REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

MINISTERE DE L'ENSEIGNEMENT SUPERIEUR

ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITE LARBI BEN M'HIDI – OUM EL BOUAGHI

FACULTE DES SCIENCES EXACTES ET DES SCIENCES DE LA NATURE

ET DE LA VIE

DEPARTEMENT DE MATHEMATIQUES ET INFORMATIQUE

N° d'ordre : .....

Série : .....

THÈSE

En vue de l'obtention du diplôme

de DOCTORAT DE 3ème CYCLE EN INFORMATIQUE

Option : Ingénierie des Systèmes Distribués

Formalisation de la méthodologie PASSI : Une approche dirigée par les modèles

Préparée par : Mr. MAZOUZ Mihoub

Devant le jury composé de :

Président	DERDOURI Lakhdar	Maitre de conférences A	Université d'Oum El Bouaghi
Rapporteur	MOKHATI Farid	Professeur	Université d'Oum El Bouaghi
Co-rapporteur	BADRI Mourad	Professeur	Université de Québec à trois Rivières, Canada
Examinateurs	HAMRI Salah	Maitre de conférences A	Université d'Oum El Bouaghi
	LAFIFI Yacine	Professeur	Université de Guelma

2017/2018

Je dédie cette thèse à :

Mes parents qui m'ont accompagné et soutenu durant ces années

Ma femme qui m'a vraiment encouragé pour terminer ce travail

Mes frères Mohamed & Yaakoub et mes sœurs

Mon neveu Uwais

Remerciements

Il m'est agréable d'adresser mes sincères remerciements à tous ceux qui m'ont apporté de près ou de loin, aide et conseils lors de l'élaboration de cette thèse de doctorat. Je voudrais remercier en particulier :

Monsieur le Professeur **FARID Mokhati** (Professeur et directeur du laboratoire RELA(CS)^2 chez l'université de L'arbi ben M'hidi à Oum El Bouaghi) pour son enthousiasme et la motivation qu'il m'a donné, sa disponibilité et leurs conseils judicieux apportés tout au long de cette thèse ;

Monsieur le Professeur **MOURAD Badri** (Professeur chercheur chez l'université de Québec à Trois rivière à Canada), pour son apport scientifique et son enthousiasme pour cette recherche ainsi que pour les remarques constructives émises malgré la langue distance qui nous sépare ;

Monsieur le Docteur **LAKHDAR Derdouri** (Maitre de conférences -classe A-, chez l'université de L'arbi ben M'hidi à Oum El Bouaghi) pour avoir accepté la lourde tâche, d'être le président du jury.

Monsieur le Docteur **SALAH Hamri** (Maitre de conférences -classe A- chez l'université de L'arbi ben M'hidi à Oum El Bouaghi) pour avoir accepté d'être examinateur de ma thèse.

Monsieur le Professeur **YACINE Lafifi** (Professeur chercheur chez l'université 08 Mai 1945 à Guelma) pour avoir accepté d'être examinateur de cette thèse.

Un spécial remerciement à ma sœur docteure **KARIMA Mazouz**, pour son encouragement qu'elle m'a donné.

Un spécial remerciement à mon ami, **Mr. HAMZA Benacherine** (Enseignant chercheur chez l'université d'Alger 2) pour son aide qu'il m'a donné.

Un spécial remerciement aussi à mes collègues **YOUGHOURTA**, **SAMIR**, **MOHAMED**, **SOFIANE**, **LAMINE et HICHAM**.

Je remercie également tous les membres du laboratoire RELA(CS)^2.

ملخص

لقد أثبتت تكنولوجيا العميل قدرتها وفعاليتها في تطوير الأنظمة الموزعة والمعقدة. خلال العشريتين الأخيرتين، تم اقتراح العديد من منهجيات تطوير الأنظمة متعددة-العملاء مثل: "Gaia"، "Tropos" و "PASSI". بالرغم من كون هذه المنهجيات وغيرها قد ساهمت بطريقة لافتة في مواجهة عدة تحديات في مجال تطوير الأنظمة متعددة-العملاء، إلا أن معظمها تعتمد على أدوات وصف نصف-رياضية ولا تستخدم تقنيات رياضية، الشيء الذي من شأنه جعل الوصفيات البيانية المنتجة خلال دورة التطوير عرضة لاحتوائها على التناقضات والغموض. استعمال الطرق الرياضية من شأنه سد هذه الثغرات والسماح بوصف واضح ودقيق دون أي غموض للنظام قيد التطوير. في هذه الرسالة نقترح منهجية "F-PASSI" (Formal-PASSI) كإمتداد للمنهجية "PASSI". تهدف "F-PASSI" إلى اعتماد وصفيات رياضية والاستفادة منها بُغْيَةَ تطبيق بعض التقنيات الرياضية عليها. عملية التصميم الخاصة بـ "F-PASSI" تتكون من نفس عملية تصميم المنهجية "PASSI" مع إدراج نموذج رياضي جديد. يتكون هذا الأخير من أربع مراحل ويسعى إلى توفير وصف رياضي مبني على منطق إعادة-الكتابة ولغته "Maude" عن طريق تنفيذ تحويلين متتاليين. التحويل الأول هو من نوع نموذج-إلى-نموذج و يُنْتِج نموذج "Maude" إنطلاقا من نمودج PASSI. التحويل الثاني هو من نوع نموذج-إلى-نص و يُنْتِج وصف نصي مبني على "Maude" إنطلاقا من نموذج "Maude" المُتَحَصَّل عليه من التحويل الأول. يتم استغلال الوصف الرياضي المتحصل عليه للتأكد من صحة الرسوم البيانية السلوكية المصممة خلال المراحل السابقة، وللتأكد من مدى تحقق الخصائص المتعلقة بالعملاء بصفة فردية وبالنظام متعدد العملاء بصفة مجملة وهذا قبل المرور إلى مرحلة الترميز. "F-PASSI" مُدَعّمة بأداة من تطويرنا،"F-PTK". الأداة المطورة تسهل لمستخدميها القيام بشتى مراحلها بطريقة سَلِسَةٍ سِيَمَا أنها تعتمد على طريقة تَتَبُّع من اقتراحنا أيضا. سيتم توضيح المنهجية المقترحة وكذا الأداة المطورة من خلال دراسة حالة ملموسة.

الكلمات المفتاحية: *هندسة البرمجيات الموجهة بالعملاء، المنهجية PASSI، التطوير الرياضي للأنظمة متعددة-العملاء، الوصف الرياضي، التأكيد الرياضي، التحقق الرياضي، منطق إعادة-الكتابة، اللغة Maude، التَّتَبُّع، الهندسة الموجهة بالنماذج، التحويلات "نموذج إلى نموذج" ، التحويلات "نموذج إلى نص" .*

RÉSUMÉ

Le paradigme agent a prouvé sa capacité et son efficacité dans le développement des systèmes distribués et complexes. Durant les deux dernières décennies, plusieurs méthodologies de développement des systèmes multi-agents (MAS) ont été proposées telles que Gaia, Tropos et PASSI. Bien que ces méthodologies et autres aient contribué de manière significative à relever plusieurs défis dans le domaine de développement des SMA, la plupart d'entre elles sont basées sur des notations semi-formelles et n'utilisent pas des techniques formelles, ce qui met les spécifications produites lors des phases du cycle de développement susceptibles de contenir des inconsistances, d'incohérences ou des ambiguïtés. L'utilisation de méthodes formelles fait face à ces lacunes et peut permettre une description précise et non ambigüe du système sous-développement. Dans cette thèse, nous proposons la méthodologie F-PASSI (Formal-PASSI), une extension de la méthodologie PASSI. F-PASSI vise à adopter de spécifications formelles et les exploiter pour en appliquer quelques techniques formelles. Le processus de conception de F-PASSI se compose de celui de PASSI en lui intégrant un modèle formel. Ce dernier se compose de quatre phases et vise à offrir une description formelle basée sur la logique de réécriture et son langage Maude par l'exécution de deux transformations successives. La première transformation est du type Modèle-à-Modèle (M2M) et en résulte un modèle Maude à partir d'un modèle PASSI. La deuxième transformation est du type Modèle-à-texte (M2T) et en résulte une description basée-Maude à partir du modèle Maude généré. La description formelle produite est exploitée ensuite pour valider les diagrammes comportementaux conçus dans les modèles qui précèdent le module formel, et pour vérifier des propriétés au niveau d'abstraction Multi/Single agent avant le passage au modèle de codage. F-PASSI est supportée par un outil que nous avons développé (F-PTK). L'outil développé facilite aux développeurs leurs tâches surtout le fait qu'il soit basé sur une technique de traçabilité que nous avons aussi proposée. La méthodologie proposée et l'outil développé sont illustrés à travers une étude de cas.

Mots clés : *Génie logiciel orienté-agent (GNOA), la méthodologie PASSI, développement formel des SMA, spécification formelle, validation formelle, vérification formelle, la logique de réécriture, Maude, la traçabilité, l'ingénierie dirigée par les modèles (IDM), transformations Modèle-à-Modèle, transformations Modèle-à-Texte.*

ABSTRACT

The agent paradigm has proved its ability and efficiency in modelling complex distributed applications. During the last two decades, several (Multi-agent systems) MAS development methodologies have been proposed like, for instance, Gaia, Tropos and PASSI. Although these methodologies and others have made significant contributions to meet several challenges in the MAS development field, most of them do not use formal techniques. This makes the designed diagrams prone of containing incoherencies, inconsistencies and ambiguities. Using formal methods could face these drawbacks and enables the description of the system under development in a precise and unambiguous way. In this thesis, we propose the F-PASSI (Formal-PASSI) methodology. F-PASSI is an extension of the well-known PASSI methodology. The extension consists mainly of the integration of a new formal model to the design process of PASSI. The new model consists of four phases and aims at offering a formal description based on rewriting logic and its Maude language of the MAS under development by the execution of two successive transformations. The first transformation is of type Model-to-Model (M2M) and results a Maude model from a PASSI model. The second one is of type Model-to-Text (M2T) and results a Maude-based description from the generated Maude model. The generated formal description is then used to validate some PASSI behavioural diagrams and check properties of both single & multi-agent abstraction levels before passing to the code model. F-PASSI is supported by a tool we developed, F-PTK. The developed tool facilitates the tasks for developers, especially the fact that is based on a traceability approach we proposed too. The proposed approach and the developed tool are illustrated throughout a case study.

Key words: *Agent Oriented Software Engineering (AOSE), PASSI methodology, formal development of MAS, formal specification, formal validation, formal verification, rewriting logic, Maude, traceability, model-driven engineering (MDE), Model2Model transformations, Model2Text transformations.*

Table des matières

Glossaire des abréviations

GLOA	Genie Logiciel Oriente-Agent
SMA	Systèmes Multi-Agents
SMAA	Système Multi-Agents Adaptatifs
FIPA	Foundation for Intelligent Physical Agents
UML	Unified Modeling Language
AUML	Agent-Unified Modeling Language
AML	Agent Modeling Language
AMOLA	Agent MOdeling LAnguage
FIPA-OS	Foundation for Intelligent Physical Agents-Operating System
JADE	Java Agent Development Framwork
PDT	Prometheus Design Tool
TAOM	Tool for Agent Oriented Modeling
TAOM4E	Tool for Agent Oriented Modeling for Eclipse
IDK	INGENIAS Development Kit
PTK	PASSI ToolKit
IDM	Ingénierie Dirigée par les Modèles
MF	Méthode Formelles
M2M	Transformation Modèle vers Modèle
M2T	Transformation Modèle vers Texte
T2M	Transformation Texte vers Modèle
T2T	Transformation Texte vers Texte
ADM	Architecture dirigée par les Modèles
PASSI	Process for Agent Societies Specification and Implementation
DD	Description du Domaine
IA	Identification des Agents
IR	Identification des Rôles
ST	Spécification des Tâches
DOD	Description Ontologique du Domaine
RDF	Ressource Description Framework
DOC	Description Ontologique de Communications
DR	Description des Rôles
DP	Description des Protocoles
DSSMA	Définition de la Structure du Système Multi-Agent
DCSMA	Description du Comportement du Système Multi-Agent
DSA	Définition de la Structure de l'Agent
DCA	Description du Comportement de l'Agent

RC	Réutilisation du Code
PC	Production du Code
PTK	PASSI Tool Kit
CD	Configuration de Déploiement
F-PASSI	Formal-PASSI
F-PTK	Formal- PASSI Tool Kit
PDF	Production de Description Formelle
VF	Validation Formelle
SFPS	Spécification Formelle de Propriétés du Système
VFPS	Vérification Formelle de Propriétés du Système
ADELFE	Atelier de DEveloppement de Logiciels à Fonctionnalité Emergente
FORMAAD	Formal Method for Agent-based Application Design
ASEME	Agent Systems Engineering MEthodology
ATM	Automated Teller Machine
PIN	Personal Identification Number
SASD	Single-Agent Structure Definition
MASD	Multi-Agent Structure Definition
MABD	Multi-Agent Behaviour Description
SABD	Single-Agent Behaviour Description

Table des figures

Liste des tables

CHAPITRE 01

CHAPITRE 03

CHAPITRE 04

CHAPITRE 05

CHAPITRE 06

Introduction générale

1. Contexte et motivations

Les systèmes informatiques doivent, de plus en plus, être capables de résoudre des problèmes distribués qui existent dans des environnements dynamiques. Les systèmes logiciels qui sont utilisés dans la résolution des problèmes distribués doivent pouvoir répondre dynamiquement au changement du problème et de l'environnement.

Les systèmes multi-agents sont des systèmes composés d'entités logicielles distribuées qui coopèrent ou concurrencent pour atteindre des buts individuels ou communs **[1]**. Les agents encapsulent leurs comportements et ils sont motivés par leurs buts internes. Les agents peuvent individuellement répondre, proactivement et réactivement, aux changements de leur environnement. L'approche agent est une approche à créer des systèmes logiciels qui sont capables de résoudre les problèmes distribués.

Les systèmes multi-agents sont, à leur tour, considérés comme systèmes complexes **[2]**. Chaque agent agit de façon autonome selon sa motivation. Le système est capable de se changer au cours du temps et chaque composant n'a pas une conscience complète du reste du système. Pour maîtriser la complexité des systèmes multi-agents, il était nécessaire de penser au développement de nouvelles méthodologies d'ingénierie tenant comptes de leurs spécificités.

L'ingénierie de systèmes logiciels peut impliquer plusieurs phases de développement **[3]**. L'analyse des besoins crée des modèles du domaine et des résultats du problème dans une spécification des tâches que le système doit effectuer. La phase de conception prend les résultats de l'analyse et modélise un système qui remplira les besoins. La phase d'implémentation est quand le système est écrit conformément à sa conception. Le logiciel peut alors être testé face aux besoins du système pour assurer qu'il s'effectue comme prévu.

L'ingénierie de logiciels est une tâche difficile avec beaucoup de problèmes qui peuvent surgir dû aux facteurs tels que la complexité du système **[4]**. Une spécification ambiguë des

besoins peut mener à une implémentation incorrecte du système **[5]**. Les conceptions de logiciels peuvent être inadéquates et contenir des erreurs conceptuelles qui mèneront à la défaillance du système **[6]**.

Le test de logiciels n'est pas toujours proportionné pour découvrir des défauts dans la conception de système **[6]**. La complexité du système affectera la quantité des tests exigés. Plus les interactions du système sont plus imprévisibles plus le nombre des tests exigé pour fournir une couverture adéquate s'augmente. Compter seulement sur le test pourrait mener à l'exécution des interactions qui n'ont pas été exécutées par le test. Ceci fait l'utilisation des méthodes rigoureuses de conception plus importante dans la conception des systèmes complexes, y compris des systèmes multi-agents.

Les méthodes formelles sont l'application des mathématiques pour modéliser et vérifier des systèmes logiciels ou matériels **[7]**. Elles sont des techniques basées sur les mathématiques ayant trois buts essentiels **[8]** : (1) Supporter la création des spécifications décrivant les vraies exigences de l'utilisateur. (2) Assurer que les implémentations satisfassent la spécification. (3) Augmenter la confiance dans le sens que le système développé n'est pas seulement correct mais connu pour être correct. Toutefois, L'utilisation des méthodes formelles ont des avantages et d'inconvénients.

L'utilisation des méthodes formelles dans le génie logiciel peut mener à une spécification non-ambiguë d'un système qui peut être formellement vérifiée pour s'assurer que le système soit consistant. L'utilisation d'une méthode formelle pour concevoir des systèmes a permis de réduire le nombre de défauts de conception introduits dans un système logiciel et de réduire le nombre de tests requis par le système **[9]**. D'où, la conception devrait être analysée avant qu'elle ne soit transformée en code de programmation **[6]**.

Depuis l'apparition de la technologie agent, plusieurs méthodologies ont été proposées dans la littérature (telles que, Gaia **[10, 11, 12]**, INGENIAS **[13]**, Tropos **[14, 15]**, PASSI **[16, 17]**) pour faciliter le développement des systèmes multi-agents. Ces méthodologies utilisent différents formalismes et langages, et basées souvent sur des plateformes de développement existantes.

PASSI (Acronyme de *Process for Agent Societies Specification and Implementation*) est une méthodologie pas à pas *besoin-vers-code* (*Requirement-to-code*) pour le développement des systèmes multi-agents. PASSI intègre des concepts du génie logiciel orienté objet et des approches de l'intelligence artificielle en utilisant la notation bien standardisée UML. PASSI se compose d'un processus de conception incrémental et itératif couvrant la plupart des phases du cycle de vie de développement et un langage de

modélisation étendu d'UML. La réutilisation est un concept clé dans PASSI, elle est effectuée à travers les patrons de conception et supportée par un outil PTK (PASSI ToolKit) **[18]**.

Malgré qu'elle a prouvé son efficacité dans le développement des systèmes multi-agents dans le domaine de la robotique et de systèmes d'informations de façon générale, nous pouvons signaler trois lacunes principales pour PASSI: (1) L'absence de liens de traçabilité explicites entre les différent éléments conceptuels produits durant les différentes phases de développement, ce qui peut compliquer les tâches aux développeurs en cas de modification d'un ou plusieurs éléments ; (2) La notation sur laquelle PASSI est basée (UML), est une notation semi-formelle, ce qui met les diagrammes conçus susceptibles de contenir des incohérences ou des ambiguïtés ; (3) L'outil PTK qui supporte PASSI n'adopte aucune technique de traçabilité, ce qui peut engendrer des erreurs potentielles par les développeurs.

La logique de réécriture a été introduite par *Jose Meseguer* **[19, 20]** pour décrire les systèmes concurrents. Elle permet de penser de manière correcte sur les systèmes concurrents ayant des états et évoluant en termes de transitions. En effet, la logique de réécriture unifie plusieurs modèles formels qui expriment la concurrence comme, par exemple, les systèmes de transitions libellés **[21]**, les réseaux de Petri **[22]** et CCS **[23]**. Les énoncés de base de cette logique sont appelés : règles de réécriture et ont la forme : $t \rightarrow t'$, où t et t' sont des termes algébriques décrivant un état partiel du système concurrent. Une règle de réécriture, dans ce cas, décrit un changement d'un état partiel vers un autre si une certaine condition C soit vraie. Plusieurs langages supportant cette logique existent dans la littérature. Les principales implémentations sont : CafeOBJ **[24]**, ELAN **[25]** et Maude **[26, 27]**.

Dans le cadre de cette thèse, nous nous intéressons au Maude. Développé à SRI International et l'université de Illinois à Urbana-Champaign, au Etats-Unis avec quelques collaborations en Espagne. Maude est un langage déclaratif de haute performance. Il est considéré comme étant le système de réécriture le plus rapide.

2. Objectifs

Dans l'objectif de rendre le développement des SMA, en utilisant la méthodologie PASSI, plus rigoureux nous nous somme donnés comme objectif la formalisation de cette méthodologie à l'aide de Maude. Pour cela, nous avons tracé les trois sous-objectifs suivants :

- ✓ ***Proposer une approche de traçabilité explicite bidirectionnelle pour PASSI*** :

  La traçabilité joue un rôle important dans le développement des systèmes informatiques, particulièrement ceux complexes. PASSI a besoin d'une traçabilité explicite afin de faciliter la compréhension du SMA en cours de développement et pour mieux gérer les changements qui se produisent durant les différentes phases de développement.

- ✓ ***Étendre la méthodologie PASSI pour supporter un développement formel des SMA*** :

  La logique de réécriture permet de penser de manière correcte sur les systèmes concurrents ayant des états et évoluant en termes de transitions. L'intégration de telle logique dans le processus de développement de PASSI peut permettre une spécification plus exacte et plus claire du SMA sous-développement, une validation et vérification formelle de sa conception avant de passer à la phase de codage. Cela permet d'éviter et d'empêcher, le plutôt possible, les éventuelles erreurs commises. Par conséquence, ceci mène à une production des SMA plus corrects et plus conformes aux besoins des utilisateurs.

- ✓ ***Rendre la nouvelle extension de PASSI accessible :***

  Formaliser PASSI pourrait la rendre inaccessible aux développeurs qui ne sont pas bien fameliarisés avec les techniques formelles. Afin de surmonter cette éventuelle difficulté, il a été décidé de développer les outils nécessaires supportant telle méthodologie qui permettraient d'aider à l'automatisation de ses phases de développement et de masquer leurs complexités aux développeurs (réduire son interaction directe avec les spécifications formelle).

3. Contributions

Afin d'accomplir les objectifs décrits dans la section précédente, cette thèse apporte quelques contributions. Dans un premier temps, nous avons proposé un méta-modèle de traçabilité bidirectionnelle explicite pour la méthodologie PASSI **[28].** Le méta-modèle proposé définit des liens de traçabilité explicites entre les différents éléments conceptuels qui sont produits durant les différentes phases de développement.

Dans un second temps, nous avons proposé F-PASSI (Formal-PASSI) **[29]**, une extension de la méthodologie PASSI. Cette extension consiste en l'intégration dans PASSI d'un nouveau modèle formel basé sur la logique de réécriture que nous avons développé. Le

modèle intégré est implémenté à l'aide de Maude et vise à offrir une description formelle du SMA sous-développement afin de l'exploiter dans le but d'en appliquer certaines techniques formelles telle que la validation par simulation et la vérification de modèles.

L'approche proposée est supportée par un outil appelé F-PTK que nous avons développé (qui reste sous-amélioration jusqu'à maintenant) comme extension de l'outil PTK de PASSI. F-PTK permet de générer automatiquement des spécifications formelles Maude. Cet outil utilise les techniques de l'ingénierie dirigée par les modèles (MDE) ; la transformation de modèle de type modèle-à-modèle (M2M) puis de modèle-à-texte (M2T). En outre, l'outil supporte les différentes phases de développement et se base sur le méta-modèle de traçabilité proposé.

4. Organisation de la thèse

Le reste de la thèse est organisé comme suit :

Chapitre 01 / Le génie logiciel orienté-agent : Dans ce chapitre, un bref aperçu sur le domaine du génie logiciel orienté agent est fait en termes de ses principaux axes tels que les standards liés aux SMA, les formalismes ainsi que les différentes plateformes utilisées dans le développement des SMA.

Chapitre 02 / Méthodologies de développement des SMA : L'accent est mis, dans ce chapitre, sur la description d'une sélection de méthodologies de développement des SMA qu'elles ont attiré beaucoup d'attention de la communauté du génie logiciel orienté agent, en les classant en deux catégories : Les méthodologies basées sur les modèles et celles formelles.

Chapitre 03 / La méthodologie PASSI : Ce chapitre est consacré à la description détaillée de la méthodologie PASSI. A la fin du chapitre, une comparaison entre les méthodologies discutées est faite en se basant sur un certain nombre de critères.

Chapitre 04 / La logique de réécriture & Maude : Dans ce chapitre, nous présentons la logique de réécriture et ses règles d'inférence. Un ensemble de langages implémentant la logique de réécriture sont mentionnés en se concentrant en détails sur le langage Maude et ses extensions.

Chapitre 05 / Formal-PASSI : Une Formalisation du processus PASSI : Ce chapitre est consacré à la description de contributions mentionnées au-dessus. D'abord, nous commençons ce chapitre par présenter le méta-modèle de traçabilité proposé. Ensuite, l'extension formelle de la méthodologie PASSI, F-PASSI, est décrite via l'explication des

différentes phases du modèle formel (le nouveau modèle intégré). Puis, l'outil F-PTK, qui support F-PASSI, est décrit avec quelques détails techniques.

Chapitre 06 / Étude de cas : Le distributeur automatique (ATM) : Ce chapitre décrit l'étude de cas, distributeur au guichet automatique (ATM). L'étude de cas est utilisée pour illustrer F-PASSI.

Chapitre 01 / Le génie logiciel orienté-agent

---

1. Introduction

Au cours des dernières années, les agents sont devenus une technologie puissante pour le développement des applications distribuées complexes. Le génie logiciel orienté-agent vise l'application des principes du génie logiciel et de l'intelligence artificielle à l'analyse, la conception et l'implémentation de systèmes informatiques **[30]**. Dans la littérature, plusieurs plateformes, outils, formalismes et méthodologies de développement des systèmes

multi-agents ont été proposés depuis l'apparence du paradigme agent. L'objectif principal étant de faciliter le développement des applications basées agent.

Dans ce chapitre, nous allons donner un aperçu sur le domaine du génie logiciel orienté-agent. La deuxième section donne quelques définitions pour le terme : *génie logiciel orienté-agent*. Quelques standards liés aux SMA sont cités dans la section 3. Les formalismes utilisés pour la modélisation des SMA sont présentés dans la quatrième section. Ensuite, une sélection de plateformes de développement des SMA est représentée dans la section 5 tout en mettant l'accent sur la plateforme JADE. La section 6 est consacrée à la citation d'un ensemble outils utilisés dans le développement des SMA. Dans la section 7, nous discutons les méthodologies de développement des SMA, ce qui va paver le chemin pour le chapitre suivant. La conclusion de ce chapitre est donnée dans la huitième section.

2. Définition du génie logiciel orienté-agent

Comme le concept Agent n'a aucune définition standard, aucune définition standard n'existe, à notre connaissance, pour le terme génie logiciel orienté-agent (GLOA). Arnon Sturm et al **[30]** ont mentionné que le génie logiciel orienté-agent implique l'application des principes du génie logiciel et de l'intelligence artificielle pour analyser, concevoir et implémenter des systèmes informatiques.

Mike Wooldridge **[31]** est parmi les fondateurs du GLOA. Nous avons essayé de le contacter pour obtenir sa propre définition du GLOA, et il nous a donné, littéralement en anglais, la définition suivante : « *It means a process of software engineering in which autonomous agents feature as first-class elements in the analysis, design, and implementation of computer systems, in the same way that objects feature as first-class entities in object-oriented systems*"[1]. Nous pouvons traduire cette définition en français par : « Il signifie le processus du génie logiciel dans lequel, les agents autonomes sont considérés comme éléments de première classe dans l'analyse, la conception et l'implémentation des systèmes informatique de la même manière que les objets sont considérés comme entités de première classe dans les systèmes orienté objet ».

Le domaine du GLOA a évolué au cours des deux dernières décennies pour aborder de nombreux sujets tels que les techniques de modélisation, les frameworks d'implémentation et les méthodologies de développement des SMA **[30]** (Figure 1.1). Le génie logiciel orienté-agent est un nouvel paradigme du génie logiciel qui a surgi pour appliquer les meilleurs

[1] Cette définition est récupérée du Pr. Michael Wooldridge par un contact direct via E-mail.

pratiques dans le développement des SMA en tenant en compte les spécificités des entités logicielles utilisées, les agents.

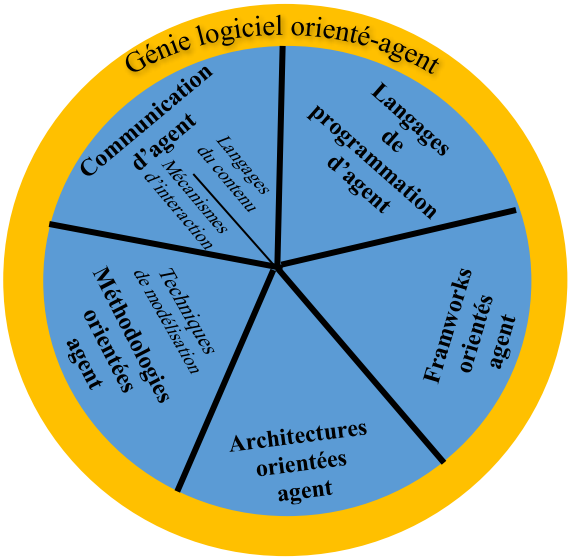

Figure 1.1 : La carte thématique du génie logiciel orienté-agent **[30]**.

3. La FIPA

3.1 Définition

FIPA[1] (acronyme de **F**oundation for **I**ntelligent **P**hysical **A**gents) est une organisation des standards de la société informatique IEEE qui promouvait la technologie basée-agent et l'interopérabilité de ses standards avec d'autres technologies. La FIPA, l'organisme de standards pour les agents et les systèmes multi-agents a été officiellement accepté par L'IEEE comme son onzième comité de standards en 2005.

3.2 Les Spécifications de la FIPA

Les spécifications de la FIPA représentent une collection de standards visant à promouvoir l'interopérabilité des agents hétérogènes et les services qu'ils peuvent représenter. Chaque spécification passe par plusieurs états (Figure 1.2) jusqu'à sa standardisation ou bien son abandonnement.

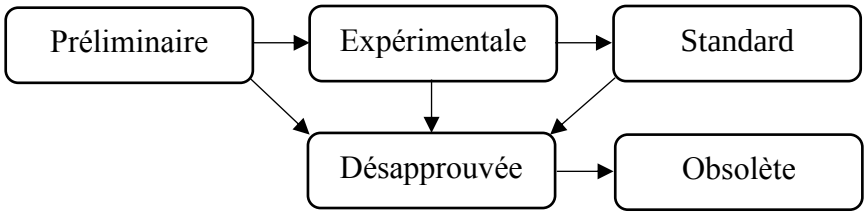

Figure 1.2 : cycle de vie de spécifications de la FIPA **[32]**.

[1] http://www.fipa.org/

La table 1.1 explique les différents stages du cycle de vie des spécifications.

Stage	Description	Nombre actuel de spécifications
Préliminaire	Une spécification dans ce stage est considérée comme une version draft sous construction.	**03**
Expérimentale	Une spécification dans ce stage est considérée comme une version stable pour une période de deux ans, où jusqu'à ce qu'elle soit promue à l'état « standard ».	**15**
Standard	Une spécification dans ce stage est considérée comme un standard stable qui est officiellement publié et approuvée par la FIPA. Pour atteindre l'état standard, une spécification doit être implémentée avec succès dans une ou plus plateforme d'agent conforme au FIPA.	**26**
Désapprouvé	Une spécification peut être déconseillée de chacun des états du cycle de vie (préliminaire, expérimentale, standard) en préparation d'un état final d'obsolète après l'écoulement d'une période de grâce. Cette période est nécessaire pour assurer que la spécification est définitivement inutile au regard du standard de la FIPA et devrai effectivement être obsolète.	**29**
Obsolète	Une spécification peut être rendue obsolète à partit l'état « Désapprouvé » après l'expiration de sa période de grâce.	**25**

Table 1.1: Les différents stages du cycle de vie d'une spécification FIPA **[32]**.

Dans les sous sections suivantes, nous allons mettre en évidence une sélection de spécifications standards clés.

3.2.1 Spécification de l'architecture abstraite (SC00001)

Cette spécification fournit un point de référence commun et immuable pour les implémentations conformes à la FIPA qui capture les caractéristiques les plus critiques et saillantes des SMA. L'architecture définit dans un niveau abstrait comment les agents peuvent localiser et communiquer par l'échange de messages qui sont codés dans un langage de communication d'agents (la sous-section suivante). Les items clefs spécifiés par ce standard sont **[33]** :

- *Les messages d'agents* : Les messages sont la forme fondamentale de communication entre agents. La structure d'un message est un ensemble de valeurs clés écrites en FIPA ACL (la sous-section suivante). Le contenu d'un message est exprimé dans un langage de contenu tels que, FIPA-SL (sous-section 3.2.3). Les messages peuvent récursivement contenir autres messages dans leurs contenus et doivent contenir les paramètres clés tels que les noms de l'expéditeur et du récepteur. Avant la transmission, les messages doivent être encodés dans une charge utile (payload) et une enveloppe de transport de messages pour le protocole particulier utilisé.

- *Uu Service de transport de messages* : Il est défini comme moyen d'envoyer et de recevoir des messages de transport entre les agents. Ceci est considéré comme un élément mandataire dans les SMA de la FIPA.
- *Un service d'annuaire d'agents : Il* est défini comme un dépôt d'informations partagé dans lequel les agents peuvent publier leurs entrées d'annuaire d'agents et dans lequel ils peuvent chercher d'entrées d'annuaire d'agents de leurs intérêts. Ceci est considéré comme un élément mandataire dans les SMA de la FIPA.
- *Un service d'annuaire de services :* Il est défini comme un dépôt partagé dans lequel les agents et les services peuvent découvrir des services. Un annuaire de services peut également être utilisé pour stocker les descriptions des services. Ceci est considéré comme un élément mandataire dans les SMA de la FIPA.

3.2.2 Spécification de la structure d'un message ACL (SC00061)

Un message FIPA-ACL contient un ensemble d'un ou plusieurs paramètres d'un message **[34]**. Précisément, quelques paramètres sont nécessaires pour une communication effective varieront selon la situation. Le seul paramètre qui est mandataire dans tous les messages ACL est le performatif, bien qu'il soit prévu que la plupart des messages ACL contiendront également des paramètres de l'expéditeur, du destinataire et du contenu.

La table suivante présente les paramètres d'un message FIPA-ACL.

Paramètre	Description
Performative	Le type de l'acte de communication du message.
Sender	L'identité de l'expéditeur du message.
Receiver	L'identité des destinataires prévus du message.
Reply-to	L'agent vers lequel les messages subséquents sont dirigés.
Content	Le contenu du message.
Language	Le langage dans lequel le contenu est exprimé.
Encoding	Le codage du contenu du message.
Ontology	Une référence vers une ontologie pour donner un sens aux symboles dans le contenu du message.
Protocol	Le protocole d'interaction utilisé pour structurer une conversation.
Conversation-id	Une identité unique d'un fil de conversation.
Reply-with	Une expression à utiliser par un agent répondant pour identifier le message.
In-reply-to	Une référence à une action antérieure à laquelle le message est une réponse.
Reply-by	Le temps/la date indiquant quand une réponse doit être reçue.

Table 1.2: Les paramètre d'un message FIPA-ACL **[34]**.

La figure 1.3 montre un exemple d'un message FIPA-ACL.

```
(request
:sender (agent-identifier :name salim@salimdomain.com)
:receiver (agent-identifier :name mihoub@mihoubdomain.com)
:ontology travel-assistant
:language FIPA-SL
:protocol fipa-request
:content
""((action
(agent-identifier :name mihoub@mihoubdomain.com)
(book-hotel :arrival 15/10/2006
:departure 05/07/2002 ... )
))""
)
```

Figure 1.2 : Un exemple simple d'un message FIPA-ACL.

3.2.3 Spécification du langage du contenu SL (SC00008)

Le langage sémantique de la FIPA (FIPA-SL) **[35]** est utilisé pour définir la sémantique intentionnelle pour les actes de communications de la FIPA comme une logique des attitudes mentales et des actions formalisées dans un langage du premier ordre. Il est défini en termes d'une grammaire d'expression de chaine de caractères définie comme étant une sous-grammaire de la syntaxe la plus générale s-expression[1].

3.2.4 Spécification de la bibliothèque des actes de communication (SC00037)

Le langage ACL de la FIPA définit une communication en termes d'une fonction ou une action appelée l'acte de communication (AC), effectué par l'acte de communiquer. Ce standard fournit une bibliothèque de tous les ACs spécifiés par la FIPA. Selon le document officiel de la FIPA **[36]**, la bibliothèque est constituée de vingt-deux actes de communications. La table suivante représente une collection des ACs de la FIPA.

Acte de communication	Description
Accept Proposal	L'action d'accepter une proposition soumise précédemment pour effectuer une action.
Inform	L'agent expéditeur informe l'agent récepteur qu'une proposition donnée est vraie.
Request	L'agent expéditeur demande au récepteur d'effectuer une action.

[1] Une expression symbolique (s-expression) est une notation d'une structure de listes imbriquée. Son origine se trouve dans la famille des langages de programmation Lisp dont la syntaxe entière consiste en ces expressions. https://igor.io/2012/12/06/sexpr.html

Confirm	L'agent expéditeur informe l'agent récepteur qu'une proposition donnée est vraie, où le récepteur est connu d'être incertain au sujet de la proposition.
Cancel	L'action d'un agent informant un autre agent que le premier agent a l'intention que le deuxième agent effectue une action.
Propose	L'action de soumettre une proposition pour effectuer une certaine action en donnant certaines préconditions.
Not Understood	L'agent expéditeur de l'acte (par exemple, i) informe l'agent récepteur (par exemple, j) qu'il a perçu que « j » a effectué une certaine action, mais que je n'ai pas compris ce que « j » a juste fait. Un cas commun particulier est que « i » indiques à « j » que je n'ai pas compris le message que « j » a juste envoyé à « i ».

Table 1.3 : Quelques actes de communication de la FIPA **[36]**.

3.2.5 Spécification des protocoles d'interaction

Un protocole d'interaction (PI) est considéré comme un patron typique d'échange de messages. La FIPA a défini plusieurs standards représentant des PIs tels que, « Request » **[37]**, « Query » **[38]**, « Contract Net » **[39]** et « Propose » **[40]**. La figure 1.4 montre le digramme de séquence représentant le protocole d'interaction « Propose » de la FIPA.

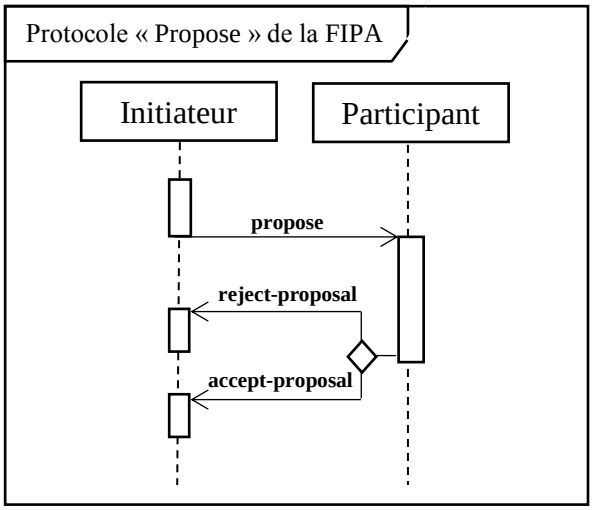

Figure 1.4 : Le protocole d'interaction « Propose » de la FIPA **[40].**

Le protocole d'interaction représenté par la figure ci-dessus permet à un agent de proposer aux agents récepteurs que l'initiateur fera les actions décrites dans l'acte communicatif propose lorsque l'agent récepteur accepte la proposition.

4. Formalismes de modélisation des systèmes multi-agents

Parmi les missions fondamentales du génie logiciel orienté-agent est la proposition d'un ensemble de notations et de formalisme qui sont destinés à aider à comprendre un tel système et qui sont cabales de modéliser et de représenter les concepts relatifs aux SMA. Plusieurs

formalismes ont été proposés dans la littérature. Dans les sous-sections suivantes, nous donnons un aperçu d'une sélection des formalismes les plus connus et qui ont fait l'objet de plusieurs articles dans la modélisation des SMA tout en les classant sous deux catégories : Les formalismes formels et ceux semi-formels

4.1 Formalismes semi-formels

Les formalismes semi-formels comptent sur l'aspect graphique. Les concepts relatifs aux SMA sont représentés chacun par un certain graphique. Un des avantages les plus importants de ce type est celui de la visualisation des modèles conçus ce qui facilite leur compréhension et augmente leur lisibilité. Trois formalismes sont sélectionnés :

4.1.1 Agent UML

Agent UML[1] (AUML) est une notation qui supporte la modélisation des systèmes multi-agents. AUML utilise la notation standardisée UML et l'étend afin de représenter les agents, leurs comportements et les interactions entre eux **[41]**. AUML utilise le diagramme de classe UML et lui insérant plusieurs compartiments afin de représenter les caractéristiques de l'agent telles que les comportements, les capacités, les services et les protocoles **[42]**. AUML introduit la stéréotype « agent » pour montrer qu'une classe représente un agent. Le diagramme de classes AUML considère une nouvelle relation entre agents, « *order* ». Cette dernière est particulièrement utilisée dans les hiérarchies pour montrer qu'un agent supérieur dans la hiérarchie demande d'un agent inférieur dans la hiérarchie à faire quelque chose.

AUML adopte une approche en couches pour modéliser les protocoles d'interactions entre agents. La première couche représente le protocole global en utilisant des diagrammes de séquences, des paquetages et des modèles (*templates*). La deuxième couche représente les interactions entre agents en utilisant des diagrammes de séquence, de collaboration, d'activité et d'états. La troisième couche représente le traitement d'agent interne en utilisant des diagrammes d'activités et d'états-transitions.

Poggi, A et al **[43]** ont proposé un diagramme de déploiement AUML, un diagramme de déploiement UML amélioré par des concepts basés-agent, entre autres, le concept de mobilité.

4.1.2 AML

AML **[44]** (acronyme de Agent Modeling Language) est un langage de modélisation semi-formel basé sur la superstructure d'UML 2.0 **[45]** pour la spécification, la modélisation

[1] www.auml.org

et la documentation des systèmes en termes de concepts tirés de la théorie des SMA. AML peut s'appliquer au contexte des systèmes explicitement conçus en utilisant les concepts des SMA. En général, AML peut être utilisé quand il est approprié ou utile pour construire des modèles qui, entre autres :

- Sont constitués d'un nombre d'entités autonomes, concurrents et/ou asynchrones (éventuellement proactives) ;
- Comprennent des entités qui sont capables d'observer et/ou interagir avec leur environnement ;
- Utilisent des interactions complexes et des services agrégés.

Plusieurs sources ont contribué dans la conception d'AML telles que, UML1.5 **[46]**, UML 2.0 **[45]**, OCL 2.0 **[47]**, AUML **[41]**, i* **[48]**, Gaia **[10,11,12]**, PASSI **[15, 16]**, Prometheus **[49, 50]**.

4.1.3 AMOLA (Agent MOdeling LAnguage)

AMOLA (acronyme de Agent MOdeling Language) **[51]** est un langage qui fournit la syntaxe et les sémantiques pour créer des modèles des SMA couvrant la phase d'analyse et de conception. Il supporte une approche modulaire de conception d'agents et introduit les concepts du control intra/inter agent. Le contrôle intra-agent définit le cycle de vie de l'agent par la coordination des différents modules qui implémentent ses capacités. Le contrôle inter-agents définit les protocoles qui régissent la coordination de la société d'agents. La modélisation du contrôle intra/inter-agent est basée sur les diagrammes d'états. La phase d'analyse s'appuie sur les concepts de capacité et de fonctionnalité. La méthodologie ASEME (Chapitre suivant) est basée sur AMOLA.

4.2 Formalismes formels

Les formalismes formels comptent sur l'aspect de précision. Les concepts relatifs aux SMA sont représentés mathématiquement sans aucune ambigüité. Un des avantages les plus importants de ce type est celui de l'exactitude et la possibilité d'appliquer des techniques formelles de vérification et de validation ce qui permet de détecter les erreurs le plus tôt possible **[52]**. Trois formalismes sont sélectionnés :

4.2.1 La notation Z

La notation Z, né au laboratoire d'informatique de l'université d'Oxford, est un langage de spécification formelle **[53]**. Cette notation, ainsi que son extension Z-objet (qui supporte les principes du paradigme objet), est basée sur une théorie et une logique : la théorie des

ensembles et la logique du premier ordre. La première fournit la fondation pour construire une structure abstraite du système informatique sous spécification. La deuxième est utilisée pour exprimer le comportement du système.

Dans le domaine des SMA, la notation Z fait l'objet de plusieurs travaux où les auteurs ont essayé de l'utiliser pour spécifier les SMA. Smith, G et al **[54]** ont introduit MAZE, une extension de la notation Z-Objet **[55]** afin qu'elle puisse supporter les SMA. Brandão, A et al **[56]** ont présenté AgentZ qui étend la notation Z et Z-objet afin de fournir une notation permettant de vérifier les modèles de conception des SMA.

4.2.2 Les machines abstraites

La machine abstraite est un concept très proche de certaines notations bien connues dans la programmation sous le nom du *module, classe* ou *type abstrait de données*. Elle basée sur la théorie des ensembles et le calcul des prédicats.

Une machine abstraite peut être considérée comme une calculatrice qui est caractérisée par la présence d'une mémoire invisible (la partie statique de la machine) et d'un certain nombre de boutons (la partie dynamique de la machine). La mémoire forme l'état de la machine et les différents boutons sont des opérations qu'un utilisateur peut l'activer afin de modifier l'état en question **[57]**.

Les machines abstraites représentent le mécanisme de base de la méthode formelle B **[57]** qui est introduite par Jean-Raymond pour la spécification, la conception et le codage des systèmes logiciels tout en suivant un processus de raffinement. Cette dernière fait l'objet d'un ensemble d'articles sur la spécification des SMA. Fadil, H et al. **[58]** utilisent la méthode B pour modéliser formellement les interactions entre agents afin de les vérifier ensuite et prouver les spécifications UML initiales et ils ont illustrés leur approche à travers le protocole standardisé Contract-Net **[39]** comme une étude de cas. Jemni Ben Ayed, L et al. **[59]** ont présenté une technique de spécification et de vérification pour les protocoles d'interactions dans les SMAs en combinant AUML et la méthode Event-B[1] **[60]**. Dans leur technique, le protocole d'interaction est modélisé dans un diagramme de protocole AUML et translaté ensuite dans la méthode Event-B. Les propriétés de sécurité et de vivacité requises sont ajoutées à la spécification dérivée pour vérification à l'aide de l'outil B4free[2].

[1] Event-B est une extension de la méthode B qui prend en compte la notion d'évènement.
[2] http://www.b4free.com

4.2.3 Maude

Maude est un langage de spécification formelles pour la spécification et la vérification des systèmes concurrents. Il est basé sur la logique de réécriture. Maude et la logique de réécriture seront abordés en détails dans le chapitre 04. Maude fait l'objet de plusieurs articles dans lesquels il est utilisé pour la modélisation et la spécification des SMA. Parmi les travaux qui utilisent Maude comme formalisme de spécification des SMA, on cite le travail de Mokhati et al **[61]** qui ont utilisé Maude pour spécifier puis valider les protocoles d'interaction entre agents. Mohamed Amin, L., et al **[62]** ont utilisé Maude pour décrire formellement organisations des agents.

4.2.4 Réseau de Petri

Les réseaux de Petri **[63, 64]** est un framework de modélisation formelle pour les systèmes concurrents, distribué et complexes très connus. Ma, B **[65]** a introduit une approche pour modéliser les SMA par les réseaux de Petri colorés hiérarchiques **[66]** où chaque agent est modélisé par un système de réseau de Petri coloré et le SMA composé de plusieurs agents est modélisé par un système de réseaux de Petri colorié et hiérarchique. Zhu, Weifeng et al **[67]** ont présenté une méthode appelée BDIPN de modélisation des croyance, désirs et les intentions des agents BDI[1].

5. Plateformes de développement des SMA

Depuis l'apparence du paradigme agent, plusieurs plateformes ont été construites pour faciliter l'implémentation et le déploiement des SMA. Dans cette section, nous allons représenter quelques plateformes en concentrant beaucoup plus sur la plateforme la plus utilisée et reconnue (dans notre estimation) par les développeurs des SMA, JADE.

5.1 FIPA-OS

FIPA-OS[2] (acronyme de FIPA Open Source) est une plateforme d'agent ouverte (open source) implémentée cent pour cent (100 %) en Java. La plateforme supporte la communication entre les différents agents en utilisant un langage de communication d'agent qui se conforme aux standards d'agents de la FIPA **[68]**. La concentration clef de cette platform est le fait qu'elle supporte l'ouverture. Cette dernière est supportée naturellement

[1] BDI (acronyme de Belief-Desire-Intention) est une architecture d'agent où chaque agent possède un ensemble d'informations sur l'environnement et sur autres agents situés dans l'environnement dans lequel il est situé, et possède un ensemble de buts à atteindre (Desires) avec un ensemble de plans (Intention) à suivre pour accomplir les buts.

[2] http://fipa-os.sourceforge.net/index.htm

par le paradigme agent lui-même et par la conception de la plateforme elle-même dont les parties ont un couplage faible de sorte que des extensions et des innovations pour soutenir la communication d'agent peuvent se produire dans plusieurs domaines. FIPA-OS est disponible dans sa version actuelle 2.2.0 et téléchargeable directement via le lien : https://sourceforge.net/projects/fipa-os/files/ .

5.2 JACK

JACK[1] **[69]** est une plateforme commerciale pour le développement des SMA. JACK est une implémentation mature du paradigme BDI. Elle inclut un langage de programmation orienté-agent permettant aux développeurs d'écrire les agents directement en utilisant les concepts d'agents (plans, buts, croyances, etc.). Le langage de la plateforme JACK étend Java afin de supporter une programmation orientée-agent. Il définit des nouvelles classes, interfaces et méthodes, comme il fournit des extensions à la syntaxe de Java afin de supporter les nouvelles classes orientées-agent. Toutes ces extensions sont implémentées en tant que des plug-in Java. La plateforme Jack fournit aussi un environnement de développement intégré (IDE) et un ensemble d'outils de débogage.

5.3 MADkit

MADkit[2] **[70]** (acronyme de Multi Agent Development kit) est une plateforme permettant un développement générique des systèmes multi-agents basé sur le concept d'organisation. L'aspect clef de MADkit est que, contrairement aux approches conventionnelles qui sont principalement centrées sur l'agent, MADkit suit une approche centrée sur l'organisation. Par conséquent, MADkit est construit sur le modèle organisationnel AGR (Agent/Groupe/Rôle) et ne repose pas sur un modèle d'agent prédéfini. Les agents jouent des rôles dans des groupes et créent ainsi des sociétés artificielles **[71]**. La dernière version de MADkit est 5 (téléchargeable depuis le site référencié au pied de la page).

5.4 Jason

Jason[3] **[72]** est un interpréteur d'une version étendue du langage basé sur l'architecture BDI, AgentSpeak. Il implémente les sémantiques opérationnelles de ce langage et fournit une plateforme pour le développement des systèmes multi-agents. Jason est disponible et open-source et il est distribué sous la license GNU LGPL.

[1] http://www.aosgrp.com/products/jack/
[2] http://www.madkit.net/madkit/
[3] http://jason.sourceforge.net/wp/

5.5 JADE

Jade[1] (acronyme de Java Agent Development Framwork) est une plateforme implémentée complètement en Java qui facilite l'implémentation et l'exécution des systèmes multi-agents par un intergiciel qui se conforme aux spécifications de la FIPA[2] **[73]**. Jade est gratuit et distribué par *Telecom Italia* sous les termes et les conditions de la licence LGPL[3]. La version actuelle de Jade est 4.5.0 et elle est téléchargeable depuis le site Web référencié en pied de la page. Jade est parmi les plateformes les plus utilisées et les plus populaires dans l'implémentation des systèmes multi-agents. Il est basé sur les principes suivants **[74, 75]** :

- **L'interopérabilité** : Jade est conforme aux spécifications de la FIPA. En conséquence, les agents Jade peuvent interopérer avec autres agents qui se conforment au même standard.
- **Uniformité et portabilité** : Jade fournit un ensemble homogène des interfaces de programmation (APIs) qui sont indépendants du réseau sous-jacent et la version de Java. Le runtime Jade fournit les mèmes APIs pour les environnements J2EE (Java 2 Platform, Enterprise Edition), J2SE (Java 2 Platform, Standard Edition) et J2ME (Java 2 Platform, Micro Edition).
- **Facilité d'utilisation** : La complexité de l'intergiciel est cachée derrière un ensemble des APIs simples et intuitifs.
- **La philosophie** « Pay-as-you-go » : Les programmeurs n'ont pas besoin d'utiliser toutes les fonctionnalités fournies par l'intergiciel.
- **La mobilité :** Dans les environnements J2SE, Jade supporte la mobilité du code ainsi que l'état de l'exécution. Autrement dit, un agent peut arrêter l'exécution sur un hôte, émigre sur un autre hôte à distance (sans besoin d'avoir le code installé déjà sur cet hôte) et de continuer son exécution du point où qu'il a s'était interrompu.

Une plateforme JADE comprend trois composants de base : (1) Un environnement d'exécution dans lequel les agents existent. (2) Une bibliothèque de classes Java utilisable directement ou par extension pour développer les agents. (3) Suite d'outils graphiques permettant l'administration et la gestion des agents actifs. La figure 1.5 montre les éléments architecturaux principaux d'une plateforme JADE.

[1] http://jade.tilab.com/
[2] http://www.fipa.org/
[3] https://www.gnu.org/licenses/old-licenses/lgpl-2.1.en.html

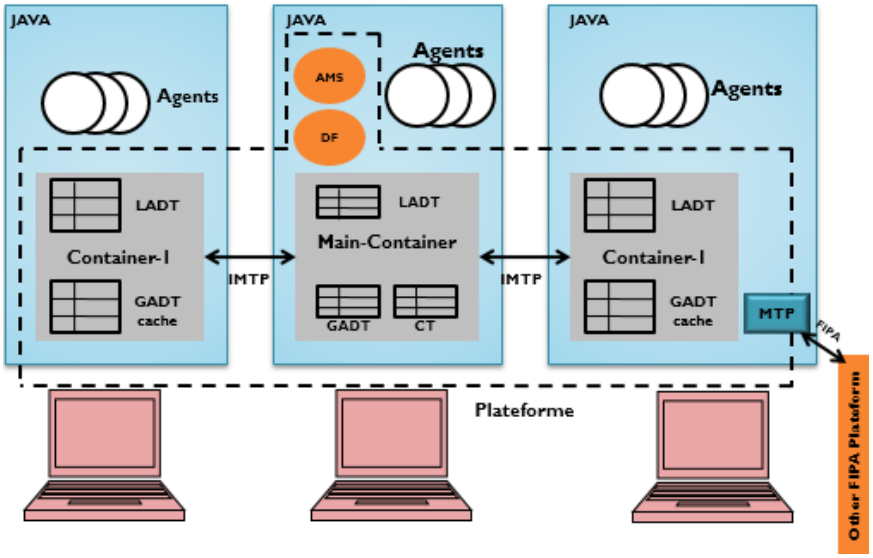

Figure 1.5 : Les éléments architecturaux principaux d'une plateforme JADE **[73].**

La figure 1.6 montre le diagramme de classes des différents éléments architecturaux d'une plateforme JADE. Une plateforme d'agents Jade est composé d'un ou plusieurs conteneurs. Chaque conteneur d'agents est un environnement multithreads composé d'un thread d'exécution pour chaque agent. Il gère localement un ensemble d'agents (création, attente et destitution) et assure le traitement de communications (routage des messages, répartition des messages ACL et gestion des messages vers l'extérieur). La plateforme doit avoir un conteneur principal actif. Ce dernier possède deux agents spéciaux : *AMS* (Agent Management System) et *DF* (Directory Facilitator) qui offre le service des pages jaunes.

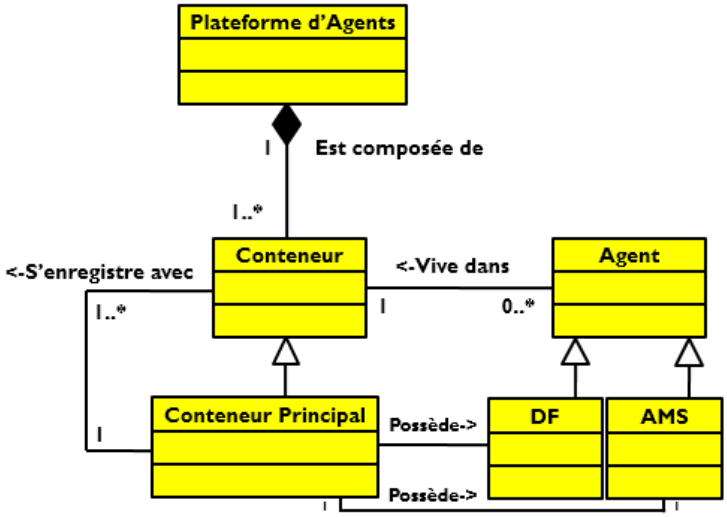

Figure 1.6 : Diagramme de classes des éléments architecturaux de JADE **[73].**

Jade fait l'objet de plusieurs tentatives pour l'étendre et l'améliorer. Jadex[1] (acronyme de JADE eXtension) est une extension de la plateforme JADE en adoptant l'architecture BDI

[1] https://sourceforge.net/projects/jadex/

(Belief-Desire-Intention) par l'utilisation de la technologie XML **[76]**. Braun, P., et al **[77]**, les auteurs ont essayé d'étendre Jade par un nouveau service de mobilité appelé Kalong. Madrigal-Mora Cristián et al **[78, 79]** ont proposé JadeOrgs qui étend Jade en introduisant les deux concepts : Organisation et rôles, comme des entités du premier niveau disponibles en temps d'exécution. S, Zerrougui et al **[80]** ont proposé une approche pour améliorer le service des pages jaunes au cas où le système multi-agents soit de taille importante (scalabilité). Dans **[81, 82]**, les auteurs proposent et implémentent un add-on appelé Jade-Leap qui peut s'exécuter, non seulement sur les ordinateurs personnels (PCs), mais aussi sur les téléphone mobiles Android. Jade-Leap fournit les mêmes APIs par rapport au Jade afin qu'un agent développé pour s'exécuter sur Jade puisse s'exécuter sur Jade-Leap sans aucune modification. Vishnuvardhan Mannava et al. ont décrit leurs efforts dans **[83]** pour intégrer les protocoles JXTA[1] dans JADE afin de faciliter les communications inter-agent via l'Internet. Dans **[84]**, Iva Bojic et al. ont introduit le framework JBehaviourTrees qui étend les comportements Jade avec des arbres de comportements (BTs). BTs sont construits à travers la composition des tâches de base augmentant la possibilité de la modularité et de réutilisation du code. Bergenti, F., et al. **[85]** ont proposé et décrit JADEL. JADEL est conçu pour supporter une implémentation effective des agents et SMA basés-JADE en supportant quatre abstractions de base : agents, comportements, ontologies de communication et protocole d'interaction.

6. Outils de développement des SMA

Les outils de développement représentent un additif important pour la réalisation pratique d'applications logicielles, principalement parce qu'ils facilitent l'automatisation d'activités de développement et sont capables de cacher la complexité aux développeurs **[86]**. Dans cette section, une collection d'outils est représentée.

6.1 PDT

PDT (acronyme de Prometheus Design Tool) **[87]** est un outil graphique qui supporte la conception structurée des SMA. Il supporte la méthodologie Prometheus **[49, 50]**, mais peut également être utilisé plus généralement. Cet outil est écrit en Java et disponible pour téléchargement[2].

[1] La technologie JXTA est un ensemble de protocoles ouverts qui permettent à tout appareil connecté sur le réseau, depuis des téléphones portables et PDAs sans fil aux PCs de communiquer et de collaborer de manière P2P. http://www.oracle.com/technetwork/java/index-jsp-136561.html

[2] http://www.cs.rmit.edu.au/agents/pdt.

Il est basé sur des entités spécifiques d'agent telles que buts, plans, perceptions, actions, protocoles. Dans **[88]**, une version de PDT[1] intégrable dans la plateforme Eclipse, ce qui permet aux utilisateurs d'accomplir le cycle de développement complet d'une application orientée agent dans un IDE et également hériter d'un ensemble riche de fonctionnalités de développement fournies par Eclipse.

6.2 Agent Factory

Agent factory[2] **[89]** est une collection open-source d'outils, de plateformes et de langages qui supporte le développement et le déploiement des SMA. Ce framework est divisé en deux parties : La prise en charge du déploiement des agents sur les ordinateurs portables, les ordinateurs de bureau et les serveurs (AFSE : Agent Factory Standard Edition) ; et la prise en charge du déploiement d'agents sur des périphériques restreints tels que les téléphones mobiles et les capteurs (AFME : Agent Factory Micro Edition). AFSE combine une plateforme d'agent conforme à la FIPA à part entière avec la prise en charge d'une gamme de langages et d'architectures d'agents. Il est entièrement intégré à Eclipse de manière à simplifier la prise en charge de nouveaux langages et architectures.

6.3 TAOM

L'environnement nommé par TAOM (acronyme de Tool for Agent Oriented Modeling) **[90]** est l'outil développé initialement comme un modélisateur supportant la méthodologie Tropos **[9]**. Un premier prototype de cet outil **[91]** a été développé et distribué sous la licence GPL2.

TAOM4E[3] résulte de la réingénierie de la version précédente du modélisateur TAOM et étend ses fonctionnalités pour supporter la méthodologie Tropos depuis l'ingénierie plut-tôt des exigences jusqu'à la génération de code BDI. TAOM4E supporte un développement dirigé par les modèles des logiciels orienté-agents suivant la méthodologie Tropos. Cet outil est disponible en tant qu'un plug-in Eclipse.

6.4 IDK

IDK[4] (acronyme de INGENIAS Development Kit) **[92]** est un framework pour l'analyse, la conception et l'implémentation des SMA. Il est basé sur les spécifications des méta-modèles SMA adopté par la méthodologie Ingenias **[13]**.

[1] https://sites.google.com/site/rmitagents/software/prometheusPDT/downloads
[2] http://agentfactory.ucd.ie/index.php/Main_Page
[3] http://selab.fbk.eu/taom/
[4] http://ingenias.sourceforge.net/

6.5 PTK

PTK (acronyme de PASSI ToolKit) est l'outil supportant la méthodologie PASSI. La méthodologie PASSI ainsi que l'outil PTK seront représentés dans le chapitre 3.

7. Méthodologies de développement des SMA

Au début de notre recherche, nous avons essayé de trouver une définition standard du terme « *terminologie* ». Néanmoins, nous avons trouvé un grand conflit entre les deux termes : « *Méthodologie* » et « *Méthode* ». Dans cette thèse, le terme « *Méthodologie de développement des SMA* » signifie : *Un guide pour planifier, contrôler et structurer le processus de développement d'un SMA avec des composants spécifiques tels que des phases, des tâches, des méthodes, des techniques et d'outils*.

Depuis l'apparence du paradigme agent, plusieurs méthodologies de développement des SMA ont été proposées dans la littérature telles que Gaia **[10, 11]**, Message **[93]**, ADELFE **[94, 95, 96]**, Prometheus **[49, 50].** La méthodologie Gaia **[10, 11]** est considérée comme la méthodologie la plus connue dans la littérature. Pour cela, nous donnons dans cette section une description de la méthodologie Gaia et nous allons laisser la description d'autres méthodologies pour le chapitre suivant.

Inventé par Michael Wooldridge[1] en 2000, Gaia **[10, 11]** est considérée comme la méthodologie générique la plus connue par la communauté du GLOA. Elle voie un système comme une organisation composée d'une collection de rôles. Son processus de développement se compose de trois (3) phases principales **[11]** : *L'analyse* et *la conception architecturale et la conception détaillée (*Figure 1.7).

Dans la phase d'analyse, les sous-organisation légèrement couplées du SMA entier. Ensuite, pour chacune de ces dernières, quatre modèles abstraits basiques : *(a) Le modèle environnemental*, afin de capturer les caractéristiques et leurs représentations de l'environnement du SMA, *(b) le modèle préliminaire de rôles*, afin de capturer les activités clefs à être jouées dans le SMA. *(c) le modèle préliminaire d'interactions*, afin de capturer les interdépendances basiques entre les rôles et leurs protocoles correspondants ; et *(d) Un ensembl*e de règle organisationnelles, exprimant les contraintes globales qui doivent être à la base du fonctionnement du SMA.

[1] Michael Wooldridge est considéré parmi les pères du paradigme agent, aussi il est l'auteur du livre référence intitulé par « An Introduction To Multi-Agent Systems ». Les intéressés peuvent consulter sa page web : http://www.cs.ox.ac.uk/people/michael.wooldridge/

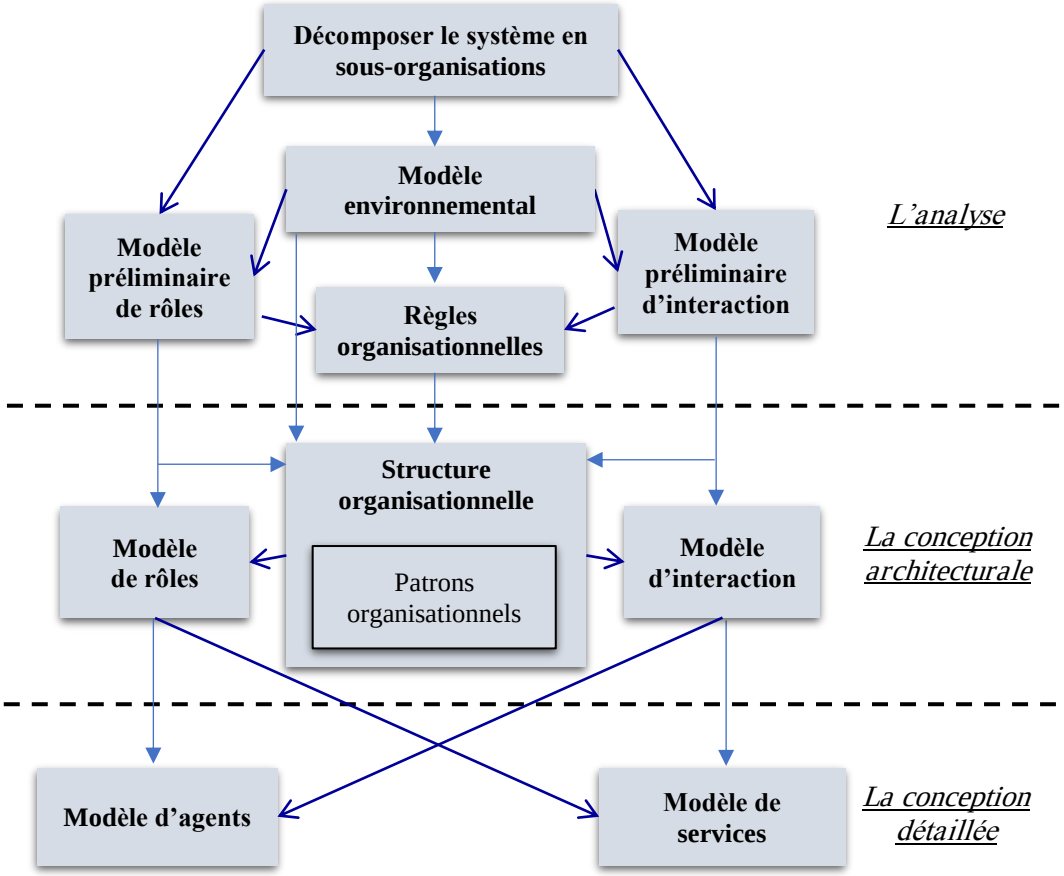

Figure 1.7 : Processus de conception et modèles de Gaia **[11]**.

Dans la phase de conception détaillée, la structure organisationnelle du SMA, la topologie du SMA entier et de chaque sous-organisation sont définies. La topologie complète les buts et les règles organisationnelles des sous-organisations. Ces activités résultent de deux modèles : *Modèle (complet) de rôles* et *le modèle (complet) d'interactions*.

Enfin, la conception détaillée implique l'identification de : *(a) Un modèle d'agents*, un ensemble de classes d'agents dans le SMA implémentant les rôles identifiés et les instances spécifiques de ces classes, *(b) Un modèle de services*, exprimant les services et les protocoles d'interaction à fournir dans les classes d'agents.

Plusieurs travaux ont essayé d'étendre GAIA en termes de :

- *Phases de développement couvertes* **:** Rodriguez et al **[97]** ont introduit une phase de modélisation des besoins dans le processus de développement de Gaia. Moraitis, P et al **[98]** ont proposé un processus particulier consacré à la conversion des modèles de Gaia en code JADE.
- *Scalabilité* **[99]** : Les auteurs ont adapté Gaia pour supporter la conception des SMA ouverts et complexes (ROADMAP).

8. Conclusion

Nous avons vu dans ce chapitre les principes de bases relatifs au domaine du génie logiciel orienté-agent en partant de la définition du terme de « génie logiciel orienté-agent » ainsi que ses intérêts. Plusieurs concepts ont été standardisés tels que, Agent communication. En outre, des plateformes de développement des SMA ont été proposées dans la littérature tels que JADE, Jadex et Jack. Les méthodologies de développement des SMA visent à définir les différentes étapes, notations, outils, plateformes utilisées lors le cycle de développement. Gaia est la méthodologie la plus connue par les chercheurs de la communauté du GLOA.

Chapitre 02/ Méthodologies de développement des SMA

---

1. Introduction

L'objectif principal de la proposition des méthodologies de développement des SMA étant de faciliter leur développement. Ces méthodologies ont certes apporté de réelles avancées dans la conception et l'implémentation des SMA. Dans ce chapitre, nous allons décrire les méthodologies les plus célèbres, celles qu'elles ont contribué dans le domaine du génie logiciel orienté agent, en les classant en deux catégories : Les méthodologies dirigées par les modèles (qui utilisent des techniques de l'IDM) et celles dites formelles (qui utilisent les méthodes formelles). Si une méthodologie combine l'utilisation des deux techniques (de l'IDM et des méthodes formelles), elle sera citée sous les deux catégories. Le reste de ce chapitre est organisé comme suit : La deuxième et la troisième sections sont consacrées aux méthodologies utilisant des techniques de l'IDM et celles formelles respectivement. Nous allons conclure le chapitre dans la quatrième section.

2. Méthodologies utilisant les techniques de l'ingénierie dirigée par les modèles

2.1 L'ingénierie dirigée par les modèles

2.1.1 Définition

L'Ingénierie Dirigée par les Modèles (IDM) est un paradigme dans lequel la modélisation est considérée comme étant l'élément central d'un logiciel. Ce paradigme met le modèle au centre des préoccupations des concepteurs et des analystes. L'IDM correspond à l'utilisation systématique des modèles comme concept clé tout au long du processus d'ingénierie. En outre, elle propose une automatisation du processus d'ingénierie, des modèles au code par affinements successifs des modèles **[100]**.

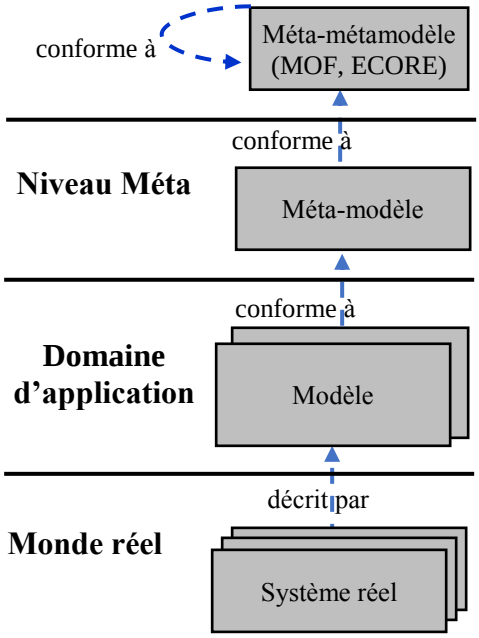

Figure 2.1 : Vue multi-niveau de l'ingénierie dirigée par les modèles.

2.1.2 Modèle, méta-modèle & méta-métamodèle

Comme illustré dans la figure 2.1, trois concepts clés peuvent être capturés dans la théorie de l'IDM : Modèle, méta-modèle et méta-métamodèle (par fois appelé mega-modèle[1] **[101]**). Nous pouvons définir un modèle comme une représentation plus ou moins abstraite du système réel. Les méta-modèles sont utilisés pour conceptualiser et définir la syntaxe abstraite des langages de modélisation. Autrement dit, un méta-modèle est encore une autre abstraction qui spécifie les propriétés du modèle lui-même **[100]**. Le niveau méta fournit

[1] Ce terme est sélectionné pour éviter la confusion avec les sens basiques des termes : modèle et méta-modèle.

des langages (méta-métamodèles, tels que MOF **[102]**, Ecore[1]) utilisés dans la spécification des méta-modèles auxquels les modèles se conforment. Les méta-métamodèles sont représentés par eux même.

2.1.3 Transformation de modèles

À côté de modèles, les transformations de modèles constituent un composant très crucial de la théorie de l'IDM **[103]**. En permettant la formalisation du mappage entre des modèles différents ou un modèle en des niveaux d'abstraction différents, les transformations de modèles facilitent l'automatisation de génération d'un modèle dit cible à partir d'un modèle dit source (qui se conforment les deux aux leurs méta-modèles correspondants) en exécutant un ensemble de règles de transformation (Figure 2.2). Les règles de transformations donnent la correspondance entre chaque élément du méta-modèle source et celui du méta-modèle cible.

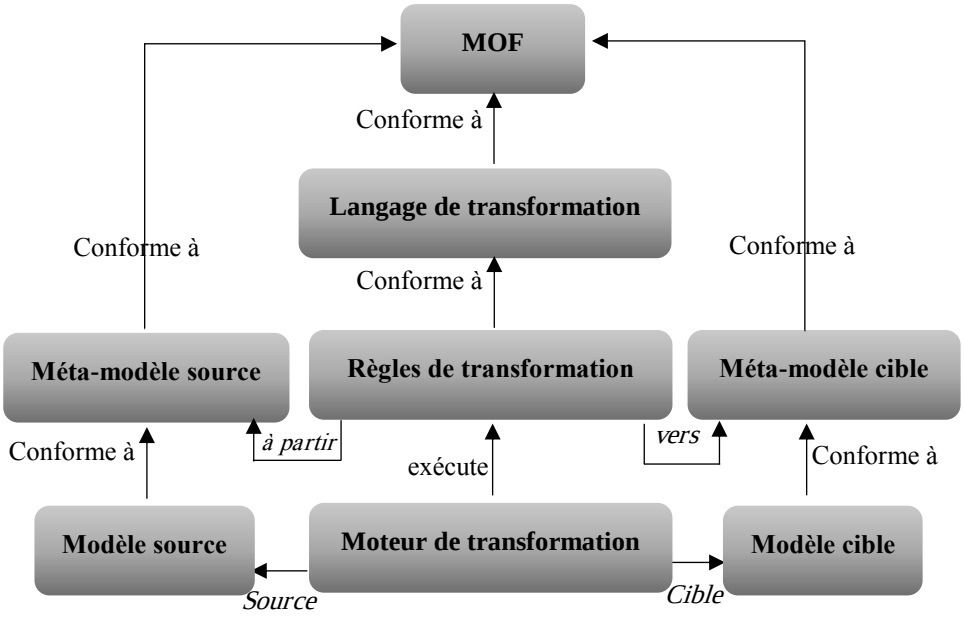

Figure 2.2 : Technique de transformation de modèles **[104].**

Le type (texte ou pas) du modèle source ou cible fait engendrer trois (3) types de transformation : *Modèle-à-modèle* (M2M), *Modèle-à-texte* (M2T) et *Texte-à-modèle* **[105]** (T2M). Les transformations de type M2T sont utilisés généralement pour générer le code source ou bien une spécification formelle textuelle à partir d'un modèle conceptuel. Autres considérations ont été mis en compte pour la classification de transformations de modèles [**106, 107]**.

[1] http://www.eclipse.org/ecoretools/overview.html

2.1.4 L'architecture dirigée par les modèles

L'architecture dirigée par les modèles **[108]** (ADM ou MDA en anglais) est une approche de l'IDM proposée par l'OMG[1] (Acronyme de Object Management Group) dans l'objectif d'améliorer le développement d'applications en termes de productivité, portabilité, interopérabilité **[109]**. Cette approche recommande la séparation de la spécification de la fonctionnalité du système et la spécification de l'implémentation de cette fonctionnalité sur une plateforme spécifique. Elle utilise trois types de modèles : CIM (Computation Independant Model) dans lequel le système est modélisé dans un niveau d'abstraction très élevé, PIM (Platform Independant Model) dans lequel la structure et le comportement du système sont définis indépendamment d'aucune plateforme d'exécution et PSM (Platform Specific Model) qui définit la solution sur une plateforme spécifique **[110]**.

2.2 Les méthodologies basées-MDE de développement des SMA

Dans cette section, quelques méthodologies dirigées par les modèles sont représentées. Dans ce type de méthodologies, les modèles dirigent le processus de développement. Les méthodologies sont représentées suivant leurs dates d'apparence chronologiques.

2.2.1 Tropos

A. Brève description

Tropos[2] **[14, 15]**, est une méthodologie de développement des SMA basée sur les modèles qui utilise les concepts relatifs au paradigme agent durant tout le processus de développement. Les notions telle que : Agent, bût, tâche et dépendance sont utilisées pour modéliser et analyser les besoins initiaux et finaux, la conception architecturale/détaillée et pour implémenter le SMA final. Particulièrement, elle concentre beaucoup plus sur la phase de spécification et d'analyse des besoins. Tropos est considérée parmi les méthodologies les plus évolutive (en termes d'amélioration d'extensions telles que Formal-Tropos, section 3.2.1) et documentée.

B. Processus de développement

Le processus de développement de Tropos est constitué de cinq (5) phases comme illustré par la figure 2.3 : *La phase d'analyse des besoins initiaux* est destinée à la compréhension et la modélisation des paramètres organisationnels existants où le SMA va être introduit. L'organisation est représentée en termes d'acteurs orientés-bût qui se

[1] https://www.omg.org/mda/
[2] http://www.troposproject.eu/

dépendent socialement pour atteindre leurs objectifs. *La phase d'analyse des besoins finaux* se commence par les sorties de la phase précédente et introduit le SMA dans les paramètres organisationnels. Dans *la phase de conception architecturale,* l'architecture globale du SMA est définie en termes des agents interagissant. *La phase de conception détaillée* raffine encore la spécification du système en définissant les fonctionnalités à implémenter dans chaque agent aussi bien que les protocoles d'interaction. *La phase d'implémentation et de test* est destinée au développent des agents du système et à leur vérification. Il s'agit de s'assurer de la conformité du fontionnement de l'interaction des agents à leur spécification. Les modèles conçus tout au long du processus de développement de Tropos sont créés en utilisant le langage de modélisation conceptuelle i* **[48]** et complétés par les diagrammes d'activité et de séquence d'UML**.**

Figure 2.3 : Processus de développement de Tropos **[14]**.

C. Adoption de l'IDM

Tropos est associée avec un outil de conception, TAOM4E[1] (**A**cronyme de **T**ool for visual **A**gent **O**riented **M**odelling for the **E**clipse platform). Cet outil est basé sur le plug-in EMF[2] et GEF[3] de la plateforme Eclipse. Morandini, M et al. **[111, 112]** ont proposé Tropos4AS, une extension de la méthodologie Tropos afin de supporter la modélisation des systèmes auto adaptatifs et l'outil t2x (Tropos2Jadex) qui génère le code pour plateforme JADEX est aussi présenté. Bertolini, D., et al **[113]** ont introduit et décrit un environnement de développement qui supporte le processus de développement complet de Tropos en lui enrichissant par des techniques de transformation de modèles. Cet environnement utilise un plugin Eclipse de transformation comme étant un moteur de transformation de modèles, *Tefkat*[4]. L'environnement introduit par Bertolini et al, utilise QVT[5] pour transformer le modèle de capacités des agents (exprimé via des entités et des structures Tropos) en la définition du processus sous-jacent à l'exécution des capacités (exprimés via des diagrammes d'activités AUML -sous-section 4.1.1 du chapitre 01-). Une deuxième transformation peut être effectuée par l'environnement afin de générer un ensemble de

[1] http://selab.fbk.eu/taom/
[2] https://www.eclipse.org/modeling/emf/
[3] https://www.eclipse.org/gef/
[4] http://tefkat.sourceforge.net/
[5] https://www.omg.org/spec/QVT/1.3/PDF

classes JADE à partir de diagrammes d'activité et de séquence qui spécifient les comportements des agents.

2.2.2 ADELFE

A. Brève description

ADELFE[1] **[94, 95, 114]** (acronyme de **A**telier de **DE**veloppement de **L**ogiciels à **F**onctionnalité **E**mergente) est une méthodologie proposée en 2003 où le but étant d'aider et de guider les concepteurs durant le développement des systèmes multi-agents adaptatifs (SMAA) qui sont utilisés dans des situations dans lesquelles les besoins sont incomplètement exprimée, l'environnement est imprédictible ou le système est ouvert. La théorie des SMMA **[115]** est basée sur le concept d'auto-organisation. Depuis son apparence, la méthodologie ADELFE est étendue et améliorée en termes de phases couvertes par son processus de développement **[94]** et en termes d'adoption d'une approche de simulation lors de conception **[95].**

B. Processus de développement

Le processus de développement de la méthodologie ADELFE **[116]**, en sa deuxième version, se compose de cinq phases (Figure 2.4). La première phase (*Les besoins Préliminaires*) concentre sur la collecte d'informations sur le client et ses besoins. Une description consensuelle du problème est faite en termes des besoins fonctionnels et non-fonctionnels ce qui doit conférer au système ses limites et ses contraintes. Dans la deuxième phase (Les b*esoins finaux*), l'objectif est de valider les besoins et le détailler via une description des acteurs, un modèle des cas d'utilisation et des scénarios. La caractérisation de l'environnement du système et l'identification des échecs parmi ses interactions conduisent, à la fin de cette phase, à une conclusion sur l'adéquation des SMA de traiter le problème décrit. Durant la quatrième phase (*Phase d'analyse*), les entités sont caractérisées comme passives ou actives et leurs interactions sont décrites. Ceci, permet à connaitre l'adéquation des SMAA pour traiter le problème du client ou pas. Si le résultat est positif, toutes les interactions entres les entités sont décrites et les échecs de coopération sont identifiés ce qui contribue à l'identification des agents. La quatrième phase (*La phase de conception*) détaille l'architecture en termes de modules et une architecture orientée-agent de la solution est définie. La cinquième phase (*La phase de l'implémentation*) définit une

[1] https://www.irit.fr/ADELFE/

architecture orientée-composant indépendante d'aucune plateforme d'implémentation des SMA.

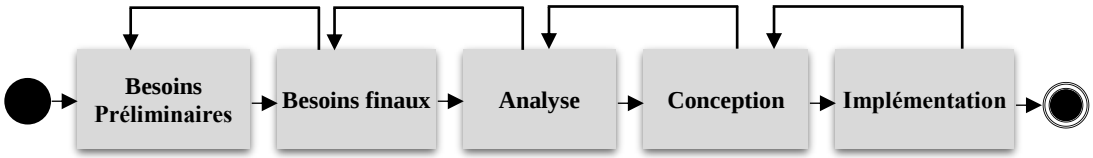

Figure 2.4 : Processus de développement de la méthodologie ADELFE 2.0 **[116]**.

C. Utilisation de techniques de l'IDM

ADELFE propose deux méta-modèles correspondants aux deux langages de modélisation, AMAS-ML (Adaptative MultiAgent Systems-Modeling Language) **[116]** et μADL [**117]** (micro-Architecture Description Language). AMAS-ML, qui est utilisé dans la phase de conception, formalise les concepts de la théorie des SMAA et μADL, qui est utilisé dans la phase de l'implémentation, concentre sur une architecture spécifique. Le modèle résultant de la phase de conception est utilisé comme une entrée de la phase d'implémentation et résulte un modèle μADL via une transformation de modèle. Ensuite, le code conforme à la plateforme JavAct[1] **[118]** du SMAA est généré via une transformation M2T grâce à l'outil MAY[2].

2.2.3 Prometheus

A. Brève description

Prometheus **[49, 50]** est une méthodologie proposée dans le but d'assister et guider les développeurs dans le développement des applications basés-agent, particulièrement, les agents BDI. Les SMA conçus par Prometheus peuvent être implémentés dans n'importe quelle plateforme d'implémentation couvrant de telles abstractions, la plateforme JACK est beaucoup plus préférée. Elle est supportée par un ensemble d'outils facilitant les différentes étapes de développement tels que, PDT[3] (**P**rometheus **D**esign **T**ool) **[87]**.

B. Processus de développement

Le processus de développement de Prometheus est itératif. Il couvre trois phases **[119]** (Figure 2.5) : *La spécification du système, la conception architecturale* et *la conception détaillée*. Dans la phase de spécification du système, le système est spécifié en termes de buts et scénarios des cas d'utilisation ; l'interface du système avec

[1] https://www.irit.fr/smac/fr/javact
[2] https://www.irit.fr/redmine/projects/may
[3] https://sites.google.com/site/rmitagents/software/prometheusPDT

l'environnement est décrite en termes d'actions, percepts et données externes ; et les fonctionnalités sont définies dans la phase de conception architecturale, les types d'agents sont identifiés, la structure globale du système est capturée et les scénarios des cas d'utilisations sont développés en protocoles d'interaction. Dans la phase de conception détaillée, chaque agent identifié est détaillé et développé en termes de capacités, données, évènements et plans.

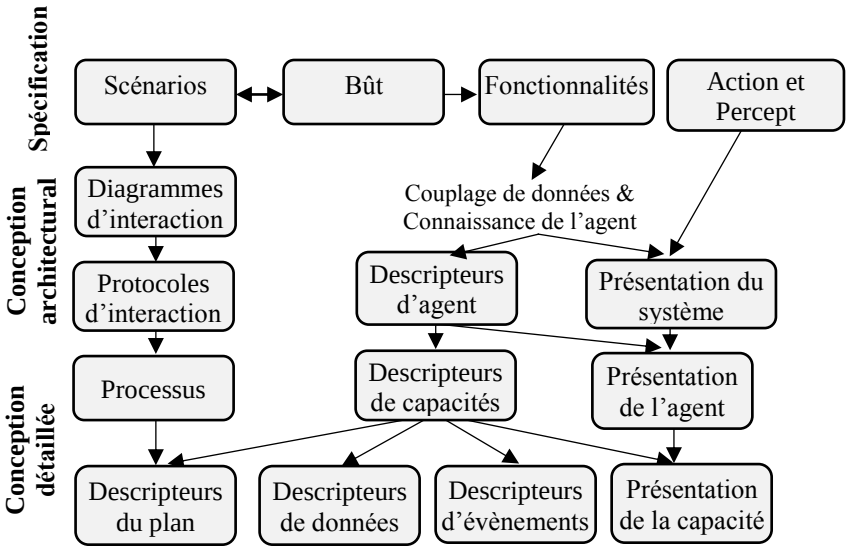

Figure 2.5 : Processus de développement de la méthodologie Prometheus **[119]**.

C. Utilisation de techniques de l'IDM

L'outil PDT est redéveloppé sous forme d'un plug-in intégrable dans la plateforme Eclipse **[88].** Cette version améliorée de PDT permet, entre autres, de générer un squelette de code conforme à la plateforme de développement des SMA, JACK par l'exécution d'une transformation du type M2T. Elle permet aussi de générer un diagramme d'interaction AUML à partir d'une notation textuelle **[120]** en effectuant une transformation de type T2M **[121]**.

2.2.4 INGENIAS

A. Brève description

INGENIAS[1] **[13]**, est une évolution de la méthodologie MESSAGE **[93]** qui fournit une notation pour la modélisation des SMA et une collection bien définie d'activités pour guider le processus de développement d'un SMA dans les phases d'analyse, de conception, de vérification et de génération du code, supportée par un ensemble d'outils intégrés, IDK[2] **[92]**.

[1] http://ingenias.sourceforge.net/
[2] https://sourceforge.net/projects/ingenias/

Elle est basée sur les cinq concepts : Agent, organisation, buts/tâches, environnement et interactions.

B. Processus de développement

Le processus de développement de la méthodologie INGENIAS suit un processus bien connu, le processus unifié **[92]**. INGENIAS rajoute une définition des activités qui peuvent être développées dans les étapes d'analyse et de conception, spécialement, pour identifier, définir et associer les éléments qui construisent le SMA. En générale, INGENIAS a deux workflows : l'analyse et la conception **[122]**. Au cours du workflow d'analyse, des cas d'utilisation sont générés et leurs acteurs sont définis, l'architecture du système esquissée avec un modèle d'organisation. Ensuite, les cas d'utilisation et les interactions qui leur sont associées sont raffinés, les modèles d'agents qui détaillent les éléments de l'architecture du système sont développés, les tâches dans les modèles d'organisation sont décrites et le modèle de l'environnement est aussi raffiné. Pendant le workflow de conception, la première tâche consiste à générer un prototype, ensuite, des raffinements doivent être introduits, les modèles d'interaction sont spécifiés, les tâches et les buts sont modélisés et les modèles d'agents sont définis. Finalement, les relations qui règlent le comportement organisationnel doivent être décrites. La phase d'implémentation est abordée par la génération du code Jade grâce à un ensemble d'outils. Gómez-Sanz et al. **[123]** ont étendu INGENIAS pour couvrir la phase du test et du débogage.

C. Utilisation de techniques de l'IDM

INGENIAS utilise le standard MOF (Meta-Object-Facility) pour décrire les méta-modèles des cinq vues. A partir duquel, *IDK MAS Model Editor* est instancié pour offrir la possibilité de créer et éditer les modèles d'INGENIAS **[124]**, de leurs documentations, de maintenance et de génération de code. Afin de générer du code Jade, des modèles (templates) de code sont liés aux concepts du méta-modèle d'INGENIAS. La génération du code est effectuée grâce au plug-in *IAF* (INGENIAS Agent Framework) sous forme d'une transformation M2T. Une autre transformation de type T2M est effectuée grâce au plug-in *CodeUploader- tool* assurant que les changements apportés sur le code généré sont chargées sur les modèles (ingénierie inverse).

2.2.5 ForMAAD

A. Brève description

Zeineb Graja et al. ont essayé de reformuler la méthodologie ForMAAD (section 3.2.2 du même chapitre) en termes de l'approche MDA en utilisant le langage AML. Des extensions ont été faites pour supporter la nouvelle version de la méthodologie **[125]**.

B. Processus de développement

Le processus de ForMAAD dans sa version MDA se compose en deux étapes principales : *La représentation orientée-modèle* des différents phases (spécification et conception) en utilisant le langage AML et *la translation vers un langage formel*. Le résultat de la première étape est constitué d'un ensemble de modèles : 1) Le modèle de spécification des exigences où, selon la notation AML, une entité est représentée par une classe (*class*), une société d'agents est représentée par une organisation (*organization*), un objectif est représenté par un but décidable (*decidable goal*) et un objectif commun d'une société d'agent est représenté par une contrainte (*constraint*) associée à l'organisation correspondante. 2) Le modèle de définition de la stratégie de coopération. 3) Le modèle de définition de la structure de l'organisation. 4) Le modèle de définition du comportement collectif et 5) le modèle de définition du comportement individuel. La deuxième étape se focalise sur la translation des modèles de la première étape au langage formel $\boldsymbol{T_{emporal}Z}$ **[126].**

Une vérification formelle est appliquée après chaque étape du Processus de développement. La spécification formelle translatée peut être importée par l'outil Z/EVES **[127]** afin de prouver les théorèmes nécessaires pour garantir la satisfaction des exigences de l'utilisateurs.

C. Utilisation de techniques de l'IDM

La version dirigée par les modèles de la méthodologie ForMAAD enrichit la version ordinaire par une conception en premier plan basée sur le langage semi-formel AML et utilise des techniques de transformation de modèles en suivant l'approche MDA pour générer automatiquement un code exécutable. Une transformation de type M2T est effectuée pour produire une spécification $\boldsymbol{T_{emporal}Z}$ à partir de diagrammes AML conçus lors des phases du cycle de développement.

2.2.6 ASEME

A. Brève description

ASEME[1] **[128, 129]** (Acronyme de : the Agent Systems Engineering MEthodology) est une méthodologie dirigée par les modèles pour le développement des systèmes basés-agent qu'utilise le langage AMOLA (section 4.5 du premier chapitre). ASEME s'inspire aussi d'autres méthodologies telles que Tropos (section 2.2.1 du même chapitre) et Gaia (section 7 du chapitre précédent) en termes de diagrammes utilisés. ASEME utilise des concepts tels que : capacité, fonctionnalité, contrôle intra/inter gent.

B. Processus de développement

Le processus de développement de ASEME est itératif. Il se compose de six (6) phases de développement (Figure 2.6) : La phase d'analyse des exigences, la phase d'analyse, la phase de conception, la phase d'implémentation, la phase de vérification et la phase d'optimisation. Les trois premières phases sont de haut-en-bas (top-down), par contre, les trois dernières sont de bas en haut (bottom-up). Dans chacune de ces phases, trois niveaux d'abstractions sont définis. Le premier est le niveau macro (société) où la fonctionnalité globale du SMA est modélisée. Ensuite, au niveau micro (agent) chaque partie de la société (agents) est modélisée. Enfin, plus de concentration sur les détails qui composent chacune des parties de l'agent sont définis dans le troisième niveau.

Durant la phase d'analyse des exigences, les acteurs qu'ils vont agir avec le système sont identifiés, leurs buts sont associés et les exigences spécifiques relatives à chaque but d'un acteur sont définies. Dans la phase d'analyse, les acteurs de la phase précédente sont transformés en rôles, les capacités d'agents sont définies et décomposées en activités et les fonctionnalités que doivent être utilisées afin de réaliser chaque activité sont soulignées par l'analyste.

Dans la phase de conception, les contrôles intra-agents implémentant des protocoles spécifiques d'interaction sont définis à travers la définition des rôles nécessaires et l'interaction entre eux, les contrôles inter-agents sont implémentés, chaque capacité est définie au moyen de ses fonctionnalités, les structures de données et les algorithmes qui devraient être implémentés. Les modèles définis durant la phase de conception (modèles de contrôle intra/inter-agent et diagramme de composants) sont des modèles PIM.

[1] http://aseme.tuc.gr/

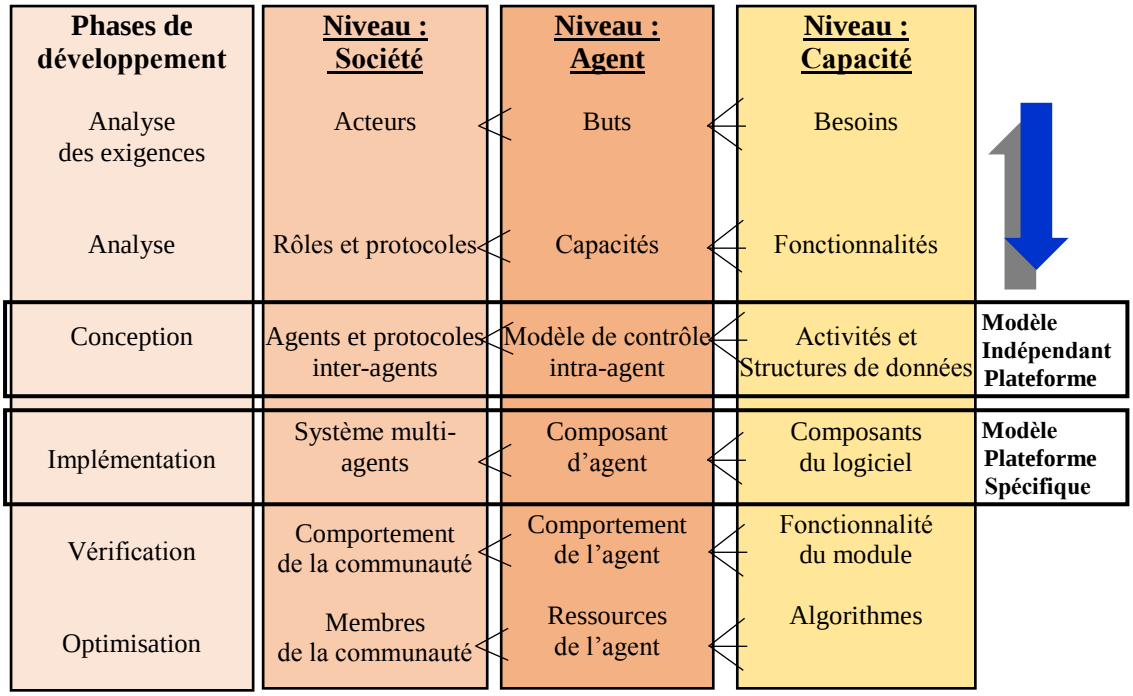

Figure 2.6 : Processus de développement d'ASEME **[129]**.

Dans la phase d'implémentation, la plateforme ou les langages de programmation sont sélectionnés pour implémenter les modules des différents agents et, par conséquence, des modèles PSM sont créés. La fonctionnalité du SMA est vérifiée en comparaison avec ses exigences durant la phase de vérification en parallèle pour les trois niveaux d'abstraction. La phase d'optimisation vise à optimiser le système (les algorithmes sans le temps d'exécution ou la consommation des ressources, le nombre de capacités exécutées en parallèle, le nombre d'agents instanciés/détruits).

C. Utilisation de techniques de l'IDM

ASEME applique l'IDM au développement des SMA, ainsi les modèles d'une phase de développement précédente sont transformés aux modèles de la phase suivante. Les différents modèles sont créés pour chaque phase de développement et la transition d'une phase à l'autre est aidée par une transformation automatique de modèle comprenant des transformations M2M, M2T et T2M. *SAG2SUC*, *SUC2SRM*, *SRM2IAC* et *IAC2JADE* sont des outils développés pour assurer l'automatisation des différentes transformations. Ces outils sont implémentés dans un environnement de développement intégré qui s'appelle ASEME IDE (téléchargeable depuis la page « Downloads » du site Web de la méthodologie) en utilisant un ensemble de plug-in d'Eclipse comme, par exemple, EMF[1], Xpand[2], Xtext[3].

[1] https://www.eclipse.org/modeling/emf/
[2] https://wiki.eclipse.org/Xpand
[3] https://www.eclipse.org/Xtext/

3. Méthodologies utilisant les méthodes formelles

3.1 Les méthodes formelles

3.1.1 Définition

Les méthodes formelles sont des techniques basées sur les mathématiques. Elles sont utilisées dans l'informatique pour décrire les propriétés des systèmes matériels et/ou logiciels **[8]**. Elles offrent un framework avec lequel, des systèmes grands et complexes peuvent être spécifiés, développés et vérifiés d'une manière systématique plutôt qu'une manière ad-hoc. La rigueur mathématique des méthodes formelle permet aux utilisateurs d'analyser et vérifier les systèmes informatiques à n'importe quelle partie du cycle de développement : L'ingénierie des besoins, la spécification, la conception, l'implémentation, le test et la maintenance **[130]**.

3.1.2 Développement des SMA avec les méthodes formelles

Le développement formel des SMA implique la définition d'une méthodologie où les méthodes formelles joue un rôle dominant dans leur développement. Dans la littérature, trois alternatives ont été capturées **[131]** : La dérivation formelle, l'intégration avec une méthodologie existante ou bien la proposition de nouvelles méthodologies.

A. Dérivation formelle

Les travaux appartenant à cette catégorie visent à réaliser un SMA en se basant sur une spécification donnée. La dérivation peut être considéré comme une forme de transformation modèle-au-code. Vasconcelos, W., et al **[132]** ont présenté comment un SMA codé en utilisant la programmation logique est inféré à partir d'une spécification puis transformé en suite vers un système adapté aux besoins des utilisateurs.

B. Intégration avec une méthodologie existante

Les travaux appartenant à cette catégorie visent à améliorer des méthodologies qui existent déjà en introduisant des techniques formelles. Formal-Tropos (Sous-section 2.2.1) fait partie de cette catégorie.

C. Proposition d'une nouvelle méthodologie

Autres chercheurs ont choisi de proposer de nouvelles méthodologies formelles. La méthodologie ForMAAD (Sous-section 2.2.2) et le travail de Ball (Sous-section 2.2.3) font partie de cette catégorie.

3.2 Méthodologies formelles de développement des SMA

3.2.1 Formal Tropos

A. Brève description

Formal Tropos[1] **[133]** est une extension de la méthodologie Tropos **[14, 15]** par une spécification formelle des exigences. Cette extension fournit un langage de spécification qui offre les concepts primitifs de la spécification des besoins initiaux de i* **[48]** (acteur, but, dépendance stratégique) et les complète par un langage de spécification temporel inspiré du projet KAOS **[134]**. La notation i* permet la description des aspects structurels du modèle des exigences précoces tels que le réseau de relations et de dépendances entre les acteurs. Formal Tropos permet de représenter également les aspects dynamiques du modèle en décrivant, par exemple, comment le réseau de relations évolue au fil du temps. La représentation et l'analyse de ces aspects dynamiques permettent une compréhension plus précise du modèle des exigences précoces et révèlent des lacunes et des incohérences qui ne sont en aucun cas triviales à découvrir sans l'aide d'outils d'analyse formels. Afin de supporter l'analyse automatisée des spécifications de Formal Tropos, une technique de vérification est proposée et implémentée dans l'outil T-Tool[2] qui est basé sur le vérificateur de modèle symbolique NuSMV **[135]**.

B. Processus de développement

Le Processus de développement de Tropos formelle se compose de trois étapes essentielles : L'étape de spécification, l'étape de translation et l'étape de vérification. Dans la première étape, l'analyste spécifie formellement les exigences en utilisant le langage de spécification défini par *Tropos formelle*. Ensuite, une translation systématique de la spécification formelle est réalisée grâce à l'outil T-tool[1] vers un langage intermédiaire ayant une sémantique claire, indépendant de la spécification de Tropos formelle et indépendant de toute technique particulière d'analyse. Enfin, une version améliorée du vérificateur de modèle NuSMV effectue une analyse formelle sur la spécification du langage intermédiaire.

C. Utilisation de méthodes formelles

Formal Tropos utilise un langage de spécification formelle des exigences. Cette spécification formelle est exploitée ensuite pour effectuer *une vérification de cohérence* (Est-ce que la spécification admet des scénarios valides), *une validation d'assertion* (Est-ce que

[1] http://disi.unitn.it/~ft/
[2] http://disi.unitn.it/~ft/ft_tool.html

tous les scénario du système respectent certaines propriétés d'assertion), *une vérification de possibilité* (Est-ce qu'il y a un scénario pour le système qui respecte certaines propriétés de possibilité) et une *animation* permettant aux utilisateurs d'explorer interactivement les scénario valides du système.

3.2.2 ForMAAD

A. Brève description

ForMAAD **[136]** est une méthodologie formelle qui utilise le langage formel $\boldsymbol{T_{emporal}Z}$ pour le développement des systèmes multi-agents à travers sept (7) étapes de raffinement en couvrant la description de l'aspect structurel ainsi que l'aspect comportemental du SMA. ForMAAD est supportée par des extensions apportées à StarUML[1].

B. Processus de développement

La méthodologie ForMAAD suit un processus de développement composé de deux (2) phases principales : *La phase de spécification* et *la phase de conception*. Dans la première phase, les exigences des utilisateurs qui correspondent, dans une société d'agents, à un objectif commun (le but que les agents doivent accomplissent) sont décrites d'une façon abstraite. Cet objectif commun est décomposé en sous-buts (buts locaux). La spécification de l'environnement (composé généralement d'un espace de travail et d'entités passives et éventuellement des entités actives -agents-) dans lequel les agents évoluent est aussi possible. La deuxième phase est basée sur un ensemble de raffinements successifs du comportement collectif ainsi que l'individuel (Figure 2.7) pour produire une spécification plus détaillée du SMA.

[1] http://staruml.io

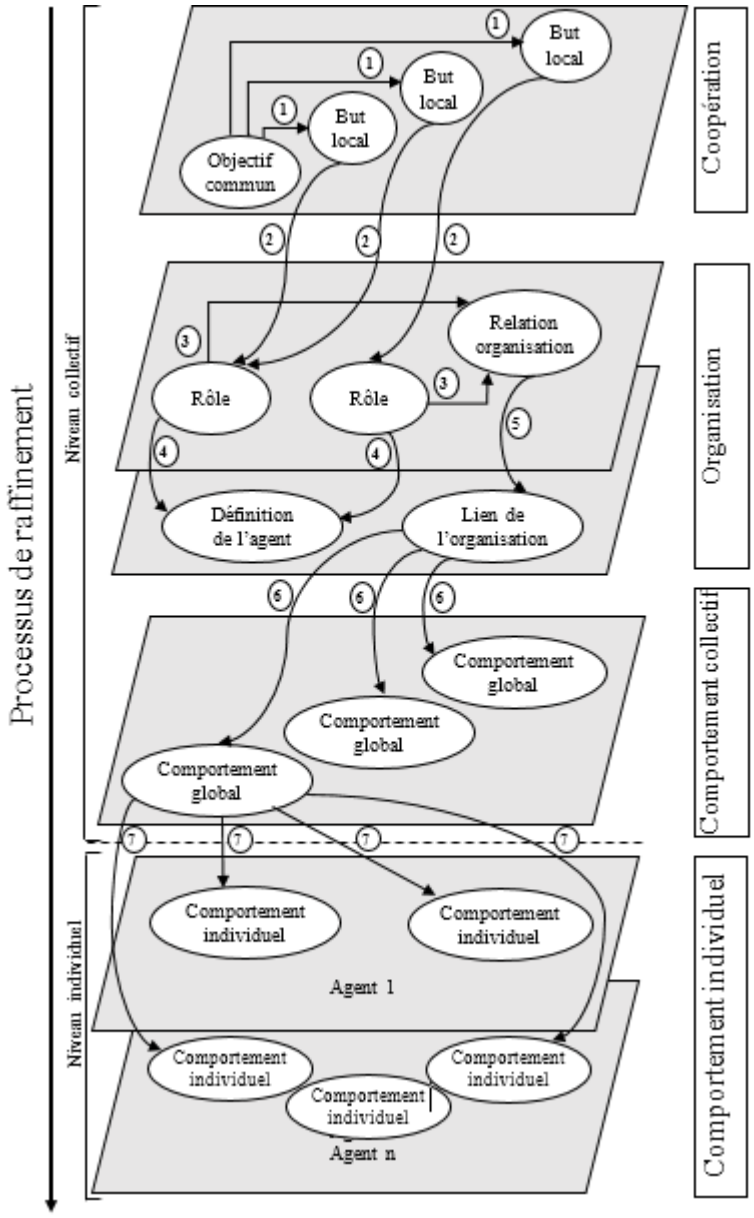

Figure 2.7 : Les étapes de raffinements de la phase de conception de ForMAAD **[137]**.

La vérification de la spécification résultante satisfait aux exigences est considérée comme une tâche essentielle réalisée progressivement pendant les étapes de raffinement.

C. Utilisation de méthodes formelles

Le langage formel utilisé par la méthodologie ForMAAD pour spécifier les systèmes multi-agents est le langage ***Temporal*Z.** Ce dernier intègre la logique temporelle linéaire dans la notation Z. La notation est un langage de spécification formel orienté-modèle (la description d'une application se fait en termes d'états et d'opérations sur eux définis dans des modules mathématiques) basé sur la théorie d'ensembles et la logique de prédicats du premier ordre. L'intégration de concepts de la logique temporelle linéaire dans la notation Z

a permis d'exploiter les outils accompagnant la notation Z, comme Z/EVES **[127]** pour la vérification des preuves d'obligation de chaque étape de raffinement.

3.2.3 Processus incrémental de développement des SMA en Event-B

A. Brève description

PID-SMA-B (Processus incrémental de développement de SMA en Event-B) est proposée par Elisabeth Ball **[138, 139]** combine l'utilisation de modèles informels basés sur des concepts centraux aux systèmes multi-agents (buts, interaction d'agents et rôle) avec autres modèles formels basée sur la méthode formelle Event-B **[60]**.

B. Processus de développement

Le Processus de développement est divisé en deux étapes. Le but de la première étape est de modéliser le système comme un ensemble de buts décrivant les interactions du système (Elaboration partielle du diagramme de buts). Les modèles Event-B sont créés en analysant les buts et les relations utilisés pendant la construction des modèles de buts du système. Dans cette étape, la perspective des modèles Event-B est d'un seul système d'agents qui change l'état lorsque les agents interagissent. La deuxième étape prend le diagramme de buts et les modèles Event-B développés dans la première étape et les raffine pour modéliser les agents interagissant ainsi que l'environnement. Les modèles Event-B sont ensuite raffinés pour intégrer les rôles et les communications des agents. Ils sont ensuite raffinés afin d'introduire les ressources pour le système, puis le modèle du système est décomposé en rôles d'agents synchronisés et modèles de ressources. Des conseils (guide) pour simplifier la transformation de diagrammes de buts en modèles formels sont fournis. Le guide a pour but de faciliter la décharge des obligations de preuve et d'exploiter les prouveurs automatiques de la boîte à outils RODIN[1].

C. Utilisation de méthodes formelles

L'adoption des méthodes formelle dans cette méthodologie réside dans l'utilisation de la méthode bien connue Event-B. Event-B est une méthode formelle pour la modélisation et le raisonnement sur les systèmes réactifs et distribués.

[1] http://www.event-b.org/, https://www3.hhu.de/stups/handbook/rodin/

4. Conclusion

Nous avons vu dans ce chapitre les principes de bases relatifs à l'ingénierie dirigée par les modèles et aux méthodes formelles. Un ensemble de méthodologies utilisant des techniques intervenant de l'ingénierie dirigée par les modèles et de méthodes formelles ont été décrites. Les méthodologies Tropos, Prometheus, ADELFE, INGENIAS et ForMAAD utilisent la technique de transformation de modèles dans leurs processus de développement, ce qui facilite la tâche pour les développeurs et vise à automatiser le processus de développement. A l'opposé de ces dernières méthodologies qui utilisent des formalismes semi-formels durant le cycle de développement, ForMAAD, Formal Tropos et PID-SMA-EB utilisent des formalismes formels pour éliminer les éventuelles. Certaines de ces méthodologies couvrent la plupart des phases du cycle de développement telles que ASEME et TROPOS. En effet, l'intégration de techniques d'IDM et de méthodes formelles est très utile pour assurer la qualité des SMA produits au bout du cycle de développement. Dans le chapitre suivant, nous allons présenter et détailler la méthodologie PASSI, la méthodologie qui représente la brique de base de cette thèse.

Chapitre 03/ La Méthodologie PASSI

---

1. Introduction

Plusieurs méthodologies de développement des SMA ont été proposées dans la littérature. L'une des méthodologies couvrant la plupart du cycle de développement et utilisant un ensemble important de standards tels que (A)UML, RDF[1], etc.., c'est la méthodologie PASSI. PASSI fait l'objet de plusieurs articles dans lesquels les auteurs ont essayé de l'utiliser pour la conception des SMA dans des domaines différents **[140, 141]** et de l'étendre pour qu'elle soit utilisée dans d'autres domaines **[142]**.

Le reste de ce chapitre est organisé comme suit : Tout d'abord, un petit aperçu sur la méthodologie PASSI fait l'objet de la deuxième section. Dans la section 3, le méta-modèle de SMA selon PASSI est expliqué. Dans la 4ème section, nous allons détailler le processus de développement de PASSI. La section 5 est consacré à la démonstration de l'outil PTK supportant PASSI. Ensuite, la traçabilité dans PASSI est discutée dans la 6ème section. La septième section est consacrée à la comparaison entre Gaia, les méthodologies décrites dans le deuxième chapitre et PASSI. Enfin, nous conclurons ce chapitre dans la section 8.

2. Le méthodologie PASSI

PASSI[2] **[16, 17]** (Acronyme de *Process for Agent Societies Specification and Implementation*) est une méthodologie pas à pas besoin-vers-code (*Requirement-to-code*) pour la conception et le développement des SMA. PASSI intègre des concepts du génie logiciel orienté objet et des approches de l'intelligence artificielle en utilisant la notation bien standardisée UML. Elle se réfère aux standards les plus diffusés comme (A)UML, FIPA, Java, RDF. PASSI se compose d'un processus de conception incrémental et itératif couvrant la plupart des phases du cycle de vie de développement et un langage de modélisation étendu d'UML. La réutilisation est un concept très important dans PASSI, elle est effectuée à travers les patrons de conception et supportée par un outil : PTK (PASSI ToolKit) **[18]**.

3. Le méta-modèle de SMA adopté dans PASSI

Les méta-modèles adoptés par les méthodologies de développement des SMA offrent un langage de haut niveau pour le développement des SMA en termes de concepts de la technologie agent. Le méta-modèle adopté dans PASSI est divisé en trois domaines **[17]**.

[1] https://www.w3.org/RDF/
[2] http://www.pa.icar.cnr.it/passi/Passi/PassiIndex.html

3.1 Domaine du problème

Cette partie du domaine (Figure 3.1) contient les composants décrivant les besoins que le SMA en cours de développement doit les accomplir ensuite comme, par exemple, agent, tâche, plan, les messages échangés entre deux rôles (*Message_Rôle-Rôle*). Ces composants sont reliés directement aux phases du modèle des besoins du système de PASSI (section 4.1).

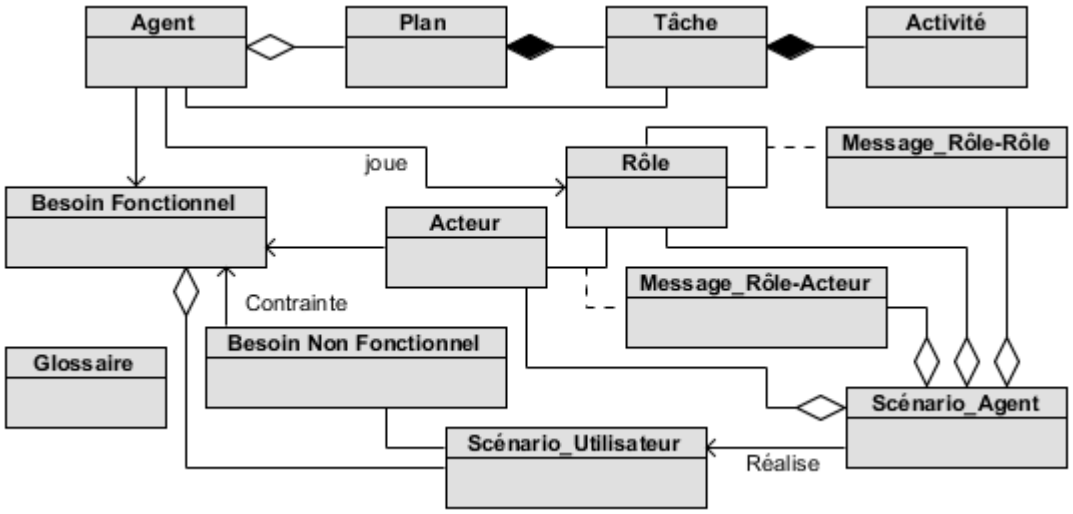

Figure 3.1 : Méta-modèle d'un SMA selon PASSI (La partie « domaine du problème ») **[17]**.

3.2 Domaine de l'agence

Cette partie du méta-modèle (Figure 3.2) représente les composants utilisés pour définir une solution du problème comme, par exemple, la communication, les éléments de l'ontologie (*ElementOntologie*), les protocoles d'interaction (*Protocol-Interaction-Agent)*. Ces composants sont reliés au modèle de la société d'agents (sous-section 4.2).

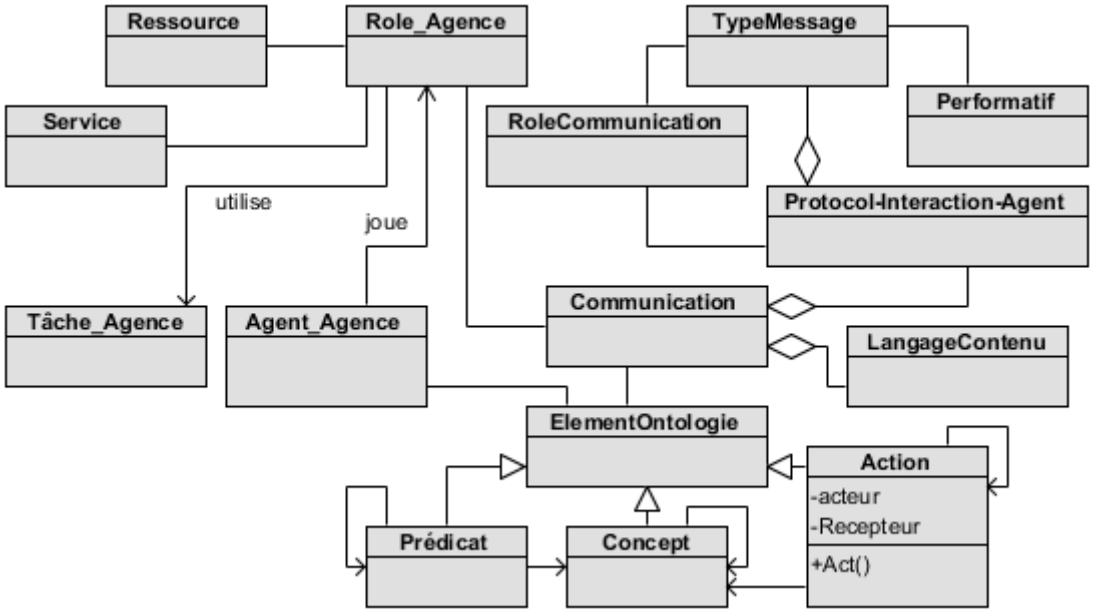

Figure 3.2 : Méta-modèle d'un SMA selon PASSI (La partie « domaine de l'agence ») **[17]**.

3.3 Domaine de la solution

Finalement, les composants du domaine de l'agence sont mappés aux éléments de la plateforme d'implémentation (conforme au FIPA) adoptée. Il représente le niveau code de la solution et l'étape finale de raffinement (Figure 3.3).

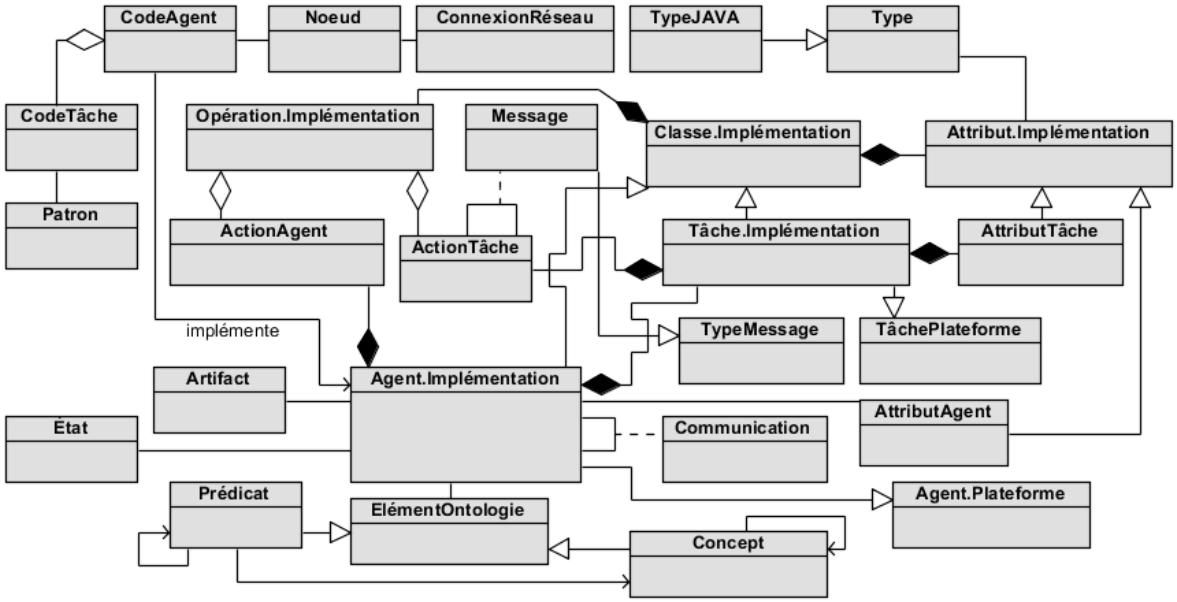

Figure 3.3 : Méta-modèle d'un SMA selon PASSI (La partie « domaine de la solution ») **[17]**.

4. Le processus PASSI

Le processus de conception PASSI est incrémental et itératif composé de cinq (5) modèles et une activité de test. La figure suivante montre le processus de conception PASSI. Chaque modèle se compose d'une phase ou plus.

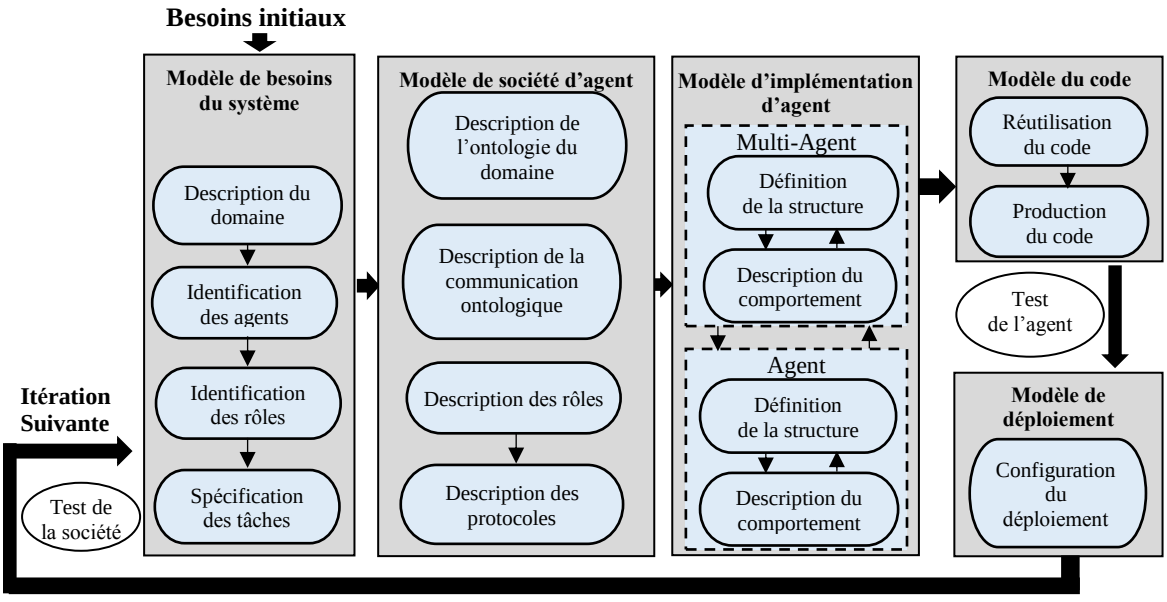

Figure 3.4 : Le processus de développement PASSI **[16, 17]**.

4.1 Modèle des besoins du système

C'est le modèle responsable de l'identification des besoins du système en fonction de fonctionnalités, agents, rôles et de tâches. Il se compose de quatre (4) phases :

4.1.1 Description du domaine (DD)

Durant cette première phase, les besoins fonctionnels du système sont capturés à travers un diagramme des cas d'utilisation. Chaque besoin fonctionnel est représenté par un cas d'utilisation. La figure suivante montre une partie d'un simple exemple d'un diagramme de description du domaine.

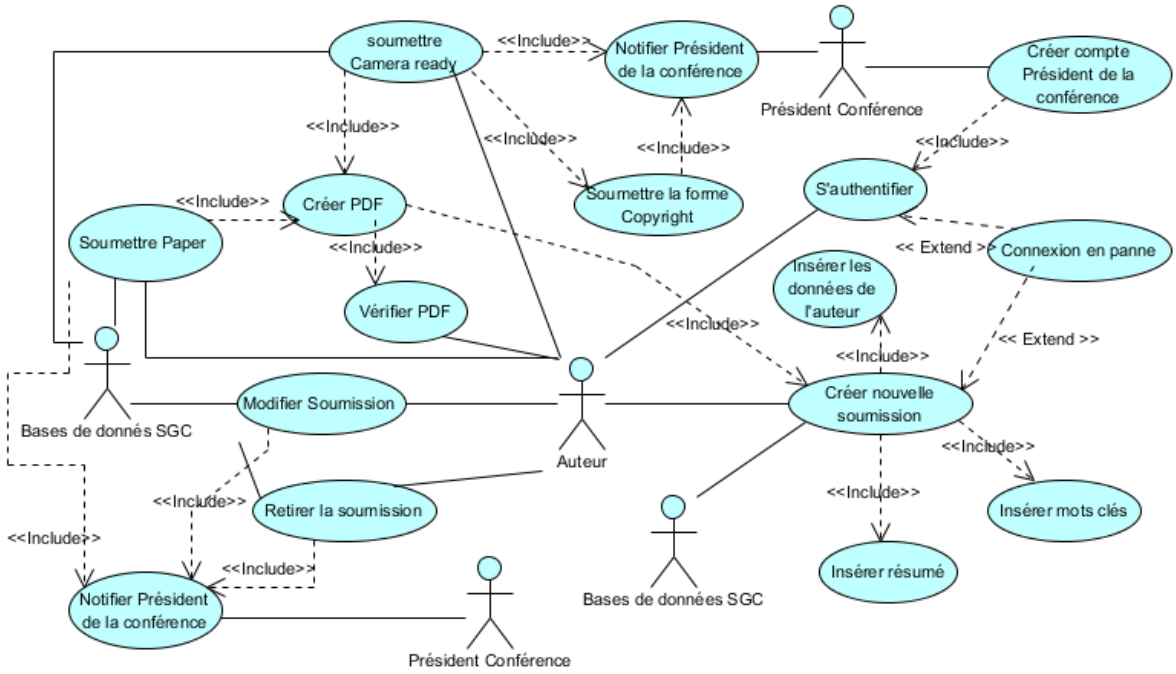

Figure 3.5 : Exemple d'un diagramme de description du domaine **[17]**.

4.1.2 Identification des agents (IA)

Dans cette phase, les besoins identifiés dans la phase précédente sont regroupés en packages. Chaque package représente un agent et les besoins qui le composent représentent ses fonctionnalités. La figure 3.6 montre une partie d'un exemple de diagramme d'identification d'agents. Le diagramme identifie trois agents : « *GestionnaireSoumission* », «*GestionnaireFichiers* » et « *GestionnaireCompte* ».

Comme illustrés dans la figure 3.6, les relations entre les cas d'utilisation du même agent suivent la syntaxe et les stéréotypes d'UML usuelles. Par contre, les relations entre les cas d'utilisation de différents agents sont stéréotypées par *« communicate »*. Les relations de communication (relations stéréotypées par *« communicate »*) entre agents sont dirigées de l'initiateur vers le participant **[17]**.

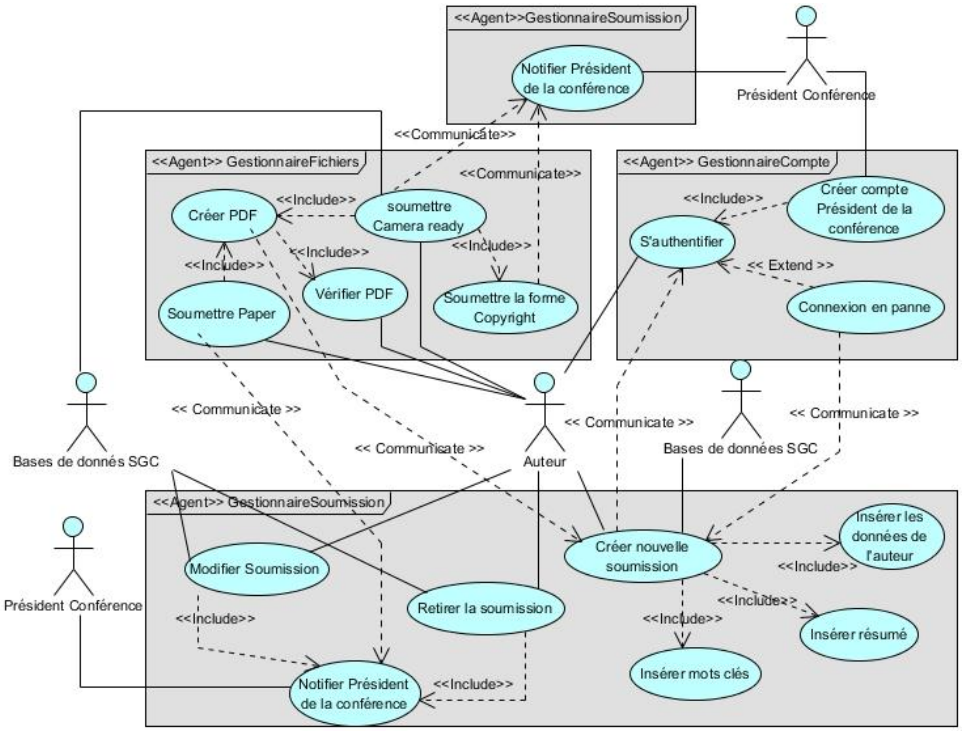

Figure 3.6 : Exemple d'un diagramme d'identification des agents **[17]**.

4.1.3 Identification des rôles (IR)

Les rôles qui seront joués dans des différents scénarios par les agents composant le système sont identifiés à travers plusieurs diagrammes de séquence, un scénario par un diagramme de séquence. Chaque ligne de vie représente un rôle en suivant la syntaxe suivante : *<Rôle> : <Nom_Agent>*. Dans le scénario décrit par le diagramme illustré dans la figure 3.7, l'agent *« GestionnaireFichiers »* joue les quatre (04) rôles suivants : *«InterfaceGraphique»*, *«GestionnairePapier»*, *«GestionnairePDF»* et *«InterfaceBDD»*, l'agent *«GestionnaireSoumission»* joue un seul rôle, *«Notificateurr»*.

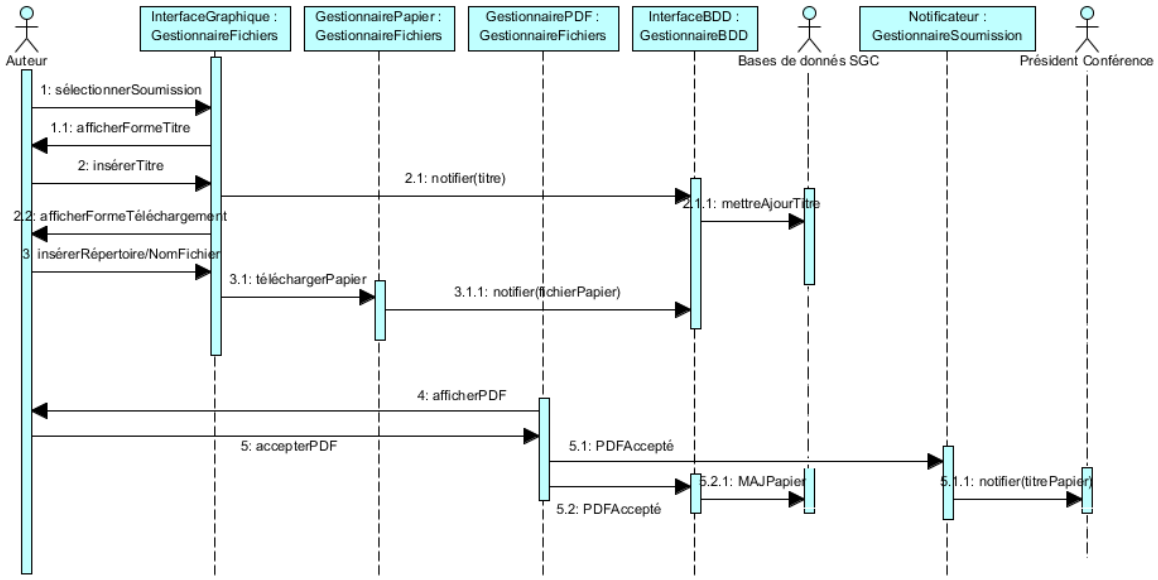

Figure 3.7 : Exemple d'un diagramme d'identification des rôles (un scénario) **[17]**.

4.1.4 Spécification des tâches (ST)

Dans cette phase, l'accent est mis sur chaque comportement d'agent pour concevoir un plan qui pourrait satisfaire les besoins de l'agent en déléguant ses fonctionnalités à un ensemble de tâches. Pour chaque agent, un diagramme d'activité en deux couloirs est conçu. Un des couloirs contient un ensemble d'activités symbolisant les tâches de l'agent en cours, alors que l'autre couloir contient quelques activités représentant les autres agents interagissant avec l'agent en cours. Les relations entre les activités signifient soit des messages entres les taches et les agents interagissant ou bien des communications entre les tâches du même agent. La figure suivante montre un exemple d'un diagramme de spécification de tâches.

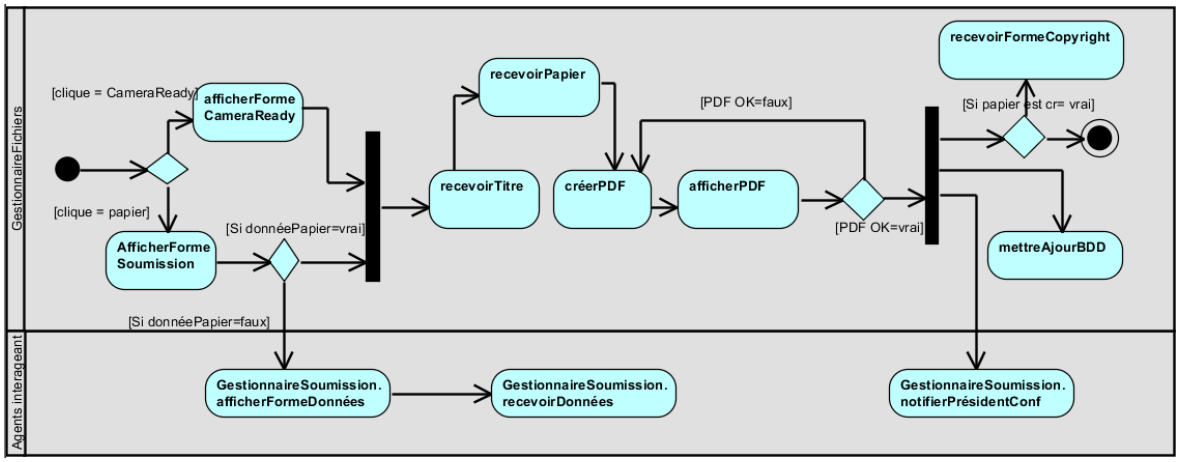

Figure 3.8 : Exemple d'un diagramme de spécification des tâches **[17]**.

4.2 Modèle de la société d'agents

Dans ce modèle, le système est vu comme une société d'agents. Il se compose de quatre (04) phases :

4.2.1 Description de l'ontologie du domaine (DOD)

Durant cette phase, une représentation ontologique du domaine en termes de concepts, de prédicats et d'actions est spécifiée en utilisant un diagramme de classes. Cette représentation offre une formalisation de l'environnement où les agents vivent ainsi que les connaissances attribuées aux agents avec leurs communications. La figure 3.9 montre un exemple d'un diagramme de description de l'ontologie du domaine.

Les éléments de l'ontologie (concepts, prédicats et des actions) sont reliés en utilisant les trois relations standards d'UML : La généralisation, l'agrégation et l'association **[17].**

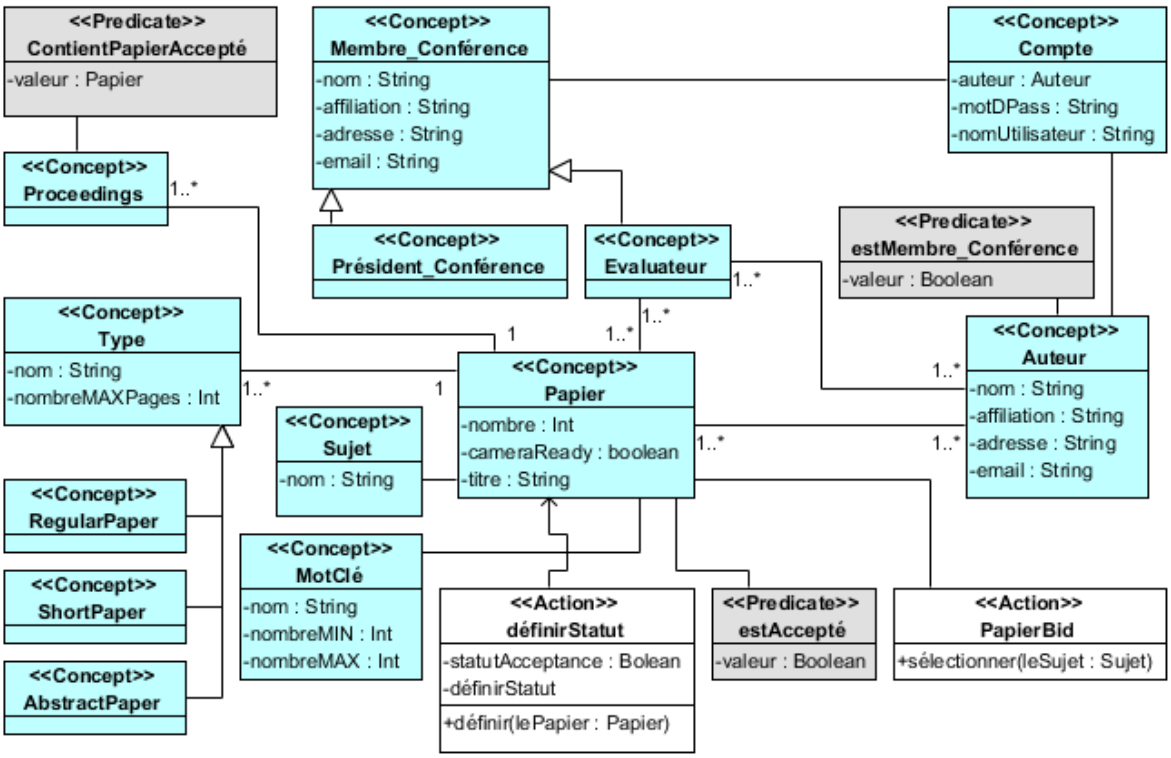

Figure 3.9 : Exemple d'un diagramme de description de l'ontologie du domaine **[17].**

4.2.2 Description ontologique de communications (DOC)

Cette phase vise la description de toutes les interactions entre les agents via un diagramme de classes. Chaque agent est représenté par une classe et chaque interaction entre deux agents est représentée par une association entre leurs classes. Selon les standards de la FIPA, une communication est constituée par des actes de communication qui sont regroupés dans des protocoles d'interaction qui définissent la séquence de messages attendus. Pour

ceci, chaque communication est caractérisée par trois attributs groupés dans une classe d'association. Ces attributs sont : l'ontologie, langage de contenu et le protocole d'interaction. Les rôles joués par les agents lors d'une communication sont rapportés au début et à la fin de la ligne de l'association. Dans l'exemple montré dans la figure 3.10, l'agent « GestionnaireSoumission » (en jouant le rôle « *InterfaceAuteur* ») commence une communication avec l'agent « GestionnaireCompte » (en jouant le rôle « gestionnaire*Erreur* »). Les concepts de l'ontologie utilisés lors de cette communication sont : *Auteur* et *Compte,* le langage utilisé est le langage bien standardisé RDF et le protocole suivi est le protocole standardisé par la FIPA, *Inform*.

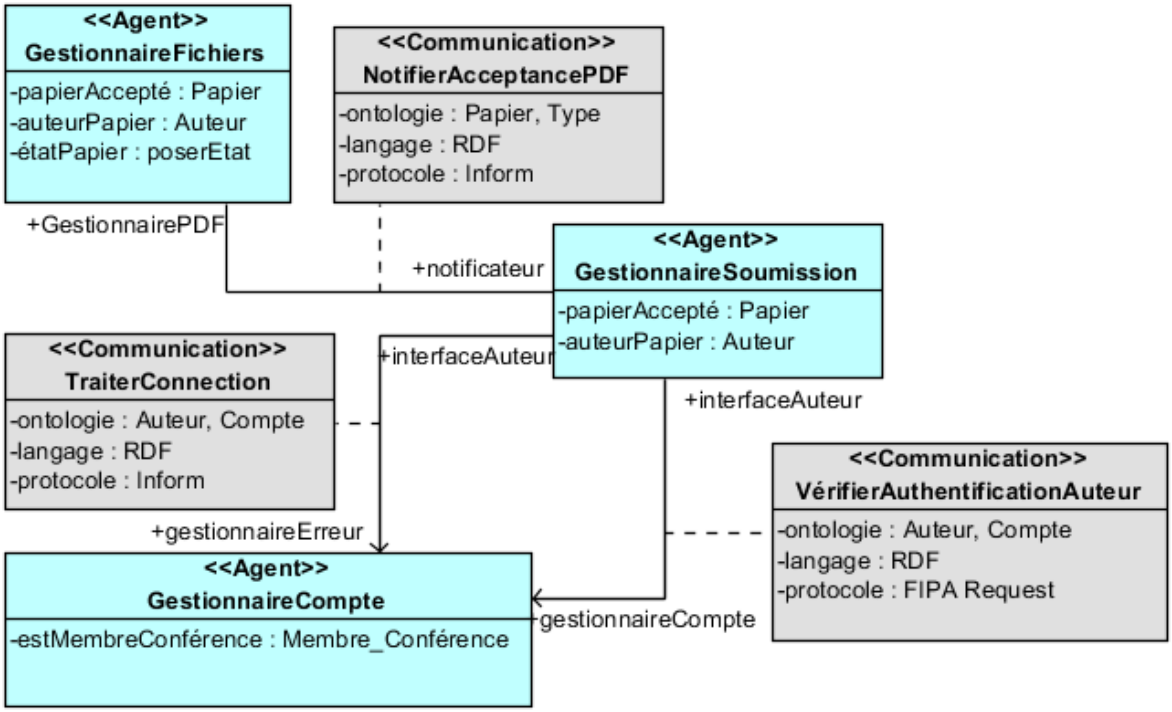

Figure 3.10 : Exemple d'un diagramme de description ontologique de communication **[17]**.

4.2.3 Description de rôles (DR)

Les rôles identifiés précédemment dans la phase d'identification de rôles sont décrits en plus de détail dans cette phase en fonction de tâches qui les composent. Les rôles sont décrits via un diagramme de paquetages. Chaque rôle est représenté par une classe indépendante où chacune de ses opérations représente une tâche. Les classes représentant les rôles du même agent sont regroupées dans un paquetage. Les rôles peuvent être reliés par une des relations suivantes : [ROLE_CHANGE], [SERVICE_DEPENDENCY], [RESSOURCE_AVAILA-BILITY] ou [COMMUNICATION_AVAILABILITY]. Dans l'exemple illustré dans la figure 3.11, le rôle « *InterfaceAuteur* » (de l'agent : *GestionnaireSoumission*) dépend du rôle «*GestionnaireCompte* » pour lui générer un mot de passe pour un tel auteur. La relation

[ROLE_CHANGE] donne un aperçu sur le cycle de vie des agents en fonction de leurs rôles joués.

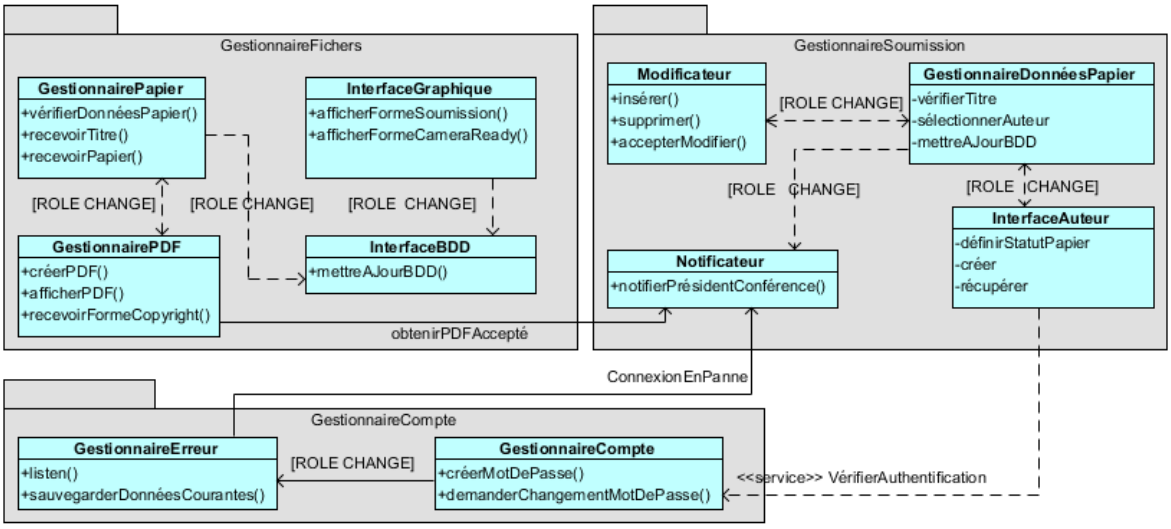

Figure 3.11 : Exemple d'un diagramme de description de rôles **[17]**.

4.2.4 Description de protocoles (DP)

Cette phase vise à décrire les protocoles de communication rapportés dans le diagramme de description ontologique de communication s'ils ne sont pas standardisés par la FIPA. Chaque protocole non standard est décrit par un diagramme de séquence AUML.

4.3 Modèle d'implémentation des agents

Ce modèle offre l'architecture de la solution en termes de classes et de méthodes dans deux niveaux d'abstraction : Agent et multi-agents. Il se compose de quatre (04) phases réparties en deux niveaux d'abstraction.

4.3.1 La définition de la structure du système multi-agents (DSSMA)

La structure générale du système en cours de développement est spécifiée par un diagramme de classes dans cette phase. Chaque agent est représenté par une classe où les méthodes représentent ses tâches. Les connaissances (pièces de l'ontologie du domaine) de chaque agent sont spécifiées comme des attributs. Les liens entre les agents signifient des communications entre eux. La figure 3.12 montre un exemple d'un diagramme produit à la fin de cette phase.

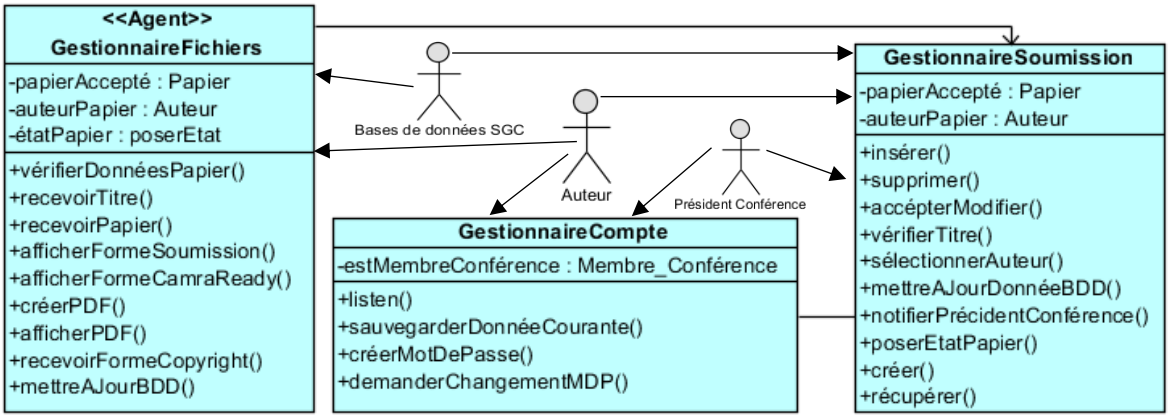

Figure 3.12 : Exemple d'un diagramme de DSSMA **[17]**.

4.3.2 La description du comportement du système multi-agents (DCSMA)

Cette phase vise la description du comportement collectif du système en termes de tâches, des actions et les messages échangés entre les agents. La description est effectuée grâce à un ou plusieurs diagrammes d'activité en couloirs. Chaque couloir est consacré à une tâche d'un agent (Un agent est représenté par plusieurs couloirs représentants leurs tâches). La figure 3.13 illustre une partie d'un diagramme DCSMA.

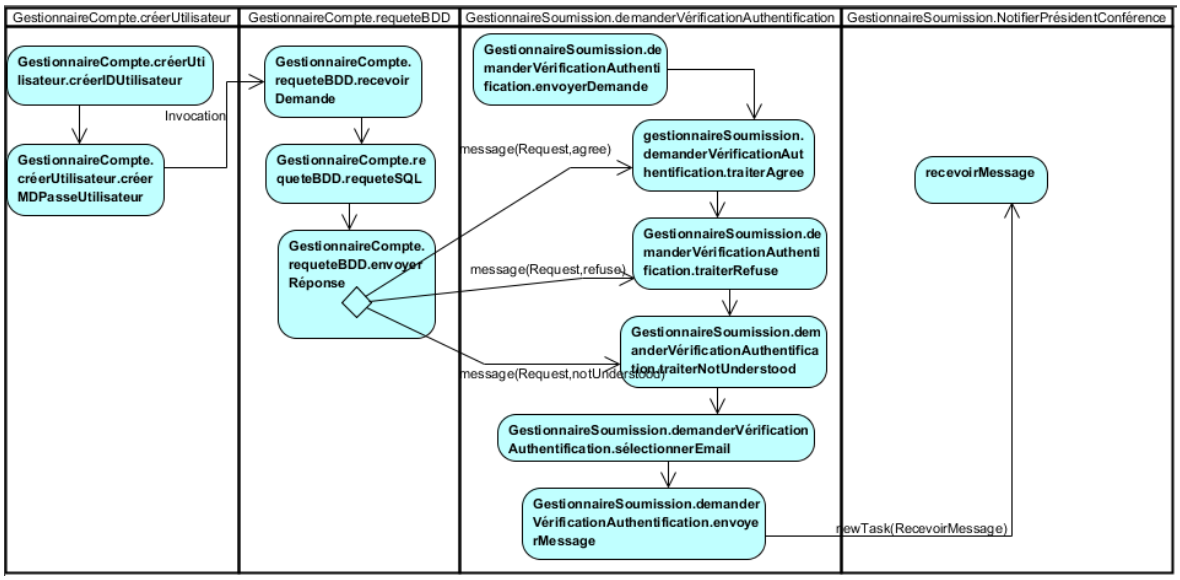

Figure 3.13 : Une partie d'un diagramme de DCSMA **[17]**.

4.3.3 La définition de la structure des agents (DSA)

Dans cette phase, les structures internes des agents sont décrites avec plus de détails à travers la description détaillée de leurs tâches. Un diagramme de classes est produit pour chaque agent. Une classe est consacrée à l'agent lui-même où ses connaissances (pièces de l'ontologie du domaine) peuvent être spécifiées par des attributs. Chacune des classes restantes représente une tâche particulière où les opérations représentent les actions qui la composent.

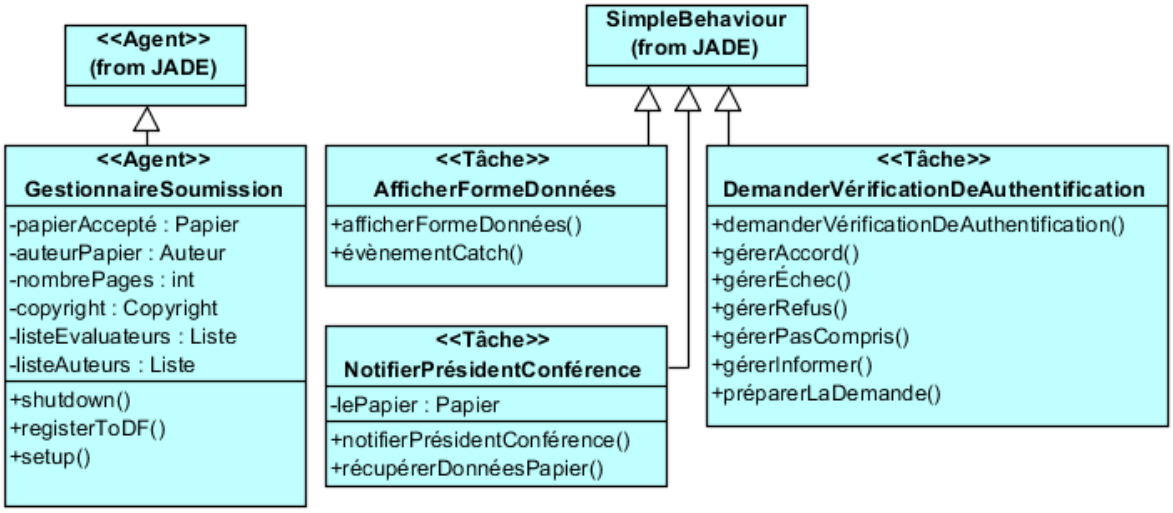

Figure 3.14 : Une partie d'un diagramme de DSA **[17]**.

La figure 3.14 montre une partie du diagramme de DSA de l'agent « Gestionnaire*Soumission* » où la plateforme choisie était Jade. Les trois tâches héritent de la classe « *SimpleBehaviour* » de Jade.

4.3.4 La description du comportement individuel des agents (DCA)

Dans cette phase, le comportement individuel de chaque agent/tâche est décrit par un diagramme d'états-transitions ou bien un diagramme d'activité. La figure suivante montre un exemple très simple d'un diagramme de DCA.

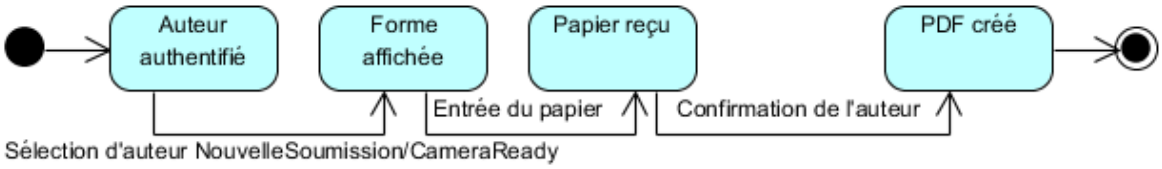

Figure 3.15 : Une partie d'un diagramme de DCA **[17]**.

4.4 Modèle du code

C'est le module de codage offrant la solution au niveau plus concret. Il se compose de deux (2) phases.

4.4.1 Réutilisation du code (RC)

Durant cette phase, le développeur peut réutiliser les patrons de conception prédéfinis. Ces patrons de conception sont constitués de diagrammes et du code correspondant. L'intégration des patrons de conception est supposée être effectuée par l'outil PTK (Section 5) et l'application AgentFactory[1]. Les patrons sont classés selon deux critères : *Le contexte d'application* et *la fonctionnalité* **[143]**. Le premier critère détermine si le patron de

[1] http://www.pa.icar.cnr.it/passi/AgentFactory/AgentFactoryIndex.html

conception est appliqué sur un agent, plusieurs agents ou bien sur les éléments qui le composent (Action, Comportement, Composant et Service). Le deuxième critère exprime la fonctionnalité du patron (Accès aux ressources locales, communication, élaboration et mobilité). La table 3.1 montre les patrons de conception prédéfinis de PASSI en les classant selon les deux critères.

		Contexte d'application			
		Action	***Comportement***	***Composant***	***Service***
Fonctionnalité	***Accès aux ressources locales***	53		*Generic Agent, Parallel Resource Sharing, Sequential Resource Sharing, PublishSubscribe, Resource Caching*	
	Communication	66	*Request (I/P), Query (I/P), Inform (I/P), ContractNet (I/P)*		*Request, Query, Inform, ContractNet*
	Elaboration	44		*Planner*	
	Mobilité	7			*Explorer*

Table 3.1 : Classification de patrons de conception prédéfinis de PASSI. (I/P) indique la présence des patrons pour les deux rôles : Initiateur et Participant. Les patrons de la catégorie « Action » ne sont pas mentionnés par nom à cause de leur nombre **[143]**.

4.4.2 Production du code (PC)

Après la réutilisation éventuelle des patrons de conception prédéfinis, le code de la solution est généré par l'outil PTK (Section 5). Le programmeur doit compléter le corps de méthodes vides.

4.5 Modèle de déploiement

Il est composé d'une seule phase.

4.5.1 La configuration de déploiement (CD)

Un diagramme de déploiement est utilisé pour décrire l'allocation des agents aux différentes unités de traitement avec toute contrainte de migration ou de mobilité. La figure 3.16 montre un exemple simple d'un diagramme de déploiement.

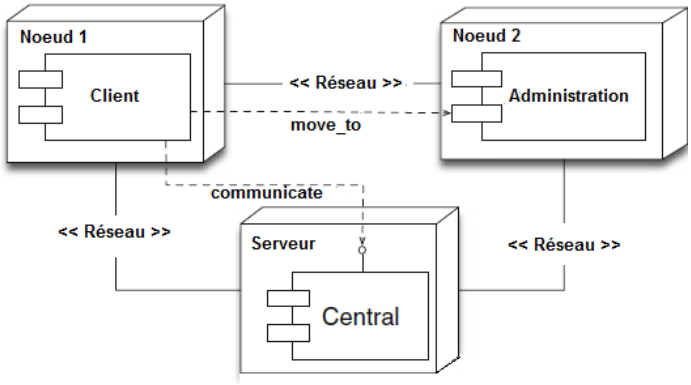

Figure 3.16 : Un exemple simple d'un diagramme de configuration de déploiement **[17]**.

4.6 Activité de test

L'activité de test dans PASSI est divisée en deux niveaux : *Test de l'agent* où un framework construit sur Jade est implémenté **[18]**. Les classes les plus importantes de ce framework sont : « *Test* » pour tester une tâche spécifique d'un agent et « *TestGroup* » pour tester toutes les tâches d'un agent spécifique. Le deuxième niveau est le *test de la société d'agents* où un test d'intégration est effectué pour valider les résultats de l'itération courante **[17]**.

5. PASSI ToolKit (PTK)

La méthodologie PASSI est supportée par un outil appelé PTK[1] (Acronyme de PASSI Toolkit). L'outil est offert sous format d'un plugin intégrable dans l'environnement de modélisation IBM Rational Rose[2]. Il guide les développeurs tout au long des phases de développement de PASSI en vérifiant la consistance des digrammes conçus et des rapports textuels sont générés. Les développeurs doivent lire les rapports générés pour corriger éventuellement les erreurs. PTK offre la possibilité de générer automatiquement un squelette de code à partir les diagrammes PASSI. PTK est disponible et téléchargeable depuis le site *sourceforge*[1]. La figure 3.17 montre le menu principal du plugin.

[1] https://sourceforge.net/projects/ptk/
[2] http://www-03.ibm.com/software/products/fr/enterprise

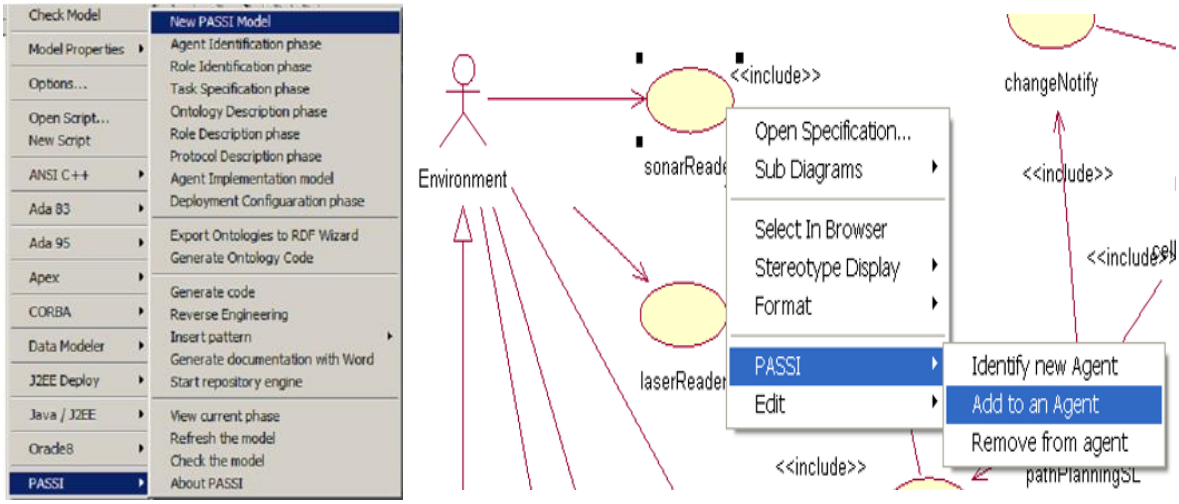

Figure 3.17 : Menu principal de l'outil PTK.

6. La traçabilité dans PASSI

Dans le chapitre décrivant la méthodologie PASSI **[17]**, les auteurs ont mentionné les dépendances entre les différents éléments de conception qui sont produits tout au long du processus de conception de PASSI et comment les diagrammes sont construits. Les dépendances représentent des liens implicites entre les éléments de conception, par exemple, les liens qui existent entre les éléments partageant les mêmes noms tels que *Tâche* (Domaine du problème, figure 3.1), *Tâche_Agence* (Domaine d'agence, figure 3.2) et *Tâche _Implémentation* (Domaine de solution, figure 3.3). Un méta-modèle est proposé dans la section 2 du cinquième chapitre pour mettre des liens de traçabilité bidirectionnels explicitement.

7. Comparaison

Dans cette section, nous résumons par une comparaison et nous discutons les méthodologies décrites précédement. La comparaison est basée sur un ensemble de critères proposés déjà dans la littérature et autres proposés par nous-même. En effet, plusieurs travaux de comparaison et d'évaluation des méthodologies de développement des SMA ont été publiés. La table 3.2 illustre les travaux (cités par ordre chronologique) que nous avons capturé de la littérature, les méthodologies évaluées ou comparées ainsi que les critères adoptés. Certains auteurs ont classifié les critères de comparaison et d'évaluation en dimensions tels que *Sabas, A., et al.* **[144]** (Méthodologie, représentation, agent, organisation, coopération, technologie), en aspects tels que *Dam, K. H. and M. Winikoff* **[145]** (Concepts, langage de modélisation, processus et pragmatique).

Travail	Les méthodologies comparées/évaluées	Critères adoptés																								
		Couverture du cycle de développement	Concepts adoptés	Notation (Formalisme)	Réutilisation	Outillage	Documentation	Complexité de compréhension/utilisation	Restriction aux utilisateurs	Domaines d'application	L'architecture de l'agent	Traçabilité	Raffinement	Scalabilité	Evolution	Expressivité	Processus de développement	Consistance	Couverture de l'aspect I/C	Implication Utilisateur	Modèle de développement	Approche de développement	Utilisation des standards	Plateformes ciblés	Auto-organisation	Gestion de la complexité
Sabas et al. *(2002)* **[144]**	MMTS **[146]**, AODSRT **[147]**, AOMEM **[148]**, HLIM **[149],** MAS-CommonKADS **[150]**, Gaia **[11,12]**, MASB **[151]**, MaSE **[152]**, CoMoMAS **[153]**	×	×	×	×		×													×	×	×				×
Tran, Q.-N. N., et al. (2003) **[154]**	/	×	×	×	×			×		×				×				×				×				
Dam, K. H. and M. Winikoff (2004) **[145]**	MaSE **[152]**, Prometheus **[49, 50]** Tropos **[14, 15]**	×	×	×	×			×				×	×	×				×								
Sturm, A. and O. Shehory (2004) **[155]**	Gaia **[11, 12]**	×	×				×	×		×	×			×		×										
Tran, Q.-N. N., et al. *(2005)* **[156]**	MaSE **[152]**, Gaia **[11, 12]**, MMTS **[146]**,	×	×			×	×	×		×	×		×	×		×		×			×					×
Sudeikat, J., et al *(2005)* **[157]**	MaSE **[152]**, Tropos **[14, 15]**, Prometheus **[49, 50]**	×	×	×		×	×	×			×						×									×
Abdelaziz, T. H. S., et al. (2007) **[158]**	Gaia **[11, 12]**, MaSE **[152]**, HLIM **[149]**	×	×	×		×																				

Al-Hashel, E., et al. (2007) **[159]**	ROADMAP **[99]**, Prometheus **[49, 50]**, MaSE **[152]**	×	×	×	×	×		×				×						×							×
Cernuzzi, L. and F. Zambonelli *(2008)* **[160]**	MaSE **[152]**, Tropos **[14, 15]**	×		×		×	×	×			×		×	×		×		×							×
Elamy, A.-H. H. and B. Far (2008) **[161]**	Gaia **[11, 12],** MaSE **[152],** Tropos **[14, 15],** Agent-SE **[162],** MASSIVE **[163],** Prometheus **[49, 50],** MESSAGE **[93],** MAS-CommonKADS **[150],** PASSI **[16, 17]**	×		×	×					×		×	×			×		×							×
Yubo, J., et al. (2009) **[164]**	MaSE **[152]**, Gaia **[11, 12]**, Tropos **[14, 15].**	×	×	×		×	×	×		×		×		×		×		×							×
Garcia, E., et al. (2009) **[166]**	RT_MESSAGE **[165],** INGENIAS **[13],** ANEMONA **[167],** GORMAS **[168]**	×	×	×	×		×	×		×	×		×	×	×	×	×	×				×		×		×
Sturm, A. and O. Shehory (2014) **[169]**	Gaia **[11, 12],** INGENIAS **[13]**, MaSE **[152],** PASSI **[16, 17]**, Prometheus **[49, 50]**, Tropos **[14, 15].**	×	×	×		×	×	×		×	×			×		×									×

Table 3.2 : Travaux précédents d'évaluation et de comparaison des/entre méthodologies de développement des SMA.

Dans notre comparaison, nous adoptons un ensemble de critères proposés déjà dans la littérature en leur ajoutant d'autres critères relatifs à l'IDM et aux méthodes formelles. Nous classifions les critères adoptés en quatre (4) classes :

❖ **Méthodologie :** Cette classe se compose de 6 critères :

- ***Domaine d'application*** : Le(s) domaine(s) où la méthodologie est plus appropriée.
- ***Scalabilité*** : La capacité d'être utilisée pour le développement des SMA plus larges.
- ***Facilité d'utilisation*** : Est-ce que la méthodologie est facile à comprendre/utiliser ?
- ***Phases de développement*** : Quelles sont les phases du cycle de développement qui sont couvertes par la méthodologie : L'**A**nalyse, la **C**onception (**A**rchitecturale ou **D**étaillée), l'**I**mplémentation, la **v**alidation, la vérification, le **t**est, le **d**éploiement et la **m**aintenance.
- ***Support*** : La disponibilité de documentation nécessaire (site/page Web, études de cas, document explicatifs, guides) qui facilite l'utilisation de la méthodologie ainsi que la disponibilité des outils supportant la méthodologie.
- ***Evolution*** : Y a-t-il des extensions, améliorations de la méthodologie ?
- ***Utilisation des standards*** : Quelles sont les standards utilisés par la méthodologie.

❖ **Modélisation :** Cette classe se compose de 7 critères :

- ***Concepts*** : La méthodologie utilise-t-elle les concepts : Agent, rôle, tâche, acteur, but, plan, organisation, société, environnement, communication, interaction, message, service, mobilité, ressource, ontologie).
- ***Formalisme utilisé*** : Quels sont les formalismes de modélisation utilisés que ce soit semi-formels ou formels (UML, AUML, AML, etc.)
- ***Gestion de la complexité*** : Est-ce que le SMA est modélisé en différents niveaux d'abstractions ?
- ***Expressivité*** : La capacité de représenter les concepts qui se réfèrent à l'aspect structurel/comportemental de l'agent/SMA, Connaissances des agents.
- ***Raffinement*** : Est que le SMA est raffiné progressivement tout au long du cycle de développement ?
- ***Traçabilité*** : Est-ce qu'une traçabilité explicite est bien définie entre les différends diagrammes/modèles produits durant le cycle de développement de la méthodologie ?
- ***Assurance de la cohérence*** : Les développeurs peuvent-ils vérifier/assurer la cohérence entre les modèles conçus ?

La table 3.3 montre la comparaison entre les méthodologies décrites dans le deuxième chapitre et la méthodologie PASSI en se basant sur les critères cités ci-dessus.

	Critères relatifs à la méthodologie								Critères relatifs à la modélisation																						
Méthodologies	**Domaine d'application**	**Scalabilité**	**Facilité d'utilisation**	**Phases de développement**	**Support**		**Evolution**	**Standards utilisés**	**Concepts**																	**Formalisme**	**Gestion de complexité**	**Expressivité**		**Raffinement**	**Traçabilité**	**Assurance de la cohérence**
					Documentation	Outillage			Agent	Role	Tache	Acteur	But	Plan	Organisation	Société	Environnement	Communication	Interaction	Message	Service	Mobilité	Ressource	Ontologie			Aspect Individuel	Aspect Colleltif			
Gaia	/	M	M	A,CA, CD	M	F	M	/	✓	✓			✓		✓		✓		✓	✓	✓		✓		Propre schéma		✓	✓	✓		
Tropos	/	M	M	A,CA, CDI	TH	TH	TH	AUML	✓	✓		✓	✓	✓			✓		✓	✓	✓		✓		i*/AUML	✓	✓	✓	✓		
Formal Tropos	/	M	M	A	H	H	/	AUML	✓	✓		✓	✓	✓			✓		✓	✓	✓		✓		language de Formal Tropos	✓	✓	✓	✓		
Ingenias	/	M	B	A C I	H	H	M	UML	✓	✓	✓	✓	✓		✓		✓	✓	✓	✓	N	N	✓	N	UML, sa propre notation	✓	✓	✓	✓		✓
ADELFE 2.0	**SA**	H	M	A C,I	TH	TH	TH	UML	✓			✓	✓	✓	✓		✓	✓	✓	✓			✓		UML AMAS-ML µADL	✓	✓	✓	✓		✓
Prometheus	/	H	M	S,CA CD	M	M	M		✓	✓		✓	✓	✓			✓	✓	✓	✓							✓	✓	✓		✓
PID-SMA-EB	/	B	B		M	F	M	/	✓	✓	×	×	✓	×	×	×	✓	✓	✓	✓	×	×	✓	N	EVENT-B	✓	✓	✓	✓		✓
ForMAAD	/	M	M	S C	F	M	M		✓	✓			✓		✓	✓	✓	✓	✓	✓					AML Ztemporel	✓	✓	✓	✓		?
ASEME	**M**	M	M	A,C I,V,O	H	H	M		✓	✓	✓	✓	✓			✓			✓	✓			✓	✓	AMOLA	✓	✓	✓	✓	✓	✓
PASSI	/	M	H	A, C, I, D, T	H	M	H	Java, UML, XML RDF, FIPA	✓	✓	✓	✓		✓		✓		✓	✓	✓	✓	✓	✓	✓	UML	✓	✓	✓	✓		

Table 3.3 : Comparaison entre les méthodologies selon les critères de « Méthodologie » et de « Modélisation & Notation ».

❖ **Utilisation des techniques de l'ingénierie dirigée par les modèles :**

Cette classe se compose de 4 critères :

- ***Méta-modèle*** : Existence d'un méta-modèle bien défini d'un SMA selon la méthodologie.
- ***MDA*** : l'approche MDA est-elle adoptée dans le processus de développement de la méthodologie ?
- ***Transformation de modèles*** *:* Type de transformation de modèles : Modèle source/cible (M2M, M2T, T2M ou T2T), horizontalement (même niveau d'abstraction) ou verticalement (Niveaux d'abstraction différents) ?
- ***Phase d'application*** : La phase de développement dans laquelle la technique de transformation de modèles est utilisée.
- ***Outillage*** : Quelles sont les outils assurant l'application des techniques de l'IDM utilisées ?

❖ **Utilisation des méthodes formelles :** Cette classe se compose de 2 critères :

- ***Utilisation*** *:* Quel est l'objectif de l'utilisation des méthodes formelles : Spécification, implémentation, vérification ou validation.
- ***Langage formel utilisé*** : Quel est le langage utilisé, la logique sur laquelle il est basé, l'exécutabilité de la spécification formelle, outillage correspondant au langage formel utilisé.

La table 3.4 montre la comparaison entre les méthodologies décrites dans le deuxième chapitre et la méthodologie PASSI en se basant sur les critères relatifs à l'utilisation de techniques de l'IDM et de méthodes formelles.

	Critères relatifs à IDM							Critères relatifs à l'utilisation de méthodes formelles							
			Transformation de modèles					Utilisation				Langage formel			
Méthodologies	**Métamodèle**	**MDA**	Langage de transformation	**Type**		**Phase(s) d'application**	**Outillage**	Spécification	Implémentation	Validation	Vérification	Language formel	Logique	Exécutabilité de la spécification	Outillage
				Modèle source & Modèle cible	Verticale / Horizontal										
Gaia	✓														
Tropos	✓	✓	Tefkat	M2M, M2T	H, V	CD, I	TAOM4E, T-Tool, t2x, etc								
Formal Tropos				T2T	H			✓	×	×	✓	Propre language basé sur KAOS	Logique temporelle linéaire	×	T-Tool, NuSMV
Ingenias	✓	×	IAF	M2T, T2M	V	I	CodeUploader-tool, IDK MAS Model Editor								
ADELFE	✓	×	MAY	M2M, M2T	H, V	C, I	ATL, ADELFE-Toolkit, May								
Prometheus	✓	✓	Eclipse	M2T, T2M	H,V	CD	EPDT								
PID-SMA-EB								✓	✓	×	✓	Language B	Logique de prédicats	×	Rodin
ForMAAD	×	✓	Jscripts	M2T	H	C	Profile StarUML	✓	×	✓	×	Temporel Z	LTL	×	The Z/EVES
ASEME	×	✓	Non cité	M2M, M2T, T2M	H, V	A, C, I	SAG2SUC, SUC2SRM, SRM2IAC et IAC2JADE								
PASSI	✓		Non cité	M2T	V	C	PTK								

Table 3.4 : Comparaison entre les méthodologies selon les critères relatifs à l'IDM et aux MF.

8. Conclusion

La méthodologie PASSI, détaillée dans ce chapitre, est une méthodologie ayant un processus de conception itératif et incrémental couvrant la plupart du cycle de développement. Son processus de conception se compose de plusieurs phases réparties sur cinq modèles de niveaux d'abstraction différents avec deux activités de test. PASSI est basée sur la réutilisation en utilisant les patrons de conception et sur un ensemble de standards tels que (A)UML, RDF, architecture d'agents FIPA et JAVA. PASSI est supportée par une boite à outils (PTK, AgentFactory).

Bien qu'elle ait prouvé son efficacité dans le développement des systèmes multi-agents dans le domaine de la robotique et de systèmes d'informations de façon générale, nous pouvons signaler trois lacunes principales sur PASSI: (1) L'absence de liens de traçabilité explicites entre les différent éléments conceptuels produits durant les différentes phases de développement, ce qui peut compliquer les tâches aux développeurs en cas de modification d'un ou plusieurs éléments ; (2) La notation sur laquelle PASSI est basée (UML), est une notation semi-formelle, ce qui met les diagrammes conçus susceptibles de contenir des incohérences ou des ambiguïtés ; (3) L'outil PTK qui supporte PASSI n'adopte aucune technique de traçabilité, ce qui donne aux développeurs la possibilité de faire des erreurs. Nous pérsentons dans le chapitre suivant la logique de réécriture et le langage Maude. Ce langage est utilisé dans le cadre de cette thèse pour formaliser la méthodlogie PASSI.

Chapitre 04/ Logique de réécriture & Maude

1. Introduction

Nous allons présenter dans ce chapitre la logique de réécriture et son langage, Maude et ses extensions. Le reste de ce chapitre est organisé comme suit : Tout d'abord, un petit aperçu sur la logique de réécriture fait l'objet de la deuxième section. Ensuite, le langage Maude est bien détaillé dans la section 3. Enfin, nous allons terminer ce chapitre par une conclusion dans la section 4.

2. La logique de réécriture

2.1 Définition

La logique de réécriture a été introduite par *Jose Meseguer* **[19, 20]** pour l'objectif de décrire les systèmes concurrents. Elle permet de penser de manière correcte sur les systèmes concurrents ayant des états et évoluant en termes de transitions. En effet, la logique de réécriture unifie plusieurs modèles formels qui expriment la concurrence comme, par exemple, les systèmes de transitions libellés **[21]**, les réseaux de Petri **[22]** et CCS (Calculus of Communicating Systems) **[23]**. Les énoncés de base de cette logique sont appelés : règles de réécriture et ont la forme : *t -> t'*, où *t* et *t'* sont des termes algébriques décrivant un état partiel du système concurrent. Une règle de réécriture, dans ce cas, décrit un changement d'un état partiel vers un autre si une certaine condition C soit vraie. Formellement, une théorie de réécriture est un triplet : R = (Σ, E, R) où :

- (Σ, E) : Une théorie équationnelle avec un symbole de fonctions Σ et d'équations E et ;
- R un ensemble de règles de réécriture libellés de la forme générale :

r : t -> t' (règles de réécriture inconditionnelles) ou bien

r : t -> t' if condition (règles de réécriture conditionnelles)

Les règles de réécriture inconditionnelles indiquent que : Le terme t devient t', par contre, celles conditionnelles indiquent que : Le terme t devient *t'* si certaine condition soit vraie.

2.2 Règles d'inférence

Une théorie de réécriture a un ensemble de règles d'inférence **[19, 20]**.

- ***La réflexivité*** : Pour chaque t ∈ T_Σ (X), $\frac{}{t \to t'}$
- ***L'égalité :*** $\frac{u \to v \quad E \vdash u = u' \quad E \vdash v = v'}{u' \to v'}$

- ***La congruence :*** Pour chaque f : $k_1 \ldots k_n \rightarrow k$ dans Σ, et $t_i, t_i' \in T_\Sigma(X)$, $1 \leq i \leq n$,

$$\frac{t_1 \rightarrow t_1' \quad \ldots \; t_n \rightarrow t_n'}{f(t_1, \ldots, t_n) \rightarrow f(t_1', \ldots, t_n')}$$

- ***Le remplacement :*** Pour chaque substitution $\theta : X \rightarrow T_\Sigma(X)$, et pour chaque règle r : $t \rightarrow t'$ dans R, avec, $vars(t) \cup vars(t') = \{x_1, \ldots, x_n\}, et\ \theta(x_1) = p_1, 1 \leq l \leq n$, alors

$$\frac{p_1 \rightarrow p_1' \quad \ldots \quad p_n \rightarrow p_n'}{\theta(t) \rightarrow \theta'(t')}$$

  Où pour $1 \leq i \leq n,\ \theta'(x_i) = p_i'$, et pour chaque x ∈ X - $\{x_1, \ldots, x_n\},\ \theta'(x) = \theta(x)$.

- **La transitivité :**

$$\frac{t_1 \rightarrow t_2 \qquad t_2 \rightarrow t_3}{t_1 \rightarrow t_3}$$

Ces règles d'inférence peuvent être visualisées par la figure suivante :

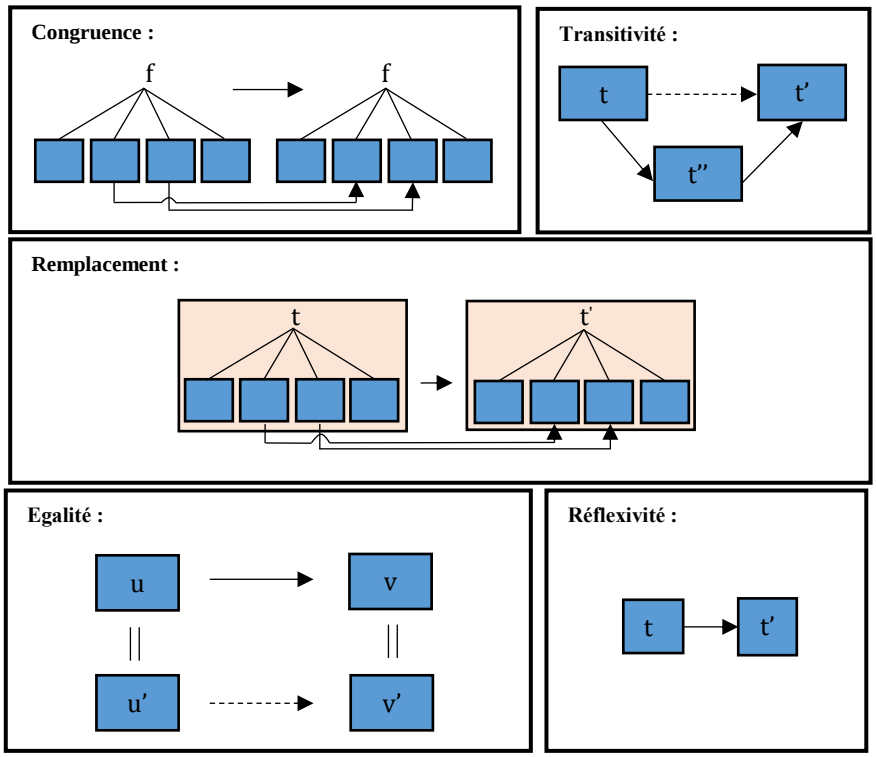

Figure 4.1 : Visualisation des règles d'inférence d'une théorie de réécriture **[19, 20]**.

La règle de réflexivité dit que pour chaque état *t*, il y a une transition *idle* dans laquelle rien ne change. La règle de l'égalité spécifie que les états sont en fait des classes d'équivalence modulo les équations *E*. La règle de congruence est une forme très générale du *parallélisme latéral* de sorte que chaque opérateur *f* peut être considéré comme un constructeur d'état parallèle permettant à ses arguments d'évoluer en parallèle. La règle de remplacement prend en charge une forme différente de parallélisme, qui pourrait s'appeler « *parallelism under one's feet* », comme en plus de réécrire une instance du côté gauche d'une règle à l'instance latérale droite correspondante, les fragments d'état dans la

substitution des variables de la règle peuvent également être réécrits. Finalement, la règle de transitivité nous permet de construire des calculs simultanés plus longs en les composant séquentiellement **[20]**. Parmi les implémentations de la logique de réécriture, nous citons CafeOBJ **[24]**, ELAN **[25]** et Maude **[26, 27]**.

2.3 Implémentation

Plusieurs langages ont essayé d'implémenter la logique de réécriture. Les principales implémentations sont : CafeOBJ, ELAN et Maude.

2.3.1 CafeOBJ

CafeOBJ[1] est un langage pour l'écriture de spécification et de vérification formelle. Il implémente la logique équationnelle par la réécriture et peut être utilisé comme un système interactif de preuve de théorème. Il est développé à l'institut japonais avancé des sciences et de la technologie (JAIST), comme une extension du langage équationnel OBJ[2].

2.3.2 ELAN

ELAN[3] est développé au *laboratoire lorrain de recherche en informatique et ses applications*[4] en France (LORIA) en concentrant beaucoup plus sur l'utilisation des stratégies pour guider la réécriture. Les techniques développées dans ce contexte ont été transférées vers un nouveau langage, TOM[5]. Ce dernier représente une extension du langage Java avec des capacités basées sur la réécriture (héritées du langage ELAN) et situé dans un niveau intermédiaire entre la programmation au bas niveau et la spécification à haut niveau.

2.3.3 Maude

Maude[6], le langage et le système déclaratif de haute performance, est développé à SRI International[7] et l'université de Illinois à Urbana-Champaign, aux Etats-Unis avec quelques collaborations en Espagne. Maude est considérée parmi les systèmes de réécriture les plus rapides. Il prend en charge la spécification de la logique de réécriture et équationnelle à la fois et la programmation d'une large gamme d'applications. Plus de détails sur le langage Maude seront donné dans la section suivante.

[1] https://cafeobj.org
[2] http://cseweb.ucsd.edu/~goguen/sys/obj.html
[3] http://elan.loria.fr
[4] http://www.loria.fr
[5] http://tom.loria.fr/wiki/index.php5/Main_Page
[6] http://maude.cs.illinois.edu/w/index.php?title=The_Maude_System
[7] https://www.sri.com

En plus, quelques auteurs ont essayé de combiner ou intégrer l'un de ces langages avec/dans un autre pour exploiter les caractéristiques spécifiques de chacun d'eux. Adrián Riesco Rodríguez et al **[170, 177, 172]** ont présenté récemment un interpréteur implémenté en Maude appelé CafeInMaude[1] pour les spécifications CafeOBJ, ce qui permet d'utiliser les outils offerts de Maude avec des spécifications de CafeOBJ.

2.4 Domaines d'application

La logique de réécriture est une logique universelle très expressive **[173]**. Ceci donne la logique de réécriture des bonnes capacités de représenter un très grand ensemble de systèmes. Maude est un langage de programmation déclaratif à usage général, il n'y en principe aucune limite aux applications qui pourraient être développées en l'utilisant **[173]**. Nous allons citer ici quelques types d'applications pour lesquelles la logique de réécriture et ses langages semblent particulièrement bien adaptés.

2.4.1 Représentation des modèles de calcul

Plusieurs modèles de calcul, y compris ceux concurrent, peuvent être directement et naturellement exprimés en tant que théorie de réécriture et peuvent être exécutés comme des modules système en Maude. Parmi les modèles de calcul qu'ont été représentés dans la logique de réécriture, on cite : ρ-calculus **[174]**, les systèmes de réduction combinatoire **[175]**, les systèmes de transitions libellées **[176]**, les réseaux de Petri (réseaux contextuels, algébriques, colorés, temporisés) **[177]** et les π-calculus **[178]**.

2.4.2 Définition de la sémantique des langages de programmation

La logique de réécriture est un framework sémantique prometteur pour spécifier formellement les langages de programmation comme des théories de réécriture. Etant donné que ces spécifications peuvent habituellement être exécutées dans Maude, elles deviennent en fait des interpréteurs pour les langages en question. En outre, telles spécifications formelles permettent à la fois un raisonnement formel et une variété d'analyses formelles pour les langages ainsi spécifiés.

2.4.3 Représentation des logiques

La généralité et l'expressivité de la logique de réécriture comme un framework sémantique pour les systèmes concurrents a également une contrepartie logique. En effet, la logique de réécriture est un framework logique prometteur dans lequel de nombreuses

[1] https://github.com/ariesco/CafeInMaude

logiques de différentes natures peuvent etre spécifiées en Maude (telles que, la logique linéaire propositionnelle **[179]** qui a été spécifiée en Maude par Clavel **[180])**.

2.4.4 Spécification et construction d'outils formels

Les prouveurs de théorèmes et d'autres outils formels ont des systèmes d'inférences sous-jacent qui peuvent être naturellement spécifiés et prototypés dans la logique de réécriture. Plusieurs outils formels développés de cette façon font partie de l'environnement formel de Maude comme par exemple : Church-Rosser **[181]**, l'outil de vérification de cohérence **[182]**, l'outil de terminaison de Maude **[183]**, Real-Time Maude (voir section 3.4.1.C). Plus loin des outils de Maude, plusieurs outils sont aussi construits comme, par exemple, l'outil JavaFAN[1] (Java Formal ANalysis) qui supporte la définition et l'analyse formelles de programmes JAVA **[184]**.

2.4.5 Spécification et analyse des protocoles de communication

Grâce à sa flexibilité dans la modélisation des objets distribués avec différents modes de communication et d'interaction, la logique de réécriture est très bien adaptée à la spécification et l'analyse des protocoles de communication comme, par exemple, le travail des chercheurs à l'université de Stanford et de SRI **[185]** qui a montré l'utilisation du langage Maude pour analyser la préconception d'un nouveau protocole de diffusion pour les réseaux actifs.

2.4.6 Spécification, analyse et vérification des conceptions de logiciels et des architectures

La logique de réécriture a été utilisée par plusieurs auteurs dans ce domaine afin de permettre une analyse, validation et vérification formelles des conceptions de logiciels. Des travaux pertinents dans cette direction comprend le travail de Wirsing, M. *et al* **[186]** qui consiste la transformation systématique des diagrammes UML vers des spécifications exécutables en Maude, qui peuvent être ensuite utilisées pour exécuter et analyser formellement la conception.

La logique de réécriture a été également utilisée pour augmenter la puissance analytique des notations architecturales telles que les langages de description architecturale (ADLs). Dans ce contexte, on cite, par exemple, le travail de *Rademaker, A., et al* **[187]**, où ils ont donné une sémantique basée-logique de réécriture pour le langage de description architecturale CBabel permettant l'exécution et la vérification des descriptions CBabel en

[1] http://fsl.cs.illinois.edu/index.php/JavaFAN

Maude. Dans ce cas, les architectures décrivant les applications complexes peuvent être vérifiées formellement (concernant les propriétés de blocage et de la consistance de synchronisation).

2.4.7 Spécification e analyse formelle des langages des domaines spécifiques

Les langages de modélisations sont usuellement définis en termes de leurs syntaxe concrète et abstraite. Ceci permet le développement rapide des langages et d'outils associés (éditeurs, etc.) mais, ne permet pas la représentation de leurs sémantiques comportementales. José E. Rivera et ses collègues de l'université de Málaga en Espagne **[188]**, ont exploré l'utilisation de Maude comme une notation formelle pour décrire des modèles, des méta-modèles et leurs comportements dynamiques en les faisant susceptibles d'une analyse et simulation formelles.

3. Maude

Défini par J. Meseguer, le langage Maude est l'une des implémentations les plus fortes et performantes de la logique de réécriture. Maude est un langage déclaratif de haut niveau et un système de haute performance prenant en charge les calculs des deux logiques, équationnelle et de réécriture, pour une large gamme d'applications.

3.1 Caractéristiques

Maude et son environnement formel peuvent être utilisés comme un langage de programmation déclarative, un langage de spécification formelle exécutable, et comme un système de vérification formelle. Voici les caractéristiques essentielles de Maude **[27]** :

- ✓ ***La simplicité*** : Les programmes/spécifications devraient être simples que possible et avoir une signification très claire.
- ✓ ***L'expressivité*** : Très grande gamme d'applications devrait être naturellement décrite par Maude : de systèmes séquentiels déterministes à des systèmes très concurrents et indéterministes, de petits systèmes aux grands systèmes, de spécifications abstraites aux implémentations concrètes.
- ✓ ***La performance*** : Les implémentations concrètes devraient générer des systèmes de performance compétitive avec d'autres langages de programmation efficaces.
- ✓ ***Menu d'un ensemble d'outils*** : Plusieurs outils ont été développé pour le système Maude (voir section 3.4.1).

3.2 Concepts de base

Dans cette sous-section, nous allons présenter la syntaxe générale d'une spécification Maude. En effet, il existe deux niveaux séparés de Maude : Core Maude et Full-Maude. Tout ce qui est dans Core Maude est en Full Maude.

3.2.1 Core Mode

A. Identificateurs

Dans Core Maude, les identificateurs sont les éléments syntaxiques basiques utilisés pour nommer les modules, les sortes, et pour former les noms d'opérateurs. Un identificateur est une séquence de caractères ASCII telle que :

- Elle ne comporte aucun caractère d'espace ;
- Les caractères '{', '}' , '(', ')', '[', ']' et ',' sont spéciaux qui coupent une séquence de caractères en différents identificateurs.
- Le caractère '' est utilisé comme un caractère d'espace pour indiquer qu'un espace blanc ou des caractères spéciaux ne coupent pas la séquence de caractères.

B. Modules

Les unités de base d'une spécification et programmation en Maude sont les modules. L'adoption de ce mécanisme de structuration facilite la compréhension des grands systèmes, augmente la réutilisation des composants et localise les effets des changements du système **[189]**. En effet, la version Core Maude supporte deux types de modules : fonctionnels et systèmes. La différence entre ces deux types réside dans les capacités de description de chacun d'eux.

❖ ***Modules fonctionnels***

Les modules fonctionnels définissent les sortes de données et les opérateurs sur ces données via des théories équationnelles. Les sortes de données sont composées des éléments qui peuvent être appelées par termes. Un module fonctionnel est déclaré suivant la syntaxe suivante :

fmod NOM-MODULE **is**

….

Endfm

Équations

Les équations sont déclarées en Maude en utilisant le mot clé « *eq* » si elle est inconditionnelle ou bien le mot clé « *ceq* » si elle est conditionnelle, suivi par un terme, le signe d'égalité « = » puis un terme représentant la partie droite de l'équation (suivi par « if »

suivi par une condition si elle est conditionnelle). Optionnellement, elles peuvent être suivies par une liste d'attributs de déclaration (*label, metadata, nonxec et owise*).

La forme générale d'une équation inconditionnelle est comme suit :

$$eq\ \langle Terme-1\rangle =\ \langle Terme-2\rangle\ [\ \langle Attributes\ de\ déclarations\rangle\ ]\ .$$

La forme générale d'une équation conditionnelle est comme suit :

$$ceq\ \langle Terme-1\rangle =\ \langle Terme-2\rangle$$
$$if\ \langle EqCondition-1\rangle/\backslash\ .../\backslash\langle EqCondition-k\rangle\ [\ \langle Attributes\ de\ déclarations\rangle\ ]\ .$$

Memberships

Les memberships sont déclarées en utilsiant le mot clé « mb » si elle est inconditionnelle ou bien le mot clé « cmb » si elle est conditionnelle, suivi par un terme, suivi par « : », suivi par une sorte (voir la sous-section C) suivi par « if » suivi par une condition si elle est conditionnelle). Comme les équations, les memberships peuvent optionnellement avoir des attributs de déclarations.

Une membership inconditionnelle a la forme générale suivante :

$$mb\ \langle Terme\rangle :\ \langle Sorte\rangle\ [\ \langle Attributs\ de\ déclarations\rangle\ ]\ .$$

Une membership conditionnelle a la forme générale suivante :

$$cmb\ \langle Terme\rangle =\ \langle Sorte\rangle$$
$$if\ \langle EqCondition-1\rangle/\backslash\ .../\backslash\langle EqCondition-k\rangle\ [\ \langle Attributs\ de\ déclarations\rangle\ ]\ .$$

La figure suivante montre un exemple d'un module fonctionnel.

```
fmod COORD-COMPLEX-TYPE is
        including FLOAT .
        including BOOL .
        sort Coord .
        op _;_ : Float Float -> Coord .
        op empty : -> Coord .
        op getLatitude : Coord -> Float .
        op getLongitude : Coord -> Float .
        op equals : Coord Coord -> Bool .
        vars Lat Lon x1 y1 x2 y2 : Float .
        eq getLatitude ( Lat ; Lon ) = Lat .
        eq getLongitude ( Lat ; Lon ) = Lon .
        eq equals ( x1 ; y1 , x2 ; y2 ) = if (x1 == x2) and (y1 == y2) then true else false fi .
endfm
```

Figure 4.2 : Exemple d'un module fonctionnel.

❖ ***Modules systèmes***

Les modules systèmes spécifient une théorie de réécriture. Un module système a des sortes, opérateurs et peut avoir des équations et des règles de réécriture qui peuvent être conditionnelles.

Un module système est défini en Maude suivant la syntaxe suivante :

mod NOM-MODULE **is**

….

endm

Règles de réécriture

L'addition qu'un module système offre par rapport au module fonctionnel, est la possibilité de spécifier de règles de réécritures. Une règle de réécriture décrit un changement d'un état partiel du système vers un autre état si, éventuellement, une certaine condition C soit vraie. Une règle de réécriture inconditionnelle a la forme $l : t \rightarrow t'$ (*if C* si elle est conditionnelle), où t et t' sont des termes du même type, et *l* représente l'étiquette de la règle. Une règle de réécriture inconditionnelle est représentée en Maude par la syntaxe suivante :

$$rl\ [\ \langle étiquette \rangle\ ]\ :\ \langle Terme - 1 \rangle\ \Rightarrow\ \langle Terme - 2 \rangle\ [\ \langle Attributs\ de\ déclarations \rangle\ ]\ .$$

Une règle de réécriture inconditionnelle est représentée en Maude par la syntaxe suivante :

$$crl\ [\ \langle étiquette \rangle\ ]\ :\ \langle Terme - 1 \rangle\ \Rightarrow\ \langle Terme - 2 \rangle$$
$$if\ \langle Condition - 1 \rangle\ /\backslash\ \ldots\ /\backslash \langle Condition - k \rangle\ [\ \langle Attributs\ de\ déclarations \rangle\ ]\ .$$

La figure suivante montre un exemple d'un module système.

```
mod BB-TEST is
        sort Expression .
        ops a b bingo : -> Expression .
        op f : Expression Expression -> Expression .
        rl a => b .
        rl b => a .
        rl f(b, b) => bingo .
endm
```

Figure 4.3 : Exemple d'un module système **[27]**.

C. Sorte et sous-sortes

Une sorte est une catégorie de valeurs. Par exemple, le nombre cinq il pourrait s'agir d'une valeur de « Integer ». Elle est déclarée en utilisant le mot clé « sort » suivi par un identificateur qui représente le nom de la sorte, suivi par un espace comme suit **[190]** :

$$sort\ \langle identificateur \rangle\ .$$

Nous pouvons aussi déclarer un ensemble de sortes en utilisant le mot clé « sorts » comme suit : $sorts\ \langle identificateur - 1 \rangle\ \ldots\ \langle identificateur - k \rangle\ .$

Les sortes peuvent être trié partiellement par une relation de sous-sorte comme suit :

$$subsort\ \langle sorte - 1 \rangle < \langle sorte - 2 \rangle\ .$$

La déclaration précédente affirme que la première sorte est une sous-sorte de la deuxième.

D. Opérateurs

Un opérateur est déclaré en Maude en utilisant le mot clé « *op* » suivi par « : », suivi une liste des sortes de ses arguments, suivie par la sorte de son résultat. Optionnellement, Il peut être suivi par une déclaration d'attributs (ils offrent des informations supplémentaires sur l'opérateur : sémantique, syntaxique, etc..). La déclaration d'un opérateur a la forme générale suivante :

$op\ \langle nomOpérateur \rangle : \langle sorte - 1 \rangle \ldots \langle sorte - k \rangle -> \langle sorte \rangle\ [\ \langle Attributs\ d'opérateurs \rangle\ ]\ .$

Si un opérateur n'a aucun argument, il s'appelle une constante.

E. Variables

Une variable est déclarée en Maude par la syntaxe composée du mot clé « *var* », d'un identificateur représentant le nom de la variable, une « : », et un identificateur représentant sa sorte comme suit **[186]** :

$$var\ \langle\ idVariable\ \rangle\ :\ \langle\ idSorte\ \rangle\ .$$

Les variables de la même sorte peuvent être déclarées en utilisant le mot clé « vars » comme suit :

$$vars\ \langle\ idVariable - 1\ \rangle \ldots \langle\ idVariable - k\ \rangle\ :\ \langle\ idSorte\ \rangle\ .$$

3.2.2 Full Maude

Full Maude est implémenté en Maude lui-même et étend Core Maude avec une notation orientée objet, où des concepts basiques du paradigme objet tels que les classes, les objets et les messages peuvent être spécifiés naturellement dans des modules dits orientés objet.

A. Modules orientés-objet

Un module orienté objet est déclaré en Maude suivant la déclaration suivante :

omod NOM-MODULE **is**

….

endom

Un module orienté objet peut contenir de déclarations de classes, d'objets, etc.

❖ ***Classes***

Une classe est définie en utilisant le mot clé « *classe* » suivi par un identifiant représentant le nom de la classe, suivi par « | », suivi par un ensemble de déclarations d'attributs séparées

par « , ». Chaque déclaration d'attribut a la forme *attr : Sorte,* où *attr* est l'identificateur de l'attribut, et *Sorte* est son type. Une déclaration d'une classe a la forme suivante :

$$class\ idClasse\ |\ attr-1\ :\ Sorte1, \ldots, attr-1\ :\ Sorte-k\ .$$

Les classes sans attributs sont déclarées suivant la syntaxe :

$$class\ C\ .$$

La notion d'héritage est bien supportée par la version Full Maude grâce au mot clé « *subclass* ». La syntaxe générale d'une relation d'héritage est comme suit :

$$subclass\ \ sousClasse < classeMère\ .$$

Les sortes requises afin de décrire un système orienté objet sont : Object, Msg et Configuration.

❖ ***Configurations***

Une configuration est un multiensemble d'objets et de messages qui représente un état possible du système. Une configuration va avoir la forme suivante (l'ordre n'a aucune importance) :

$$\langle\ obj-1\ \rangle\ \ldots\ \langle\ obj-k\ \rangle\ \langle\ messg-1\ \rangle\ \ldots\ \langle\ messg-n\rangle$$

Où, $\langle\ obj-1\ \rangle\ \ldots\ \langle\ obj-k\ \rangle$ sont des objets, $\langle\ messg-1\ \rangle\ \ldots\ \langle\ messg-k\ \rangle$ sont des messages.

Une règle de réécriture pour un système orienté objet a, en général, la forme suivante :

$$rl\ [\langle étiquette\rangle\ ]\ :\ \langle\ obj-1\ \rangle\ \ldots\ \langle\ obj-k\ \rangle\ \langle\ messg-1\ \rangle\ \ldots\ \langle\ messg-n\rangle$$
$$=>\langle\ obj'-1\ \rangle\ \ldots\ \langle\ obj'-j\ \rangle\ \langle\ obj-k+1\ \rangle\ldots\ \langle\ obj-m\ \rangle$$
$$\langle\ messg'-1\ \rangle\ \ldots\ \langle\ messg'-p\rangle$$

Où, $\langle\ obj'-1\ \rangle, \ldots, \langle\ obj'-j\ \rangle$ sont des versions mises à jour de $\langle\ obj-1\ \rangle, \ldots\ , \langle\ obj-j\ \rangle$ pour $j \leq k$, $\langle\ obj-k+1\ \rangle\ldots\ \langle\ obj-m\ \rangle$ sont des objets nouvellement crées, et $\langle\ messg'-1\ \rangle\ \ldots\ \langle\ messg'-p\rangle$ sont des nouveaux messages.

Les utilisateurs peuvent définir n'importe quelle syntaxe d'objet ou de messages qui est pratique.

La figure suivante montre un exemple très simple d'un module orienté objet.

```
omod ACCOUNT-CONCEPT is
        pr STRING .
        class Account | accountN : String, owner : Oid .
endom
```

Figure 4.4 : Exemple d'un module orienté objet.

3.2.3 Importation des modules

Trois types essentiels peuvent être utilisés pour importer les modules en Maude : L'inclusion, la protection et l'extension.

- ***Importation par inclusion***

L'importation de modules par inclusion permet l'ajout de nouveaux termes aux sortes définies dans le module importé, ainsi que la redéfinition des termes existants dans le module importé. Elle se fait en utilisant le mot clé « *including* » (ou bien l'abréviation « *inc* ») suivi par un caractère d'espace suivi par le nom du module à importer, suivi par un caractère d'espace puis un point. Les deux importations par inclusion suivantes sont équivalentes :

including STRING . et inc STRING.

- ***Importation en extension***

Ce type d'importation de modules permet l'ajout de nouveaux termes aux sortes définies dans le module importé, mais aucune redéfinition est permise. Elle se fait en utilisant le mot clé «*extending*» suivi par un caractère d'espace suivi par le nom du module à importer, suivi par un caractère d'espace puis un point. Les deux importations en extension suivantes sont équivalentes :

extending STRING . et ext STRING.

- ***Importation en protection***

Ce type d'importation de modules ne permet ni l'ajout de nouveaux termes ni la redéfinition des termes existants dans le module importé. Elle se fait en utilisant le mot clé « *protecting*» (ou bien l'abréviation « *pr* ») suivi par un caractère d'espace suivi par le nom du module à importer, suivi par un caractère d'espace puis un point. Les deux importations en protection suivantes sont équivalentes :

protecting STRING . et pr STRING.

3.3 L'environnement Maude

Comme nous avons indiqué précédemment, le langage Maude (la version actuelle de Maude est 2.7.1[1]) est menu d'un environnement permettant l'exécution des spécifications/programmes Maude. Les utilisateurs doivent, dans ce cas, écrire leurs programmes/spécifications Maude dans des fichiers textes en utilisant des applications similaires à celle de bloc note (de Windows), puis de les charger depuis l'environnement Maude. Ceci soulève de plusieurs difficultés dans la détection des erreurs syntaxique qui sont faites durant l'écriture des spécifications/programmes Maude. Plusieurs outils ont été

[1] http://maude.cs.illinois.edu/w/index.php?title=Maude_download_and_installation

développés dans la littérature pour faciliter aux utilisateurs de détecter les erreurs syntaxiques et de bien interagir avec l'environnement de base Maude. On cite ici, *Maude Workstation*[1] (téléchargeable depuis le lien au-dessous de la page) et *Maude Development Tool*[2] qui définit des plug-ins intégrables dans l'IDE Eclipse. En utilisant des outils comme ceux cités ici, les utilisateurs de Maude vont trouver beaucoup de facilités lors de l'écritures de spécifications/programmes.

3.4 Outils et extensions

Plusieurs outils ont été introduits dans la littérature supportant le langage Maude. Il est également étendu par plusieurs extensions permettant d'enrichir son expressivité dans des domaines plus précis. Dans ce qui suit, nous allons citer un ensemble de ces outils et extensions en leurs donnant des brèves descriptions. Nous avons donné plus d'importance concentration aux outils/extensions utilisés dans notre contribution.

3.4.1 Outils

A. LTL model checker

La vérification de modèle (model checking) est une technique de vérification qui explore tous les états possibles du système de manière brute. Le vérificateur de modèle (model checker) ou bien l'outil effectuant la vérification de modèles examine tous les scénarios possibles du système d'une manière systématique. De cette façon, nous pouvons montrer qu'un modèle d'un système donné satisfait vraiment une certaine propriété **[191]**. La figure 4.5 montre le schéma général de la technique de vérification de modèles.

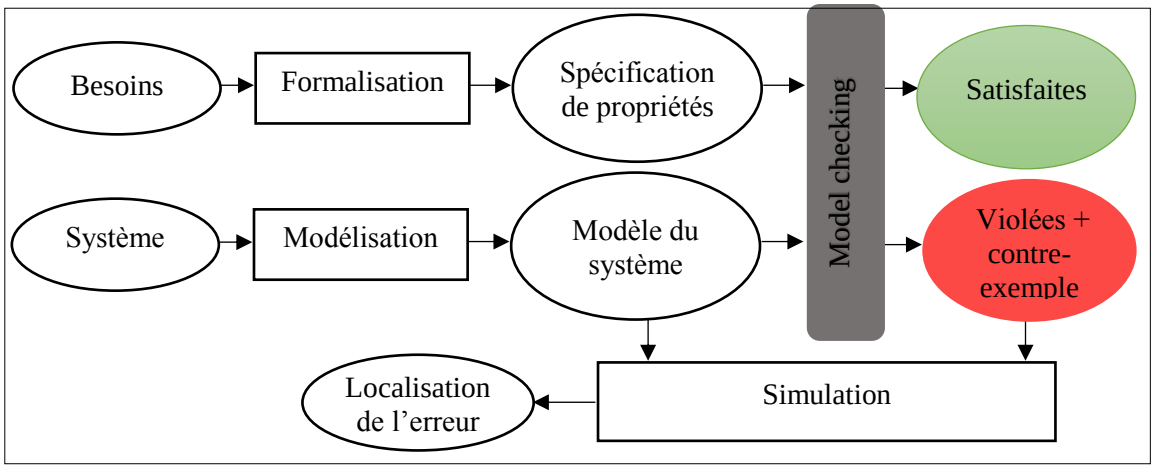

Figure 4.5 : Schéma général de la technique de model-checking **[191]**.

[1] http://www.lcc.uma.es/~duran/MaudeWorkstation/
[2] http://mdt.sourceforge.net

Maude LTL model-checker[1] est l'outil effectuant de vérification des modèles (sous-section 2.1.2.C du chapitre 02), où le modèle du système ainsi que les propriétés à vérifier sont spécifiés en Maude. Il a été conçu dans le but de combiner le langage Maude avec un moteur de model checking LTL (Linear Temporal logic) qui bénéficie des techniques de vérification de modèles à-la-volée (on-the-fly) **[192]**. Les deux figures suivantes montrent les modules fonctionnels les plus essentiels dans Maude LTL model checker, *SATISFACTION et MODEL-CHECKER.*

```
fmod SATISFACTION is
        protecting BOOL .
        sorts State Prop . *** ligne  3
        op _|=_ : State Prop -> Bool [frozen] .
endfm
```

Figure 4.6 : Le module fonctionnel SATISFACTION **[27]**.

```
fmod MODEL-CHECKER is
        protecting QID .
        including SATISFACTION .  *** 3
        including LTL .
        subsort Prop < Formula .  *** 5
        *** transitions and results
        sorts RuleName Transition TransitionList ModelCheckResult .
        subsort Qid < RuleName .
        subsort Transition < TransitionList .
        subsort Bool < ModelCheckResult .
        ops unlabeled deadlock : -> RuleName .
        op {_,_} : State RuleName -> Transition [ctor] .
        op nil : -> TransitionList [ctor] .
        op __ : TransitionList TransitionList -> TransitionList [ctor assoc id: nil] .
        op counterexample : TransitionList TransitionList -> ModelCheckResult [ctor] .
        op modelCheck : State Formula ~> ModelCheckResult [special (...)] .
endfm
```

Figure 4.7 : Le module fonctionnel MODEL-CHECKER **[27]**.

Etant donné les modules M spécifiant le système, et un état initial appelé, par exemple, *init* de type $State_M$, nous pouvons vérifier les propriétés en logique temporelle linéaire d'un système donné à partir d'un état initial (une configuration) en suivant les étapes suivantes **[192]** :

1) Définir un nouveau module appelé, par exemple, *CHECK-M*, qui importe les modules *M* et le module prédéfinit « MODEL-CHECKER » ;

2) Déclarer, *subsort* $State_M$ < *State* (*State* est l'une des sortes clés définies dans le module *SATISFACTION, figure 4.6, ligne 03)* ;

3) Définir la syntaxe des prédicats d'états que nous désirons utiliser au moyen des constantes et d'opérateurs de type *Prop* (sous-sorte de *Formula* dans le module *MODEL-CHECKER, figure 4.7, ligne 5*) ;

[1] http://maude.cs.illinois.edu/tools/lmc/

4) Définir la sémantique des prédicats d'états au moyen d'équations.

Une fois la sémantique des prédicats d'états a été définie, nous pouvons ensuite vérifier toute formule LTL, dite, *form* impliquant tels prédicats. Nous le faisons en Maude par l'expression : *init* |= *form* . Si la formule de propriété soit satisfaite, alors nous obtenons le résultat « *true* », sinon, nous obtenons un contre-exemple.

B. Predicate Abstraction in Maude

Malgré que la vérification de modèles soit une technique de vérification très puissante pour prouver un ensemble de propriétés des systèmes d'états finis, de nombreux systèmes intéressants sont infinis et, par conséquent, en dehors de la portée des algorithmes de vérification de modèles standard **[193]**. Dans ces cas, et dans les cas (ensemble infini d'états) où l'espace d'états est fini mais trop grand pour être vérifié (par la technique de vérification de modèles), l'abstraction de prédicat (Predicate Abstraction[1]) c'est une technique qui peut faire face à cette difficulté. Cette technique consiste à calculer un système d'état fini simulant le système original, dans le sens qu'une certaine propriété soit vraie dans le système dit simulant, alors elle doit être vraie aussi pour le système original **[193]**. Le prototype d'abstraction de prédicat de Maude automatise la construction du système abstrait ayant un ensemble d'états fini.

C. Real-Time Maude

Afin d'offrir la capacité de spécifier et d'analyser les systèmes temps-réel, l'outil Real-Time Maude[2] (téléchargeable depuis le lien au bas de la page) est implémenté **[194]**. Cet outil offre plusieurs techniques d'analyse incluant, la réécriture temporisée pour des objectifs de simulation et la recherche. Il a prouvé son adaptation assez bien pour l'analyse de l'exactitude et la performance des systèmes temps-réels larges et complexes **[195]**, incluant les protocoles de communication **[196]** et les algorithmes des réseaux de capteurs sans fils **[197]**

3.4.2 Extensions

A. Maude-Strategy

Maude-Strategy[3] **[198]** est une extension de Maude implémentée en Maude lui-même. Cette extension a comme objectif de contrôler, explicitement, la façon dont un terme est

[1] http://formal.cs.illinois.edu/kbae/pred/
[2] http://heim.ifi.uio.no/peterol/RealTimeMaude/
[3] http://maude.sip.ucm.es/strategies/

réécrit. L'originalité de Maude-Strategy, est de rendre possible de spécifier la stratégie de l'application des règles de réécriture, ce qui assure une séparation totale entre les règles de réécriture et leur control. L'absence de tel langage permettant cette séparation, nous pousse à coder l'ordre d'application des règles de réécritures dans les règles de réécriture elles-mêmes, ce qui les met très complexes et illisibles. Les stratégies sont définies dans des *modules de stratégies* **[198]**. Un module de stratégies est déclaré en suivant la déclaration suivante :

smod NOM-MODULE **is**

….

endsm

Les stratégies basiques sont combinées au moyen de plusieurs combinateurs, incluant : concaténation, itération, si-alors-sinon. Il est possible de définir plusieurs modules de stratégies pour un module système (ou orienté-objet) afin d'exprimer les différentes formes possibles de réécriture. Une stratégie *E* est décrite comme une opération qui, lorsqu'elle est appliquée à un terme donné *t*, produit un ensemble de termes, éventuellement vide **[198]** :

$$\_@\_ : Strat \times T_{\Sigma}(X) \rightarrow P(T_{\Sigma}(X))$$

L'opération est étendue aux ensembles de termes telles que :

$$Si\ T \subseteq P(T_{\Sigma}(X))\ et\ E \in Strat\ alors\ E @ T = \bigcup_{t \in T} S @ t\ .$$

Voici une brève description d'un sous-ensemble des stratégies du Maude-Strategy.

- ***Idle et Fail :*** Les deux premières stratégies de base sont l'identité (représenté par *Idle*) et l'échec (représenté par *Fail*). L'application de la première stratégie (Idle) renvoie le terme inchangé : $Idle\ @\ t = \{t\}$. L'application de la stratégie d'échec (Fail) renvoie l'ensemble vide comme un résultat : $Fail\ @\ t = \emptyset$
- ***Stratégies élémentaires :*** A partir de libellés de règles, il est possible de construire des stratégies afin de contrôler l'application des règles et de répéter le longtemps possible l'application d'une règle ou d'une stratégie. Une règle libellée est considérée comme une stratégie élémentaire et le résultat de l'application d'une règle libellée *L* sur un terme t renvoie l'ensemble des termes atteints par l'application de la règle *L*.
- ***Expressions Régulières :*** L'expression des stratégies élémentaires peut être combiné par l'utilisation les opérateurs de : concaténation (;), union (|), itérations (E^* pour zéro ou plus itérations, E^+ pour une ou plus itération).
  - *op _;_ : Strat Strat -> Strat [assoc] .*
  - *op _|_ : Strat Strat -> Strat [assoc comm] .*
  - *op _* : Strat -> Strat .*

➢ *op _+ : Strat -> Strat .*

L'application de la concaténation de deux stratégies E et E' sur un terme t renvoie comme résultat tous les résultats de l'application de E' sur tous les résultats de l'applications de E sur t :

$$[(E\ ;\ E')\ @\ t] = [E'\ @\ [E\ @\ t]$$

L'application de l'union des deux stratégies E et E' sur un terme t a comme résultat tous les résultats de l'application des deux stratégies E et E' séparément sur le terme t.

$$[(E \mid E')\ @\ t] = [E\ @\ t] \cup [E'\ @\ t]$$

Les opérateurs d'itération (E^* et E^+) sont utilisés pour définir des stratégies concaténant avec succès la même stratégie :

$$[E^+\ @\ t] = \bigcup_{i\geq 1}[E_i\ @\ t],\ où\ \ E_1 = E\ and\ E_n = (E\ ;\ E_{n-1})\ for\ n > 1.$$

$$[E^*@\ t] = [(idle \mid E+)\ @\ t].$$

➢ ***Stratégies conditionnelles Régulières :*** Par ailleurs, Maude-Strategy définit des opérateurs de choix qui prennent la forme générale suivante :

Si E alors E' sinon E'' (où E ? E' : E'')

Lorsqu'elle est appliquée sur un terme t, cette forme agit comme suit : La stratégie E est initialement appliquée au t, si E est évaluée successivement ($E\ @\ t \neq \emptyset$), la stratégie E' sera appliquée sur l'ensemble des termes obtenus de l'évaluation de E, sinon ($E\ @\ t = \emptyset$), E'' sera appliquée sur le terme initial t. Parmi les opérateurs dérivés de cette forme générale, nous distinguons l'opérateur *orelse* qui agit comme suit : Lorsque E est appliquée avec succès, le résultat est obtenu, mais si elle a échoué, alors E' sera appliquée sur le terme initial. Autrement dit, E orelse E' = Si E alors idle else E'.

La figure suivante illustre un exemple d'un module de stratégies. Dans ce module, quatre stratégies sont identifiées : *Branch0, Branch1, Branch2* et *Protocol*. Par exemple, la stratégie « *Branch1* » (ligne 07) spécifie que la règle de réécriture libellée par « Send-The-Nearest-Ambulance » doit être appliquée en parallèle avec la séquence (opérateur de concaténation « ; »)) des deux règles libellées respectivement par « *Transformation-1* » et « *Send-Police-Patrol* ».

```
(smod BUISNESS-PROCESS-PROTOCOL is

  strat Branch0 : @ Configuration .
  sd Branch0 := ( First-Order ; Is-Reported ) .

  strat Branch1 : @ Configuration .
  sd    Branch1 := ( Send-The-Nearest-Ambulance | ( Transformation-1 ; Send-Police-Patrol )  )! . *** 7

  strat Branch2 : @ Configuration .
  sd    Branch2 := ( Book-The-Nearest-Hospital ; Branch1 ; Mark-Accident-As-Reported ) .

  strat Protocol : @ Configuration .
  sd    Protocol := ( Branch0 ; ( Transformation-2 orelse Branch2 ) ) .

endsm)
```

Figure 4.8 : Exemple d'un module de stratégies.

B. Mobile Maude

Maude mobile[1] est un langage d'agents mobile spécifié et prototypé en Maude lui-même en l'étendant pour supporter les notions de la mobilité **[199]**. Les deux notions clés sont *processus* et objet *mobiles*. Les processus sont considérés comme des environnements informatiques où les objets mobiles peuvent résider. Les objets mobiles, ayant leurs codes propres, peuvent se déplacer entre les différents processus dans différentes locations, et peuvent communiquer de manière asynchrone **[200]**.

3.5 Commandes de Maude

Dans cette sous-section, nous donnons une liste non-exhaustive des commandes Maude utilisées pour interagir avec son environnement. Les commandes sont représentées dans la table 4.1 en leurs donnant des brèves descriptions.

[1] http://maude.cs.uiuc.edu/maude1/mobile-maude/ et http://maude.sip.ucm.es/mobilemaude/

N°	Commande (Abréviation)	Forme générale	Description
1	*reduce* *(red)*	reduce {in module : } terme .	Elle cause la réduction du terme spécifié par le bais des équations et des axiomes des memberships dans le module donné. Si la clause 'in' est omise, le module en cours est mis en considération.
2	*rewrite* *(rew)*	rewrite {[borne]} { in module :} terme .	Elle cause la réécriture du terme spécifié par le bais des règles, des équations et des axiomes des membershipes dans le module donné. L'interpréteur par défaut les applique à l'aide d'une stratégie top-down et s'arrête lorsque le nombre d'applications de règles atteint la borne donnée. Aucune règle ne sera appliquée si une équation peut être appliquée. Si la clause 'in' est omise, le module en cours est mis en considération. Si la clause de la borne supérieure est omise, l'infinité est prise en considération.
3	frewrite (frew)	frewrite {[borne {, nombre}]} { in module :} terme .	Comme la commande « rewrite », elle cause la réécriture du terme spécifié par le bais des règles, des équations et des axiomes des membershipes dans le module donné. Mais, l'interpréteur par défaut les applique à l'aide d'une stratégie peu différente (afin d'assurer l'équité locale et l'équité des règles) et s'arrête lorsque le nombre d'applications de règles atteint la 'borne' donnée.
4	*continue* *(cont)*	continue {nombre} .	Les commandes de réécritures (à l'exception de « reduce ») peuvent être suivies par la commande « continue » si le module en cours n'a pas changé depuis la dernière réécriture. Elle permet la continuation de réécrire le terme (réécrit par la dernière commande de réécriture supportée) par 'nombre' fois (ou plus).
5	*srewrite* *srew*	srewrite T using E	Elle réécrit le terme T en utilisant l'expression de stratégie E.
6	*search*	search [n, m] in _ModId _ : _Terme-1_ _TypeRecherche_ _Terme-2 _ such that _Condition _ .	Elle permet d'explorer (suivant une stratégie de la plus profonde au premier [breadth-first]) l'espace des états atteignables en différentes manières à partir du 'Terme-1' suivant le type 'typeRecherche' **(=>1** : une seule étape, **=>+** : une étape ou plus, **E*** zéro ou plus d'étapes, **=> !** : seulement les états finaux sous forme canoniques [|ceux qu'ils ne peuvent pas être réécrits ensuite] qui sont permis) dans le module 'ModId'. L'argument 'n'offre une borne sur le nombre de solutions désirées. L'argument 'm'spécifie le profond maximum de la recherche.
		search T using B	Elle effectue une recherche à partir de T selon l'expression de recherche B.

Table 4.1 : Quelques commandes (cinq de réécriture et une de recherche) de Maude **[27]** (Les clauses optionnelles sont mises entre deux accolades {}).

4. Conclusion

Dans ce chapitre, nous avons donné un aperçu sur une logique très solide et très universelle, la logique de réécriture. Plusieurs implémentations ont été réalisée pour la logique de réécriture, entre autres, Maude. Maude est un langage de spécification formelle exécutable, où les spécifications peuvent être exécutées facilement grâce à son environnement disponible gratuitement sur le Web. Deux versions différentes existent pour Maude, Core Maude et Full Maude. Cette dernière, se caractérise essentiellement par la prise en charge des notions supplémentaires en utilisant des abréviations pratiques du paradigme objet. Nombreuses extensions ont été proposées pour Maude pour enrichir son expressivité des systèmes des domaines particuliers (Real-Time Maude pour les systèmes temps-réels, etc.). Une extension assez importante de Maude est Maude-Strategy, en l'utilisant, l'ordre d'exécution des règles de réécriture peut être contrôlable explicitement par des stratégies définies en dehors les modules dans lesquels les règles de réécriture sont définies. Maude est menu d'un ensemble très important d'outils très puissants, entre autres, Maude Model-Checker utilisable dans la phase de vérification. Plusieurs travaux ont été proposés dans la littérature en appliquant la logique de réécriture en générale et son langage Maude en particulier dans la spécification, l'analyse, et la vérification formelle des systèmes appartenant aux différents domaines. Les systèmes distribués peuvent être naturellement modélisés en Maude en tant qu'un ensemble d'entités, faiblement couplés par un mécanisme de communication approprié. Le lengage Maude et son extension maude-Strategy sont utilisés dans le reste de ce manuscrit pour formaliser la méthodlogie PASSI.

Chapitre 05/ Formal-PASSI : Formalisation du processus PASSI

---

1. Introduction

Le processus de conception de PASSI est un processus itératif et incrémental. Ceci nécessite l'adoption de règles de traçabilité très explicites pour faciliter les tâches aux développeurs tout au long le cycle de développement. Le fait que la méthodologie PASSI soit basée sur une notation semi-formelle (UML), ceci rend les diagrammes conçus susceptibles de contenir, éventuellement, des incohérences ou des ambiguités, et rend l'activité du test moins efficace. Dans ce chapitre, nous allons présenter notre contribution qui constite principalement de : (1) La proposition d'un méta-modèle de traçabilité

bidirectionnelle pour la méthodologie PASSI, (2) La proposition d'une approche de formalisation de la méthodologie PASSI en l'étendant grâce à la logique de réécriture et son langage Maude, Formal-PASSI ; (3) Le développement d'un outil baptisé F-PTK(Formal-PTK), une extension de l'outil PTK supoortant Formal-PASSI.

Le reste de ce chapitre est organisé comme suit : Tout d'abord, notre méta-modèle de traçabilité pour PASSI est proposé et discuté dans la section 2. Ensuite, la version formelle de PASSI est discutée et expliquée en détails dans la section 3. Dans la section 4, l'outil développé F-PTK est illustré. La cinqième section est consacrée à une discussion sur Formal-PASSI. Enfin, nous allons conclure ce chapitre dans la sixième section.

2. Un méta-modèle de traçabilité pour PASSI

La traçabilité joue un rôle important dans le développement de systèmes informatiques, particulièrement, ceux complexes. PASSI a besoin d'une traçabilité explicite afin de faciliter la compréhension du SMA en cours de développement et pour mieux gérer les changements qui se produisent durant le cycle de développement. Dans cette section, nous allons proposer un méta-modèle (Figure 5.1) décrivant des liens de traçabilité explicites et bidirectionnels qui accompagnent les besoins fonctionnels depuis leur identification dans la phase de description du domaine à travers le cycle de développement du processus PASSI.

Comme la figure 5.1 montre, quelques informations doivent être ajoutées aux spécifications des éléments conceptuels pour créer des liens de traçabilité bidirectionnels entre eux lors les différentes phases du cycle de développement de PASSI :

- Un identifiant unique « id » (ID selon la syntaxe du DTD de l'XML) ;
- Un nom ;
- Un ou plusieurs références (IDREF/IDREFS selon la syntaxe du DTD de l'XML) vers le(s) élément(s) précédent(s) et/ou le(s) élément(s) suivant(s).

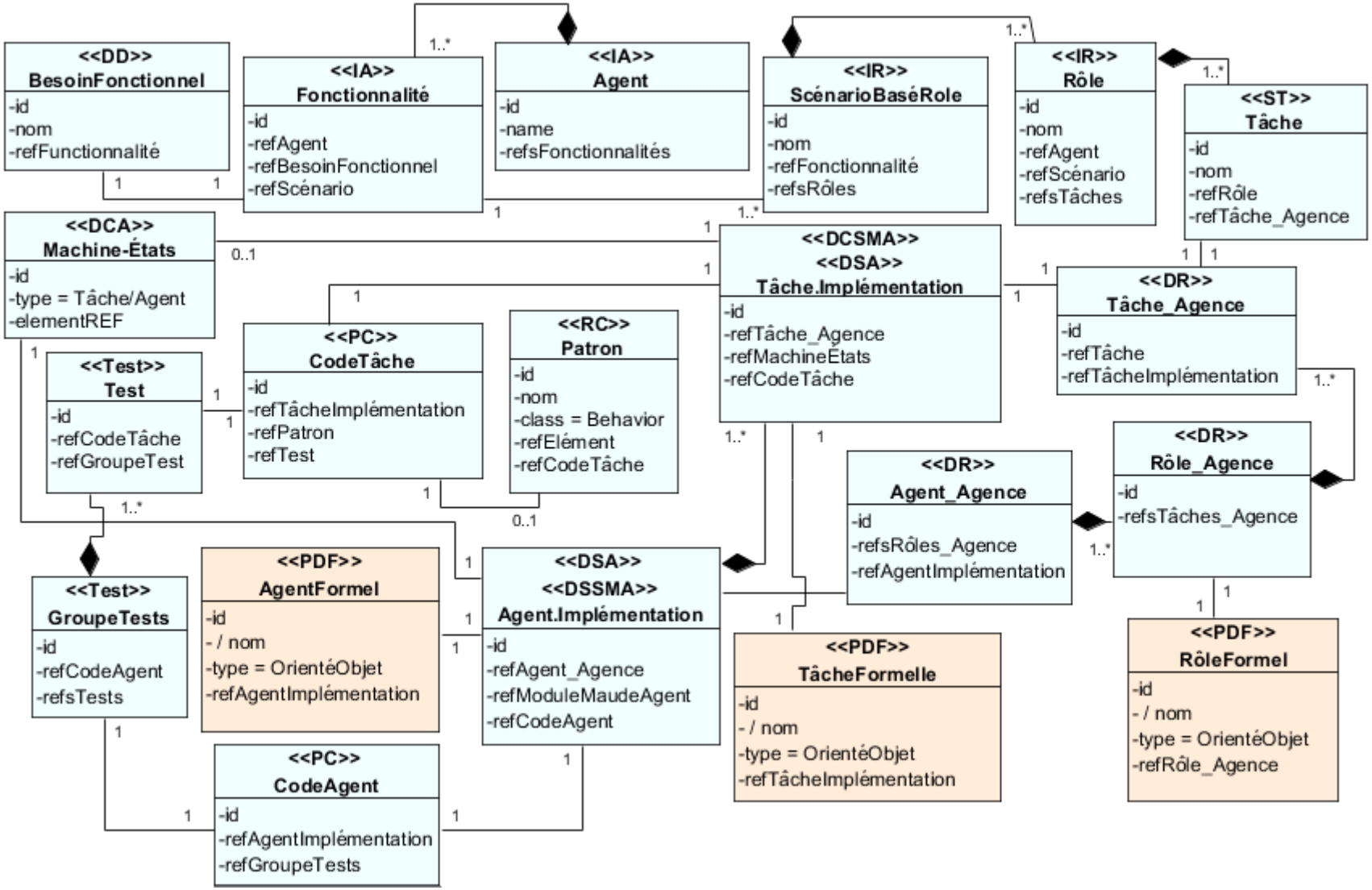

Figure 5.1 : Le méta-modèle de traçabilité Besoin2Code pour Formal-PASSI.

Le méta-modèle illustré par la figure 5.1 peut se lire comme suit : Chaque besoin fonctionnel identifié dans la phase de DD est associé ensuite dans la phase d'IA à un agent spécifique en tant qu'une fonctionnalité (l'attribut *refAgent*). Un ou plusieurs scénarios « *ScénarioBaséRôle* » (composé d'un ou plusieurs rôles) explore(nt) chaque fonctionnalité dans la phase IR. De chaque rôle, plusieurs tâches sont identifiées dans la phase ST et ainsi de suite jusqu'à l'activité du test. Par la création de liens de traçabilité explicites, le développeur peut connaitre, à chaque moment, l'origine d'une classe de tâche dans la phase DSA et quel est le test qui lui est associé. Les éléments « AgentFormel », « RoleFormel » et « TacheFormelle » sont ajoutés au métamodèle pour s'adapter à la méthodologie F-PASSI que nous avons proposée (la section suivante) et pour associer chacun d'eux à sa description formelle.

3. Le processus Formal-PASSI

Dans cette section, nous présentons F-PASSI (Formal-PASSI), une extension de la méthodologie PASSI afin de formaliser ses diagrammes en générant des spécifications formelles équivalentes, et de donner au concepteur la capacité d'appliquer quelques techniques formelles (comme celle du Model-Checking) sur les spécifications formelles générées. Comme la figure 5.2 montre, un nouveau modèle « Modèle formel » (en couleur jaune) est intégré dans le processus de conception de PASSI. Ce modèle formel est basé sur la logique de réécriture et son langage Maude (et son extension Maude-Strategy), et vise à offrir une description formelle du SMA sous-développement qui sera validée et vérifiée par la suite. Le modèle formel se compose de quatre (4) phases :

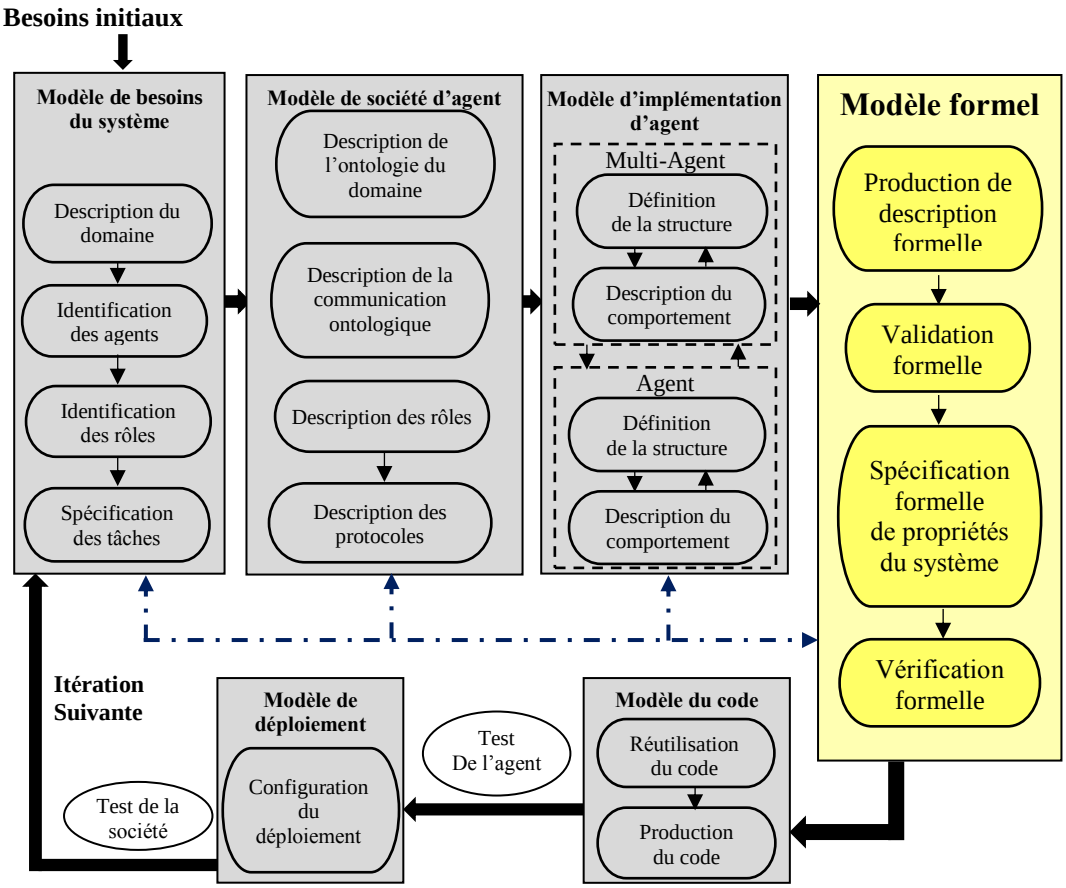

Figure 5.2 : Le processus Formal-PASSI.

3.1 Production de la description formelle (PDF)

Dans cette phase, une spécification Maude est générée à partir de diagrammes conçus pendant les différentes phases qui précèdent le modèle formel : DOD, *DR, DSSMA, DSA, DCSMA et DCA.* A la fin de cette phase, une description formelle basée-Maude couvrant les connaissances partagées par agents (les pièces de l'ontologie du domaine), la structure et le comportement du système dans les deux niveaux d'abstraction multi/individuel, sera disponible pour être exploitée dans les phases suivantes.

Pour structurer la spécification formelle basée-Maude, nous avons décidé d'effectuer sa génération en deux étapes. La première étape est une transformation de type *modèle-vers-modèle (M2M)* où le modèle source varie selon le diagramme de PASSI courant et le modèle cible est celui du langage Maude qui est conforme au méta-modèle qu'nous avons élaboré pour notre approche (Figure 5.3). La deuxième étape est une transformation de type *modèle-vers-texte (M2T)* où le modèle source est celui généré dans la première étape. A la fin de la première étape, un modèle « *M'* » conforme au méta-modèle Maude est généré. La deuxième

étape résulte d'une spécification textuelle basée-Maude « *SpecMaude* » générée à partir du modèle « *M'* » comme il est montré dans la figure 5.3.

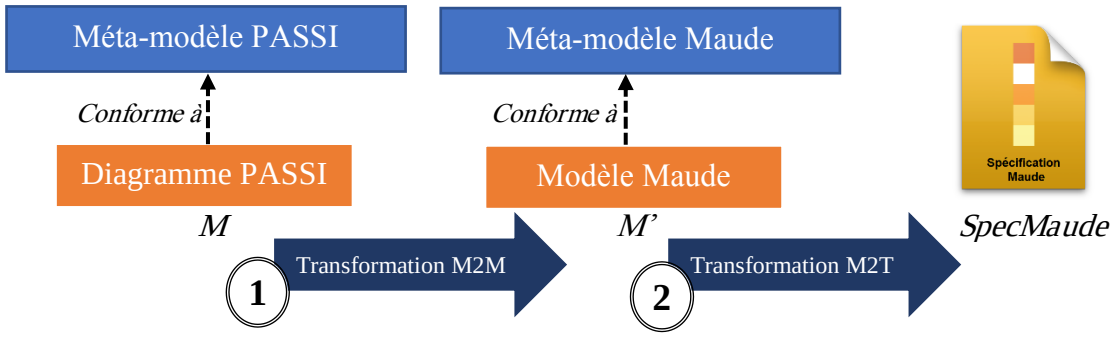

Figure 5.3 : Processus de génération de la description formelle.

Le processus montré dans la figure 5.3 est supporté par l'outil F-PTK (section 4) que nous avons développé.

3.1.1 Un méta-modèle pour Maude

Pour effectuer la phase de production de description formelle basée-Maude, nous avons élaboré un méta-modèle pour le langage Maude, la figure suivante le montre. Le méta-modèle élaboré représente les éléments de base d'une spécification Maude (section 3.2 du chapitre 04).

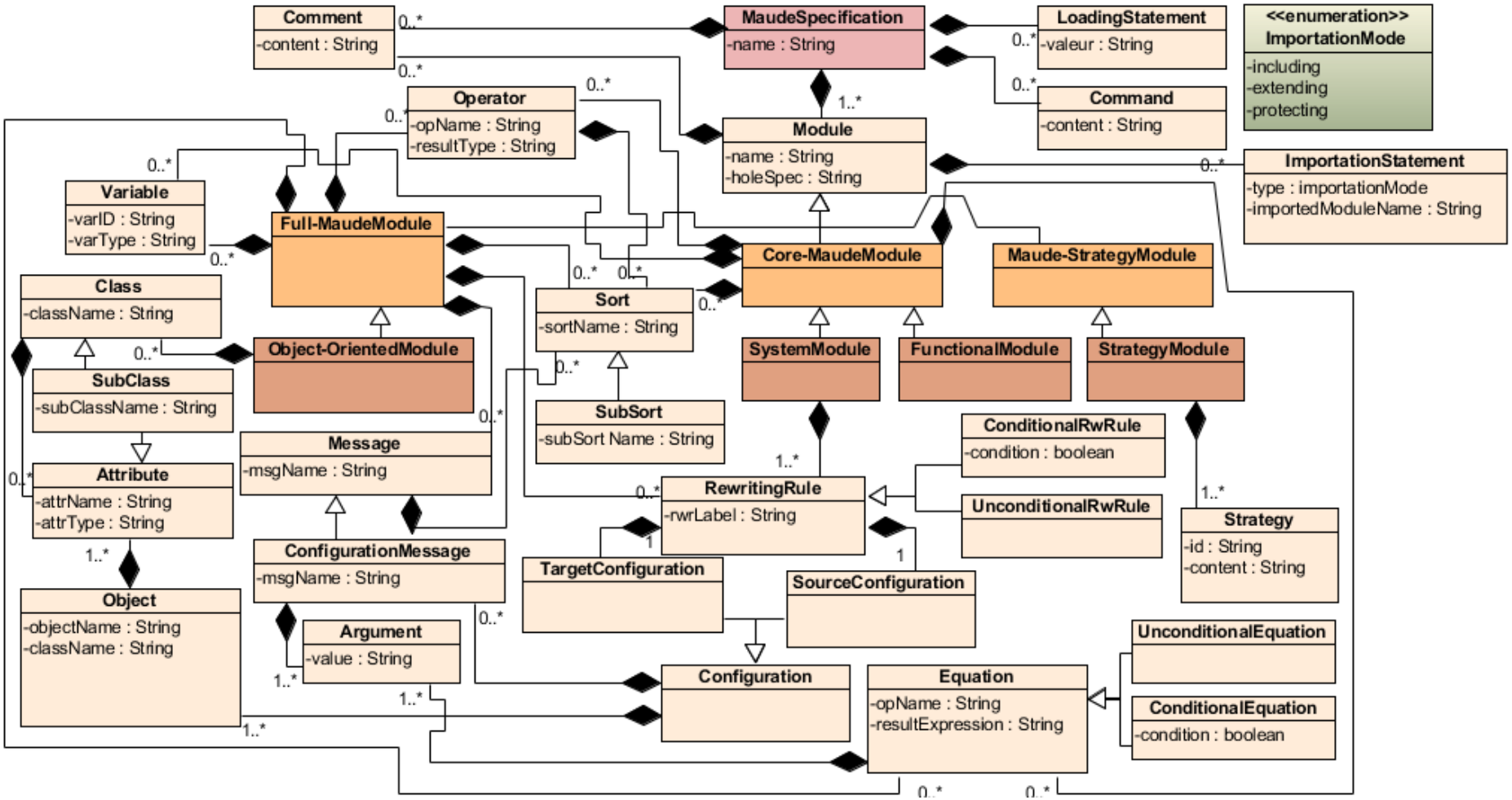

Figure 5.4 : Le méta-modèle Maude élaboré.

3.1.2 Les modules Maude générés

Après avoir effectué cette première phase du modèle formel, un ensemble de modules Maude (fonctionnel, systèmes, orientés-objet ou stratégique) sont générés. La figure suivante montre les modules Maude générés.

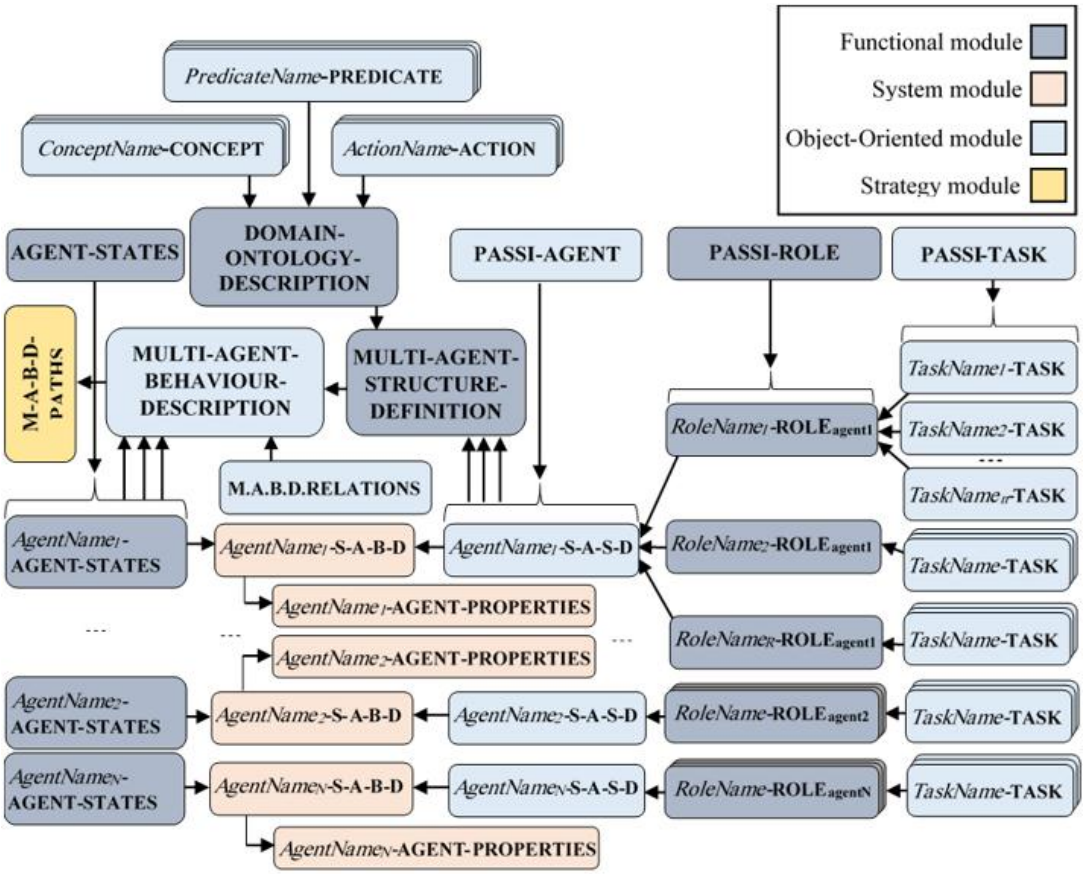

Figure 5.5 : Les module Maude générés.

La sous-section suivante décrit les règles de transformations pour les différents diagrammes de PASSI vers le langage Maude et les modules montrés dans la figure 5.5 seront abordés.

3.1.3 Règles de transformation PASSI vers Maude

A. La partie statique de la transformation (Les éléments essentiels de PASSI en Maude)

Selon la terminologie PASSI, une application basée-agent est composée d'agents, les agents jouent des rôles, les rôles sont constitués de tâches (tasks) et les tâches sont constituées d'actions. La table 5.1 représente les concepts de base sur lesquels la méthodologie PASSI est basée avec leurs représentations en Maude. La représentation de

ces éléments en Maude représente la partie statique (fixe pour chaque SMA) de la transformation.

Concepts de base de PASSI	Représentation en Maude	Description
Tâche (Task)	**(omod** PASSI-TASK **is** **inc** STRING . **class** Task | superClassTaskName : String . **op** noneTask : -> Cid . **op** noneAction : -> Msg . **** JADE commun methods for all* ****subclass Tasks* **msgs** action done : ParametersList -> Msg . **endom)**	Ce module définit la classe « *Task* » qui représente le concept de tâche. Puisque les tâches vont être translatés ensuite dans le niveau code en comportements -behaviours- (selon la terminologie du framework Jade), l'attribut *superClassTaskName* exprime le type du comportement (par exemple, OneShotBehaviour, CyclicBehavi-our). Les attributs *NoneTask* et *noneAction* expriment le fait que l'agent n'a encore effectué aucune tâche ni action. Les méthodes en commun : *action*, et *done* (selon le framework Jade) pour les sous-classes sont exprimées via des messages.
Rôle (Role)	**(fmod** PASSI-ROLE **is** **sorts** Role, NextPlayedRole . **op** noneRole : -> Role . **** Specifying that the agent is in* ****initialization step, no role played yet* **endfm)**	Le concept de « *Role* » est représenté par un module fonctionnel dans lequel un type « Role » est défini. Dans ce module, le type *NextPlayedRole* est aussi défini pour exprimer la relation [ROLE_CHANGE] spécifiée durant la phase de description de rôles (DR). Pour exprimer que l'agent n'a encore joué aucun rôle, l'opérateur *noneRole* est défini.
Agent	**(omod** PASSI-AGENT **is** **pr** PASSI-ROLE . **pr** PASSI-TASK . **pr** AGENT-STATES . **** PASSI Agent class declaration* **class** Agent | playsRole : Role, performsTask : Task , executesTaskAction : Msg, currentState : AgentState . **** JADE commun methods for all* **** subclass agents* **msgs** setup registerToDF takeDown : ParametersList -> Msg . **endom)**	Le concept « *Agent* » est représenté par un module orienté-objet dans lequel une classe appelée *Agent* ayant quatre (4) attributs est définie. La classe devrait être héritée par chaque classe d'agent. Les quatres attributs sont : 1) ***playsRole***: de type Role (défini dans le module importé PASSI-ROLE), détermine le rôle joué par l'agent dans un instant donné. 2) ***performsTask***: de type Task (défini dans le module importé PASSI-TASK), détermine la tâche effectuée par l'agent dans un instant donné. 3) ***executesTaskAction***: de type Msg (prédéfini dans Full-Maude), détermine l'action exécutée par l'agent dans un instant donné. 4) ***currentState***: de type AgentState (défini dans le module importé, AGENTS-STATES, voir Figure 5.13), exprime l'état de l'agent dans un instant donné parmi tous ses états possibles spécifiés dans le diagramme DCA. Les méthodes communes : *setup*, *registerToDF* et *takedown* (selon la terminologie du framework JADE) pour toutes les sous-classes d'agents sont exprimées par des messages.

Table 5.1 : Représentation de concepts de base de PASSI en Maude.

B. La partie dynamique de la transformation

❖ *Description de l'ontologie du domaine vers Maude (DOD2Maude)*

Le diagramme de DOD est constitué d'un ensemble d'éléments ontologiques (Concepts, prédicats et actions) reliés éventuellement par des relations qui peuvent être de type composition, généralisation ou bien de type association (Figure 5.6).

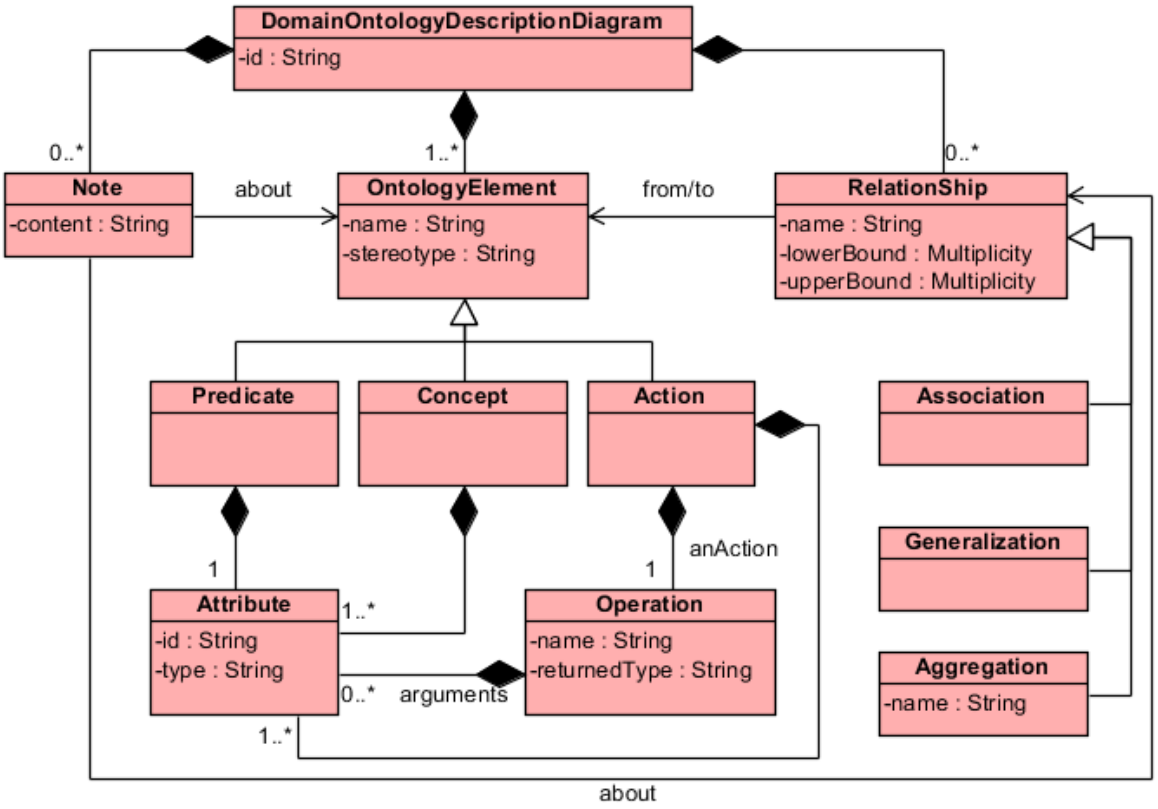

Figure 5.6 : Le méta-modèle du diagramme DOD.

La figure suivante montre un pseudo-algorithme représentant les règles de transformation d'un diagramme de description de l'ontologie du domaine vers le méta-modèle Maude.

```
Pour chaque anOntologyElement : OntologyElement du méta-modèle DODDiagram faire
    générerUnModuleOrientéObjet(unObjectOrientedModule){
      Si anOntologyElement est un Concept alors
           unObjectOrientedModule.name = anOntologyElement.name + "-CONCEPT")
       else Si anOntologyElement est un Predicate alors
                 anObjectOrientedModule.name = anOntologyElement.name + "-PREDICATE")
              else unObjectOrientedModule.name = anOntologyElement.name + "-ACTION");
            FinSi
      FinSi

    }
```

```
anObjectOrientedModule.addClass(aClass){
      aClass.className = anOntologyElement.name;
   }
   Pour chaque anAttribute : Attribute de anOntologyElement faire
      aClass.addAttribute(aMaudeClassAttribute){
        aMaudeClassAttribute.attrName = unAttribute.id;
        aMaudeClassAttribute.attrType = unAttribute.type;
      }
   FinPour
   Pour chaque anOperation : Operation de anOntologyElement faire
      anObjectOrientedModule.addMessage(aMessage){
          aMessage.msgName = anOperation.name;
          Pour chaque anArgument : Attribute de anOperation faire
              aMessage.addSort(aSort){
                  aSort.sortName = anArgument.type;
              }
          finPour.
      }
   FinPour

  Pour chaque aRelationship : Relationship de DODDiagram faire
    si aRelationship.from == anObjectOrientedModule alors
      anObjectOrientedModule.addImportationStatement(anImportationStatement){
        anImportationStatement.type = "protecting";
        anImportationStatement.importedModuleName = aRelationship.To.name;
      }
    finSi
    Si aRelationship est de type "Generalization" alors
          anObjectOrientedModule.addSubClass(aSubClass){
             aSubClass.subClassName = aRelationShip.To.name;
          }
    finSi
  FinPour
 Pour chaque aNote : Note de DODDiagram faire
    Si aNote.about == anOntologyElement alors
        anObjectOrientedModule.addComment(aComment){
          aComment.content = anote.Content;
        }
    FinSi
 FinPour.
FinPour.
```

Figure 5.7 : Règles de transformation du DOD vers Maude.

- Un concept portant le nom « *ConceptName* » est translaté comme une classe du même nom définie dans un module orienté objet avec le nom « CONCEPT-NAME-CONCEPT ».
- Un prédicat portant le nom « *PredicateName* » est translaté comme une classe du même nom définie dans un module orienté objet avec le nom « *PREDICATE-NAME-PREDICATE* ».
- Une action portant le nom « *ActionName* » est translatée comme une classe du même nom définie dans un module orienté objet avec le nom « *ACTION-NAME-ACTION* ». Les méthodes d'une action sont représentées en Maude par des messages.

Tous les modules représentant les éléments de l'ontologie (concepts, prédicats et actions) sont importés dans un module fonctionnel appelé « *DOMAIN-ONTOLOGY-DESCRIPTION* » (Figure 5.8).

```
(fmod DOMAIN-ONTOLOGY-DESCRIPTION is
  *** Importation de modules représentant les éléments ontologiques
  pr CONCEPT-NAME1-CONCEPT .
                  …
  pr CONCEPT-NAMEn-CONCEPT .
  ***
  pr PREDICATE-NAME1-PREDICATE .
                  …
  pr PREDICATE-NAMEn- PREDICATE .
  ***
  pr ACTION-NAME1-ACTION .
                  …
  pr ACTION-NAMEn- ACTION .
endfm)
```

Figure 5.8 : Représentation de l'ontologie du domaine en Maude.

❖ ***Descriptions des rôles et de la structure d'agent vers Maude (DR2Maude et DSA2Maude)***

Dans le diagramme de DR, chaque rôle est décrit par une classe où chacune de ses méthodes représente une tâche. Un rôle nommé « *RoleName* » est translaté en Maude par un opérateur de type « *Role* » défini dans un module fonctionnel appelé « *ROLENAME-ROLE* » (Figure 5.9, ligne 09) dans lequel tous les modules représentant ses tâches doivent être importés (lignes 04, 05 et 07). Puisque le type « *Role* » est défini dans le module « *PASSI-ROLE* », ce dernier est aussi importé (ligne 02).

```
(fmod ROLENAME-ROLE is  *** 01
  pr PASSI-ROLE . *** 02
          ***
  pr TASKNAME1-TASK . *** 04
  pr TASKNAME2-TASK . *** 05
          ***
  pr TASKNAMEN-TASK . *** 07
  *** Déclaration du role
  op RoleName : -> Role . *** 09
endfm)
```

Figure 5.9 : Représentation d'un rôle en Maude.

Dans le diagramme de DSA d'un agent, toutes ses tâches sont décrites par des classes indépendantes. Pour cela, nous avons choisi de représenter chaque tâche en Maude par une classe (héritant de la classe *TASK* qui est définie dans le module PASSI-TASK, voir table 5.1) du même nom dans un module orienté-objet. Toutes les actions d'une tâche sont spécifiées en tant que messages comme illustré dans la figure suivante.

```
(omod TASK-NAME is
  pr PASSI-TASK .
  pr METHOD .
  *** Déclaration de la classe de tâche
    class TaskName | attribute1 : Sort1, attributeN : Sortn .
    subclass TaskName < Task .
  *** Task action(s)
  msg action1 : Argument1Type ArgumentNType -> Msg .
  msg action2 : Argument1Type ArgumentNType -> Msg .
endom)
```

Figure 5.10 : La représentation d'une tâche en Maude.

Un diagramme de DSA d'un agent appelé « *AgentNam*e » est donc translaté en Maude par un module orienté objet « *AgentName-SINGLE-AGENT-STRUCTURE-DEFINITION* » comme illustré par la figure 5.11.

```
(omod AGENTNAME-SINGLE-AGENT-STRUCTURE-DEFINITION is
  pr PASSI-AGENT . *** Le Module qui définit la classe de base « Agent »
  pr METHOD . *** Ce module définit des types requis pour spécifier les paramètres d'une méthode
  *** Importation des modules représentant les rôles de l'agent
  pr ROLENAME1-ROLE . *** 05
  pr ROLENAME2-ROLE . *** 06
          ***
  pr ROLENAMEN-ROLE . *** 08
  *** Importation du module représentant la description de l'ontologie du domaine
  pr DOMAIN-ONTOLOGY-DESCRIPTION .  *** 10
  *** Déclaration de la classe d'agent
  class AgentName | attribute1 : Type1,  attribute2 : Type2, attributeN : TypeN .   *** 12
  subclass AgentName < Agent . *** 13
endom)
```

Figure 5.11 : Translation du diagramme de DSA en Maude.

Comme illustré dans la figure 5.11, un agent est translaté en Maude par une classe (ligne 12) héritant de la classe de base « *Agent* » (ligne 13) qui est définie et expliquée dans la table 5.1. Tous les modules représentant les rôles d'un agent (dans lesquels les modules représentant leurs tâches sont importés) sont importés (lignes 05,06 et 08). Le module qui représente la description de l'ontologie du domaine est aussi importé (ligne 10).

Par conséquence, le diagramme de DSSMA est translaté en Maude par un module fonctionnel (Figure 5.12) dans lequel tous les modules représentant les structures individuelles des agents sont importés (lignes 2, 3 et 5).

```
(fmod MULTI-AGENT-STRUCTURE-DEFINITION is

   inc AGENTNAME1-SINGLE-AGENT-STRUCTURE-DEFINITION . *** 02
   inc AGENTNAME2-SINGLE-AGENT-STRUCTURE-DEFINITION . *** 03
                ***
   inc AGENTNAMEN-SINGLE-AGENT-STRUCTURE-DEFINITION . *** 05

endfm)
```

Figure 5.12 : Translation du diagramme de DSSMA en Maude.

❖ ***Description du comportement d'agent vers Maude (DCA2Maude)***

Les états spécifiés dans le digramme de DCA d'un agent appelé « *AgentName* » sont translatés en Maude par des opérateurs définis dans un module fonctionnel nommé « *AGENT-NAME-AGENT-STATES* » comme illustré dans la figure 5.13 (lignes 05, 06 et 08).

Pour exprimer qu'un ensemble d'états représente les états d'un agent « *AgentName* » et pas un autre, le type « *AgentNameAgentState* » est défini (ligne 03). Ce dernier est un sous-type de la sorte « *AgentState* » (ligne 04) qui est défini dans le module « *AGENT-STATES* » (voir figure 5.13, ligne 2). Les états communs pour tous les agents sont déclarés comme des opérateurs dans le même module (ligne 04)

```
(fmod AGENT-NAME-AGENT-STATES is
  pr AGENT-STATES .    *** 02
  sort AgentNameAgentState . *** 03
  subsort AgentNameAgentState < AgentState . *** 04
  ops AgentState1,  *** 05
      AgentState2,  *** 06
          ***
      AgentStateN : -> AgentNameAgentState . *** 08
endfm)
```

Figure 5.13 : Le module spécifiant les états d'un agent en Maude.

❖ ***Description du comportement du SMA vers Maude (DCSMA2Maude)***

Comme cité dans **[17]**, les actions des tâches (TaskActions) dans le diagramme DCSMA sont connectés par les relations suivantes : *Invocation, Done, NewTask, Message*. Ces relations sont définies dans le module M-A-B-D-RELATIONSHIPS (voir figure 5.14, lignes :12, 13 et 14). Outre ces relations, le module définit aussi des types comme (ligne 6) *OntologyElement* (concept, prédicat ou action) ; *Performative* qui signifie la performative de communication mentionnée dans la relation *Message*. Afin d'exprimer qu'une tâche a été instanciée, qu'une action de tâche (TaskAction) a été exécutée et qu'un message a été envoyé, nous avons défini respectivement les trois messages : « *TaskInstantiated* », « *TaskActionExecuted* » et « *MessageSent* » (lignes 16, 17 et 18). Le message *FinalState* vise à exprimer l'état final d'un scenario dans le diagramme DCSMA (ligne 20).

```
(omod M-A-B-D-RELATIONSHIPS is
    ...
   inc STRING .
   pr PASSI-ROLE .
   pr PASSI-TASK .
   sorts OntologyElement Performative . *** 06
   subsort Cid < OntologyElement .
   subsort String < Performative .
   sorts Initiator Participant .
   subsort Cid < Initiator .
   subsort Cid < Participant .
   ***
   msgs invocation Done : Msg -> Msg . *** Les relations entre les actions des tâches      *** 12
   msg newTask : Task -> Msg . *** Relations entre les tâches >                          *** 13
   msg message : OntologyElement Performative -> Msg .                                   *** 14
   ***
    msg TaskInstantiated : Cid Role Cid -> Msg . *** tâche, rôle,  agent                 *** 16
    msg TaskActionExecuted : Msg Cid Cid -> Msg . *** Action Task Agent                  *** 17
    msg MessageSent : Initiator Participant OntologyElement Performative -> Msg .       *** 18
   ***
    msg FinalState : ParametersList -> Msg . *** 20
endom)
```

Figure 5.14 : Le module M-A-B-D-RELATIONSHIPS.

Le diagramme de DCSMA est translaté en Maude par un module orienté-objet appelé « MULTI-AGENT-BEHAVIOUR-DESCRIPTION » comme la figure 5.15 montre. Dans ce module, le module représentant la structure du SMA (ligne 02), les modules décrivant les états des agents (lignes : 03, 04 et 05) et le module « M-A-B-D-RELATIONSHIPS » (lignes : 06) sont importés.

Toutes les transactions décrites dans MABD sont translatées en Maude par des règles de réécriture (lignes 07, 08 et 10).

```
(omod MULTI-AGENT-BEHAVIOUR-DESCRIPTION is
  …
 inc MULTI-AGENT-AGENT-STRUCTURE-DEFINITION . *** 02
 inc AGENTNAME1-AGENT-STATES .  *** 03
 inc AGENTNAME2-AGENT-STATES .  *** 04
 inc AGENTNAMEN-AGENT-STATES . *** 05
 inc M-A-B-D-RELATIONSHIPS .          *** 06
  …
[c]rl [<transition1>] : <expr1> => <expr2> .*** 07
[c]rl [<transition2>] : <expr3> => <expr4> .*** 08
  …
[c]rl [<transitionN>] : <expr5> => <expr6> . *** 10

endom)
```

Figure 5.15 : La représentation du diagramme DCSMA en Maude.

Tous les chemins d'exécution du diagramme de DCSMA sont capturés automatiquement (grâce à l'outil F-PTK, section 4) et représentés en tant que stratégies (graçe au Maude-Strategy) dans un module de stratégies appelé « MULTI-AGENT-BEHVIOUR-DESCRIPTION-PATHS » (Figure 5.16).

```
(smod MULTI-AGENT-BEHVIOUR-DESCRIPTION-PATHS is
  …
 strat path0 : @ Configuration .
 sd path0 :=   …
  …
 strat path1 : @ Configuration .
 sd path1 :=   …
  …
 strat pathN : @ Configuration .
 sd pathN :=   …

endsm)
```

Figure 5.16 : Le module MULTI-AGENT-BEHVIOUR-DESCRIPTION-PATHS.

3.2 Validation formelle (VF)

La particularité de la description formelle générée, sachant qu'elle est développée en utilisant des objets, des messages et des règles de réécriture, est le fait qu'elle est exécutable. Comme Maude est un environnement très versatile en termes de simulation, il est possible de définir un état initial personnalisé (configuration initiale) et d'exécuter cette configuration du système. Deux diagrammes (dans la version actuelle de F-PASSI) sont considérés par la phase de validation : diagramme de description du comportement d'agents (DCA) et celui de description du comportement du SMA (DCSMA).

Pour le premier (DCA), le processus de validation commence en introduisant une ou plusieurs configurations initiales d'un agent avec sa connaissance (éléments de l'ontologie).

Pour le deuxième diagramme (DCSMA), le processus de validation commence en introduisant une ou plusieurs configurations initiales composées de modules représentant tous les agents du SMA ainsi que leurs connaissances qu'ils en ont besoin.

Après que la simulation soit exécutée, le concepteur doit lire les résultats obtenus de différentes configurations initiales et de juger s'ils sont attendus ou non. Si les résultats obtenus sont indésirables, il devrait jeter un coup d'œil sur le diagramme de DCA ou de DCSMA pour certaines modifications.

3.3 Spécification formelle des propriétés du système (SFPS)

Dans cette phase, le concepteur (qui est supposé être familiarisé avec les concepts de la logique temporelle linéaire (LTL) et du langage Maude) est chargé de spécifier formellement un ensemble de propriétés (désirables ou non) du SMA pour les vérifier pendant la phase suivante. Comme un point de départ, tous les états d'un agent appelé « AgentName1 » devraient être spécifiés en tant que prédicats élémentaires dans un module système appelé « AGENT-NAME1-PREDICATES ». Les propriétés à vérifier vont être construits en composant ces prédicats élémentaires via les opérateurs de LTL.

3.4 Vérification formelle des propriétés du système (VFPS)

Dans cette dernière phase du modèle formel, une vérification de modèles (Model-Checking) d'un ensemble de diagrammes comportementaux de PASSI est effectuée. Le Model-Checking vise à appliquer une analyse exhaustive de tous les chemins d'exécutions possibles d'un système, et à identifier si un ensemble de propriétés spécifiées dans la phase précédente sont satisfaites ou pas. L'application de cette technique sur une description formelle générée précédemment, est très importante pour vérifier les diagrammes DCSMA et DCA. L'application du Model-Checking avant de passer au modèle de code aurait l'avantage d'éviter la propagation des erreurs subtiles, introduites au niveau de modèles précédents (Modèle de besoins du système, modèle de la société d'agents et le modèle d'implémentation), avec le reste du processus de développement (Modèle du code, test d'agent, modèle de déploiement et test de la société d'agents).

4. Un outil supportant Formal PASSI : Formal-PTK

4.1 Aperçu

Pour rendre *Formal-PASSI* valide et son adoption plus large par les chercheurs et éventuellement par l'industrie, nous devons offert aux utilisateurs le(s) outil(s) nécessaire(s) pour y faire face. Pour cela, nous avons développé une boite à outils prototype pour supporter

Formal-PASSI appelée F-PTK (Formal-PASSI Toolkit) en utilisant l'environnement de développement intégré C++ Builder XE7[1]. La figure 5.17 montre l'outil développé (qui reste sous-amélioration).

Parmi les options offertes par F-PTK dans sa version 1.0, on cite : (1) L'édition des différents diagrammes du processus PASSI ; (2) La détection automatique des différents chemins d'exécution définis dans le diagramme de DCSMA et les translater en stratégies Maude à utiliser dans la phase de validation/vérification formelle ; (3) L'assurance de la consistance des diagrammes édités en se basant sur le méta-modèle de traçabilité propsoé ; (4) La sérialisation des diagrammes pour une utilisation ultérieure (vers un fichier XML); (5) La génération de la description formelle basée-Maude du SMA ; (6) la sérialisation de la description formelle générée en un fichier XML; (7) La validation de la description formelle générée ; (8) La vérification et la satisfaction d'un ensemble de propriétés.

Outre que F-PTK supporte l'extension qu'nous avons proposée pour la méthodologie PASSI, il est caractérisé principalement par rapport à PTK par le fait qu'il est basé sur le méta-modèle de traçabilité que nous avons proposé **[28]**. Ceci va guider les développeurs pendant la conception de différents diagrammes et faciliter leurs tâches.

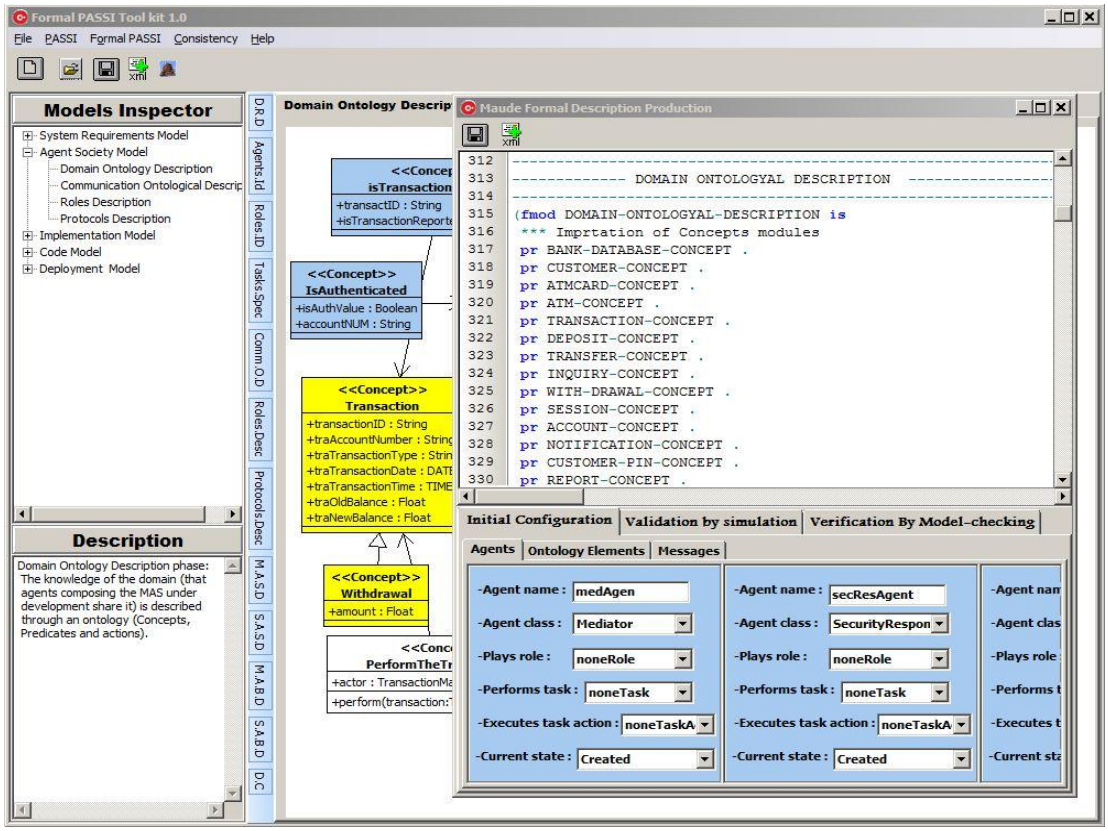

Figure 5.17 : L'interface préliminaire de l'outil Formal-PASSI toolkit (FPTK).

[1] https://www.embarcadero.com/fr/products/cbuilder

4.2 Détails techniques

4.2.1 Exécution des règles de transformation

La phase de production de la description formelle dans le processus F-PASSI, ainsi que l'opération de sérialisation des modèles, sont considérées comme des transformations de type *modèle vers texte (M2T)*. Dans la littérature, il existe plusieurs approches pour l'exécution des règles de transformation dans le domaine de l'IDM. L'approche que nous avons utilisée dans notre processus de formalisation de la méthodologie PASSI, est celle par programmation, où les règles de réécriture sont directement programmées en utilisant un langage de programmation (C++ dans notre cas). La figure suivante montre la classe *ProjetFormalPASSI* avec ses opérations principales permettant la transformation de différents diagrammes de PASSI en Maude.

```
class ProjectFormalPASSI{
  public :
    //...
    // Attributs
    static DiagrammeDescriptionOntologieDomaine *dod;
    static DiagrammeDéfinitionStructureSMA *dssma;
    static DiagrammeDescriptionComportementSMA *dcsma;
    static DéfinitionStructureAgent *dsa;
    static DescriptionComportementAgent *dca;
    static ModèleMaude *unModèleMaude;
    // Opérations
    void static DOD2Maude();
    void static DSSMA2Maude();
    void static DCSMA2Maude();
    void static DSA2Maude();
    void static DSA2Maude();
    //...
};
```

Figure 5.18 : Une partie du code source de F-PTK, la classe « *ProjetFormalPASSI* ».

La figure 5.19 représente la méthode statique *DOD2Maude* qui implémente le pseudo-algorithme (figure 5.7) permettant la transformation d'un diagramme de DOD en Maude.

```
void ProjetFormalPASSI::DOD2Maude(){
    OntologyElement *anOntologyElement = ProjetFormalPASSI::
                DiagrammeDescriptionOntologieDomaine->ontologyElementsHead;
    while (anOntologyElement != NULL){
        ObjectOrientedModule *anObjectOrientedModule =
         new ObjectOrientedModule(anOntologyElement->generateMaudeModuleName());
        Class *aClass = anObjectOrientedModule->addAClass(anOntologyElement->name);
        // ----
        Attribute *anAttribute = anOntologyElement->attributesHead;
        while (anAttribute != NULL) {
            aClass->addAttribute(anAttribute->identifier, anAttribute->type);
            if(anObjectOrientedModule->
                isThisImportationStatementExist(anAttribute->type.UpperCase()) == false)
               anObjectOrientedModule->
               addAnImportationStatement(protecting, anAttribute->type.UpperCase());
            anAttribute = anAttribute->nextAttribute;
        }
        // ----
        Operation *anOperation = anOntologyElement->operationsHead;
        while (anOperation != NULL){
            Message *aMessage = anObjectOrientedModule->addMessage
                (anOperation->name);
            Attribute * anOperationArgument = anOperation->argumentsHead;
            while (anOperationArgument != NULL) {
                aMessage->addAParameterType(anOperationArgument->type);
                if(anObjectOrientedModule->
                  isThisImportationStatementExist
                  (anOperationArgument->type.UpperCase()) == false)
               anObjectOrientedModule->addAnImportationStatement(protecting,
               anOperationArgument->type.UpperCase());
                anOperationArgument = anOperationArgument->nextAttribute;
            }
            if(anObjectOrientedModule->
                isThisImportationStatementExist
                 (anOperation->returnedType.UpperCase()) == false)
              anObjectOrientedModule->addAnImportationStatement
              (protecting, anOperation->returnedType.UpperCase());
            anOperation = anOperation->nextOperation;
        }
        Relationship *aRelationship = ProjectFormalPASSI::
             DiagrammeDescriptionOntologieDomaine->relationshipsHead;
        while (aRelationship != NULL) {
            if (aRelationship->source->name == anOntologyElement->name) {
                anObjectOrientedModule->addAnImportationStatement(protecting,
                    aRelationship->target->generateMaudeModuleName());
                if (aRelationship->type == Generalization) {
                    anObjectOrientedModule->addASubClass
                        (anOntologyElement->name, aRelationship->target->name);
                }
            }
            aRelationship = aRelationship->nextRelationship;
        }
        Note *aNote = ProjetFormalPASSI::
              DiagrammeDescriptionOntologieDomaine->notesHead;
        while(aNote != NULL){
          if(aNote->target->name == anOntologyElement->name){
            StringLine *aStringLine = aNote->contentLinesHead;
            while(aStringLine != NULL){
              anObjectOrientedModule->addAComment(aStringLine->line);
              aStringLine = aStringLine->nextLine;
            }
          }
          aNote = aNote->nextNote;
        }//...
```

Figure 5.19 : Une partie du code source de F-PTK, La méthode statique « *DOD2Maude* ».

4.2.2 Description XML de la spécification Maude adoptée

La description formelle générée durant la première phase du modèle formel du processus F-PASSI, est textuelle basée-Maude. Pour stocker cette description formelle, nous avons utilisé le langage XML. XML (*eXtensible Modeling Language*), standardisé par la W3C[1], est un langage flexible de représentation de documents textuels pour les stocker et/ou transférer, en leurs donnant une telle sémantique compréhensible par la machine. Nous avons développé une grammaire représentée en utilisant DTD[2] (*Document Type Definition*) représentant la structure générale de la description basée-Maude adoptée. La figure suivante montre le fichier DTD développé.

Maude.dtd*

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT MaudeSpecification (LoadingStatement*, Comment*, Module+)>
<!ELEMENT LoadingStatement (#PCDATA)>
<!ELEMENT Comment (#PCDATA)>
<!ELEMENT Module (Comment*,
                  ImportingStatement*,
                  SortDeclaration*,
                  SubSortDeclaration*,
                  Constant*,
                  OperationDeclaration*,
                  VariableDefinition*,
                  OperationDefinition*,
                  MessageDeclaration*,
                  RewritingRule*)>
<!ELEMENT ImportingStatement EMPTY>
<!ELEMENT SortDeclaration EMPTY>
<!ELEMENT SubSortDeclaration EMPTY>
<!ELEMENT Constant EMPTY>
<!ELEMENT OperationDeclaration (ParameterType*)>
<!ELEMENT VariableDefinition EMPTY>
<!ELEMENT OperationDefinition (Argument*)>
<!ELEMENT MessageDeclaration (ParameterType+)>
<!ELEMENT ParameterType (#PCDATA)>
<!ELEMENT Argument EMPTY>
<!ELEMENT RewritingRule (SourceConfiguration, TargetConfiguration)>
<!ELEMENT SourceConfiguration (ConfigurationMessage*, Object+)>
<!ELEMENT TargetConfiguration (ConfigurationMessage*, Object+)>
<!ELEMENT Object (Attribute*)>
<!ELEMENT Attribute EMPTY>
<!ELEMENT ConfigurationMessage EMPTY>

<!ATTLIST Module type (Functional | System | ObjectOriented)
                      #REQUIRED name CDATA #REQUIRED>
<!ATTLIST ImportingStatement type (including | protecting | extending)
                      #REQUIRED moduleName CDATA #REQUIRED>
<!ATTLIST SortDeclaration name CDATA #REQUIRED>
<!ATTLIST SubSortDeclaration SubSort CDATA #REQUIRED SuperSort CDATA #REQUIRED>
<!ATTLIST Constant name CDATA #REQUIRED type CDATA #REQUIRED>
<!ATTLIST OperationDeclaration  name CDATA #REQUIRED resultType CDATA #REQUIRED>
<!ATTLIST VariableDefinition name  CDATA #REQUIRED type CDATA #REQUIRED>
<!ATTLIST OperationDefinition name CDATA #REQUIRED resultExpression CDATA #REQUIRED>
<!ATTLIST MessageDeclaration name CDATA #REQUIRED>
<!ATTLIST Object name CDATA #REQUIRED className CDATA #REQUIRED>
<!ATTLIST Attribute name CDATA #REQUIRED type CDATA #REQUIRED>
<!ATTLIST ConfigurationMessage name CDATA #REQUIRED>
```

Figure 5.20 : Le fichier DTD développé pour la description basée-Maude.

[1] https://www.w3.org/XML/
[2] Un DTD définit la structure, les éléments légaux et les attributs d'un document XML

5. Discussion

L'intégration conjointe des méthodes formelles et la technique de transformation de modèles intervenant de l'IDM dans le processus de conception de PASSI a un impact très important sur ce dernier. La table 5.2 montre une comparaison entre PASSI et F-PASSI en se basant sur les critères décrits dans le chapitre 03.

	Critères relatifs à l'utilisation de l'IDM							**Critères relatifs à l'utilisation de méthodes formelles**							
			Transformation de modèles					Utilisation				Langage formel			
				Type											
	Méta-modèle	MDA	Langage de transformation	Modèle source & modèle cible	Verticale ou Horizontale	Phases d'application	Outillage	Spécification	Implémentation	Validation	Vérification	Langage formel	Logique	Exécutabilité de la spécification	Outillage
PASSI	✓	×	Non cité	M2T	V	C	PTK								
F-PASSI	✓	×	C++	M2M, M2T	H, V	C	F-PTK	✓		✓	✓	Maude, Maude-Strategy	Logique de réécriture	✓	L'environnement Maude

Table 5.2 : Comparaison entre PASSI et Formal-PASSI.
(H : horizontale, V : verticale et C : conception)

La nouvelle version de PASSI, Formal-PASSI, couvre la lacune de la semi-formalité de PASSI (qui est basé sur UML) graçe à l'utilisation du langage de spécification formelle, Maude (qui est basé sur la logique de réécriture) pour produire une description formelle en eliménant, éventuellement, les ambiguïtés dans les différents diagrammes de PASSI. L'utilisation de la technique de transformation de modèle (M2M et M2T) qui représente le cœur et l'âme de l'IDM augemente la productivité de la description formelle et facilité la tâche au développeur.

6. Conclusion

Dans ce chapitre, nous avons proposé un méta-modèle de traçabilité pour la méthodologie PASSI. Le méta-modèle proposé définit des liens explicites de traçabilité entre les différent éléments conceptuels produits durant le cycle de développent et appartenant aux niveaux d'abstraction différents. Ceci facilite beaucoup les tâches aux développeurs surtout celles de test et de maintenance. Notre deuxième contribution détaillée dans ce chapitre consiste à la proposition d'une nouvelle approche de formalisation de la méthodologie PASSI, Formal-PASSI. Le processus de conception de F-PÄSSI est constitué par celui de PASSI en lui intégrant un modèle formel. Ce dernier vise à offrir une description formelle basée sur le langage Maude. La description formelle est produite automatiquement grâce à l'outil F-PTK que nous avons développé (Transformation M2M puis M2T). La description formelle est ensuite validée et vérifiée en spécifiant un ensemble de propriétés relatifs aux comportements indivduel ou collectif des agents. L'outil développé supporte les deux contributions. Dans le chapitre suivant, nous allons appliquer notre approche sur une exemple simple mais réaliste pour montrer et illustrer l'apport de notre contribution.

Chapitre 06/ Etude de cas : Distributeur Automatique (ATM)

---

1. Introduction

L'approche que nous avons proposée sera illustrée dans ce chapitre à travers une étude de cas, celle du guichet de distribution automatique (ATM, Automated Teller Machine). Le reste de ce chapitre est organisé comme suit : Tout d'abord, l'étude de cas est brièvement décrite dans la section 2. Dans la troisième section, notre conception de l'étude de cas est représentée. Dans la section 4, le méta-modèle de traçabilité proposé est appliqué sur l'ATM. La cinquième section est consacrée à l'application de l'extension formelle que nous avons proposée sur l'étude de cas. Enfin, nous conclurons ce chapitre dans la sixième section.

2. Description de l'étude du cas (Guichet de distribution automatique)

Le SMA que nous avons choisi pour valider et illustrer Formal-PASSI contrôlera un guichet de distribution automatique simulé. Ce dernier est composé d'un lecteur de bande magnétique pour lire une carte ATM, une console pour interagir avec les clients, une fonte

pour déposer des enveloppes, un distributeur en espèces et une imprimante pour imprimer les reçus des clients.

Le client doit insérer une carte ATM et entrer ensuite un numéro d'identification personnel (PIN). Les informations de la carte insérée et le PIN saisi seront envoyés au système de la banque pour validation avant chaque ouverture de session. Après avoir validé la carte et le PIN, le client pourra alors effectuer une ou plusieurs transactions. Le client pourrait prendre sa carte lorsqu'il ne souhaite plus d'effectuer une transaction ou lorsqu'il décide d'interrompre une transaction en cours.

L'ATM permet aux clients de : (1) Effectuer un retrait en espèces du compte associé à la carte insérée ; (2) Effectuer un dépôt sur tout compte associé à la carte insérée ; (3) Effectuer un transfert d'argent entre deux comptes associés à la carte insérée ; (4) effectuer une nouvelle avoir sur tout compte associé à la carte insérée ; et (5) Annuler la transaction en cours si le bouton « Annuler » est appuyée.

3. La conception de l'ATM à travers PASSI

3.1 Identification des agents (IA)

Notre conception pour l'ATM a abouti à l'identification de trois (3) agents :

- ***Le médiateur (Mediator Agent) :*** C'est l'agent responsable de l'affichage d'informations (sur les options offertes, après une transaction effectuée avec succès, etc) sur la console de l'ATM ;
- ***Le gestionnaire de transactions (TransactionManager) :*** C'est l'agent chargé daccomplir de différentes transactions, du rapport des transactions et de l'impression de reçus de transactions effectuées avec succès ;
- ***Le responsable de la sécurité (SecurityRespensible) :*** C'est l'agent responsable de la vérification des cartes de clients, l'authentification des clients et de l'assurance de la confidentialité lorsqu'une transaction soit en progression.

La figure 6.1 montre le diagramme IA conçu pour l'ATM.

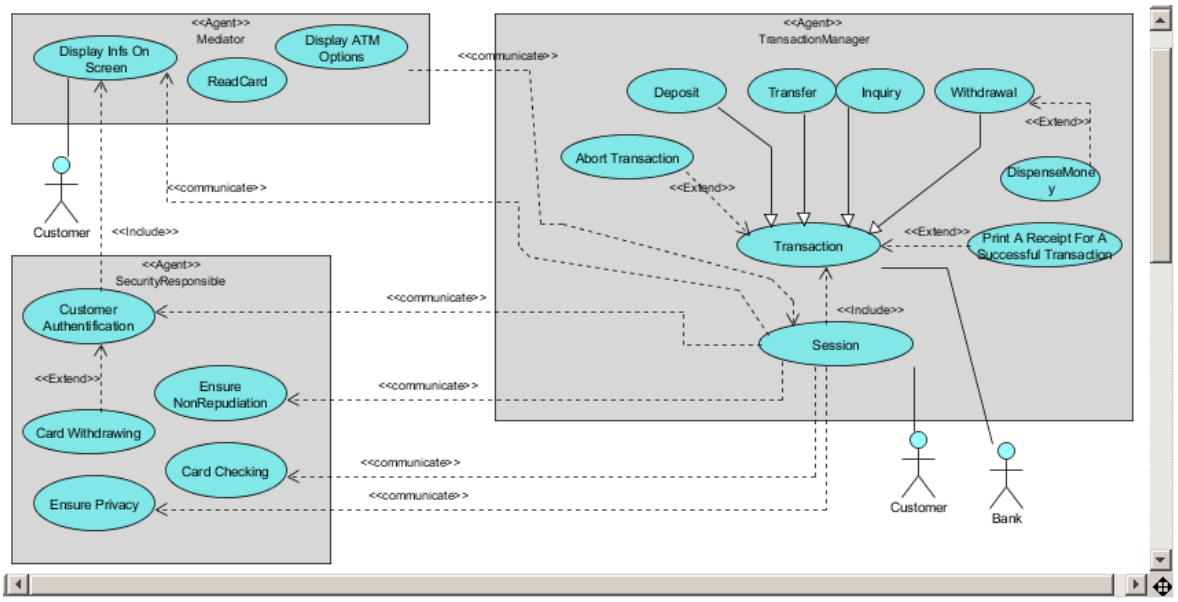

Figure 6.1 : Diagramme de IA pour l'ATM.

3.2 Identification des rôles (IR)

La figure 6.2 montre le diagramme d'identification des rôles que nous avons conçu pour l'ATM (Scénario : session). Ce scénario décrit la situation d'ouverture d'une session et l'accompissement d'une transaction avec succès.

Le diagramme de IR identifie deux rôles pour l'agent « TransactionManager », quatre rôles pour l'agent « Mediator » et trois rôles pour l'agent « SecurityResponsible ». La table suivante décrit brièvement les rôles identifiés.

Agents	**Rôles**			
Mediator	***CardReader***	***Amount Checker***	***Dispenser***	***Displayer***
	L'agent joue ce rôle lors de la lecture d'une carte ATM insérée.	L'agent joue ce rôle lors de la vérification de la montant disponible dans l'ATM.	L'agent joue ce rôle lors de la distribution de l'argent.	L'agent joue ce rôle lors de l'affichage d'informations sur la console que le client interagit avec.
Security Responsible	***Account Checker***	***Authenticator***	***Saver***	
	L'agent joue ce rôle lors de la vérification de la validité d'un compte.	L'agent joue ce rôle lors de l'authentification des clients à travers leurs PIN.	L'agent joue ce rôle lors de la sécurisation de la transaction en cours.	
Transaction Manager	***Performer***	***Reporter***		
	L'agent joue ce rôle lors de l'accomplissement d'une transaction.	L'agent joue ce rôle lors de sauvegarde des informations d'une transaction.		

Table 6.1 : Les rôles identifiés du scénario pour l'ATM.

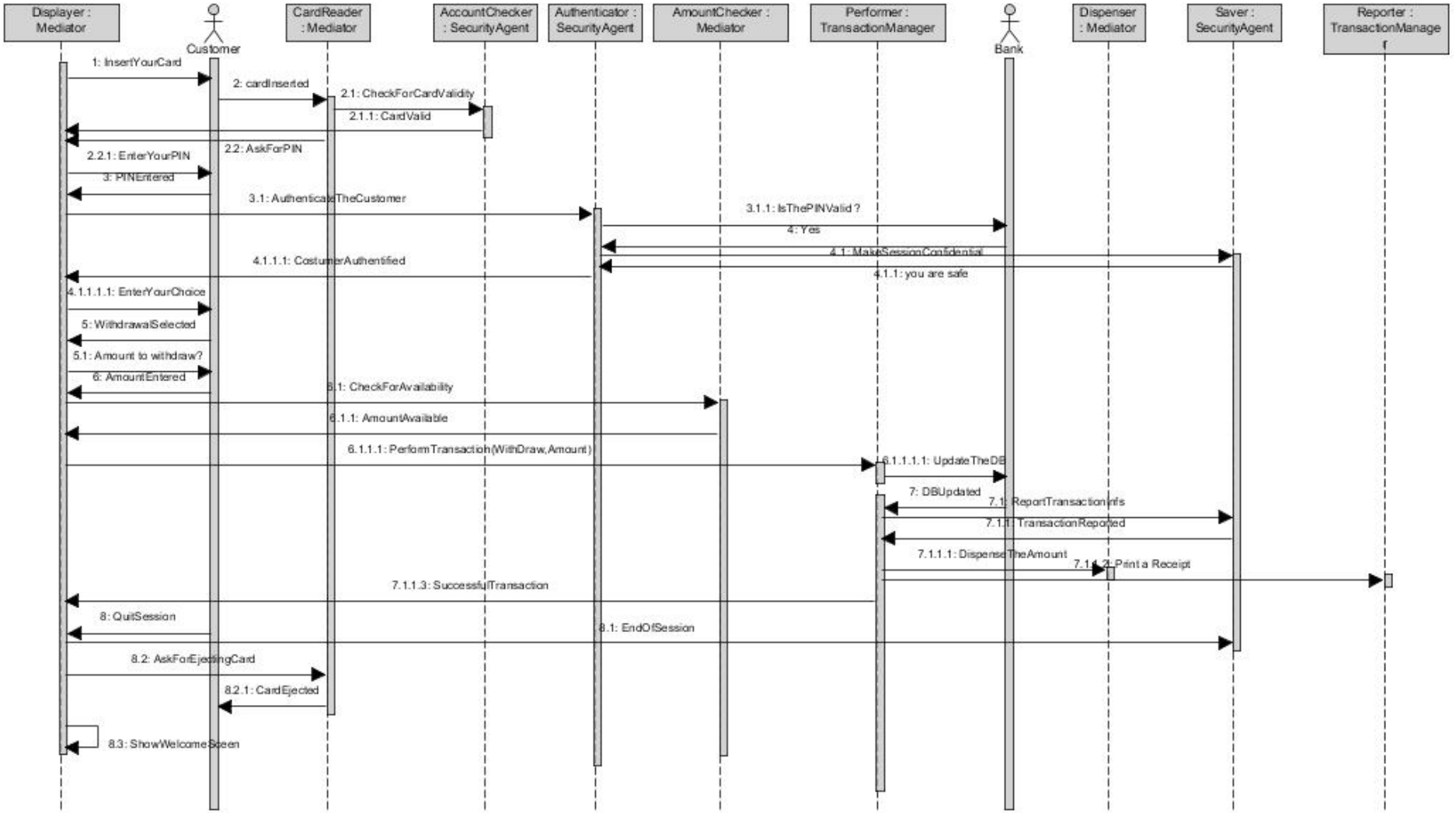

Figure 6.2 : Diagramme de IR pour l'ATM.

3.3 Description de l'ontologie du domaine (DOD)

Dans cette phase, les connaissances du domaine de l'étude du cas sont décrites d'une perspective ontologique. Par exemple, le concept « Transaction » est caractérisé par son identificateur, sa date, son temps, etc. (voir figure 6.3).

Le fait que « withdrawal », « Inquiry », « Transfer » et « Deposit » sont de types transaction, ils sont considérés et identifiés comme étant des concepts héritant le concept « Transaction ».

Le prédicat « IsTransactionPerformed » est identifié, comme la figure 6.3 montre, pour savoir si une transaction est bien effectuée (isTransPerfValue=true) ou pas (isTransPerfValue=false).

3.4 Description des rôles (DR)

Les rôles que nous avons identifiés dans le diagramme de IR sont décrites en fonction de leurs tâches dans le diagramme de DR (voir figure 6.4). Chaque rôle est composé d'une ou plusieurs tâches qui sont représentées en tant que méthodes.

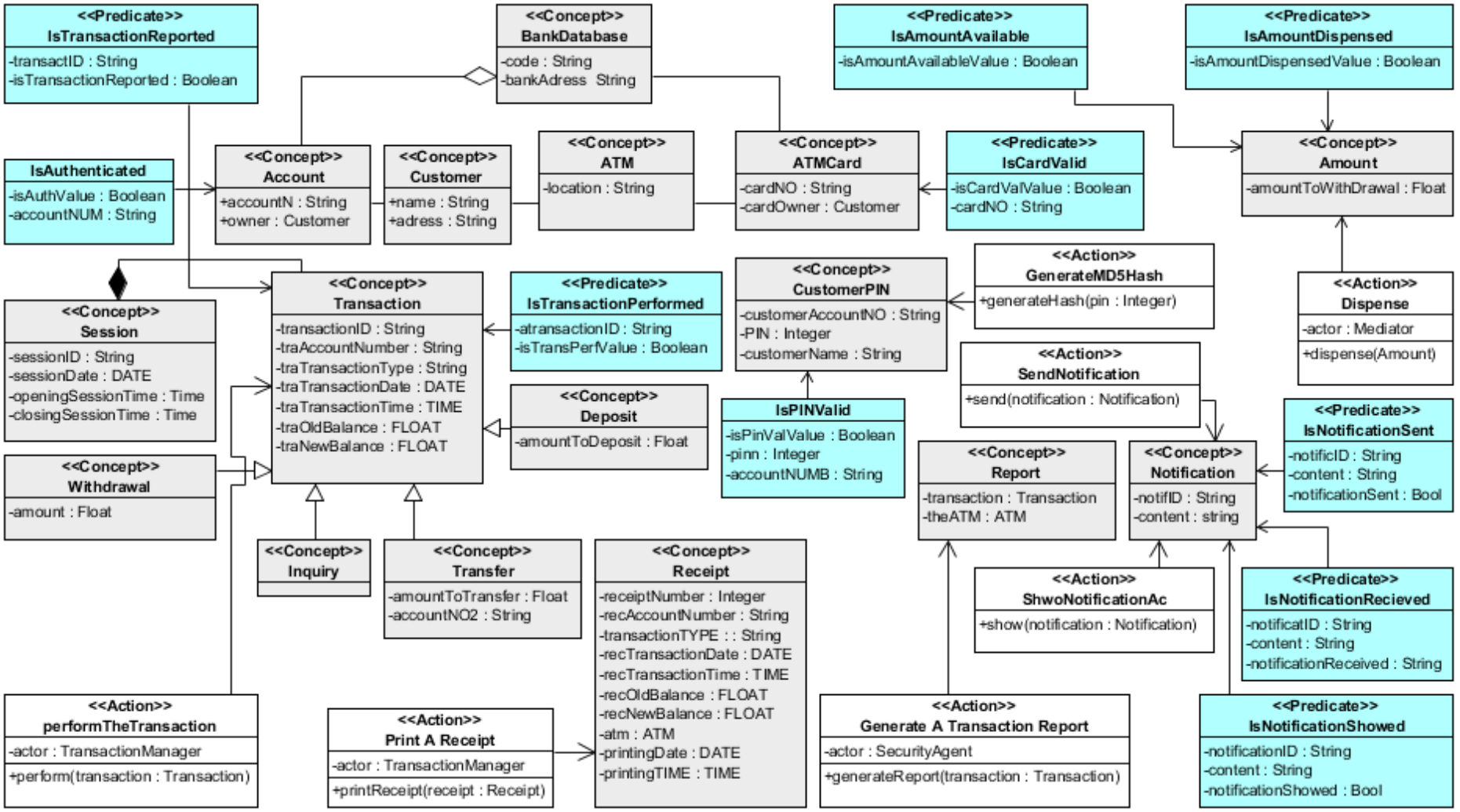

Figure 6.3 : Diagramme de DOD pour l'ATM.

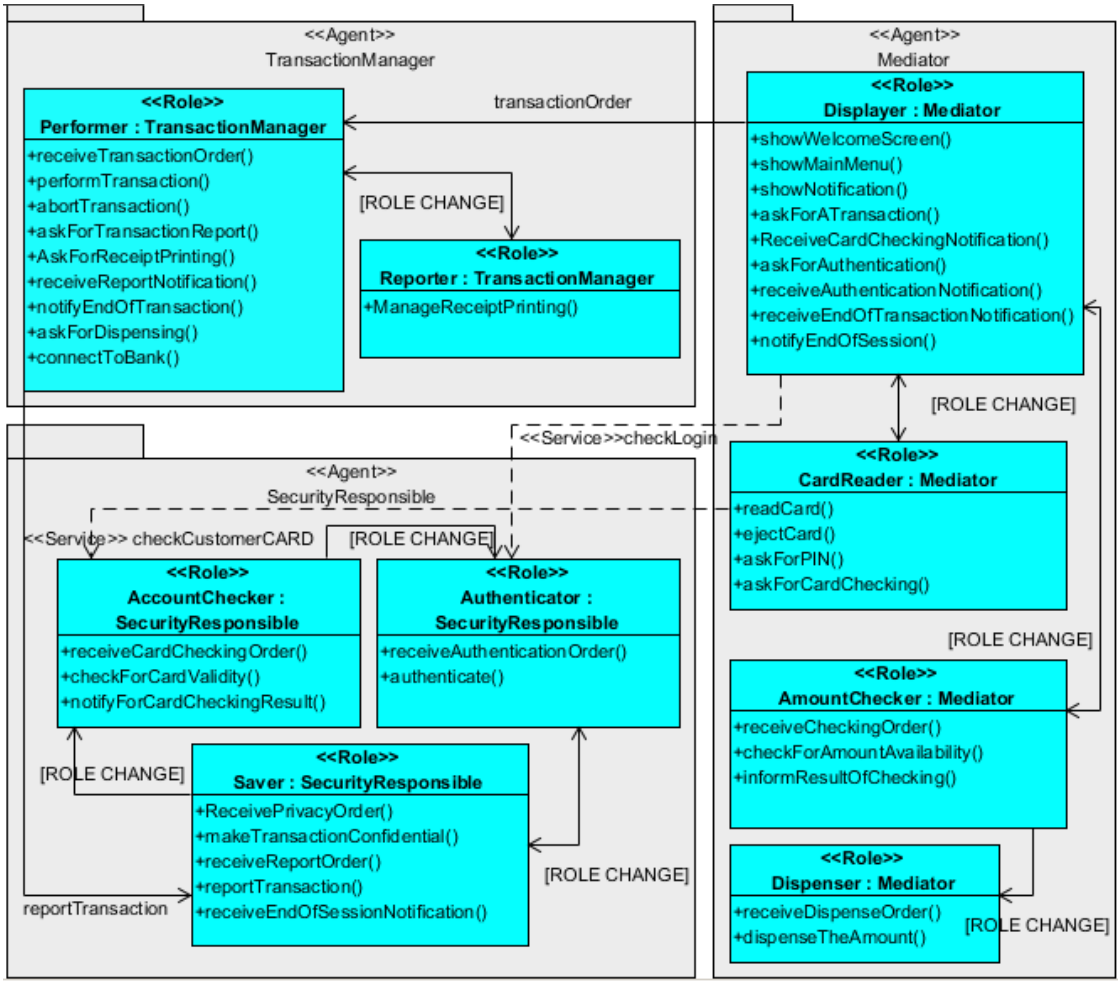

Figure 6.4 : Diagramme de DR pour l'ATM.

3.5 Définition des structures des agents (DSA)

Les figures 6.5, 6.6 et 6.7 montrent les diagrammes de DSA pour les agents de l'ATM.

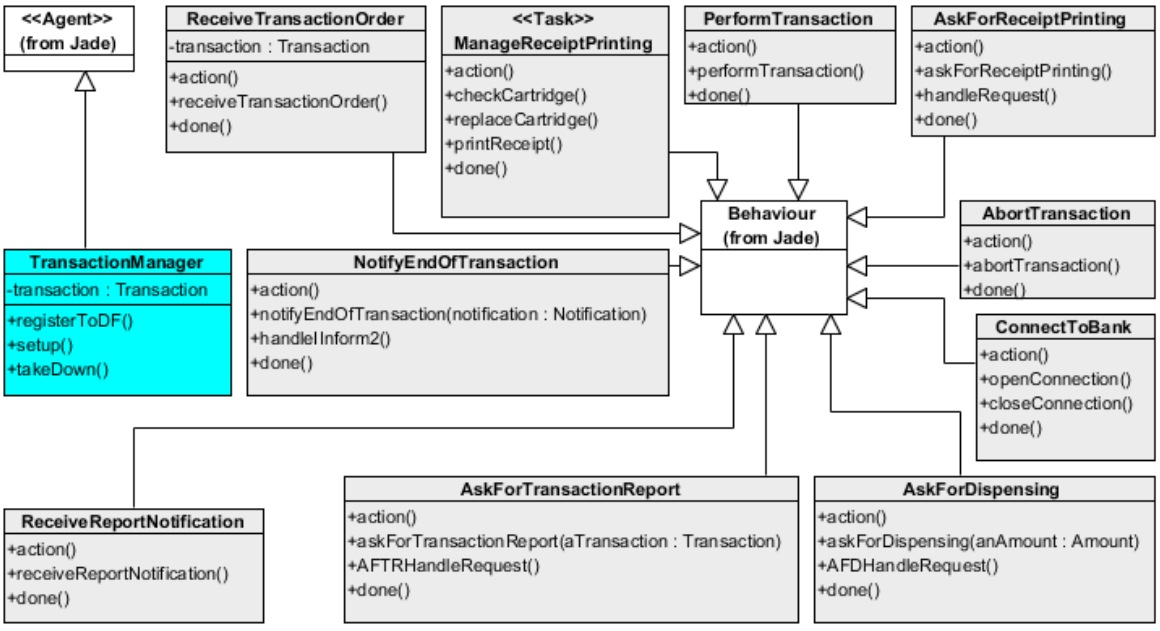

Figure 6.5 : Diagramme de DSA pour l'agent « TransactionManager ».

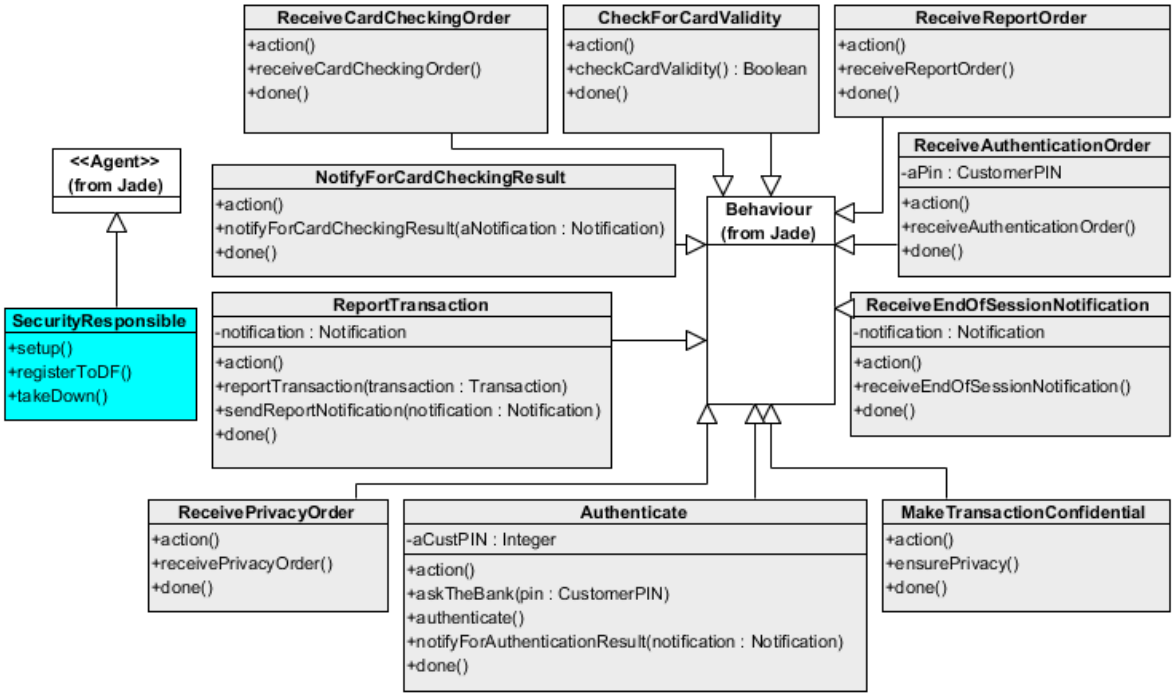

Figure 6.6 : Diagramme de DSA pour l'agent « *SecurityResponsible* ».

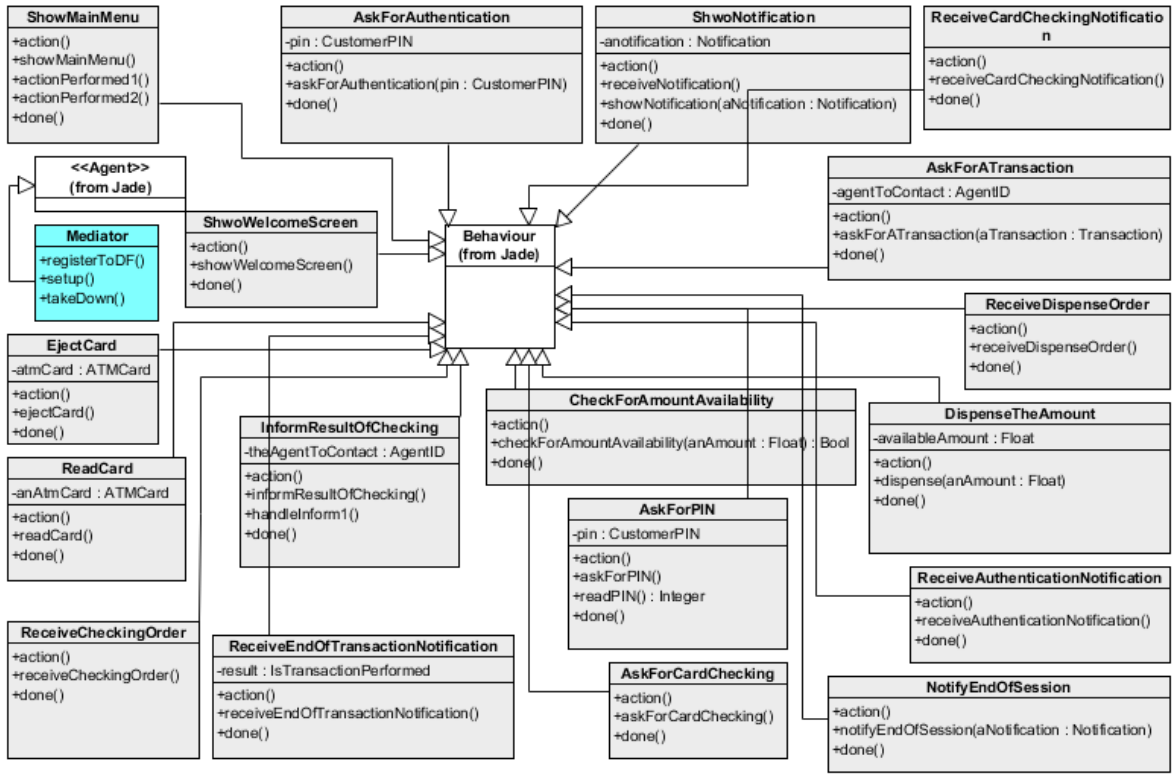

Figure 6.7 : Diagramme de DSA pour l'agent « Mediator ».

3.6 Description du comportement collectif (DCSMA)

La figure 6.8 représente une partie du diagramme DCSMA que nous avons conçu pour l'ATM. Le diagramme montre comment les actions de tâches sont exécutées, les messages qui sont envoyés entre les différents agents ou tâches. Par exemple, le message « message

(Notification, Informe) » est envoyé par l'agent « SecurityResponsible » (son action de tâche « sendReportNotification ») à l'agent « TransactionManager » (son action de tâche « receiveReportNotification »).

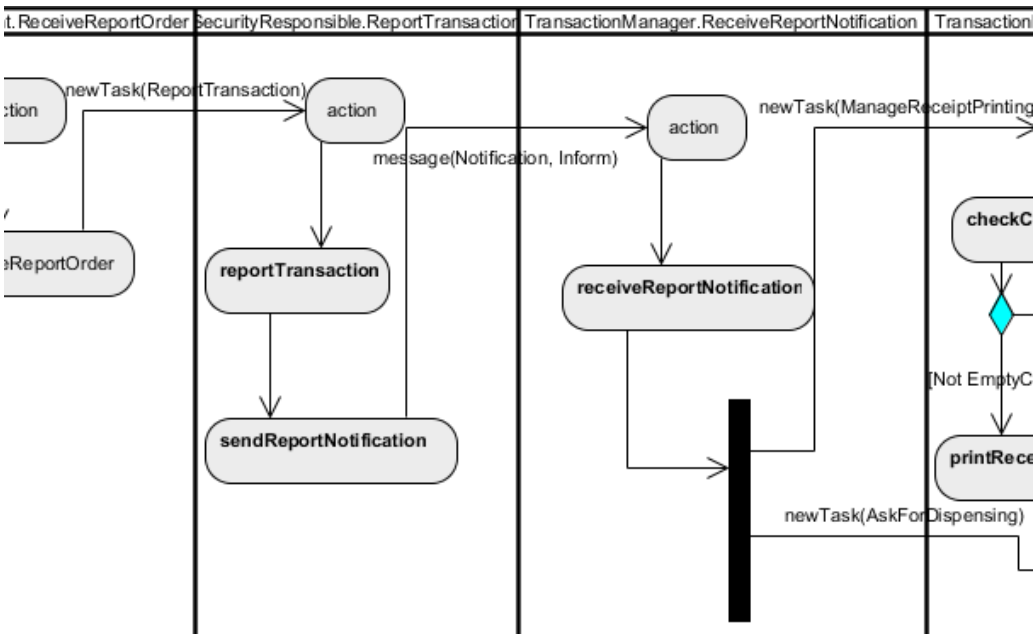

Figure 6.8 : Une partie du diagramme DCSMA pour l'ATM.

3.7 Description de comportements des agents (DCA)

Pour décrire le comportement individuel de chaque agent, nous avons utilisé le diagramme d'états-transitions. Seulement le diagramme de DCA de l'agent « TransactionManager » qui sera montré (Figure 6.9). Nous avons identifié douze (12) états pour cet agent. Par exemple, après avoir demandé la distribution d'argent de l'agent « Mediator » (L'état : *AskingForMoneyDespensing*), l'agent « Transaction Manager » sera dans l'état *NotifiyingForTransactionEn*d en exécutant l'action *notifyEndOfTransaction*.

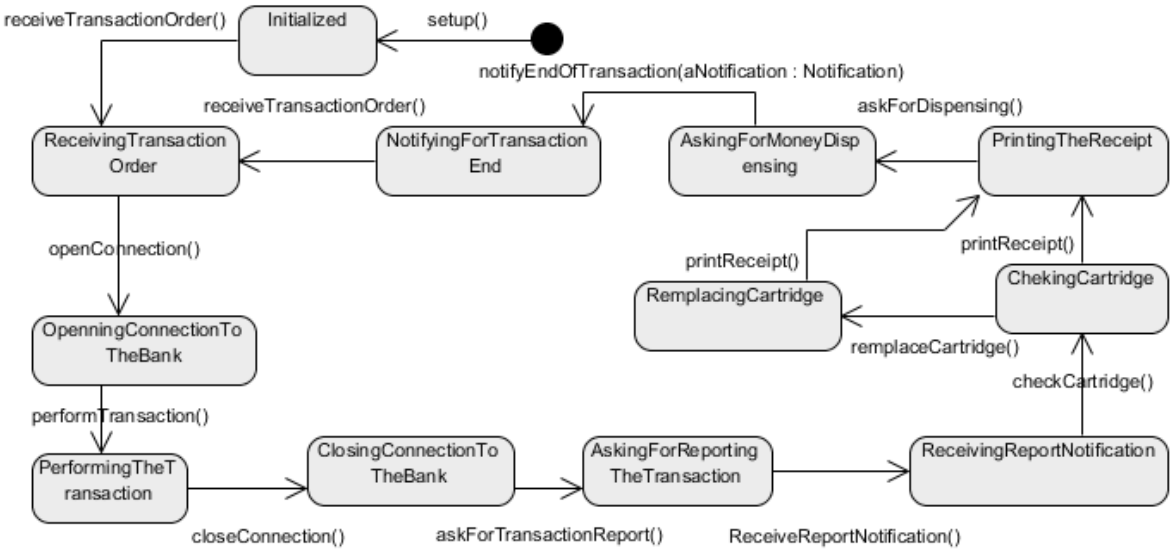

Figure 6.9 : Diagramme de DCA pour l'agent « TransactionManager ».

4. Liens de traçabilité pour l'ATM

Le besoin fonctionnel « Session » qui est identifié durant la phase de déscription du domaine, est attaché ensuite à l'agent « TransactionManager » en tant qu'une fonctionnalité durant la phase d'identiication d'agents. Cette fonctionnalité est explorée par un scénario aux rôles durant la phase d'identification de rôles. Le scénario illustré par la figure 6.2 identifie, entre autres, deux rôles : « Performer » et « Reporter ». Le rôle « Performer » est ensuite bien décrit par le rôle d'agence du même nom dans la phase de description de rôles. Ce dernier est composé de plusieurs tâches d'agence, entre autres celle « ReceiveTransactionOrder ». Cette dernière utilise, entre autres, l'élément de l'ontologie (concept) : *Transaction*. Chacun de ces éléments seront représentés en Maude (RôleFormel, TâcheFormelle, AgentFormel, ElementOntologieFormel). À la fin du processus de conception de F-PASSI, la tâche citée précédemment sera testée en utilisant un « Test » et l'agent « TransactionManager » sera testé par un « TestGroup ». Tous les éléments sont liés les uns aux autres dans les deux sens.

5. Application du modèle formel

Dans cette section, nous allons appliquer le modèle formel que nous avons proposé sur l'étude du cas ATM. Dans chaque phase, seulement un ensemble de modules qui seront montrés.

5.1 Production de la description formelle

Grâce à F-PTK, une spécification Maude du MAS sous-développement est produite. Comme nous avons cité précédemment, les éléments de l'ontologie du domaine (concepts, prédicats et actions) sont translatés en Maude par des classes définies dans des modules orienté objet. La figure suivante montre la représentation Maude correspondante du prédicat « IsAuthenticated ». L'attribut « isAuthValue » de type booléen, exprime si le client ayant le compte ATM « accountNum » soit authentifié ou pas.

```
(omod IS-AUTHENTICATED-PREDICATE is
  pr BOOL .
  pr STRING .
  class IsAuthenticated | isAuthValue : Bool, accountNUM : String .
endom)
```

Figure 6.10 : Le prédicat « IsAuthenticated » en Maude.

La structure de l'agent « TransactionManager » est représentée en Maude, comme la figure 6.11 illustre, par un module orienté objet. Une classe avec le même nom que celui de l'agent est identifiée (ligne : 979). Cette classe (comme toute autre classe d'agent) doit hériter (ligne 980) la classe « Agent » (définie dans le module AGENT-PASSI, voir Table 6.1). Les rôles joués par cet agent (Performer et Reporter) sont capturés depuis le diagramme RD et les modules dans lesquels ils sont définis sont importés (lignes : 973, 974) aussi bien que l'ontologie du domaine (ligne 976).

```
(omod TRANSACTION-MANAGER-SINGLE-AGENT-STRUCTURE-DEFINITION is
  pr PASSI-AGENT . *** ligne 970
  …
  *** Importation des modules de Roles
  pr PERFORMER-ROLE . *** ligne 973
  pr REPORTER-ROLE . *** ligne 974
  *** Importation du module de description de l'ontologie du domaine
  pr DOMAIN-ONTOLOGY-DESCRIPTION .*** ligne 976
  *** Importation des modules des différent types de Maude
  *** Déclaration de la classe d'agent
  class TransactionManager | transaction : Oid .*** ligne 979
  subclass TransactionManager < Agent . *** ligne 980
endom)
```

Figure 6.11 : Le module DSA de l'agent « TransactionManager » en Maude.

La figure 6.12 montre le module Maude qui représente le diagramme de DCSMA. Ce module importe les modules suivants : 1) Le module MASD (ligne 1014), 2) Le module M-A-B-D-RELATIONSHIPS (ligne 1016), 3) Tous les modules représentant les états des différents agents qui composent le SMA (lignes 1018, 1019 et 1020).

```
(fmod MULTI-AGENT-STRUCTURE-DEFINITION is
   inc MEDIATOR-SINGLE-AGENT-STRUCTURE-DEFINITION .
   inc SECURITY-RESPONSIBLE-SINGLE-AGENT- STRUCTURE-DEFINITION .
   inc TRANSACTION-MANAGER-SINGLE-AGENT-STRUCTURE-DEFINITION .
endfm)
*****************************
(omod MULTI-AGENT-BEHAVIOUR-DESCRIPTION is
   inc MULTI-AGENT-STRUCTURE-DEFINITION .***ligne : 1014
   ***
   inc M-A-B-D-RELATIONS . ***ligne : 1016
   ***
   inc MEDIATOR-AGENT-STATES . ***ligne : 1018
   inc TRANSACTION-MANAGER-AGENT-STATES .***ligne: 1019
   inc SECURITY-RESPONSIBLE-AGENT-STATES . ***ligne : 1020
   …
endom)
```

Figure 6.12 : Le module MULTI-AGENT-STRUCTURE-DEFINITION et une partie du module MULTI-AGENT-BEHAVIOUR-DESCRIPTION.

Toutes les relations apparaissant dans le diagramme de DCSMA qui relient les actions de tâches sont translatées en Maude par des règles de réécriture. L'exécution de chaque règle de réécriture affecte les états des agents impliqués et les éléments ontologiques utilisés.

La figure 6.13 illustre la règle de réécriture (libellée par : MABD-35, ligne : 1433) qui représente l'exécution de l'action de la tâche « notifyForAuthenticationResults » dans le cas d'un PIN invalide. Dans ce cas, l'état de l'agent « SecurityResponsible » est changé de « SendingAuthenticationResult » à « SendingAuthenticationResult » (lignes : 1438 et 1446). Aussi, l'objet du prédicat « IsAuthent » avec la valeur « false » et un objet de notification « notif » avec le contenu « Your Pin is invalid, please enter a correct one » sont générés (lignes : 1441, 1442 et 1443).

```
rl[ MABD-35 ] : *** ligne :1433
   invocation(notifyForAuthenticationResult(notif))
   < custPIN : CustomerPIN | customerAccountNO : accno, PIN : pin, customerName : custName >
   < secRes: SecurityResponsible | playsRole : Authenticator, performsTask : Authenticate,
   executesTaskAction : authenticate(EmptyParametersList),
   currentState: AuthenticatingTheCustomer > ***ligne : 1438
   =>
   message(Notification, "Inform")
   < "isAuthent" : IsAuthenticated | isAuthValue : false, accountNUM : accno > ***ligne : 1441
   < "notif" : Notification | notifID : notID, ***ligne : 1442
   content : "Your Pin is invalid, please enter a correct one" > , ***ligne : 1443
   < secRes : SecurityResponsible | playsRole : Authenticator, performsTask : Authenticate,
   executesTaskAction : notifyForAuthenticationResult(notif),
   currentState : SendingAuthenticationResult > . ***ligne : 1446
```

Figure 6.13 : Une règle de réécriture du module MULTI-AGENT-BEHAVIOUR-DESCRIPTION.

La figure suivante montre une partie de l'ensemble de stratégies capturées du diagramme DCSMA et définies dans le module stratégique MULTI-AGENT-BEHAVIOUR-DESCRIPTION-PATHS.

```
(smod MULTI-AGENT-BEHAVIOUR-DESCRIPTION-PATHS is
  strat Root : @ Configuration .
  sd Root := ( MABD-01 ; MABD-02 ; MABD-03 ; MABD-04 ;
               MABD-05 ; MABD-06 ; MABD-07 ; MABD-08 ;
               MABD-09 ).
  …
  strat Parall1-1 : @ Configuration .
  sd Parall1-1 := ( Branch1-6 | Branch1-7 )! .
  *** Case of well passed scenario
  strat Path1 : @ Configuration . *** ligne 2215
  sd Path1 := ( Root ; Branch1-1 ; Branch1-2 ; Branch1-3 ; *** ligne 2216
                Branch1-4 ; Branch1-5 ; Parall1-1 ; Branch1-8 ) . *** ligne 2217
  …
endsm)
```

Figure 6.14 : Une partie du module stratégique représentant les différents chemins du diagramme de DCSMA.

5.2 Validation formelle

Une fois la description basée-Maude du SMA soit générée, une validation par simulation devient possible. La figure 6.15 montre une configuration initiale dans laquelle un client appelé « Mazouz Salim » (ligne : 2321) choisit d'effectuer une opération de retrait d'un montant de € 500,00 (ligne 2323). Les trois agents sont initialisés à la première fois (lignes : 2326, 2328 et 2331).

```
2317 eq initialConfig =
2318    newTask(ShowWelcomeScreen)
2319    < "custPIN" : CustomerPIN | customerAccountNO : "b23070025156", PIN : 1234, customerName : "Mazouz Salim" >
2320    < "atmCard" : ATMCard | cardNO : "b23070025156", cardOwner : "Mazouz Salim" >
2321    < "customer" : Customer | name : "Mazouz Salim", adress : "PO Box n 3045 -04000- OEB, Algeria" >
2322    < "anAmount" : Amount | amountToWithDrawalOrDeposit : 500.0 >
2323    < "withdrawal" : WithDrawal | amount : 500.0, transactionID : "transS129", traAccountNumber : "8873997",
2324      traTransactionType : "WithDrawal" >
2325    < "medAgent" : Mediator | playsRole : noneRole, performsTask : noneTask,
2326         executesTaskAction : setup( EmptyParametersList ), currentState : Initialized >
2327    < "secRes" : SecurityResponsible | playsRole : noneRole,  performsTask : noneTask,
2328         executesTaskAction : setup( EmptyParametersList ), currentState : Initialized >
2329    < "traManagerAgent" : TransactionManager | transaction : "trans", playsRole : noneRole,
2330         performsTask : noneTask, executesTaskAction : setup( EmptyParametersList ),
2331         currentState : Initialized > .
```

Figure 6.15 : Une configuration initiale.

Les résultats de la simulation de la configuration initiale de la figure 6.15 en exécutant la stratégie « Path1 » (lignes 2215, 2216 et 2217 de la figure 6.14) sont montrés dans la figure 6.16. Cette stratégie illustre le cas dans lequel la carte insérée du client soit valide, le client soit authentifié et la transaction de retrait soit effectuée avec succès.

```
Elapsed time: 00:00:16.342
rewrites: 3733 in 6286818272ms cpu (719ms real) (0 rewrites/second)
rewrite with strategy :
result Configuration :
  FinalState(EmptyParametersList)< "IsAmountAvai" : IsAmountAvailable | isAmountAvailableValue : true >
  < "anAmount" : Amount |  amountToWithDrawalOrDeposit : 2.3e+4 >
  < "atmCard" : ATMCard | cardNO : "b2307025156",cardOwner : "customer" >
  < "customer" : Customer | adress : "PO Box n 3045 -04000- OEB, Algeria",name : "MazouzSalim" >
  < "isAmountDispensed" : IsAmountDispensed | isAmountDispensedValue : true >
  < "isAuthent" : IsAuthenticated | accountNUM : "CLE",isAuthValue : true >
  < "isCardVal" : IsCardValid | cardNo : " b2307025156"; isCardValValue : true >
  < "isTransPerf" : IsTransactionPerformed | aTransactionID : "transS129",isTransPerfValue : true >
  < "medAgent" : Mediator | currentState : EjectingCard,executesTaskAction : ejectCard(EmptyParametersList),
      performsTask : EjectCard,playsRole : Displayer >
  < "notif" : Notification | content : "<Withdrawal> Transaction has been performed succefully",notifID : "01" >
  < "secRes" : SecurityResponsible | currentState : SendingReportNotification,
      executesTaskAction : sendReportNotification("notif"),
      performsTask : ReportTransaction,playsRole : Saver >
  < "traManagerAgent" : TransactionManager | currentState : NotifyingForTransactionEnd,
      executesTaskAction : notifyEndOfTransaction("notif"),performsTask : NotifyEndOfTransaction,
      playsRole : Performer,transaction : "trans" >
  < "withdrawal" : WithDrawal | amount : 5.0e+4,traAccountNumber : "8873997", traNewBalance : 7.256e+4,
     traOldBalance : 7.256e+4,traTransactionDate : "27/11/2015",traTransactionTime : "16:57:12:5",
      transactionID : "transS129" >

Maude>
```

Figure 6.16 : Résultats de simulation (scénario bien passé) par la commande « srew initialConfig en utilisant la staratégie d'exécution « Path1 »).

Les résultats de cette phase donnent au développeur plus d'informations sur les agents à travers leurs états, les rôles qu'ils jouent, les tâches qu'ils effectuent et les actions qu'ils exécutent en plus de valeurs actuelles des attributs des éléments ontologiques. Par exemple, le prédicat « isCardVal » (encadré en noir dans la figure 6.16) nous donne l'information que la carte ATM insérée par le client est valdie.

5.3 Spécification formelle des propriétés du système

Dans cette phase, quelques propriétés du SMA sont identifiées et ensuite spécifiées dans la logique temporelle linéaire en tant que prédicats en Maude. La table 6.2 montre un ensemble de propriétés pour l'ATM et leurs spécifications dans LTL. Trois de ces propriétés doivent être satisfaites par le SMA (propriétés désirables). Par contre, les autres propriétés sont indésirables et le SMA ne devrait pas les satisfaire.

N°	Description de la propriété	Spécification en LTL (Maude)	Propriété désirable	Single/Multi Agent
1	Cette propriété exprime le fait que si l'agent médiateur lit le choix du client (l'etat : ReadingCustomer-Choice), il enverra éventuellement un ordre de transaction à l'agent Transaction-Manager.	MedAgent-ReadingCustomerChoice(medAgent) |-> <> MedAgent-AskingForATransaction(medAgent)	Oui	Single-Agent: Mediator
2	Cette propriété exprime le fait que si l'agent *TransactionManager* reçoit un ordre de transaction (l'état : ReceivingTransactionOrder), l'état « PerformingTheTransaction » exprimant qu'il effectue la transaction sera eu très bientôt.	TransManAgent-ReceivingTransactionOrder(transactionMan) |-> TransManAgent-PerformingTheTransaction(transactionMan)	Oui	Single-Agent: TransactionManager
3	Cette propriété exprime le fait que si l'agent SecurityResponsible reçoit un ordre de contrôle de carte, il notifiera directement les résultats de la vérification (sans l'avoir vérifié d'abord).	SecResAgent-ReceivingCardCheckingOrder (secRes) -> O SecResAgent-NotifyingCardCheckingResult (secRes)	Non	Single-Agent: SecurityResponsible
4	Cette propriété exprime le fait que si l'agent Mediator envoie un ordre de transaction à l'agent Transaction-Manager, ce dernier sera dans l'état ReceivingTransactionOrder.	MedAgent-AskingForATransaction(medAgent) -> O TransManAgent-ReceivingTransactionOrder(traManagerAgent)	Oui	Multi-Agent
5	Cette propriété exprime la situation : Toujours, si l'agent Mediator lit le choix du client, l'agent Transaction-Manager notifiera la fin de la transaction.	[] (MedAgent-ReadingCustomerChoice(medAgent) |-> TransManAgent-NotifyingForTransactionEnd(traManagerAgent))	Non	Multi-Agent

Table 6.2 : Quelques propriétés spécifiées pour l'ATM.

5.4 Vérification formelle des propriétés du système

Après la spécification des propriétés, une vérification à l'aide des techniques de vérification de modèles est appliquée. La figure 6.17 montre les résultats obtenus par le vérificateur de modèles du langage Maude. Dans les cas de propriétés désirables insatisfaites ou bien les propriétés indésirables satisfaites (propriétés n°3, n°5 de la table 6.2), le développeur est obligé d'examiner les diagrammes correspondants pour modification.

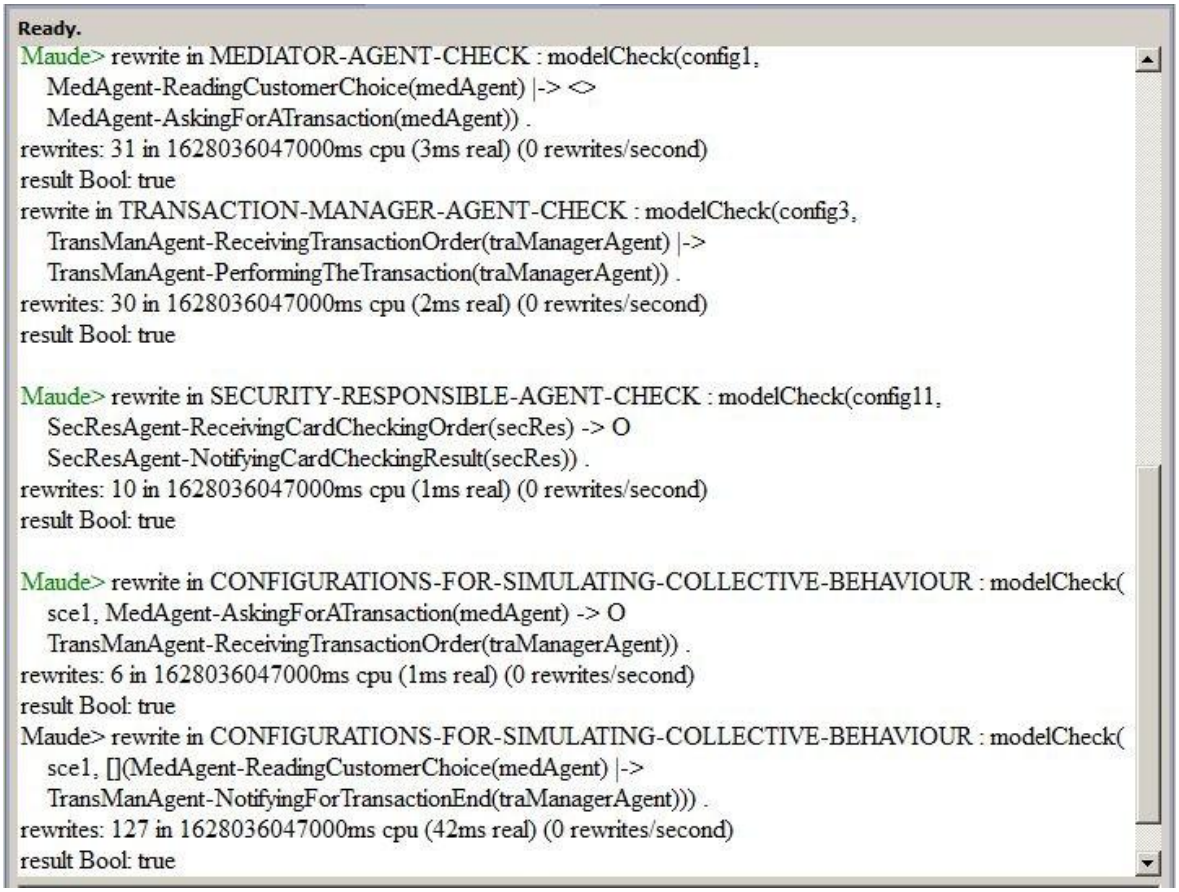

Figure 6.17 : Résultats de l'application du model-checking.

6. Conclusion

Après avoir décrit et conçu l'étude de cas que nous avons choisi pour valider la méthodologie F-PASSI, nous en avons appliqué le modèle formel proposé et des modules Maude générés grâce à l'outil F-PTK ont été montrés. L'extension formelle proposée a montré son utilité dans la détection des erreurs éventuelles qui sont faits dans les différents diagrammes de PASSI grâce à la description Maude générée. L'utilisation de la technique du model-checking pour vérifier des propriétés (niveaux d'abstraction différents : individuel/collectif) a son importance. L'outil F-PTK facilite la plupart des tâches et vise à une adoption vaste de la méthodologie F-PASSI.

Conclusion générale & Perspectives

Les systèmes multi-agents sont utilisés dans la modélisation, la simulation et le développement des systèmes distribués. Le développement de tels systèmes nécessite l'existence d'outils, langages, formalismes et méthodologies pour assister les développeurs durant les différentes phases de développement.

Dans la littérature, plusieurs méthodologies de développement de systèmes multi-agents ont été proposées comme, par exemple, Gaia, Tropos, Ingenias, ASEME, PASSI. Malgré que ces méthodologies (et autres) aient apporté des éléments de réponses importants dans le domaine du génie logiciel orienté agent, la plupart d'entre elles ne couvrent pas les phases principales du développement et sont basées sur des formalismes semi-formels.

La méthodologie PASSI, ayant un processus de développement incrémental et itératif, est l'une parmi les méthodologies qui couvrent la plupart des phases de développement, de l'analyse des besoins jusqu'à la phase du test. Elle est basée sur la notation semi-formelle, UML. Dans le cadre de cette thèse, nous avons opté pour formaliser PASSI à l'aide de Maude pour les avantages qu'elle procure. Outre qu'elle garantit un développement relativement complet, elle utilise plusieurs outils et langages intéressants et d'actualité comme JAVA, RDF, XML, UML, FIPA.

Dans cette thèse, nous avons proposé et discuté la version formelle de PASSI, F-PASSI (Formal-PASSI). Le processus de développement de F-PASSI intègre un nouveau modèle formel dans l'ancien processus. Basé sur le langage Maude, le processus d'élaboration de ce modèle est composé de quatre phases. La première phase vise à offrir une description formelle basée-Maude du système sous-développement en utilisant une technique très utile de l'ingénierie dirigée par les modèles, la transformation de modèles (M2M & M2T). La deuxième phase vise à valider la description formelle générée grâce à la validation par simulation supportée par l'outil Maude. Dans la troisième phase, le développeur spécifie des propriétés (désirables ou indésirables, individuelles ou collectives) du système en logique temporelle dans Maude. La quatrième phase vise à la vérification des propriétés spécifiées en utilisant une technique très puissante, la vérification de modèle grâce au vérificateur de modèles intégré dans Maude. En outre, des liens bidirectionnels de traçabilité ont été bien déterminés via le métà-modèle de traçabilité que nous avons proposé. F-PTK est l'outil que nous avons développé pour supporter F-PASSI.

On peut résumer l'importance du travail présenté dans cette thèse par les points suivants :

- ❖ La formalisation d'une méthodologie AOSE importante (ayant un processus incrémental et itératif qui couvre la majorité des phases de développement et qui adopte un nombre important de standards comme, UML, XML, RDF, JAVA, FIPA et), PASSI.
- ❖ La proposition d'un méta-modèle de traçabilité qui définit des liens de traçabilité bidirectionnels explicites entre les différents éléments conceptuels.
- ❖ Le développement d'un outil supportant la version formelle proposée.

Le travail présenté dans cette thèse devrait améliorer la méthodologie PASSI et combler ses lacunes en termes de formalité par l'utilisation d'un langage formel basé sur la logique de réécriture et en termes de productivité par l'utilisation d'une technique de transformation de modèle de l'ingénierie dirigée par les modèles.

En effet, notre projet de formalisation de PASSI reste ouvert. Comme perspectives, nous envisageons à courts et moyen termes :

- Introduire plus de diagrammes PASSI au processus de formalisation ;
- La formalisation de l'ensemble de patron de conceptions prédéfinis de PASSI (présentés dans la section 4.4.1 du chapitre 3) en utilisant Maude ;
- Redéveloppement de l'outil F-PTK en utilisant l'environnement libre et gratuit, Eclipse[1] et les différents projets relatifs comme Sirius[2]. Sirius se base fortement sur les techniques de l'ingénierie dirigée par les modèles comme la transformation de modèles, la méta-modélisation ;
- Améliorer l'outil F-PTK en y ajoutant la possibilité de visualiser et d'animer les résultats de la phase de validation formelle pour les rendre plus lisibles ;
- Proposer (ou utiliser) une notation graphique pour décrire les différents opérateurs de la logique temporelle linéaire afin de faciliter la phase de spécification formelle des propriétés du système aux développeurs qui ne sont pas familiarisés avec LTL.

[1]https://www.eclipse.org/home/
[2]http://www.eclipse.org/sirius/

Références

[1] Ferber, J. (1995). Les Systèmes multi-agents : vers une intelligence collective, InterEditions.

[2] Cilliers, P. (2002). Complexity and Postmodernism : Understanding Complex Systems, Routledge.

[3] Pressman, R. S. (2001). Software Engineering : A Practitioner's Approach, McGraw-Hill Higher Education.

[4] FP Brooks. No silver bullet : Essence and accidents of software engineering. IEEE Computer, 20(4) :10–19, 1987.

[5] Berry, D. M. and E. Kamsties (2004). Ambiguity in Requirements Specification. Perspectives on Software Requirements. J. C. S. do Prado Leite and J. H. Doorn. Boston, MA, Springer US : 7-44.

[6] D. Jackson. Dependable software by design. Scientific American - American Edition, 294(6) :68, 2006.

[7] Storey, N. R. (1996). Safety Critical Computer Systems, Addison-Wesley Longman Publishing Co., Inc.

[8] Wing, J. M. (1990). A Specifier's Introduction to Formal Methods. Computer 23(9) : 8-23.

[9] Hall, A. (1999). Using Formal Methods to Develop an ATC Information System. Industrial-Strength Formal Methods in Practice. M. G. Hinchey and J. P. Bowen. London, Springer London : 207-229.

[10] Zambonelli, F., et al. (2003). Developing multiagent systems : The Gaia methodology. ACM Trans. Softw. Eng. Methodol. 12(3) : 317-370.

[11] Cernuzzi, L., T. Juan, L. Sterling and F. Zambonelli (2004). The Gaia Methodology. Methodologies and Software Engineering for Agent Systems : The Agent-Oriented Software Engineering Handbook. F. Bergenti, M.-P. Gleizes and F. Zambonelli. Boston, MA, Springer US : 69-88.

[12] Cernuzzi, L., A. Molesini and A. Omicini (2014). The Gaia Methodology Process. Handbook on Agent-Oriented Design Processes. M. Cossentino, V. Hilaire, A. Molesini and V. Seidita. Berlin, Heidelberg, Springer Berlin Heidelberg : 141-172.

[13] Pavón, J. and J. Gómez-Sanz (2003). Agent Oriented Software Engineering with INGENIAS. Multi-Agent Systems and Applications III : 3rd International Central and Eastern European Conference on Multi-Agent Systems, CEEMAS 2003 Prague, Czech Republic, June 16–18, 2003 Proceedings. V. Mařík, M. Pěchouček and J. Müller. Berlin, Heidelberg, Springer Berlin Heidelberg : 394-403.

[14] Giorgini, P., et al. (2004). The Tropos Methodology. Methodologies and Software Engineering for Agent Systems : The Agent-Oriented Software Engineering Handbook. F. Bergenti, M.-P. Gleizes and F. Zambonelli. Boston, MA, Springer US: 89-106.

[15] Morandini, M., et al. (2014). The Tropos Software Engineering Methodology. Handbook on Agent-Oriented Design Processes. M. Cossentino, V. Hilaire, A. Molesini and V. Seidita. Berlin, Heidelberg, Springer Berlin Heidelberg : 463-490.

[16] Cossentino, M. (2005). From requirements to code with the PASSI methodology. In B. Henderson- Sellers & P. Giorgini (Eds.), Agent-oriented methodologies. Hershey, PA, USA : Idea Group Publishing : Chap. IV, pp. 79–106.

[17] Cossentino, M. and V. Seidita (2014). PASSI : Process for Agent Societies Specification and Implementation. Handbook on Agent-Oriented Design Processes. M. Cossentino, V. Hilaire, A. Molesini and V. Seidita, Springer Berlin Heidelberg : 287-329.

[18] Chella, A., M. Cossentino and L. Sabatucci (2004). Tools and patterns in designing multi-agent systems with PASSI. WSEAS Transactions on Communications 3(1) : 352-358.

[19] Basin, D., M. Clavel and J. Meseguer (2000). Rewriting Logic as a Metalogical Framework. FST TCS 2000 : Foundations of Software Technology and Theoretical Computer Science : 20th Conference New Delhi, India, December 13–15, 2000 Proceedings. S. Kapoor and S. Prasad. Berlin, Heidelberg, Springer Berlin Heidelberg : 55-80.

[20] Meseguer, J. (2005). A Rewriting Logic Sampler. Theoretical Aspects of Computing. ICTAC 2005 : Second International Colloquium, Hanoi, Vietnam, October 17-21, 2005. Proceedings. D. Hung and M. Wirsing. Berlin, Heidelberg, Springer Berlin Heidelberg : 1-28.

[21] Keller, R. M. (1976). Formal verification of parallel programs. Commun. ACM 19(7): 371-384.

[22] Murata, T. (1989). Petri nets : Properties, analysis and applications. Proceedings of the IEEE 77(4) : 541-580.

[23] Milner, R. (1982). A Calculus of Communicating Systems. Springer-Verlag New York, Inc.

[24] Diaconescu, R. and Futatsugi, K. (1998). CafeOBJ Report : The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification. AMAST Series in Computing, vol. 6. World Scientific.

[25] Borovanský, P., et al. (1996). ELAN : A logical framework based on computational systems. Electronic Notes in Theoretical Computer Science 4(Supplement C): 35-50.

[26] Clavel, M., F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and J. F. Quesada (2002). Maude : specification and programming in rewriting logic. Theoretical Computer Science 285(2): 187-243.

[27] Clavel, M., F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, José, Meseguer and C. Talcott (2007). All about maude - a high-performance logical framework : how to specify, program and verify systems in rewriting logic, Springer-Verlag.

[28] Mazouz, M., et al. (2015). Towards an Explicit Bidirectional Requirement-to-Code Traceability Meta-model for the PASSI Methodology. Proceedings of the International Conference on Agents and Artificial Intelligence - Volume 1. Lisbon, Portugal, SCITEPRESS - Science and Technology Publications, Lda : 203-209.

[29] Mazouz, M., Mokhati, F. and Badri, M., 2017. Formal Development of Multi-Agent Systems with FPASSI : Towards Formalizing PASSI Methodology using Rewriting Logic. Informatica, 41(2), 233-252.

[30] Sturm, A. and O. Shehory (2014). Agent-Oriented Software Engineering : Revisiting the State of the Art. Agent-Oriented Software Engineering : Reflections on Architectures, Methodologies, Languages, and Frameworks. O. Shehory and A. Sturm. Berlin, Heidelberg, Springer Berlin Heidelberg : 13-26.

[31] Woolridge, M. (2009). An Introduction to Multiagent Systems, Second Edition, John Wiley & Sons.

[32] Cycle de vie d'une spécification FIPA, http://www.fipa.org/specifications/lifecycle.html

[33] La spécification de l'architecture abstraite de la FIPA, http://www.fipa.org/specs/fipa00001/SC00001L.html

[34] La spécification officielle de la structure d'un message FIPA-ACL, http://www.fipa.org/specs/fipa00061/SC00061G.html

[35] La spécification officielle du langage du contenu FIPA-SL, http://www.fipa.org/specs/fipa00008/SC00008I.html

[36] La spécification officielle de la bibliothèque des actes communicatifs de la FIPA, http://www.fipa.org/specs/fipa00037/SC00037J.html

[37] La spécification officielle du protocole d'interaction « Request » de la FIPA, http://www.fipa.org/specs/fipa00026/SC00026H.html

[38] La spécification officielle du protocole d'interaction « Query » de la FIPA, http://www.fipa.org/specs/fipa00027/SC00027H.html

[39] La spécification officielle du protocole d'interaction « Contract Net » de la FIPA, http://www.fipa.org/specs/fipa00029/SC00029H.html

[40] La spécification officielle du protocole d'interaction « Propose » de la FIPA, http://www.fipa.org/specs/fipa00036/SC00036H.html

[41] Bauer, B., et al. (2001). Agent UML : A Formalism for Specifying Multiagent Software Systems. Agent-Oriented Software Engineering : First International Workshop, AOSE 2000 Limerick, Ireland, June 10, 2000 Revised Papers. P. Ciancarini and M. J. Wooldridge. Berlin, Heidelberg, Springer Berlin Heidelberg : 91-103.

[42] Huget, M.-P. (2003). Agent UML Class Diagrams Revisited. Agent Technologies, Infrastructures, Tools, and Applications for E-Services : NODe 2002 Agent-Related Workshops Erfurt, Germany, October 7–10, 2002 Revised Papers. J. G. Carbonell, J. Siekmann, R. Kowalczyk et al. Berlin, Heidelberg, Springer Berlin Heidelberg : 49-60.

[43] Poggi, A., et al. (2004). Modeling Deployment and Mobility Issues in Multiagent Systems Using AUML. Agent-Oriented Software Engineering IV: 4th InternationalWorkshop, AOSE 2003, Melbourne, Australia, July 15, 2003. Revised Papers. P. Giorgini, J. P. Müller and J. Odell. Berlin, Heidelberg, Springer Berlin Heidelberg : 69-84.

[44] Cervenka R. and I. Trencansky (2007). The AML Approach. The Agent Modeling Language - AML: A Comprehensive Approach to Modeling Multi-Agent Systems. Basel, Birkhäuser Basel : 31-36.

[45] La spécification officielle de l'OMG de UML 2.0, http://www.omg.org/spec/UML/2.0/About-UML

[46] La spécification officielle de l'OMG de UML 1.5, http://www.omg.org/spec/UML/1.5

[47] La spécification officielle de l'OMG de l'OCL, http://www.omg.org/spec/OCL/2.0/About-OCL

[48] Yu, E. S.-K. (1996). Modelling strategic relationships for process reengineering. University of Toronto.

[49] Padgham, L. and M. Winikoff (2003). Prometheus: A Methodology for Developing Intelligent Agents. Agent-Oriented Software Engineering III: Third International Workshop, AOSE 2002 Bologna, Italy, July 15, 2002 Revised Papers and Invited Contributions. F. Giunchiglia, J. Odell and G. Weiß. Berlin, Heidelberg, Springer Berlin Heidelberg: 174-185.

[50] Winikoff, M. and L. Padgham (2004). The Prometheus Methodology. Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook. F. Bergenti, M.-P. Gleizes and F. Zambonelli. Boston, MA, Springer US: 217-234.

[51] Spanoudakis, N. and P. Moraitis (2008). The Agent Modeling Language (AMOLA). Artificial Intelligence: Methodology, Systems, and Applications: 13th International Conference, AIMSA 2008, Varna, Bulgaria, September 4-6, 2008. Proceedings. D. Dochev, M. Pistore and P. Traverso. Berlin, Heidelberg, Springer Berlin Heidelberg: 32-44.

[52] Wilson Mwaluseke, G. and J. Bowen (2012). Combination of formal and semi-formal methods.

[53] Alagar, V. S. and K. Periyasamy (1998). The Z Notation. Specification of Software Systems. New York, NY, Springer New York: 281-360.

[54] Smith, G. and Q. Li (2014). MAZE: An Extension of Object-Z for Multi-Agent Systems, Berlin, Heidelberg, Springer Berlin Heidelberg.

[55] Alagar, V. S. and K. Periyasamy (2011). The Object-Z Specification Language. Specification of Software Systems. London, Springer London: 539-576.

[56] Brandão, A. A. F., et al. (2005). AgentZ: Extending Object-Z for Multi-agent Systems Specification, Berlin, Heidelberg, Springer Berlin Heidelberg.

[57] Jean-Raymond Abrial (1996). The B-Book, Assigning Programs to Meanings. Cambridge University Press, Cambridge.

[58] Fadil, H. and J.-L. Koning (2005). A Formal Approach to Model Multiagent Interactions Using the B Formal Method. Advanced Distributed Systems : 5th International School and Symposium, ISSADS 2005, Guadalajara, Mexico, January 24-28, 2005, Revised Selected Papers. F. F. Ramos, V. Larios Rosillo and H. Unger. Berlin, Heidelberg, Springer Berlin Heidelberg: 516-528.

[59] Jemni Ben Ayed, L. and F. Siala (2008). Specification and Verification of Multi-agent Systems Interaction Protocols Using a Combination of AUML and Event B. Interactive Systems. Design, Specification, and Verification : 15th International Workshop, DSV-IS 2008 Kingston, Canada, July 16-18, 2008 Revised Papers. T. C. N. Graham and P. Palanque. Berlin, Heidelberg, Springer Berlin Heidelberg : 102-107.

[60] Abrial, J.-R. (2010). Modelling in Event-B: System and Software Engineering, Cambridge University Press.

[61] Farid Mokhati, Brahim Sahraoui, Soufiane Bouzaher, Mohamed Tahar Kimour (2010). A Tool for Specifying and Validating Agents' Interaction Protocols: From Agent UML to Maude. Journal of Object Technology 9(3): 59-77.

[62] Mohamed Amin, L., et al. (2017). A formal framework for organization- centered multi-agent system specification: A rewriting logic-based approach. Multiagent and Grid Systems 13(4) : 395-419.

[63] C. A. Petri (1962), Kommunikation mit Automaten. PhD thesis, University of Bonn, Bonn, Germany.

[64] Peterson, J. (1981). Petri Net Theory and the Modeling of Systems, Prentice Hall PTR.

[65] Ma, B. (2005). Modeling Multi-Agent Systems with Hierarchical Colored Petri Nets. Artificial Intelligence Applications and Innovations: IFIP TC12 WG12.5 - Second IFIP Conference on Artificial Intelligence Applications and Innovations (AIAI2005), September 7–9, 2005, Beijing, China. D. Li and B. Wang. Boston, MA, Springer US: 167-171.

[66] Jensen, K. (1992). Hierarchical Coloured Petri Nets. Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use Volume 1. Berlin, Heidelberg, Springer Berlin Heidelberg: 89-121.

[67] Zhu, W. and Q. Fei (2010). A Petri-Net Modeling Method of Agent's Belief-Desire-Intention and Its Application in Logistics. Advances in Neural Network Research and Applications. Z. Zeng and J. Wang. Berlin, Heidelberg, Springer Berlin Heidelberg: 837-844.

[68] Poslad, S., et al. (2000). The FIPA-OS agent platform: Open Source for Open Standards.

[69] Winikoff, M. (2005). Jack™ Intelligent Agents: An Industrial Strength Platform. Multi-Agent Programming: Languages, Platforms and Applications. R. H. Bordini, M. Dastani, J. Dix and A. El Fallah Seghrouchni. Boston, MA, Springer US : 175-193.

[70] Jacques Ferber (2009), MadKit pas à pas, Démarrage et prise en main du logiciel MadKit, LIRMM - Université de Montpellier II.

[71] Lemercier, M. and D. Gaïti (2002). A New Organisational Framework for Network Modelling Using a Multi-Agent System. Smart Networks : IFIP TC6 / WG6.7 Seventh International Conference on Intelligence in Networks (SmartNet 2002) April 8–10, 2002, Saariselkä, Lapland, Finland. O. Martikainen, K. Raatikainen and J. Hyvärinen. Boston, MA, Springer US : 113-128.

[72] Boss, N. S., et al. (2010). Building multi-agent systems using Jason. Annals of Mathematics and Artificial Intelligence 59(3) : 373-388.

[73] Bellifemine, F. L., et al. (2007). Developing Multi-Agent Systems with JADE (Wiley Series in Agent Technology), John Wiley \& Sons.

[74] Bellifemine, F., et al. (2005). Jade — A Java Agent Development Framework. Multi-Agent Programming : Languages, Platforms and Applications. R. H. Bordini, M. Dastani, J. Dix and A. El Fallah Seghrouchni. Boston, MA, Springer US : 125-147.

[75] F. Bellifemine, G. Caire, A. Poggi, G. Rimassa (2002). Jade (White paper). URI : http://jade.tilab.com/papers/2003/WhitePaperJADEEXP.pdf

[76] Pokahr, A., et al. (2005). Jadex : A BDI Reasoning Engine. Multi-Agent Programming : Languages, Platforms and Applications. R. H. Bordini, M. Dastani, J. Dix and A. El Fallah Seghrouchni. Boston, MA, Springer US : 149-174.

[77] Braun, P., et al. (2005). Integrating a New Mobility Service into the Jade Agent Toolkit. Mobility Aware Technologies and Applications : Second International Workshop, MATA 2005, Montreal, Canada, October 17-19, 2005. Proceedings. T. Magedanz, A. Karmouch, S. Pierre and I. Venieris. Berlin, Heidelberg, Springer Berlin Heidelberg : 354-363.

[78] Madrigal-Mora, C. and K. Fischer (2009). Adding Organisations and Roles to JADE with JadeOrgs. Agent-Based Technologies and Applications for Enterprise Interoperability : International Workshops, ATOP 2005 Utrecht, The Netherlands, July 25-26, 2005, and ATOP 2008, Estoril, Portugal, May 12-13, 2008, Revised Selected Papers. K. Fischer, J. P. Müller, J. Odell and A. J. Berre. Berlin, Heidelberg, Springer Berlin Heidelberg : 98-117.

[79] Madrigal-Mora, C., et al. (2008). Implementing Organisations in JADE. Multiagent System Technologies : 6th German Conference, MATES 2008, Kaiserslautern, Germany, September 23-26, 2008. Proceedings. R. Bergmann, G. Lindemann, S. Kirn and M. Pěchouček. Berlin, Heidelberg, Springer Berlin Heidelberg : 135-146.

[80] S, Zerrougui., et al. (2017). A novel approach for scalability and performance enhancement in JADE. Journal of Multiagent and Grid Systems, volume 13, number 2 : 177-201.

[81] Weihong, Y. and Y. Chen (2013). The Development of Jade Agent for Android Mobile Phones. Proceedings of the 2012 International Conference on Information Technology and Software Engineering : Software Engineering & Digital Media Technology. W. Lu, G. Cai, W. Liu and W. Xing. Berlin, Heidelberg, Springer Berlin Heidelberg : 215-222.

[82] Moreno, A., et al. (2017). Using JADE-LEAP implement agents in mobile devices.

[83] Mannava, V., et al. (2011). A Novel Way of Providing Dynamic Adaptability and Invocation of JADE Agent Services from P2P JXTA Using Aspect Oriented Programming. Advances in Network Security and Applications : 4th International Conference, CNSA 2011, Chennai, India, July 15-17, 2011. D. C. Wyld, M. Wozniak, N. Chaki, N. Meghanathan and D. Nagamalai. Berlin, Heidelberg, Springer Berlin Heidelberg : 552-563.

[84] Bojic, I., et al. (2011). Extending the JADE Agent Behaviour Model with JBehaviourTrees Framework. Agent and Multi-Agent Systems : Technologies and

Applications : 5th KES International Conference, KES-AMSTA 2011, Manchester, UK, June 29 – July 1, 2011. Proceedings. J. O'Shea, N. T. Nguyen, K. Crockett, R. J. Howlett and L. C. Jain. Berlin, Heidelberg, Springer Berlin Heidelberg : 159-168.

[85] Bergenti, F., et al. (2017). Agent-oriented model-driven development for JADE with the JADEL programming language. Computer Languages, Systems & Structures 50 : 142-158.

[86] Pokahr, A. and L. Braubach (2009). A Survey of Agent-oriented Development Tools. Multi-Agent Programming : Languages, Tools and Applications. A. El Fallah Seghrouchni, J. Dix, M. Dastani and H. R. Bordini. Boston, MA, Springer US : 289-329.

[87] Padgham, L., et al. (2008). The Prometheus Design Tool – A Conference Management System Case Study. Agent-Oriented Software Engineering VIII : 8th International Workshop, AOSE 2007, Honolulu, HI, USA, May 14, 2007, Revised Selected Papers. M. Luck and L. Padgham. Berlin, Heidelberg, Springer Berlin Heidelberg : 197-211.

[88] Sun, H., et al. (2010). Eclipse-based Prometheus design tool. Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems : volume 1 - Volume 1. Toronto, Canada, International Foundation for Autonomous Agents and Multiagent Systems: 1769-1770.

[89] R. W. Collier (2001). Agent Factory: A Framework for the Engineering of Agent-Oriented Applications. PhD thesis, Department of Computer Science, University College Dublin.

[90] Morandini, M., et al. (2011). Tropos Modeling, Code Generation and Testing with the Taom4E Tool.

[91] Perini, A. and A. Susi (2004). Developing Tools for Agent-Oriented Visual Modeling. Multiagent System Technologies: Second German Conference, MATES 2004, Erfurt, Germany, September 29-30, 2004. Proceedings. G. Lindemann, J. Denzinger, I. J. Timm and R. Unland. Berlin, Heidelberg, Springer Berlin Heidelberg : 169-182.

[92] García-Magariño, I., et al. (2009). The INGENIAS Development Kit : A Practical Application for Crisis-Management. Bio-Inspired Systems : Computational and Ambient Intelligence : 10th International Work-Conference on Artificial Neural

Networks, IWANN 2009, Salamanca, Spain, June 10-12, 2009. Proceedings, Part I. J. Cabestany, F. Sandoval, A. Prieto and J. M. Corchado. Berlin, Heidelberg, Springer Berlin Heidelberg : 537-544.

[93] Caire, G., et al. (2004). The Message Methodology. Methodologies and Software Engineering for Agent Systems : The Agent-Oriented Software Engineering Handbook. F. Bergenti, M.-P. Gleizes and F. Zambonelli. Boston, MA, Springer US : 177-194.

[94] Picard, G. and M.-P. Gleizes (2004). The ADELFE Methodology. Methodologies and Software Engineering for Agent Systems : The Agent-Oriented Software Engineering Handbook. F. Bergenti, M.-P. Gleizes and F. Zambonelli. Boston, MA, Springer US : 157-175.

[95] Bonjean, N., et al. (2014). ADELFE 2.0. Handbook on Agent-Oriented Design Processes. M. Cossentino, V. Hilaire, A. Molesini and V. Seidita. Berlin, Heidelberg, Springer Berlin Heidelberg : 19-63.

[96] Mefteh, W., et al. (2015). ADELFE 3.0 Design, Building Adaptive Multi Agent Systems Based on Simulation a Case Study, Cham, Springer International Publishing.

[97] Rodriguez, L., et al. (2009). Improving the Quality of Agent-Based Systems : Integration of Requirements Modeling into Gaia. 2009 Ninth International Conference on Quality Software.

[98] Moraitis, P. and N. I. Spanoudakis (2006). The Gaia2Jade process for multi-agent systems development. Applied Artificial Intelligence 20: 251-273.

[99] Juan, T., et al. (2002). ROADMAP : extending the gaia methodology for complex open systems. AAMAS.

[100] Topçu, O., et al. (2016). Model Driven Engineering. Distributed Simulation : A Model Driven Engineering Approach. Cham, Springer International Publishing : 23-38.

[101] Gaševic, D., et al. (2009). Model Driven Engineering. Model Driven Engineering and Ontology Development. Berlin, Heidelberg, Springer Berlin Heidelberg : 125-155.

[102] Object Management Group, Meta Object Facility (MOF) Specification, https://www.omg.org/spec/MOF/About-MOF

[103] Sendall, S. and W. Kozaczynski (2003). Model transformation : the heart and soul of model-driven software development. IEEE Software 20(5): 42-45.

[104] Di Ruscio, D., et al. (2012). Model Transformations. Formal Methods for Model-Driven Engineering : 12th International School on Formal Methods for the esign of Computer, Communication, and Software Systems, SFM 2012, Bertinoro, Italy, June 18-23, 2012. Advanced Lectures. M. Bernardo, V. Cortellessa and A. Pierantonio. Berlin, Heidelberg, Springer Berlin Heidelberg : 91-136.

[105] Elmounadi, A., et al. (2016). Smart Text to Model Transformation a Graph Based Approach to Cover Dynamic Analysis. International Review on Computers and Software (IRECOS) ; Vol 11, No 4 (2016).

[106] Hidaka, S., et al. (2016). Feature-based classification of bidirectional transformation approaches. Software & Systems Modeling 15(3) : 907-928.

[107] Kahani, N., et al. (2018). Survey and classification of model transformation tools. Software & Systems Modeling.

[108] Mellor, S. J., et al. (2002). Model-Driven Architecture, Berlin, Heidelberg, Springer Berlin Heidelberg.

[109] Kleppe, A., Warmer, J. and Bast, W. (2003) MDA Explained : The Model Driven Architecture : Practice and Promise. Addison-Wesley, Boston.

[110] S. Shekhar, H. Xiong and X. Zhou. Cham (2017). Model Driven Architecture. Encyclopedia of GIS, Springer International Publishing : 1295-1295.

[111] Morandini, M., et al. (2008). Towards goal-oriented development of self-adaptive systems. Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems. Leipzig, Germany, ACM : 9-16.

[112] Morandini, M., et al. (2008). Automated Mapping from Goal Models to Self-Adaptive Systems. 2008 23rd IEEE/ACM International Conference on Automated Software Engineering.

[113] Bertolini, D., et al. (2006). A Tropos Model-Driven Development Environment.

[114] Bernon, C., et al. (2003). ADELFE : A Methodology for Adaptive Multi-agent Systems Engineering. Engineering Societies in the Agents World III : Third International Workshop, ESAW 2002 Madrid, Spain, September 16–17, 2002 Revised Papers. P. Petta, R. Tolksdorf and F. Zambonelli. Berlin, Heidelberg, Springer Berlin Heidelberg : 156-169.

[115] Capera, D., et al. (2003). The AMAS theory for complex problem solving based on self-organizing cooperative agents. WET ICE 2003. Proceedings. Twelfth IEEE International Workshops on Enabling Technologies : Infrastructure for Collaborative Enterprises, 2003.

[116] Rougemaille, S., et al. (2009). ADELFE Design, AMAS-ML in Action, Berlin, Heidelberg, Springer Berlin Heidelberg.

[117] Rougemaille, S., et al. (2007). Model Driven Engineering for Designing Adaptive Multi-Agents Systems.

[118] Leriche, S. and J. P. Arcangeli (2007). Adaptive Autonomous Agent Models for Open Distributed Systems. Computing in the Global Information Technology, 2007. ICCGI 2007. International Multi-Conference on.

[119] Padgham, L. and M. Winikoff (2005). Prometheus : A practical agent-oriented methodology.

[120] Winikoff, M. (2007). Defining syntax and providing tool support for Agent UML using a textual notation. Int. J. Agent-Oriented Softw. Eng. 1(2): 123-144.

[121] Padgham, L., et al. (2007). AUML protocols and code generation in the Prometheus design tool. Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems. Honolulu, Hawaii, ACM : 1-2.

[122] Pavón, J. (2006), INGENIAS : Développement Dirigé par Modèles des Systèmes Multi-Agents. Dossier d'habilitation à diriger des recherches, Université de Pierre et Marie Curie (UPMC depuis 2007). France.

[123] Gómez-Sanz, J. J., et al. (2009). Testing and Debugging of MAS Interactions with INGENIAS, Berlin, Heidelberg, Springer Berlin Heidelberg.

[124] Pavón, J., et al. (2006). Model Driven Development of Multi-Agent Systems, Berlin, Heidelberg, Springer Berlin Heidelberg.

[125] Graja, Z., et al. (2011). ForMAAD : Towards a Model Driven Approach for Agent Based Application Design, Berlin, Heidelberg, Springer Berlin Heidelberg.

[126] Regayeg, A., Hadj-Kacem, A., Jmaiel, M. (2004). Specification and Verification of Multi-Agent Applications using Temporal Z. In Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology 2004 (IAT'2004), Beijing, China : 260–266.

[127] Saaltink, M. (1997). The Z/EVES System. Proceedings of the 10th International Conference of Z Users on The Z Formal Specification Notation, Springer-Verlag : 72-85.

[128] Spanoudakis, N. and P. Moraitis (2007). The Agent Systems Methodology (ASEME) : A Preliminary Report.

[129] Spanoudakis, N. and P. Moraitis (2011). Using ASEME Methodology for Model-Driven Agent Systems Development. Agent-Oriented Software Engineering XI : 11th International Workshop, AOSE 2010, Toronto, Canada, May 10-11, 2010, Revised Selected Papers. D. Weyns and M.-P. Gleizes. Berlin, Heidelberg, Springer Berlin Heidelberg : 106-127.

[130] Woodcock, J., et al. (2009). Formal methods : Practice and experience. ACM Comput. Surv. 41(4) : 1-36.

[131] El Fallah-Seghrouchni, A., et al. (2011). Formal Methods in Agent-Oriented Software Engineering. Agent-Oriented Software Engineering X : 10th International Workshop, AOSE 2009, Budapest, Hungary, May 11-12, 2009, Revised Selected Papers. M.-P. Gleizes and J. J. Gomez-Sanz. Berlin, Heidelberg, Springer Berlin Heidelberg : 213-228.

[132] Vasconcelos, W., et al. (2004). Rapid Prototyping of Large Multi-Agent Systems Through Logic Programming. Annals of Mathematics and Artificial Intelligence 41(2): 135-169.

[133] Fuxman, A., et al. (2004). Specifying and analyzing early requirements in Tropos. Requirements Engineering 9(2) : 132-150.

[134] Dardenne, A., et al. (1993). Goal-directed requirements acquisition. Selected Papers of the Sixth International Workshop on Software Specification and Design, Elsevier Science Publishers B. V. : 3-50.

[135] Cimatti, A., E. Clarke, F. Giunchiglia and M. Roveri (2000). NUSMV : a new symbolic model checker. International Journal on Software Tools for Technology Transfer 2(4) : 410-425.

[136] Hadj-Kacem, A., et al. (2007). ForMAAD : A formal method for agent-based application design. Web Intelli. And Agent Sys. 5(4) : 435-454.

[137] Regayeg, Amira & Kallel, Slim & Hadj Kacem, Ahmed & Jmaiel, Mohamed. (2006). ForMAAD Method: An Experimental Design for Air Traffic Control. ITSSA. 1. 327-334.

[138] Ball, E. and M. Butler (2006). Using Decomposition to Model Multi-agent Interaction Protocols in Event-B. FM'06 Doctoral Symposium.

[139] Ball, E. and M. Butler (2009). Event-B Patterns for Specifying Fault-Tolerance in Multi-agent Interaction. Methods, Models and Tools for Fault Tolerance. M. Butler, C. Jones, A. Romanovsky and E. Troubitsyna. Berlin, Heidelberg, Springer Berlin Heidelberg: 104-129.

[140] Burrafato, P. and M. Cossentino (2002). Designing a multi-agent solution for a bookstore with the PASSI methodology. AOIS@CAiSE.

[141] Cubillos, C., et al. (2007). Design of an Agent-Based System for Passenger Transportation Using PASSI. Nature Inspired Problem-Solving Methods in Knowledge Engineering : Second International Work-Conference on the Interplay Between Natural and Artificial Computation, IWINAC 2007, La Manga del Mar Menor, Spain, June 18-21, 2007, Proceedings, Part II. J. Mira and J. R. Álvarez. Berlin, Heidelberg, Springer Berlin Heidelberg : 531-540.

[142] Nunes, I., et al. (2009). Extending PASSI to model multi-agent systems product lines. Proceedings of the 2009 ACM symposium on Applied Computing. Honolulu, Hawaii, ACM : 729-730.

[143] Cossentino, M., et al. (2004). Patterns Reuse in the PASSI Methodology. Engineering Societies in the Agents World IV : 4th International Workshops, ESAW 2003, London, UK, October 29-31, 2003. Revised Selected and Invited Papers. A. Omicini, P. Petta and J. Pitt. Berlin, Heidelberg, Springer Berlin Heidelberg : 294-310.

[144] Sabas, A., et al. (2002). A Comparative Analysis of Multiagent System Development Methodologies : Towards a Unified Approach.

[145] Dam, K. H. and M. Winikoff (2004). Comparing Agent-Oriented Methodologies. Agent-Oriented Information Systems : 5th International Bi-Conference Workshop, AOIS 2003, Melbourne, Australia, July 14, 2003 and Chicago, IL, USA, October 13, 2003, Revised Selected Papers. P. Giorgini, B. Henderson-Sellers and M. Winikoff. Berlin, Heidelberg, Springer Berlin Heidelberg : 78-93.

[146] Kinny, D., et al. (1996). A methodology and modelling technique for systems of BDI agents, Berlin, Heidelberg, Springer Berlin Heidelberg.

[147] Collinot, A., Drogoul, A., and Benhamou, P. Agent-oriented Design of a Soccer Robot Team, in Proceedings of the Second International Conference on Multi-Agent systems (ICMAS-96), 41-47, Kyoto, Japan, December 1996.

[148] Kendall, E. A., et al. (1996). A Methodology for Developing Agent Based Systems for Enterprise Integration. Modelling and Methodologies for Enterprise Integration : Proceedings of the IFIP TC5 Working Conference on Models and Methodologies for Enterprise Integration, Queensland, Australia, November 1995. P. Bernus and L. Nemes. Boston, MA, Springer US : 333-344.

[149] Elammari, M. and W. Lalonde (2000). An Agent-Oriented Methodology : High-Level and Intermediate Models. Proceedings of AOIS-1999, Heidelberg June 1999.

[150] Iglesias, C. A., et al. (1998). Analysis and design of multiagent systems using MAS-CommonKADS, Berlin, Heidelberg, Springer Berlin Heidelberg.

[151] Moulin, B. and M. Brassard (1996). A scenario-based design method and an environment for the development of multiagent systems, Berlin, Heidelberg, Springer Berlin Heidelberg.

[152] DeLoach, S. A. (2004). The MaSE Methodology. Methodologies and Software Engineering for Agent Systems : The Agent-Oriented Software Engineering Handbook. F. Bergenti, M.-P. Gleizes and F. Zambonelli. Boston, MA, Springer US : 107-125.

[153] Glaser, N. (1997). The CoMoMAS methodology and environment for multi-agent system development, Berlin, Heidelberg, Springer Berlin Heidelberg.

[154] Tran, Q.-N. N., et al. (2003). A Feature Analysis Framework for Evaluating Multi-agent System Development Methodologies, Berlin, Heidelberg, Springer Berlin Heidelberg.

[155] Sturm, A. and O. Shehory (2004). A Framework for Evaluating Agent-Oriented Methodologies, Berlin, Heidelberg, Springer Berlin Heidelberg.

[156] Tran, Q.-N. N., et al. (2005). A Preliminary Comparative Feature Analysis of Multi-agent Systems Development Methodologies, Berlin, Heidelberg, Springer Berlin Heidelberg.

[157] Sudeikat, J., et al. (2005). Evaluation of Agent–Oriented Software Methodologies – Examination of the Gap Between Modeling and Platform, Berlin, Heidelberg, Springer Berlin Heidelberg.

[158] Abdelaziz, T. H. S., et al. (2007). A Framework for the Evaluation of Agent-Oriented Methodologies. 2007 Innovations in Information Technologies (IIT) : 491-495.

[159] Al-Hashel, E., et al. (2007). A Comparison of Three Agent-Oriented Software Development Methodologies : ROADMAP, Prometheus, and MaSE, Berlin, Heidelberg, Springer Berlin Heidelberg.Al-Hashel, E., et al. (2007). A Comparison of Three Agent-Oriented Software Development Methodologies : ROADMAP, Prometheus, and MaSE, Berlin, Heidelberg, Springer Berlin Heidelberg.

[160] Cernuzzi, L. and F. Zambonelli (2008). Profile based comparative analysis for AOSE methodologies evaluation. Proceedings of the 2008 ACM symposium on Applied computing. Fortaleza, Ceara, Brazil, ACM : 60-65

[161] Elamy, A.-H. H. and B. Far (2008). On the Evaluation of Agent-Oriented Software Engineering Methodologies : A Statistical Approach, Berlin, Heidelberg, Springer Berlin Heidelberg.

[162] Far, B. H. (2002). Agent-SE : A Methodology for Agent Oriented Software Engineering, Tokyo, Springer Japan.

[163] Lind, Jürgen (2001). The Conceptual Framework of Massive. Iterative Software Engineering for Multiagent Systems : The MASSIVE Method. J. Lind. Berlin, Heidelberg, Springer Berlin Heidelberg : 97-120.

[164] Yubo, J., et al. (2009). A comparison of three agent-oriented software development methodologies : MaSE, Gaia, and Tropos. 2009 IEEE Youth Conference on Information, Computing and Telecommunication.

[165] Julian, V. and V. Botti (2004). Developing real-time multi-agent systems. Integr. Comput. -Aided Eng. 11(2) : 135-149.

[166] Garcia, E., et al. (2009). Masev (Multiagent System Software Engineering Evaluation Framework), Berlin, Heidelberg, Springer Berlin Heidelberg.

[167] Botti, V. and A. Giret (2008). ANEMONA : A Multi-agent Methodology for Holonic Manufacturing Systems, Springer Publishing Company, Incorporated.

[168] Argente, E., et al. (2009). MAS Modeling Based on Organizations, Berlin, Heidelberg, Springer Berlin Heidelberg.

[169] Sturm, A. and O. Shehory (2014). The Landscape of Agent-Oriented Methodologies. Agent-Oriented Software Engineering : Reflections on Architectures, Methodologies, Languages, and Frameworks. O. Shehory and A. Sturm. Berlin, Heidelberg, Springer Berlin Heidelberg : 137-154.

[170] Riesco, A. (2014). An Integration of CafeOBJ into Full Maude. Rewriting Logic and Its Applications : 10th International Workshop, WRLA 2014, Held as a Satellite Event of ETAPS, Grenoble, France, April 5-6, 2014, Revised Selected Papers. S. Escobar. Cham, Springer International Publishing : 230-246.

[171] Riesco, A., et al. (2016). CafeInMaude : A CafeOBJ Interpreter in Maude. Fundamental Approaches to Software Engineering : 19th International Conference, FASE 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2–8, 2016, Proceedings. P. Stevens and A. Wąsowski. Berlin, Heidelberg, Springer Berlin Heidelberg : 377-380.

[172] Riesco, A., et al. (2017). A Maude environment for CafeOBJ. Formal Aspects of Computing 29(2) : 309-334.

[173] Clavel, M., et al. (2007). A Sampler of Application Areas. All About Maude - A High-Performance Logical Framework : How to Specify, Program and Verify Systems in Rewriting Logic. Berlin, Heidelberg, Springer Berlin Heidelberg : 645-665.

[174] Cirstea, H. and C. Kirchner (1998). Combining Higher-Order & First-Order Computation Using ρ-calculus : Towards a semantics of ELAN Full-version.

[175] Meseguer, J. (1996). Rewriting logic as a semantic framework for concurrency : a progress report. CONCUR '96 : Concurrency Theory : 7th International Conference Pisa, Italy, August 26–29, 1996 Proceedings. U. Montanari and V. Sassone. Berlin, Heidelberg, Springer Berlin Heidelberg : 331-372.

[176] José Meseguer, et al. (1992). Conditional rewriting logic as a unified model of concurrency. Theor. Comput. Sci. 96(1) : 73-155.

[177] Stehr, M.-O., et al. (2001). Rewriting Logic as a Unifying Framework for Petri Nets. Unifying Petri Nets : Advances in Petri Nets. H. Ehrig, J. Padberg, G. Juhás and G. Rozenberg. Berlin, Heidelberg, Springer Berlin Heidelberg : 250-303.

[178] Thati, P., et al. (2004). An Executable Specification of Asynchronous Pi-Calculus Semantics and May Testing in Maude 2.0. Electronic Notes in Theoretical Computer Science 71(Supplement C) : 261-281.

[179] Moller, F. and G. Struth (2013). Propositional Logic. Modelling Computing Systems : Mathematics for Computer Science. London, Springer London : 17-55.

[180] Clavel, M. (2000). Reflection in Rewriting Logic Metalogical Foundations and Metaprogramming Applications.

[181] Durán, F. and J. Meseguer (2010). A Church-Rosser Checker Tool for Conditional Order-Sorted Equational Maude Specifications. Rewriting Logic and Its Applications : 8th International Workshop, WRLA 2010, Held as a Satellite Event of ETAPS 2010, Paphos, Cyprus, March 20-21, 2010, Revised Selected Papers. P. C. Ölveczky. Berlin, Heidelberg, Springer Berlin Heidelberg : 69-85.

[182] Durán, F. and J. Meseguer (2010). A Maude Coherence Checker Tool for Conditional Order-Sorted Rewrite Theories. Rewriting Logic and Its Applications : 8th International Workshop, WRLA 2010, Held as a Satellite Event of ETAPS 2010, Paphos, Cyprus, March 20-21, 2010, Revised Selected Papers. P. C. Ölveczky. Berlin, Heidelberg, Springer Berlin Heidelberg : 86-103.

[183] Durán, F., et al. (2008). MTT : The Maude Termination Tool (System Description). Automated Reasoning : 4th International Joint Conference, IJCAR 2008 Sydney, Australia, August 12-15, 2008 Proceedings. A. Armando, P. Baumgartner and G. Dowek. Berlin, Heidelberg, Springer Berlin Heidelberg : 313-319.

[184] Farzan, A., et al. (2004). Formal Analysis of Java Programs in JavaFAN. Computer Aided Verification : 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004. Proceedings. R. Alur and D. A. Peled. Berlin, Heidelberg, Springer Berlin Heidelberg : 501-505.

[185] Denker, G., et al. (1999). Specification and Analysis of a Reliable Broadcasting Protocol in Maude.

[186] Wirsing, M. and A. Knapp (2002). A formal approach to object-oriented software engineering. Theoretical Computer Science 285(2): 519-560.

[187] Rademaker, A., et al. (2005). A Rewriting Semantics for a Software Architecture Description Language. Electronic Notes in Theoretical Computer Science 130 : 345-377.

[188] José Meseguer, et al. (2009). Formal Specification and Analysis of Domain Specific Models Using Maude. Simulation 85(11-12) : 778-792.

[189] Maude manual 1 : http://maude.cs.uiuc.edu/maude1/manual/maude-manual-html/maude-manual_14 .html

[190] Maude manual 2 : http://maude.cs.uiuc.edu/maude2-manual/html/maude-manualch3.html

[191] Baier, C. and J.-P. Katoen (2008). Principles of Model Checking (Representation and Mind Series), The MIT Press

[192] Eker, S., et al. (2003). The Maude LTL Model Checker and Its Implementation. Model Checking Software : 10th International SPIN Workshop Portland, OR, USA, May 9–10, 2003 Proceedings. T. Ball and S. K. Rajamani. Berlin, Heidelberg, Springer Berlin Heidelberg : 230-234.

[193] Clavel, M., et al. (2007). Some Tools. All About Maude - A High-Performance Logical Framework : How to Specify, Program and Verify Systems in Rewriting Logic. Berlin, Heidelberg, Springer Berlin Heidelberg : 667-693.

[194] Ölveczky, P. C. and J. Meseguer (2004). Specification and Analysis of Real-Time Systems Using Real-Time Maude. Fundamental Approaches to Software Engineering : 7th International Conference, FASE 2004. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004. Proceedings. M. Wermelinger and T. Margaria-Steffen. Berlin, Heidelberg, Springer Berlin Heidelberg : 354-358.

[195] Ölveczky, P. C. and J. Meseguer (2008). The Real-Time Maude Tool. Tools and Algorithms for the Construction and Analysis of Systems : 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. C. R. Ramakrishnan and J. Rehof. Berlin, Heidelberg, Springer Berlin Heidelberg : 332-336.

[196] Ölveczky, P. C., et al. (2001). Specification and Analysis of the AER/NCA Active Network Protocol Suite in Real-Time Maude. Fundamental Approaches to Software Engineering : 4th International Conference, FASE 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001

Genova, Italy, April 2–6, 2001 Proceedings. H. Hussmann. Berlin, Heidelberg, Springer Berlin Heidelberg : 333-347.

[197] Ölveczky, P. C. and S. Thorvaldsen (2007). Formal Modeling and Analysis of the OGDC Wireless Sensor Network Algorithm in Real-Time Maude. Formal Methods for Open Object-Based Distributed Systems : 9th IFIP WG 6.1 International Conference, FMOODS 2007, Paphos, Cyprus, June 6-8, 2007. Proceedings. M. M. Bonsangue and E. B. Johnsen. Berlin, Heidelberg, Springer Berlin Heidelberg : 122-140.

[198] N. Martí-Oliet, José, Meseguer and A. Verdejo (2005). Towards a Strategy Language for Maude. Electron. Notes Theor. Comput. Sci. 117 : 417-441.

[199] Durán, F., et al. (2000). Principles of Mobile Maude. Agent Systems, Mobile Agents, and Applications : Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents, ASA/MA 2000, Zurich, Switzerland, September 13-15, 2000 Proceedings. D. Kotz and F. Mattern. Berlin, Heidelberg, Springer Berlin Heidelberg : 73-85.

[200] Clavel, M., et al. (2007). Mobile Maude. All About Maude - A High-Performance Logical Framework : How to Specify, Program and Verify Systems in Rewriting Logic. Berlin, Heidelberg, Springer Berlin Heidelberg : 485-522.