £ = 26

Donc $\Omega_{Buyer,\ Seller2}$ est vérifiée et $Seller1 \oplus Buyer \oplus Seller2$ est cohérent.

5.7 Conclusion

Dans ce chapitre nous avons proposé notre approche basée réseaux de Petri pour la description des architectures logicielles des systèmes multi-agents. L'approche se base sur les diagrammes d'AUML pour la construction d'une description formelle de l'architecture donc en tire profit de la large utilisation d'AUML et la possibilité de vérifier une description formelle.

Nous avons exploité la ressemblance entre les systèmes multi agent et les services web en utilisant une calasse des réseaux de Petri très utilisée dans le domaine des services web et leurs composition qui est les réseaux de Petri ouverts. Pour la vérification nous avons pris en considération la propriété de cohérence. Un cas d'étude d'un système de commerce électronique a été proposé et vérifié. La méthode utilisée n'est pas toujours applicable car la composition des agents pour obtenir l'architecture globale peut ne pas former un arbre de service, donc d'autres méthodes peuvent être utilisés ainsi que d'autres propriétés doivent être vérifiées.

Chapitre 6

Conclusions et perspectives

Dans ce travail de nous avons essayé principalement d'illuminer un nouveau axe de recherche dans le domaine d'ingénierie des systèmes orientés agents qui est la conception des systèmes orientés agents du point de vue d'architecture logicielle ainsi que les langages qui semblent être appropriés pour la description de ce nouveau type d'architectures logicielles. Bass et al [BCK98] insistent sur la notion de multi-structures où un système peut avoir plusieurs structures correspondant chacune à un point de vue et aucune d'entre elles ne peut être considérée comme celle définissant l'architecture par conséquent il existe plusieurs type de composant (processus, modules). Un autre concept est le multi-vues ou une vue est liée à une structure et il est possible de réaliser de multiple vues pour une même structure, les vues permettent de montrer sous différents angles une même structure. Nous partageons ce point de vue (multi-vues) qui correspond à notre vision pour décrire les systèmes multi-agents ayant un seul type de composant qui est l'agent et au même temps ayant plusieurs caractéristiques et aspects qu'aucun formalisme ne peut les couvrir à lui seul. Donc chaque vue de l'architecture peut être décrite en utilisant un langage ou un formalisme permettant une meilleure description de ce point de vue et par conséquent une vérification est une validation des aspects liés à cette vue. Concernant notre travail qui utilise les réseaux de pétri ouverts, c'est plutôt une description et une validation de l'architecture du point de vue comportementale. Où nous avons essayé de décrire les agents et leur interaction et vérifier que le comportement des agents ainsi que leurs interactions se terminent toujours d'une façon valide. Dans des futurs recherches nous allons essayer d'une part d'améliorer notre approche, en raffinant la description proposée et en cherchant d'autre méthodes pour la vérification de l'architecture permettant de vérifier d'autre propriétés que la

cohérence et autres formes de composition que l'arbre de service. D'autre part nous allons aussi essayer d'investiguer d'autre formalisme quand à leur adéquation pour la description et la vérification des autres vues de l'architecture des systèmes multi-agents.

Bibliograhie

[ACC02] Accord. Assemblage de composants par contrats en environnement ouvert et réparti, état de l'art sur les langages de description d'architecture(ADLs).Projet ACCORD, TechnicalReportLivrable1.1-2,RNTL,France,Juin2002.

[AG97] R. Allen, D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*. vol. 6(3), pages: 213–249, 1997.

[AHM09] Wil M. P. van der Aalst, Kees M. van Hee, Peter Massuthe, Natalia Sidorova, Jan Martijn E. M. van der Werf: Compositional Service Trees. Petri Nets 2009: pages 283-302, 2009.

[ALMSW08] W.M.P. vander Aalst, N. Lohmann, P. Massuthe, C. Stahl, and K. Wolf. From Public Views to Private Views: Correctness-by-Design for Services. *In WS-FM 2007, volume 4937 of Lecture Notes in Computer Science*, pages139-153. Springer, 2008.

[Bau01] B. Bauer. UML Class Diagrams Revisited in the Context of Agent Based Systems.In M.Wooldrige, G.Weiss, and P.Ciancarini, editors, *Proceedings of the Second International Workshop on Agent Oriented Software Engineering (AOSE-2001)*, pages 101-118, Montreal, Canada, May2001.Springer.

[BCK98] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[BHTV04] Luciano Baresi, Reiko Heckel, Sebastian Thone, and Daniel Varro. Style-based refinement of dynamic software architectures. In *WICSA'04: Proceedings of the* 4^{th} *Working International IEEE/IFIP Conference on Software Architecture*, pages 155–164, Oslo, Norway, 2004. IEEE Computer Society.

[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.

[Bra87] M. Bratman. Intention, plans, and practical reason. *Harvard University Press*, 1987.

[Bro86] R. A. Brooks. A robust layered control system for mobile robot. *IEEE Journal of Robotics and Automation*. RA-2(21) : 14-23, April 1986.

[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.

[CD95] J. Cordy, and T. Dean. A syntactic theory of Software architecture. *IEEE Trans. on software*. Eng., vol.21, no.4, April 1995.

[CG99] A. Cuppari, P. L. Guida, M. Martelli, V. Mascardi, and F. Zini. An Agent Based Prototype for Freight Trains Traffic Management. *In Proc. of FMERail'99 Workshop*, Toulouse, France, September 1999.

[Cha99] B. Caib-draa. Agents et systèmes multi-agent., *Note de cours*. Université de Laval Québec, 1999.

[CT04] R. Cervenka and I. Trencansky. Agent Modeling Language: Language Specification. Technical report, Version0.9.Technical report, Whitestein Technologies, 2004.

[CT07] R. Cervenka, I. Trencansky. AML The Agent Modeling Language : A Comprehensive Approach to Modeling Multi-Agent Systems.ISBN978-3-7643-8395-4. Birkhauser, 2007.

[Dij68] E.W. Dijkstra. The structure of the T.H.E Multiprogramming System. *Communication of the ACM*, 11(5) :314-346, May 1968.

[Feb95] J. Ferber. *Les Systèmes Multi-Agents : Vers une Intelligence Collective*. InterEditions, Paris, 1995.

[FF94] T. Finin and R. Fritzon, KQML—A Language and Protocol for Knowledge and Information Exchange. *Proceedings of the 13th International Workshop on Distributed Artificial Intelligence*, Seattle,WA, pages 126-136, 1994.

[FG98] J. Ferberand O. Gutknecht. A Meta-Model for the Analysis and Design of Organizations in Multi-Agent Systems. ICMAS'98: *Proceedings of the 3rd International Conference on Multi-Agent Systems*, IEEE Computer Society,1998.

[FGL01] S. Fonseca, M. Griss and R. Letsinger. Evaluation of the ZEUS MAS Framework, 2001.

[FIP97] Foundation for Intelligent Physical Agents (FIPA). FIPA 07 Specification part 2 – Agent Communication Language. 1997, disponible sur : http://www.fipa.org

[Fow04] Martin Flower, UML 2.0. CompusPress, Pris, 1ème edtion, 5ème triage edition, 2004.

[GACB95] Cristina Gacek, Ahmed Abd-Allah, Bradford Clark, and Berry Boehem. On the Definition of Software Architecture. In D. Garlan, editor, *Proceedings of the First International Workshop on Architectures for Software Systems – I n Cooperation with the 17th International Conference on Software Engineering*, pages 85-95, Seattle, WA, April 1995.

[Gar00] D. Garland. Software Architecture : a Roadmap. In A. Finkelstein, editor, *The Future of Software Engineering, 22nd International Conference on Software Engineering (ICSE'2000)*, IEEE Computers Society Press / ACM Press, pages 93-101, Limerik, Ireland, June 2000.

[GFM00] O. Gutknecht, J. Ferber et F. Michel. MadKit : une architecture de plate-forme multi-agent générique, rapport de recharche, Unité Mixte CNRS-Université MontpellierII, may 2000.

[GKMP04] P. Giorgini, M. Kolp, J. Mylopoulos, and M. Pistore. TheTropos Methodology: An Overview. In *Methodologies and Software Engineering for Agent Systems*, Kluwer,2004.

[GM01] M.P. Gervais and F. Muscutariu, Towards an ADL for Designing Agent-Based Systems , *in Proceedings of the 2nd International Workshop on Agent-Oriented Software Engineering (AOSE'01)*, Lecture Notes in Computer Science n°2222, Springer Verlag (Ed), May 2001, Montreal

[GP95] David Garlan and Dewayne Perry. Introduction to the Special Issue on Software Architecture. *IEEE Transactions on Software Engineering*, 21(4), April 1995.

[He96] X. He. A Formal Definition of Hierarchical Predicate Transition Nets. *Proceedings of the 17th International Conference on the Application and the Theory of Petri Nets*, Japan, June 1996.

[HG05] B. Henderson-Sellers and P. Giorgini. Agent Oriented Methodologies, IG Publishing, ISBN 1591405866. 2005

[HOB04] M. P. Huget, J. Odell, and B. Bauer. The AUML Approach. *Methodologies and Software Engineering for Agent Systems* volume11, chapter12,2004.

[IEEE00] IEEE Architecture Working Group: IEEEStd1471-2000, Recommended practice for architectural description of software-intensive systems.Tech. rep., IEEE, 2000.

[IGGV97] C. Iglesias, M. Garijo, J. Gonzalez and J. Velasco, Analysis and Design of Multiagent Systems Using MAS-CommonKADS, 4th International Workshop, ATAL'97 (Providence, Rhode Island, USA), pages 313–327, 1997.

[JWS98] N. R Jennings, M. Wooldridge et K. Sycara. A roadmap of agent research and development. *Int Journal of Autonomous Agents and Multi-Agent Systems*, 1(1): 7- 38, 1998.

[KCSS02] Mohamed Mancona Kandé, Valentin Crettaz, Alfred Strohmeier, and Shane Sendall. Bridging the gap between IEEE 1471, an architecture description language, and UML. *Software and System Modeling*, 1(2) :113-129, 2002.

[KJB03] K. Kavi, D. Jung and H. Bhambhani. Extending UML for modeling and design of multi-agent systems. ICSE '03 Workshop on Software Engineering for Large Multi-agent Systems (SELMAS '03), Portland, Oregon.

[KKBPKS03] K.M. Kavi, D.C. Kung, H. Bhambhani, G. Pandcholi, M. Kanikarla and R. Shah. Extending UML to modeling and design of multi agent systems. *Proc. of 2nd Intl Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS2003),* held in conjunction with the International Conference on Software Engineering, Portland, OR, May 3-10, 2003

[KSLB01] Phipippe Kruchten, Bran Selic, Grant Larsen, and Alan Brown. Describing software architecture with UML. In *ICSE'01 : Proceedings of the* 23rd *International Conference on Software Engineering*, IEEE Computer Society, pages 715-716, Washington, DC, USA, 2001.

[LHJD04] Imen Loulou, Ahmed Hadj Kacem, Mohamed Jmaiel, and Khalil Drira. Formal design of structural and dynamic features of publish/subscribe architectural styles. In *ECSA'07: Proceedings of the 1st European Conference on Software Architecture*, Aranjuez, Spain, September 24-26, volume 4758 of Lecture Notes in Computer Science, pages 44–59. Springer, 2007.

[LPH06] M. Li, H. Peng, and J. Hu. Research on Modeling and Description of Software Architecture of Cooperation-Oriented System. *Proceeding PRIMA 2006*

Proceeding of 9th Pacific Rim Internationnal Workshopn, Springer-Verlag, 2006.

[Luc96] David C Luckham. Rapide : A language and toolset for simulation of distributed system by partial ordering of events. *IEEE Transactions on Software Engineering*, 1996.

[Maz01] B. Mazouzi. Ingenierie des protocols d'interaction : des systems distributes aux systyèmes multi-agents , Thèse de Doctorat, Université Paris IX-Dauphine.2001

[MDK93] Jeff Magee, Naranker Dulay, and Jeffrey Kramer. Structuring parallel and distributed programs. *IEEE Software Engineering Journal*, 8(2) :73-82, Mars 1993.

[MDK94] Jeff Magee, Naranker Dulay, and Jeffrey Kramer. A constructive development environment for parallel and distributed programs. In *IWCCS'94 : Proceedings of the IEEE Workshop on Configurable Distributed Systems*, North Falmouth, Massachusetts, USA, Mars 1994.

[Met98] Daniel Métayer. Describing software architecture styles using graph grammars. *IEEETransactionsonSoftwareEngineering*,24(7): 521–533,1998.

[MFKG05] H. Mouratidis, S. Faulkner, M. Kolp and P. Giorgini,. A Secure Architectural Description Language for Agent Systems. In *4th AAMAS (AAMAS'05)*, Uthrecht, The Netherlands, 2005.

[MPW92] R. MILNER, J. PARROW and D. WALKER. Acalculus of mobile processes. *Journal of Information and Computation* 100(1):1-77.992

[MT00] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparaison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1) :70-93, January 2000.

[Mul08] M. Müller. Detecting and Resolving Mismatches Between Provided Behaviour and Required Behaviour. Master thesis, Department of Mathematics and Computer Science, technical university of Eindhoven, 2008.

[Mül96] J. P. Müller. The Design of Autonomous Agents A Layered Approach. *Volume 1177 of Lecture Notes in Artificial Intelligence*. Springer-Verlag, Heidelberg, 1996.

[Mur89] T. Murata. Petri nets: Properties, analysis and applications, *Proceedings of the IEEE*, vol. 77 n°4, pp. 541-580, 1989.

[OPB00] J. Odell, H.V.D Parunak and B. Bauer. Representing agent interaction protocols in uml. In First international workshop, AOSE 2000 on Agent-oriented software engineering (Secaucus,NJ,USA,2001), Springer-Verlag New York, Inc., pages 121–140.

[Par72] David Lorge Parnas. On the Criteria for Decomposing Systems into Modules. *Communication of the ACM*, 15(12) :1053-1058, December &972.

[PBL03] A. Pokahr, L. Braubach and W. Lamersdorf. "Jadex: Implementing a BDI Infrastructure for JADE Agents". EXP – in *search of innovation*, 3(3):76–85, 2003.

[Per03] Jorge Enrique Pérez-Martinez. Heavyweight extensions to the UML metamodel to describe the C3 architectural style. *SIGSOFT Software Engineering Notes*, 28(3):5-5, 2003.

[Pic04] Gauthier picard. Méthodologie de développement de systèmes multi-agents adaptatifs et conception de logiciels à fonctionnalité émergente, thèse de doctorat, Université Paul Sabatier de Toulouse III, 2004.

[PW00] P. Lin and W. Michael. Prometheus: A Methodology for Developing Intelligent Agents. *In proceedings of the Third International Workshop on Agent-Oriented Software Engineering*, at AAMAS'02.

[PW02] L. Padgham and M. Winikoff. Prometheus: A Methodology for Developing Intelligent Agents, AOSE, pages174–185, 2002.

[PW92] D.E Eerry and A.L. Wolf. Foundation for the study of software architecture. *Software Engineering Notes, ACM SIGSOFT*, 17(4) :40-52, October 1992.

[RG04] H. Reza and E. Grant. A Formal Approach to Software Architecture of Agent-Base Systems. *In Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'04)*, 2004.

[RG91] A. S. Rao and M. P. Georgeff. Modeling rational agents within a BDI-architecture. *In R. Fikes and E. Sandewall, editors, Proceedings of Knowledge Representation and Reasoning (KR&R-91)*, pages 473-484. Morgan Kaufmann Publishers : San Mateo, CA, April 1991.

[RMRR98] Jason E. Robbins, Nenad Medvidovic, David F. Redmiles, and David S. Rosenblum. Integrating architecture description languages with astandard design method. In *ICSE'98:Proceedingsofthe 20th international conference on*

Software engineering, pages209–218, Washington, DC, USA, 1998. IEEE Computer Society.

[Rob00] D. J. Robinson. Component based approach to agent specification. PhD thesis, Air University, Air Force institute of technology, 2000.

[SEI] Software Engineering Institute. http//www.sei.cmu.edu

[SG96] M. Shaw and D. Garland. *Software architecture Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[Spi92] J. M. Spivey. The Z Notation: A Reference Manual. Prentice-Hall, second edition, 1992.

[SWHV94] G. Schreiber, B.Wielinga, R. Hoog and W. Velde. CommonKADS: AComprehensive Methodology for KBS Development, IEEE Expert 9 6(9): 28–37, 1994.

[Syc95] K. P. Sycara. Multiagent Systems. *AI Magazine, American Association for Artificial Intelligence*, pages 79-92, 1995.

[Wal05] C.Walton.Uniting agents and web services. Agent Link News, 18:26–28,2005.

[WD00] M. F. Wood and S. A. DeLoach. An Overview of the Multiagent Systems Engineering Methodology. In *Proceedings of the First International Workshop on Agent-Oriented Software Engineering*, 10th June 2000, Limerick, Ireland. P. Ciancarini, M. Wooldridge, (Eds.) Lecture Notes in Computer Science. Vol. 1957, Springer Verlag, Berlin, January 2001.

[Wey10] D. Weyns. Architecture-Based Design of Multi-Agent Systems. Springer 2010: I-XVII, 1-224

[WHH09] D. Weyns, A. Helleboogh and T. Holyoet. How to get multi-agent systems accepted in industry?. *International Journal of Agent-Oriented Software Engineering* Volume 3 Issue 4, May 2009

[WHS06] D. Weyns, T. Holyoet, and K. Schelfthout. Multiagent systems as software architecture: another perspective on software engineering with multiagent systems. *Proceeding AAMAS'06 Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, ACM Press, 2006.

[WJK00] M. Wooldridge, N. R. Jennings and D. Kinny. The Gaia methodology for agent oriented analysis and design. *Autonomous Agents and Multi-Agent Systems* 3(3): 285–312, 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.

[XS97] H. Xu, Y. M. Shatz. A framework for model-based design of agent oriented software. *IEEE Transactions On Software engineering*, VOL.29 No.1, pages 15-30, 2003.

[XY00] D. Xu and Y. Deng. Modeling Mobile Agent Systems with High Level Petri Nets, *Proc. of the IEEE International Conference on Systems, Man, and Cybernetics (SMC'00),* pages 3177-3182, Nashville, October 2000.

[YC06] Z. Yu, Y. Cai. Object-Oriented Petri nets Based Architecture Description Language for Multi-agent Systems. *IJCSNS International Journal of Computer Science and Network Security*, VOL.6 No.1B, January 2006.

[YCJP00] H. Yim, K. Cho, K. Jongwoo and S. Park. Architecture-Centric Object Oriented Design Method for Multi-Agent Systems, In *Proc.* of *the Fourth International Conference on Multi-Agent Systems (ICMAS-2000)*,2000.

[YCWH06]Z. Yu, Y. Cai, R. Wang, J. Han, "π-net ADL: An architecture description language for multiagent systems", In Proc. *Of Internationnal Conference on Intelligent Computing*, Lecture Notes in Computer Science, vol. 3645, Springer-Verlag, 2005.

[YCX] Z. Yu, Y. Cai and H. Xu. A Novel Architecture Description Language For Multi-agent Systems Baseon π-net.

[Yu95] E. Yu. Modelling Strategic Relationships for Process Reengineering, PhD thesis, University of Toronto, Department of Computer Science, 1995.