Modélisation, spécification formelle et vérification de protocoles d'interaction: une approche basée sur les rôles

Ivan Romero-Hernandez

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque
|_/_/_/_/_/_/_/_/_/_/

THÈSE

pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : Informatique et Mathématiques Apliquées

préparée au Laboratoire de Conception et d'Intégration de Systèmes de Valence
dans le cadre de l'**École Doctorale**
Électronique, Mathématiques, Informatique, Technologies de l'Information

présentée et soutenue publiquement

par

Ivan ROMERO HERNANDEZ

le 16 décembre 2004

Titre :

Modélisation, spécification formelle et vérification de protocoles d'interaction: une approche basée sur les rôles

---

Directeur de thèse :
M. Jean-Luc Koning

JURY

Prof. Chantal Robach	, présidente
Prof. Joël Quinqueton	, rapporteur
Prof. Félix Ramos Corchado	, rapporteur
Prof. Suzanne Pinson	, examinatrice
Prof. Jean-Luc Koning	, directeur de thèse

Résumé

Dans cette thèse on présente une proposition pour la spécification de protocoles de communication utilisés dans le contexte de systèmes multiagent, et qui nous appelons *basée sur rôles*. L'idée de rôle est récurrente dans le domaine de systèmes multiagent, et représente ce qui fait un agent dans une organisation, c'est à dire, la fonctionnalité qui est accomplie par l'agent dans le système, fonctionnalité qui est exprimée très souvent sous la forme d'échanges de messages qui suivent un protocole o séquence predéfinie. En nous centrant exclusivement sur l'aspect de l'interaction, il est possible d'abstraire la fonctionnalité du système en termes de séquences de messages stéréotypées. Séquences qui peuvent être utilisées comme un méthode de spécification en choisissant une notation appropriée.

Nous proposons ici un processus de traduction à partir d'une notation visuelle inspirée de la proposition AUML, qui peut représenter le processus de communication dans un système multiagent, et le traduire vers une représentation formelle en *Promela* pour effectuer sa validation.

Abstract

In this thesis we present a proposal for communication protocol specification in the context of multiagent systems, with an approach we call *role based*. The idea of role is frequently found on the multiagent systems domain, representing very often the functionnality accomplished by the agent within the system, functionality that is usually expressed as message sequences following well defined protocols.

This proposal is based on the notion of role attribute, as it is implied on the agent-oriented methodologies MADKIT and GAIA. Roles could be seen as social attributes of agent groups, that are taken in the context of any given interaction. If we see roles like this, we could also imagine schemes to generate an actual specification with concrete agents from the generic specification using roles. To test this approch we propose a machine-assisted specification process that takes a role definiton expressed on a visual notation (AUML), and generates a formal equivalent on the language *Promela*.

Avant propos

Ce travail de recherche a été effectué au sein du Laboratoire de Conception et d'Intégration de Systèmes de l'ESISAR à Valence (ESISAR-INPG). Les travaux exposés dans le présent mémoire ont été accomplis avec le soutien du programme bilateral franco-mexicain SFERE-CONACyT.

Je remercie particulièrement Madame Chantal Robach, Professeur à l'Institut National Polytechnique de Grenoble et directrice du laboratoire LCIS, pour l'intérêt porté à mes travaux et pour l'honneur qu'elle me fait en présidant ce jury.

Je suis très touché que Monsieur Félix Ramos Corchado, Directeur de Recherches au Centre de Recherche et d'Etudes Avancés de l'Institut National Polytechnique de Mexique, ait accepté d'être rapporteur. Je le remercie vivement de sa lecture attentive et de ses remarques sur mes travaux.

J'adresse toute ma reconnaissance à Monsieur Joël Quinqueton, Professeur à l'Université de Montpellier, pour avoir accepté d'être rapporteur, ainsi que pour l'attention qu'il a porté au présent mémoire et pour ses observations.

Je tiens à exprimer ma chaleureuse gratitude à Madame Suzanne Pinson, Professeur à l'Université Paris Dauphiné, qui a accepté d'être examinateur de mon travail de recherche. Son aide précieuse a permis d'orienter efficacement mes efforts.

Que Monsieur Jean-Luc Koning, Professeur à l'Institut National Polytechnique de Grenoble, trouve ici l'expression de ma reconnaissance pour avoir accepté de diriger mes travaux de thèse. Je suis très honoré pour son intérêt et ses conseils continuels, mais sur tout, pour sa patience.

Table des matières

Table des figures

Chapitre 1

Motivation de la thèse

Étant donné le très rapide développement de la technologie des systèmes d'information, il est de plus en plus nécessaire de développer des méthodes qui assistent le développeur pendant tout ce processus.

La raison de ce besoin c'est la relation empirique qui existe entre la taille du problème et la complexité du développement. Plus un logiciel a des fonctions, plus il est compliqué à développer. Il y a longtemps en termes industriels et chronologiques, que le seuil de complexité gérable par un seul être humain a été atteint.

Le fait de gérer le travail du développement dans une équipe est bien sûr, une des solutions les plus courantes. Les problèmes de gestion liés au bon management d'un groupe de créatifs dans l'ingénierie sont des problèmes depuis longtemps étudiés, et les solutions à ces problèmes ont été adaptées de façon assez satisfaisante depuis le début aux problèmes analogues qu'on peut trouver dans le génie logiciel, spécifiquement vouée au produit-logiciel.

Pourtant, le génie logiciel comme science et comme technique pose de nombreux problèmes particuliers qui n'ont pas des parallèles dans d'autres domaines. Pour commencer, par le produit lui-même qui est plus proche d'une composition que d'un produit à créer à la fin du processus créatif.

Le logiciel aussi impose des contraintes qui viennent des sources beaucoup plus profondes, dès la théorie mathématique même qui décrit les systèmes programmables. Par exemple, le fait de ne pouvoir jamais être totalement certain qu'un logiciel donné va arriver à un certain état désirable dans son fonctionnement, vient de la très fréquente impossibilité pratique de stocker tous les possibles états de ce système.

Le manque de rigueur dans la façon dont les logiciels sont créés se voit reflété dans le manque de qualité des produits. C'est comme si on avait un comité qui désigne des montres qui sont belles, mais qui ne donnent pas de garantie qu'elles marcheront correctement jusqu'à minuit chaque jour. Ce qui serait catastrophique d'un point de vue du génie traditionnel, est tout à fait acceptable dans le génie logiciel.

Évidemment, le processus du développement doit être fait de telle manière qu'il cherche à éliminer le plus possible d'erreurs de conception ou de programmation bien avant que le produit soit présenté au public.

Pour y arriver, il y a deux approches:

1. le suivi rigoureux du travail en utilisant un processus de gestion du travail, avec une notation ad hoc pour représenter tous les aspects du problème en voie de résolution et
2. l'automatisation du développement avec l'assistance d'outils logiciels qui fournissent des méthodes complètement définies et standardisées pour développer chaque parti.

Dans notre thèse, nous nous sommes centré sur les systèmes de développement assistés par ordinateur, qui sont ceux qui ont démontré une plus grande fiabilité au moment de réaliser la vérification des certaines propriétés désirables du logiciel-produit, mais aussi les plus ardus à maîtriser et à utiliser.

Car bien sûr, il existe des limites aux approches qui se basent sur l'automatisation du développement: tout en définissant un cadre guide et un ensemble d'outils qui fournissent de l'aide aux concepteurs, ils sont aussi une contrainte pour tous les cas qui ne sont pas prévus pour les concepteurs de la méthode. Les outils risquent assez souvent de vieillir avant d'être réellement utilisés.

Dans le cas des outils expérimentaux, désignés comme plateformes de test, ils existent aussi des défis spécifiques liés à la fragmentation du travail de recherche en plusieurs approches plus ou moins dispersées et centrées sur différents aspects du développement, et qui sont en général ceux qui sont les plus connus de ses créateurs.

Cette division inhérente à la recherche dans le domaine du génie logiciel, provoque des problèmes chez les chercheurs qui voient leur travail voué à devenir un exemple unique, quelque chose comme "une seule fois et pour toujours".

Ce problème se pose aussi pour les propositions qui sont dans le courant de l'industrie, loin des préoccupations plus subtiles de la recherche. Cependant, l'industrie à force de mauvaises expériences, a fini par créer ses propres solutions à ce problème: la standardisation et les méthodologies.

Actuellement, les approches orientées objets ont atteint un degré assez élevé de maturité pour commencer à devenir réellement compatibles entre elles. Les concepts qui définissent clairement ce qui est un objet et ce qui ne l'est pas [16] ont permis de faire des grandes avancées dans l'industrie.

Cette maturité théorique est reflétée aussi dans les aspects méthodologiques liés aux systèmes OO: la naissance et la rapide acceptation de la notation UML comme le standard *de facto* dans l'industrie, révèle le degré de maturité du domaine. Il y a eu certes, des efforts concertés par agrément

mutuel, mais ces agréments n'auraient pas été possibles s'il n'y avait eu un très fort développement et recherche théorique jusqu'à l'arrivée d'un accord tacite. Accord qui encore une fois, n'aurait pas été possible sans une longue histoire d'expérimentation et propositions plus ou moins théoriques sur ce qu'est la programmation et comment la faire convenablement.

Cependant, il existe certaines personnes qui croient que les objets ne sont pas la dernière étape dans l'évolution de l'abstraction des systèmes. Tout en gardant les grands éléments apportés par les objets (abstraction et modularité, interfaces faiblement couplées, délégation des responsabilités) certains se demandent si l' approche suivante ne va pas sur la même voie mais avec de nouvelles caractéristiques: entités plus modulaires et intelligentes, interfaces plus flexibles approchant le vrai dialogue, voir même interchangeables, avec une délégation totale des responsabilités sur des modules assez autonomes et intelligents pour remplir leur fonction convenablement et avec le minimum d'intervention humaine.

L'idée d'agent comme approche pour la programmation naît précisément de ce désir de trouver des abstractions plus poussées et plus complexes capables de représenter les systèmes du monde réel d'une façon plus naturelle et, probablement, plus facile à concevoir et à implémenter.

Il serait trompeur de dire que le simple désir d'avoir une meilleure abstraction de la composition d'un système, a suffit pour justifier le concept d'agent. En vérité ils sont un ensemble de facteurs et de développements, ceux qui ont façonné l'idée qu'on se fait à présent de ce qu'est un agent et ce qui ne l'est pas.

Le domaine de l'intelligence artificiel a connu bien des approches différentes et des bouleversements dans sa pas si longue existence, et l'apparition des agents viens d'une de ces petites révolutions: L'application des concepts de systèmes experts aux systèmes distribués. L'intelligence artificielle en réseau, est en elle même une couche supplémentaire de complexité au difficile problème de représentation de l'*intelligence*, et ces facteurs plus ou moins abstraits qui la définissent: la connaissance, l'apprentissage et la capacité de déduction. Et les systèmes en réseau ont aussi ses propres problèmes bien particuliers: tels qu'assurer la vivacité, la sécurité, la conformité à un standard donné, etc., qu'il faut prendre en compte toujours et qui, selon certains experts, se voient même reflétés dans la connaissance du monde qu'un système prétendument intelligent doit avoir.

Des telles caractéristiques et les problèmes liés à son implémentation, sont d'ores et déjà présentes dans une bonne quantité des logiciels voués au domaine de systèmes distribués et à la programmation parallèle, rarement avec une approche d'intelligence artificielle quelconque.

Cette thèse est vouée à étudier une niche très particulière du problème de conception et implémentation des systèmes d'intelligence artificielle distribuée: comment correctement spécifier et valider l'interaction entre les différentes parties. Dans cette thèse on étudie le problème très particulier de

la traduction automatique de spécifications de protocoles de communication entre agents, et la façon dont laquelle ces protocoles de communication semblent suggérer une notion plus profonde qui pourrait être une caractéristique de base des agents.

1.1 Propos de la thèse

Cette thèse se penche fondamentalement sur la façon de développer des protocoles d'interaction entre agents. Pour procéder convenablement à une explication plus exhaustive, il serait aussi convenable de donner une définition d'agent et en suite de protocole d'interaction. Malheureusement, il n'existe pas encore un accord généralement accepté sur les propriétés nécessaires et suffisantes qui caractérisent les agents et qui les différencient des abstractions de plus bas niveau (e.g. objets), ce qui rend difficile de construire une structure formelle qui explique rigoureusement les propriétés des systèmes multiagent, d'une façon analogue à la théorie de types de données abstraits qui permettent de représenter les objets.

Une telle tâche seulement est possible en consultant des ouvrages de référence et en prenant une approche particulier qui soit en accord avec nôtre intérêt de recherche, et avec l'opinion de la communauté intéressé par ces sujets.

Mais d'abord, il pourrait être prudent de définir quelle était notre motivation principale quand nous avons abordé le développement de cette thèse:

1. Étudier la traduction automatique des spécifications visuelles des systèmes multiagents vers une représentation formelle à travers l'utilisation d'exemples.
2. Trouver les aspects importants qu'il peut être nécessaires d'inclure dans une spécification d'entrée et quelles seraient les propriétés vérifiables à la sortie.
3. Utiliser une notation adéquate comme point de départ et l'utiliser exhaustivement pour tester ses capacités.
4. Détecter des caractéristiques récurrentes qui pourrait être d'une nature plus profonde dans la définition d'agent, si possible.

Mais avant d'entrer dans un éclaircissement de ces points, nous allons parler de quelques concepts fondamentaux reliés à notre travail. Le premier est celui de *génie logiciel*.

1.2 Génie logiciel

La vertigineuse croissance des capacités de calcul et de mémoire des ordinateurs a permis aux développeurs de logiciels la création des systèmes

de plus en plus complexes, avec un plus grand nombre de capacités et de fonctions, pour ainsi mieux satisfaire les demandes des utilisateurs ou clients finaux.

Cependant, le fait d'agrandir les capacités d'un logiciel augmente aussi *la complexité* de celui-ci. La complexité en génie logiciel n'est pas la même que la complexité en temps d'exécution ou de mémoire, c'est plutôt une façon d'exprimer la difficulté théorique ou pragmatique qu'on doit surmonter toujours pour obtenir à la fin le produit désiré.

Pendant que la complexité d'exécution est une *mesure* quantitative précise du comportement d'un algorithme spécifique, la complexité en génie logiciel est un concept quantitatif et qualitatif, lié aux expériences pratiques du développement d'un *produit*.

Cette complexité composée croissante affecte directement le processus même de création du logiciel. Et demande des solutions de gestion, administration et méthodologiques diverses qui facilitent et qui rendent possible, la création du logiciel.

Ces solutions méthodologiques conçues pour la création sont prises en compte dans la science du génie logiciel, et produisent des lignes directrices qui suggèrent aux développeurs les actions à prendre pour faciliter le plus possible le processus de création. Pourtant, le génie logiciel n'est pas intéressé seulement par le processus de création, mais aussi pour les procès de maintenance, correction d'erreurs et éventuelle extension du produit. Lesquelles sont requises par l'industrie moderne du software, car le logiciels sont devenus des produits de consommation massive, donc les clients demandent une suivie du produit au delà de l'achat.

La génie logiciel suit les cadres méthodologiques de la programmation qui changent a travers le temps, en suivant les changements progressifs la pratique de la programmation, lesquels sont ensuite provoqués par des changements théoriques, puis pragmatiques de l'expressivité des langages de programmation. Par exemple, l'arrivage des langages structurés a provoqué la création des méthodologies qui prennent en compte la subdivision structurée du système, plus rarement mais aussi d'une façon significative, le génie logiciel propose des changements convenables aux concepteurs des langages de programmation.

1.2.1 Spécification

Une fois notre intérêt centré sur le contexte du génie logiciel, il y a d'autres concepts majeurs liés à notre travail. Le premier d'entre eux est sans aucun doute l'idée de *spécification*, ce concept de spécification est un de plus répandus et plus utilisés dans la littérature informatique. Et il est utilisé avec un nombre assez variable de significations, sans pour autant perdre l'idée centrale: Une spécification est une description de quelque chose, sans être la chose elle-même.

Cette idée assez vague, peut être épurée et adaptée en suivant des critères pragmatiques tel que Partridge et al le suggèrent dans [50]: une spécification est une description d'un système informatique, sous la forme d'une liste le plus descriptive possible de choses que ce système doit faire.

Il est intéressant de voir que selon certains commentaires [20, 50] les spécifications doivent être à la fois les plus libres possible de descriptions détaillées sur le *comment* de l'implémentation du système, et plus sur le *quoi* doit être fait, mais il reconnaissent qu'une séparation absolue est impossible (et indésirable) dans la pratique.

Et s'ils disent cela, est dû au fait que une description d'un produit logiciel est en elle-même une certaine conception de la solution. Cette dichotomie *quoi-comment* est particulièrement présente dans les spécifications qui s'aident de notations formelles et graphiques.

Dans [20], une division générale des *notations* de spécification de software est proposée, laquelle on croit pourrait servir comme point de départ dans la compréhension du problème à traiter dans cette thèse; Haywood et al donnent quatre catégories principales:

Notation en langage naturel C'est à dire, l'utilisation de la langue humaine comme outil de description du produit logiciel, et qui c'est bien sûr, la plus couramment utilisée dans l'industrie.

Notation de hypergraphe Haywood et al appellent ainsi toutes les notations qui ont des liens sémantiques uni ou bidirectionelles qui relient des éléments, par exemple, du code HTML. Ils considèrent que l'hypertexte et d'autres formes de représentation de l'information hyper-liées sont une catégorie à part du langage naturel et les notations formelles.

Notation formelle Les notations formelles sont des langages artificiels lisibles par une machine, ces langages ont des règles de syntaxe et sémantiques qui permettent de décrire sans ambiguïté les propriétés et le comportement d'un système logiciel d'intérêt. Et dans la plupart de cas, ces règles de syntaxe et sémantique permettent aussi la manipulation symbolique du *modèle,* de telle sorte que par moyen de ces manipulations formelles ont puisse prouver la présence de *propriétés désirables*, mais pas évidentes au premier regard, qui se retrouveront ainsi démontrées -ou réfutées-. Ce qui dans le lexique du domaine, est appelé *validation.*

Notation visuelle Les notations visuelles utilisent des analogies graphiques pour représenter des propriétés d'un système, en définissant des liens sémantiques entre des symboles visuels (des dessins) et des entités abstraites dépendantes du contexte du problème à traiter avec cette notation. Les notations visuelles ont une chose en commun avec les notations formelles: Elles utilisent des symboles avec une 'syntaxe' d'interrelation et une sémantique spécifique, mais

elles ne sont pas habituellement exhaustives en ce qui concerne le non ambiguïté du modèle ni l'exactitude syntaxique. Les résultats d'une notation visuelle sont des diagrammes ou schémas dans lesquels on peut représenter de façon plus intuitive les caractéristiques d'un système.

1.2.2 Modèles

Le mot *modèle* est très fréquemment retrouvé dans la littérature informatique, et assez souvent il arrive qu'il soit utilisé de façon interchangeable avec le concept de spécification, bien que le concept de spécification soit plus général. Dans son sens général, un modèle est une représentation mathématique d'un phénomène naturel ou fait par l'homme; dans l'ingénierie la notion de modèle est surtout liée à une approche *descriptive* des phénomènes, et qui permet en utilisant les propriétés *formelles* de la branche des mathématiques utilisée pour construire ce dit modèle, d'établir la validité des certaines propriétés du système modélisé (c'est à dire, un modèle est aussi un outil analytique).

Il existe une autre définition de modèle -celle qui dit qu'un modèle est une représentation mathématique qui satisfait un ensemble d'axiomes-, proposé dans la théorie des modèles logiques; définition qui est tout aussi valable et légitime mais qui ne rentre pas dans nos efforts de recherche. Dans le déroulement de ce travail, quand on parlera du modèle on fera référence au sens descriptif-analytique du terme utilisé dans l'ingénierie.

D'autre part le mot *modélisation* fait référence au processus de création d'un *modèle*, lequel est une description symbolique du système exprimée dans la *notation* choisie par le développeur. Choix qui est fait en suivant le domaine de l'application ou le type de propriétés qu'est désirable de prouver qui sont présentes dans le modèle, dans l'intérêt de ceux qui développeront le système final. Car il est accepté comme du bon sens que si le modèle a les propriétés désirables choisies, on peut estimer qu'une implémentation finale étroitement basée sur ce modèle le fera aussi.

Dans les mathématiques, depuis un certain temps [4] il existe une division commune des modèles en deux types fondamentaux:

Modélisation dans le domaine continu Dans la quel les notations utilisées sont celles des mathématiques continues. Par exemple, un modèle physique de la chute libre d'un corps.

Modélisation dans le domaine discret Dans la quel les notations utilisées proviennent des mathématiques discrètes.

1.2.3 Spécification formelle

Un autre concept important est celui de spécification formelle. Les *méthodes formelles* décrivent un large spectre de techniques où on utilise la

logique et/ou les mathématiques discrètes pour la spécification, l'analyse et l'implémentation des systèmes logiciels. Où le mot *formel* a comme sens "*tout ce qui appartient aux relations structurelles entre des éléments*" [44, 45].

Dans la *logique formelle*, par exemple, on étudie les raisonnements, cheminements ou manipulations symboliques des propositions qui sont valables sans importer le contenu de ces dernières, c'est à dire, on s'intéresse à la *forme* et au *processus* qui mène depuis un ensemble de propositions logiques quelconques, vers des inférences valides sans pour autant faire appel à la sémantique de ces propositions.

Dans le cas d'autres méthodes de modélisation formelle, on s'intéresse aussi aux relations structurelles entre éléments d'un langage et aux manipulations symboliques valides qu'on peut faire avec, car il est possible en suivant certaines manipulations de *prouver* qu'une expression symbolique en ce langage *a* (ou *n'a pas*), c'est qui est également important- une certaine *propriété désirable*.

Le grand intérêt des méthodes de spécification formelle est de réduire la nécessité de tout essayer de prouver "à la main", et de remédier aux problèmes classiques de manque de compréhension des spécifications (par le biais d'un langage non ambigu pour faire le modèle): le manque d'attention pendant le développement (en validant que le système a des propriétés désirables, et si c'est possible qu'il n'ait pas des propriétés indésirables) et la capacité limitée des développeurs humains à faire face à la complexité des petits détails. Des choses qui sont bien dans ce que Brooks[6] appelle *l'essence* du produit logiciel.

1.2.4 Spécification visuelle

Le concept qui suit en importance dans cette thèse est sans doute celui de la spécification visuelle. Comme il a été dit auparavant, la spécification visuelle des systèmes logiciels est une approche descriptive qui vise avant tout à représenter d'une façon *intuitive* les propriétés importantes du système, car il est constaté par l'expérience, que pour certains types de tâches et problèmes de développement, les diagrammes sont plus explicites et économes que les descriptions en langage naturel ou formel, en plus d'être plus compréhensibles et d'une signification plus facile à partager , une fois qu'un consensus sur la sémantique du diagramme a été atteint.

Dans la très grande variété de notations de spécification visuelle il est possible de faire une division générale:

- Notations formelles: les notations visuelles peuvent être définies comme des langages avec des règles strictes de syntaxe, sémantique et de manipulation symbolique, qui représentent d'une façon directe des entités formelles. L'utilité de ces types de notations visuelles dite formelles, est de permettre d'effectuer un travail de validation sur un modèle visuel,

analogue a celui effectué sur des spécifications purement textuelles, tout en essayant de garder la facilité de compréhension qui vient avec des diagrammes plus explicites que des expressions formelles "dures".Des exemples de cette approche formelle pour les notations sont le langage SDL[15] et Z.120[32].

- Notations semi-formelles. Sont celles qui tout en définissant un ensemble de règles pour définir l'apparence et structuration d'un ensemble d'artefacts graphiques, laissent certains détails de la sémantique ou bien implicites ou bien au jugement du développeur. Un exemple notoire de notation semi-formelle est l'ensemble d'artefacts proposé par UML 1 et 2.

Ce domaine étant très vaste il faut préciser sur quel sous domaine on se retrouve: quand on parle de spécification visuelle, on est centré sur les notations qui représentent le système logiciel d'une façon *descriptive*. Tout comme la modélisation formelle des systèmes informatiques, son équivalent visuel utilise une notation particulière pour décrire le comportement et caractéristiques des systèmes logiciels.

1.3 Organisation du mémoire

Ce mémoire de thèse est organisé en cinq parties principales. La première est une présentation de l'état de l'art des différents domaines de recherche des quels l'auteur s'est inspiré pour la réalisation de son travail. La deuxième partie est centrée sur le travail pratique accomplie, la troisième partie est constituée de la conclusion et des annexes détaillant la structure et fonctionnement des outils logiciels générés; et la quatrième partie est une compilation de papiers élaborés par les auteurs, acceptés et publiés dans différents médias.

Chapitre 2

État de l'art

2.1 Avant les agents

Avec l'apparition et maintenant avec la popularité étendue du concept *d'objet*, où les données et les fonctions qui modifient ces données sont groupées dans une même structure syntaxique, il a été de plus en plus claire que l'approche à suivre à l'avenir était une plus forte subdivision des fonctionnalités du système, en créant des entités abstraites modulaires qui ont des fonctionnalités bien définies et complémentaires, qui coopèrent pour offrir la fonctionnalité apparente du système.

Avec les objets s'est profilé l'idée *d'interaction* faible, car les objets ont une interface avec le monde extérieur formée par leurs fonctions publiques. Ainsi les objets s'engagent à satisfaire un ensemble des comportements et de conditions établies par leur propre interface, mais n'ont pas besoin de spécifier précisément comment cette fonctionnalité est réellement accomplie. C'est l'interaction faible qui rend possible la plus forte capacité de réutilisation des systèmes à objets et qui approche encore plus les systèmes centralisé à la façon dont sont conçus les systèmes distribués.

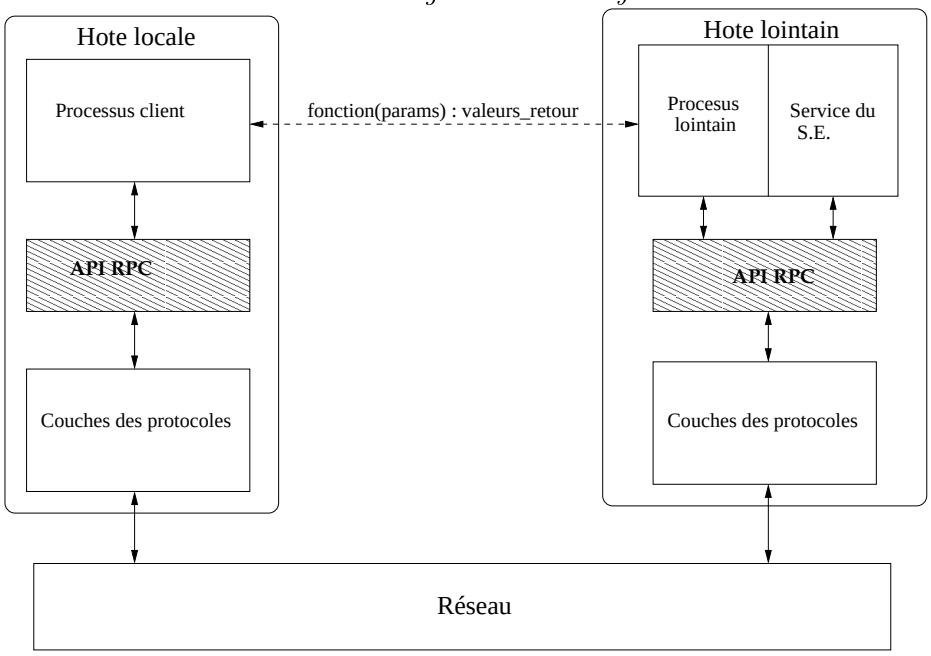
Cette idée est rapidement apparue intéressante aux chercheurs, car les systèmes distribués avaient développé déjà l'idée d'interaction faible, mais en utilisant des messages envoyés a travers un réseau, et qui peuvent abstraitement être considérés comme des messages ou invocations des méthodes parmi des objets qui ne s'exécutent pas dans le même processus. De cette idée est née la notion *d'objet distribué*.

Cependant les objets distribués ne sont pas les seuls inspirateurs de la notion d'agent, loin de là, il y a aussi bien d'autres techniques qui précèdent l'idée courante qu'on se fait de ce qu'est un agent.

2.1.1 Appels à fonction distante

Il est bien connu que dans la programmation structurée, une des entités les plus fondamentales sont les procédures/fonctions, qui sont en bref, des

FIG. 2.1 – *Structure générale d'un système RPC*



segments du code qui reçoivent des valeurs en entrée, les manipulent de façon algorithmique, et qui peuvent générer d'autres valeurs en sortie; ces segments de code sont spécialement conçus pour représenter des sections très répétées de code, vers lesquelles le flou de l'exécution d'un logiciel peut dériver à tout moment. L'idée de macro fonction, procédure et fonction son certes, contemporaines de la technique des machines programmables elle-même, mais la programmation structurée a poussé l'abstraction au-delà de son approche initiale. Dans la programmation structurée, les fonctions sont spécifiées comme des unités d'instruction avec un nom symbolique, et la notation pour l'envoi de valeurs d'entrée ainsi comme la réception de valeurs à la sortie, se rapprochent plus de la notation de fonction mathématique.

Au fur et à mesure que les systèmes en réseau se développaient, il est très tôt apparu évident que l'invocation d'une fonction pourrait être simulée de façon transparente à l'aide d'un mécanisme de communication interprocessus, en définissant un protocole de communication, et que la machine où s'exécutait le processus faisant l'appel à fonction et la machine où la fonction s'exécutait n'ont nécessairement besoin d'être le même hôte physique. Donc la dénomination 'appel à fonction distante' (Remote Procedure Call ou RPC) fait référence à toutes les stratégies et infrastructures logiciels, qui permettent d'effectuer l'invocation d'une fonction quelconque dans un hôte lointain.

La figure 2.1 montre la structure générique d'un système d'appels à fonc-

tions distantes. Ce type d'infrastructures font toutes usage d'un approche en couches qui isole l'application finale de la complexité fonctionnel inhérente au problème d'assurer la communication(établir une session depuis l'hôte locale vers l'hôte lointain, détecter la présence ou absence de la fonction invoquée, effectuer l'envois de l'information nécessaire comme paramètres à la fonction et le retour des valeurs de sortie), dont chaque une de couches remplit une partie spécifique de la fonctionnalité désirée.

Pour le programmeur, l'utilisation de la infrastructure se fait tout simplement à travers d'une interface de application (API), et il n'a pas besoin de s'occuper de tous les petits détails. Mais bien que les infrastructures d'appel à fonction distante facilitent énormément la tâche du développeur, en lui isolant le mieux possible de la complexité relative du protocole de communication soujacent au processus RPC, il y a des aspects du fonctionnement du schéma RPC qui sont impossibles à masquer, par exemple, les multiples cas d'erreur qui sont inhérents à la manipulation et transfert de données dans un réseau, ainsi que les imprévus en temps de compilation (comme le fait que la fonction lointaine ne soit plus présente dans l'hôte où on l'attend).

L'exemple le plus ancien d'une proposition d'infrastructure RPC connue par l'auteur de ce document, est celui donné dans la spécification du RFC 707[61] (proposée en 1975), qui est un cadre guide pour développer des protocoles «orientés fonction».

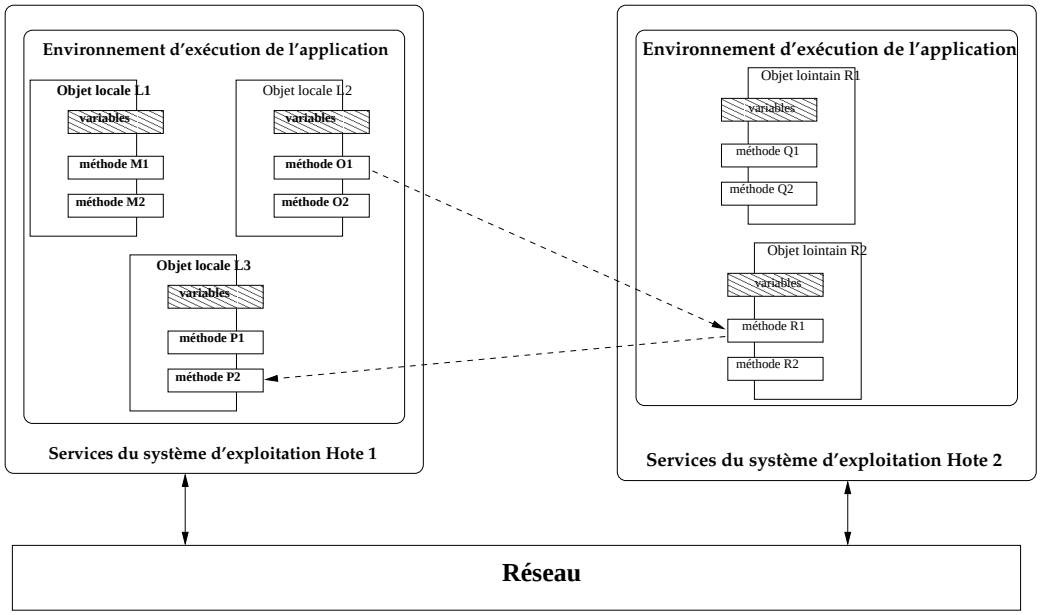
2.1.2 Objets distribués

Dès le début de l'étude de l'approche de programmation connu sur le nom de "orientée objet", il y a avait déjà des systèmes qui implémentaient des infrastructures RPC de façon solide et fiable. Du même pour les systèmes d'exploitation multi-tâche, qui étaient assez évolués pour commencer à suggérer l'idée de fils d'exécution multiples à l'intérieur d'un système fait d'objets.

Il est apparu tôt évident aux spécialistes du domaine de la programmation orientée objet, que les systèmes monolithiques composés des modules-objets pourraient bénéficier d'une plus grande flexibilité et d'un plus riche ensemble d'applications, si on étendait la notion d'appel à méthode vers une approche "appel à méthode distante". Cependant, la plus grande richesse sémantique de l'approche orientée objet permet aussi la définition de concepts (et des problèmes) bien plus sophistiqués que ses analogues dans les systèmes RPC pures.

La figure 2.2 montre un exemple de la structure générique d'un système à objets distribués. Dans cet exemple il y a deux hôtes en réseau, qui accueillent tous les deux une application développée avec une plateforme d'exécution d'objets distribués. Dans la figure, la méthode O_1de l'objet locale $L2$ fait appel a la méthode R_1 de l'objet lointain $R2$ pour accomplir sa tâche, mais à son tour, la méthode R_1 a besoin de la méthode P_2 de l'objet $L3$.

FIG. 2.2 – *Exemple de structure générale d'un système à objets distribués*



Dans un exemple semblable au dernier, mais en utilisant la programmation orientée objet centralisée, une coopération comme celle décrite auparavant est décrite et implémentée de manière triviale. Il suffit de passer explicitement les références aux instances d'objet nécessaires dans le code source, et d'invoquer les bonnes méthodes au moment venu.

Dans le cas d'une plateforme à objets distribués, même le cas les plus triviaux ont une charge non négligeable de travail de conception et développement ajoutés. Principalement, car il faut toujours assurer qu'il y a bien un objet lointain correspondant aux *attentes* de l'objet locale qui fait appel à ses fonctions, et que les objets ne tombent pas dans des inconsistances dues à son exécution parallèle et aux erreurs de communication en réseau.

Une des différences les plus importantes entre les mécanismes d'appel à fonction lointaine classiques, et ceux présents dans les systèmes d'objets distribués; est le fait que l'encapsulation des méthodes et variables dans les objets, permette d'assurer que si on attend de communiquer avec un objet quelconque de type O dans un hôte lointain, l'environnement d'exécution à objets distribués peut détecter facilement s'il est bien là. Et une fois qu'il l'a fait, on peut assurer que les *attentes* d'un objet quelconque interagissant avec cet objet de type O seront satisfaites.

Quand on parle d'attentes, on veut dire les interactions (c'est à dire, appels à méthode ou manipulation de variables internes) que les objets extérieurs peuvent effectuer sur un objet d'une classe donnée. Dans la phase de

programmation, ces interactions sont spécifiées par les méthodes et variables publiques déclarés dans une portion visible de la définition de classe, ce qui certains appellent une interface.

La notion d'interface, et en particulier, celle d'interface faiblement couplée, vient de l'idée proposée originalement dans la théorie et la programmation orientée objet, qui vise à diminuer le besoin des développeurs du système, de connaître le fonctionnement de toutes les parties d'une façon très détaillée; pour y arriver la solution la plus facile est de cacher le plus possible du fonctionnement interne d'un objet du reste de son environnement, et en laissant seulement les points d'interaction indispensables, pour que les autres objets dans l'environnement puissent faire appel aux services proposés pour l'entité. De telle sorte que les fonctionnalités soit beaucoup plus localisées dans une agglomérat abstrait facilement reconnaissables par les développeurs, et que les interactions indésirables (changement inattendu de variables para exemple), soient limités le plus possible.

Dans le cas de la programmation d'objets distribués, l'utilisation d'interfaces faiblement couplées devient beaucoup plus naturel, car les classes et les instances de classe peuvent être physique ou logiquement dans des environnements très hétérogènes, et la construction même de ces objets peut être en dehors de nôtre contrôle (en étant partie d'un système construit par un tiers, par exemple) Dans des environnements semblables, il est en fait, absolument nécessaire de connaître les services offerts pour un objet localisé dans un hôte lointain.

Des exemples d'environnements de développement et exécution à objets distribués sont les plate-formes Emerald[34], Distributed Smalltalk[5] et plus récemment, CORBA[17]. Ces systèmes pour règle générale offrent au programmeur un environnement de développement (un pré-processeur, un compilateur, un débogueur, etc.), ainsi qu'une infrastructure en temps d'exécution qui effectue tous les petits détails nécessaires au fonctionnement d'un système d'appel à méthodes en réseau.

Il faut noter que dans ces systèmes, il existe deux approches pour la définition des interfaces:

- Une approche déclarative: les interfaces des objets lointains sont connues en temps de compilation, et le compilateur produit par lui-même les appels à l'infrastructure d'objets distribués.
- Une approche réflexive: les interfaces des objets lointains sont connues en temps d'exécution, ce qui à grosso modo, fait que chaque objet dans le système distribué ait un modèle interne du comportement et de la structure de ses voisins, ce qui lui permet si nécessaire de réfléchir sur ce qu'il doit faire pour interagir avec eux. Le langage de définition d'interfaces (IDL) proposé par CORBA, est un bon exemple d'une notation pour construire des objets réflexifs.

2.1.3 UML

UML [49] (acronyme de Unified Modelling Language) est un ensemble de notations visuelles standardisées conçues pour représenter les propriétés habituellement associés à ce qu'on appelle *un système orienté objet*. UML est le résultat de la collaboration d'un groupe de spécialistes du domaine de la génie logiciel, pour arrêter la dispersion existante dans les notations utilisées dans le domaine, et créer **le** standard de facto pour modéliser les systèmes OO. L'appui initial de certains des plus importants groupes et entreprises de l'industrie, a permis que UML accomplisse bien sa fonction prévue de standard.

UML propose un ensemble d'artefacts capables de représenter un assez grand nombre de caractéristiques des systèmes orienté objets. Ces diagrammes se centrent sur trois types d'aspects des système:

Aspects statiques qui concernent les caractéristiques structurelles du système.

Aspects dynamiques c'est à dire, les comportements.

Aspects d'implantation qui sont reliés a la codification finale.

Dans cette thèse on fait référence assez souvent aux artefacts proposés par UML, et en particulier, aux diagrammes de séquence.

2.1.4 Intelligence artificielle

Les systèmes multiagents ou d'intelligence artificielle distribuée sont une des branches les plus récentes du domaine de l'intelligence artificielle. Cependant, bien avant l'existence du concept d'agent, il y a eut l'idée de créer des systèmes automatiques dits 'intelligents', capables de simuler ou imiter les processus cognitifs qui se déroulent dans les systèmes nerveux des entités douées de d'intelligence.

L'intelligence artificielle comme domaine de recherche a connu bien de bouleversements dans sa relativement courte existence. L'idée des machines programmables présentant un comportement intelligent, le concept même d'intelligence artificielle, a commencé à être façonné dans les années 40 du XXe siècle, avec des travaux fondateurs tels que "A Logical Calculus of the Ideas Immanent in Nervous Activity" par Warren McCulloch et Walter Pitts, apparu en 1943, qui est à la fois la première fois où on faisait référence à l'idée des systèmes à états, ainsi que le concept d'intelligence des machines.

Les premières approches pour créer cette intelligence des machines ont été fortement inspirées des progrès des sciences cognitives de l'époque, particulièrement la linguistique. On peut s'étonner aujourd'hui de constater que des notations aussi récurrentes et communes dans le travail d'un informaticien, comme les automates à état fini, ont eu comme justification initiale l'étude des méthodes pour reconnaître le langage humain; ce qui rentre dans

l'idée commune à l'époque qui disait que l'intelligence pouvait être simulée -ou créée- par la manipulation des constructions linguistiques. Malgré l'optimisme initial, il a très tôt paru évident que ce n'était pas le cas.

Il est clair que l'élan initial de l'intelligence artificielle étais celui de copier les processus cognitifs humains, pour obtenir à la fin un vrai système pensant; et que la meilleure façon d'y arriver était de copier les processus mentaux ou plutôt l'idée qu'on se faisait de la façon dont ces processus se déroulaient dans le système nerveux. Un anthropomorphisme tout aussi compréhensible qu'optimiste à outrance.

Actuellement l'intelligence artificielle, bien qu'encore fortement intéressée par les processus mentaux, et la façon de les imiter, dans son ensemble, est beaucoup plus conservatrice et pragmatique, en s'intéressant surtout à créer des systèmes qui ont des caractéristiques associées à l'intelligence, mais sans pour autant être par nécessité inspirée de la façon dont les humains pensent.

Par exemple, l'étude des schémas de comportement qui permettent aux insectes sociaux ou aux bandes d'oiseaux de produire des phénomènes très complexes, sans qu'il n'y ait pour autant un contrôle centralisé, sont tout à fait dans l'intérêt de l'intelligence artificielle. Il pourrait être pertinent de mentionner que dans l'ensemble des techniques associées aujourd'hui au domaine de l'IA, il existe une division de fait, qui surgit de la façon dont on aborde le problème de la représentation et la génération de l'intelligence avec une machine.

La première approche est l'approche formelle ou classique, qui est centrée fondamentalement sur l'imitation, l'émulation ou génération des processus 'intelligents', par la manipulation symbolique de structures abstraites soigneusement choisies. Les systèmes experts sont un bon exemple de cette approche.

La deuxième est l'approche connexionniste, qui vise à créer des systèmes intelligents, en définissant des structures capables de générer par la rétro-alimentation et l'auto-organisation un comportement intelligent.Un des exemples les plus connus de cette approche sont les réseaux neuronaux.

Dans le cas particulier des systèmes multiagents, l'influence de chacune de ces deux approches se reflète dans la manière dont les experts conçoivent et programment l'intelligence interne des agents. Cependant, il est possible de dire que l'approche formelle pour conceptualiser l'intelligence ainsi que ses différents aspects, est la plus influente dans le domaine, au moins beaucoup plus que les approches connexionnistes, sans pour autant nier que les approches hybrides ont aussi pris une place très importante.

Par exemple, il est presque impossible de parler d'agents sans faire référence aux approches formelles basées sur l'inférence logique. Héritage qui vient spécifiquement du domaine des systèmes experts, qui est en quelque sorte, un des moteurs initiaux de l'intelligence artificielle distribuée.

2.2 Cadres conceptuels de SMA

Les systèmes multi-agent sont communément associés à un cadre conceptuel qui sert a définir leurs propriétés d'une façon plus ou moins complète. Cette cadre conceptuel défini quel sont les propriétés ou caractéristiques qu'on considère propres aux agents, et il peut permettre idéalement de répondre aux questions fondamentales qu'on peut se poser quand on développe un système qui est supposé être un système basé en agents.

On croit que des questions qui peuvent se poser sur un programme, comme "est-ce vraiment un agent ou bien un programme?", "est-ce un objet ou bien un agent?", devraient pouvoir être répondues si on disposait d'un cadre guide qui servirait à donner une définition d'agent. De la même façon que il existe un cadre conceptuel qui permet la définition de ce qui est un objet, en qui permet de faire la différence de ce qui est un objet de ce qui ne l'est pas.

Malheureusement à ce jour-ci il n'existe pas de tel cadre guide unifié, mais plusieurs approches qui viennent des différents experts dans le domaine.

2.2.1 Agents

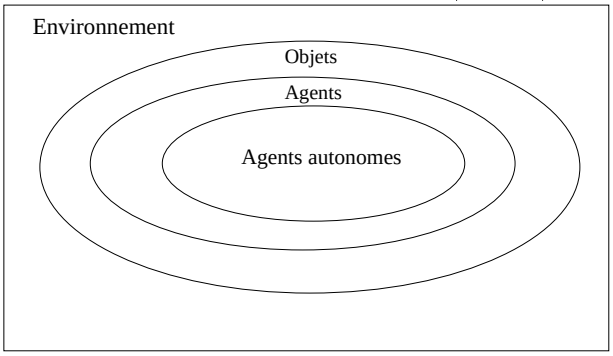
Comme il a été dit auparavant, il y a presque une définition d'agent pour chaque groupe de recherche spécialisé dans le domaine. Cette malheureuse situation a commencé à s'améliorer dans la mesure que plus de gens travaillent sur le domaine, plus d'expériences pratiques viennent à confirmer ou bien à reléguer dans l'oubli certaines propriétés présupposées faire parti de la définition d'agent, et il est maintenant possible de dire qu'une espèce de consensus a surgit de la confusion initiale.

Les agents son en général vus comme des entités autonomes situées qui réagissent aux événements d'un environnement de façon intelligente et d'accord à un plan ou agenda privé. Cette définition, bien entendu, ne prend pas en compte toutes les sous-divisions qui sont apparues dans la définition précise d'agent qui donnent pour règle générale chacun des experts du domaine.

Dans [41], Luck et d'Iverno se penchent sur la question d'un cadre guide unifié pour faire la définition de ce que c'est un système multiagent et ils proposent une définition formelle exprimée avec le langage Z de ce qui est un agent, en incluant trois niveaux d'abstraction: l'environnement/entités, les objets, les agents et les agents autonomes. La relation sémantique de ces quatre éléments est exprimé dans la figure 2.3, sous la forme d'un diagramme de Venn.

L'environnement est pour eux un ensemble d'entités, lesquelles sont simplement des modules abstraits avec des attributs, les entités peuvent aussi être des objets s'ils ont un ensemble d'opérations associées, et ces objets peuvent être des agents s'ils ont un but, finalement les agents peuvent être

FIG. 2.3 – *Diagramme hiérarchique environnement/objets/agents/autonomes*



des agents autonomes s'ils ont en plus d'un but un agenda un plan.

Ils privilégient fortement la notion de but, qui est pour eux une condition sine quoi non dans l'ensemble de l'environnement que les agents cherchent à accomplir ou satisfaire. L'agenda est pour eux le niveau maximal "d'agentitude", car sa satisfaction demande d'une capacité supérieure de raisonnement.

Sa définition se rapproche en esprit a celle donné dans [65], par Jennings et Wooldrige [64], pour eux la définition faible d'agent c'est:

> « ...un dispositif ou -le plus souvent- un logiciel qui a les propriétés suivantes »:

- Autonomie: les agents fonctionnent sans l'intervention directe des êtres humains et d'autres systèmes.
- Sociabilité: les agents interagissent avec d'autres agents ou avec des humains en utilisant un langage.
- Réactivité: les agents perçoivent leur environnement et réagissent en conséquence d'une façon opportune.
- Pro activité: les agents ne réagissent pas seulement à leur environnement, mais aussi ils prennent l'initiative en fonction de son propre agenda ou plan".

Tous ces points -sauf peut-être le dernier- sont bien connus pour ceux qui travaillent avec les systèmes distribués, alors, qu'est ce qui différencie réellement les systèmes multiagent des systèmes distribués?

Si on prend la dernière définition tel quelle est, les systèmes multiagent seraient définis principalement pour son niveau d'intelligence: sa capacité de faire des plans ou d'avoir un agenda, lesquels sont d'habitude bien au delà de la réactivité qu'on trouve dans les systèmes basés en modèles client-serveur.

Cette appelée définition faible d'agent sert comme base pour la méthodologie de développement orientée agent GAIA[63], et elle est plus ou moins reprise par beaucoup d'auteurs dans la littérature récente du domaine.

Il y a d'autres définitions que celle-là, bien entendu, comme par exemple la définition qui donne le IBM Intelligent Agent's Strategy:

> "Les agents sont des entités-informatiques qui font un ensemble de tâches de façon semi-autonome en représentation d'un tiers, qui peut à son tour être un humain ou bien un autre logiciel. Et pour cela faire, il a les connaissances nécessaires sur les désirs ou préférences de celui qu'il représente"

Cette définition peut s'appliquer parfaitement aux agents de recherche ou d'enchère, qui font quelque chose à la place d'un client humain et qui pendant un certain temps ne sont pas sous les commandes directes de celui-ci.

Cependant, elle nous semble incomplète dans le sens qu'elle peut être appliquée à n'importe quel système qui automatise des tâches et qui agit de façon semi-autonome, tel qu'un daemon de sauvegarde en réseau. Il est évident que cette définition est fortement liée aux intérêts particuliers de l'entité qui la propose, mais elle a la vertu aussi d'ajouter trois éléments nouveaux: les connaissances, les désirs et les préférences, toutes lesquelles apparaissent fréquemment dans la littérature.

2.2.2 Programmation/modélisation orientée agent

Il est bel et bien utile d'avoir un cadre guide qui définisse c'est qui est un agent, où se situe-t-il et quels sont les différents aspects ou caractéristiques qu'il faut prendre en compte pour développer un système orienté agent. Cependant, toutes ces définitions ont pour seul but de permettre d'arriver à une implémentation finale du système, ce qui est le but réel de tout le génie logiciel et des toutes les approches conceptuelles pour la programmation -ou la spécification[50], comme on verra après- existants. Une quelconque approche donnée, au moins dans l'industrie, doit faire ses preuves dans la pratique quotidienne du développement et si ses fondements théoriques sont solides et esthétiques, c'est positif mais pas déterminant dans son adoption finale.

Dans le cas de systèmes multiagent il existe beaucoup d'outils et des méthodologies associées qui permettent d'effectuer un processus de développement avec une approche orienté agent. Mais dû aux différences assez importantes entre les différentes propositions conceptuelles d'agent, elles peuvent varier de façon importante dans sa couverture des différents aspects du développement réel.

A continuation, on mentionnera quelques-unes des méthodologies avec ses outils associés, et qui sont importantes dans la littérature de systèmes multiagent actuelle, ainsi que pour notre travail.

2.2.3 L'approche voyelles

Quelques fois tout au long de ce document, on fera référence au cadre conceptuel voyelles, proposé originalement par Yves Demazeau dans [9]. Voyelles est un cadre conceptuel et méthodologique pour le développement des systèmes multiagent et aussi pour la création des méthodologies qui profitent de cette proposée capacité d'abstraction. La proposition voyelles (ou vowels, en anglais) donne une définition de système multiagent et d'agent, et considère que les propriétés d'un tel système peuvent se diviser en quatre aspects fondamentaux:

- Agents: la définition d'agent en voyelles est assez classique: la définition dit externe d'agent dit que les agents sont des entités concrètes ou virtuelles qui existent dans un environnement dans lequel elles peuvent agir et communiquer entre eux, et qui présentent aussi un certain comportement autonome.
- Environnement: l'environnement d'un agent consiste en tout ce qui entoure l'agent et qui ne fait pas partie de lui-même. Cette définition synthétisée, assez floue, peut être améliorée en disant que l'environnement de l'agent est fait de tous les éléments qui peuvent éventuellement être en relation avec l'agent dans son existence à travers le temps. Les agents existent dans un environnement, et c'est sur lui qu'ils peuvent agir, dans lequel ils communiquent et c'est aussi l'endroit abstrait à partir d'où ils obtiennent ses informations -c'est à dire, d'où ils obtiennent leurs perceptions-.
- Interaction: ceci désigne tout ce qui concerne la communication entre les agents, c'est qui est fortement lié à la nature distribuée des agents eux-mêmes. L'idée d'interaction en contraste de communication simple -tel que définie dans le cadre des systèmes distribués- est une idée fortement ancrée dans la littérature des systèmes multiagents, et le cadre voyelles considère aussi cet aspect comme très important pour mériter une analyse à approfondi séparément. Bien que l'étude de la communication entre agents ait beaucoup des points en commun avec la théorie des protocoles de communication, il y a des aspects bien particuliers de la communication entre agents qui sont mieux adressées par une approche différente. Comme par exemple, la façon la plus adéquate pour construire des messages aptes pour transporter des affirmations sur l'état mental de l'agent ou bien, la façon de construire des protocoles réellement inter opérables.
- Organisation: le dernier aspect dans lequel l'approche voyelles considère qu'un système MA peut se diviser, c'est l'organisation. L'organisation est, de façon très générale, la manière dont laquelle le développeur ou bien le système multiagent lui-même établie les responsabilités, les fonctions ou les tâches qui doivent accomplir chacun des modules qui

conforment le système, pour arriver à un certain but.

Bien évidemment, tous ces aspects sont fortement liés dans la pratique de l'implémentation d'un système multiagent quelconque. Mais l'intérêt principal du cadre voyelles, c'est justement, de proposer une séparation abstraite -et peut être quelques fois arbitraire- mais avec la suffisante généralité pour être utile pour spécifier/concevoir/développer des systèmes orientés agent.

2.2.4 Gaia

Gaia est une méthodologie pour le développement de logiciels avec une approche orientée agent, qui a été proposé par Wooldridge, Jennings et Kinny dans [63]. Cette méthodologie est assez souvent utilisée comme référence d'exemple ou comme point de comparaison pour les nouvelles approches méthodologiques.

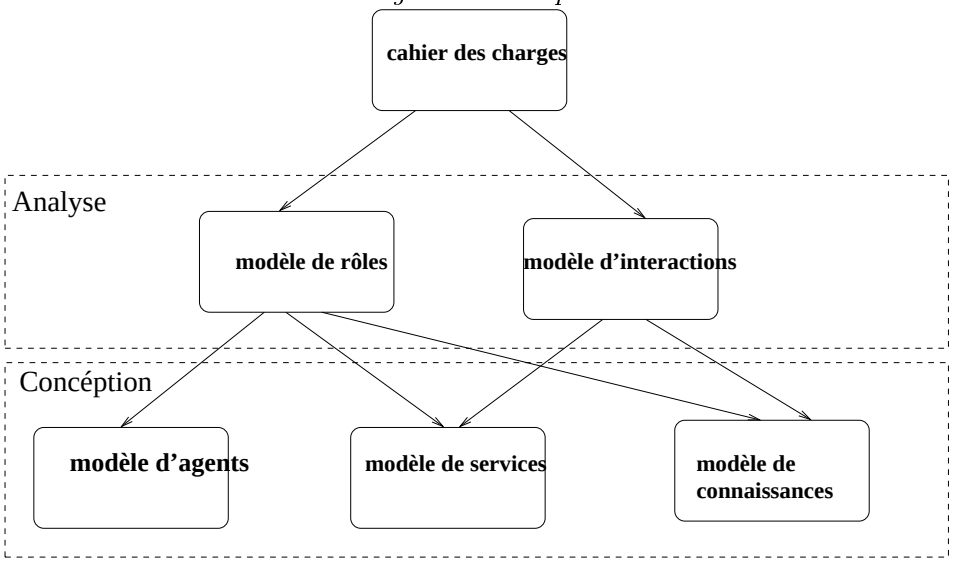
Gaia donne une intéressante définition de ce qui est un agent, qui retrouve l'idée qui est dans l'esprit de beaucoup d'experts du domaine. A savoir:

- Les agents sont des modules autonomes de taille conséquente, représentant l'équivalent d'un processus du système.
- Le développement d'un système fait avec des agents a comme hypothèse de base, qu'il existe un but commun a accomplir, et qu'il est acceptable que la solution obtenue pour y arriver ne soit pas optimale du point de vue des entités individuelles.
- Les agents sont hétérogènes. C'est à dire, il n'est pas nécessaire qu'il soient élaborés avec le même langage de programmation ou exécutés sur le même type d'environnement. Il est nécessaire par contre, qu'il puissent *interagir*.
- L'organisation du système est statique, c'est à dire, les relations de collaboration entre les agents ne changent pas en temps d'exécution.
- Les habilités ou compétences des agents ne changent pas non plus à travers le temps.
- Le système est supposé avoir un nombre raisonnable (les auteurs disent moins de 100) de types d'agent.

La méthodologie Gaia est centrée sur deux aspects fondamentaux des agents: spécifier les macro-aspects (aspects sociaux), et les micro-aspects (comportement interne de l'agent). La méthodologie elle-même suit une progression de décomposition du problème du haut-en-bas, qui va des aspects les plus visibles et abstraits de la société d'agents, et qui mène progressivement vers une définition plus détaillée et concrète de chaque élément.

La figure 2.4 présente une vue générale des activités définis par les auteurs de la méthodologie. La première partie, le cahier de charges, est laissée sans

FIG. 2.4 – *Vue générale des phases de* GAIA



définition et au choix du développeur. Ce qui est raisonnable si on considère qu'ils existent plusieurs approches pour sa génération, et que ce document exprime tout simplement, une liste de tout ce qui doit faire le système à développer.

Dans Gaia, la spécification détaillée de l'organisation représente une partie significative du travail de développement. Pour définir l'organisation du système, les auteurs font appel à un concept qui apparaît fréquemment dans la littérature: celui de rôle.

Dans toute spécification Gaia, la structure la plus générale et abstraite est *le système*. Ce système représente toutes les entités qui coopèrent pour produire la fonctionnalité voulue. Cependant, la décomposition du problème ne va pas directement à la spécification des modules-agents a partir du cahier de charges, car étant donné qu'on est en train d'extraire les catégories les plus abstraites d'abord, la catégorie suivante dans le système social artificiel, est celle de rôle.

Cette idée vient assez naturellement si on considère la façon dont nous abordons la conception d'une organisation ou système social quelconque. Tout d'abord on pense au différentes fonctionalités/responsabilités/titres qui existent dans cette organisation, avant de penser aux entités concrètes qui réalisent ces fonctions abstraites. Du même que par exemple, quand on pense à définir qu'est-ce que un match de foot-ball soccer, on pense d'abord aux joueurs et aux arbitres, ensuite aux différents *rôles* qui prennent les joueurs, tels que gardien, attaquant, etc. Bien que l'exemple soit un peu prosaïque avec des rôles directes, est un exemple acceptable de la manière dont nous abordons la conception d'une organisation en tant que telle.

Cependant, la notion de rôle en tant que telle est assez floue. Wooldridge et al proposent une définition de rôle comme la conjonction de quatre aspects:

- Responsabilités. Les responsabilités sont les actions qui doivent être accomplies par les entités appartenant a un rôle donné, et qui déterminent sa fonctionnalité dans la structure sociale. Dans la méthodologie, les responsabilités des rôles sont divisés en deux types: celles qui sont nécessaires pour assure *la sécurité* du système, et celles qui sont nécessaires pour assurer la *vivacité*. Il faut noter que les concepts de sécurité et vivacité sont plus proches de ses analogues en model-checking, mais en exprimant des faits plus pragmatiques; qu'il y ait de la *sécurité* exprime le fait que dans le système n'arrive jamais rien de *mauvais*, pendant que la vivacité exprime le fait que le rôle accompli éventuellement quelques chose *de bon*.
- Permissions. Les permissions définissent les capacités d'accès des membres d'un rôle aux ressources d'information dans l'environnement qui sont étroitement liées à ses responsabilités.
- Activités. Les activités sont toutes les opérations qui peuvent être accomplies par les membres d'un rôle sans faire appel à d'autres.
- Protocoles. Les protocoles sont des spécifications des comportements stéréotypés qui définissent la façon dont les rôles interagissent entre eux.

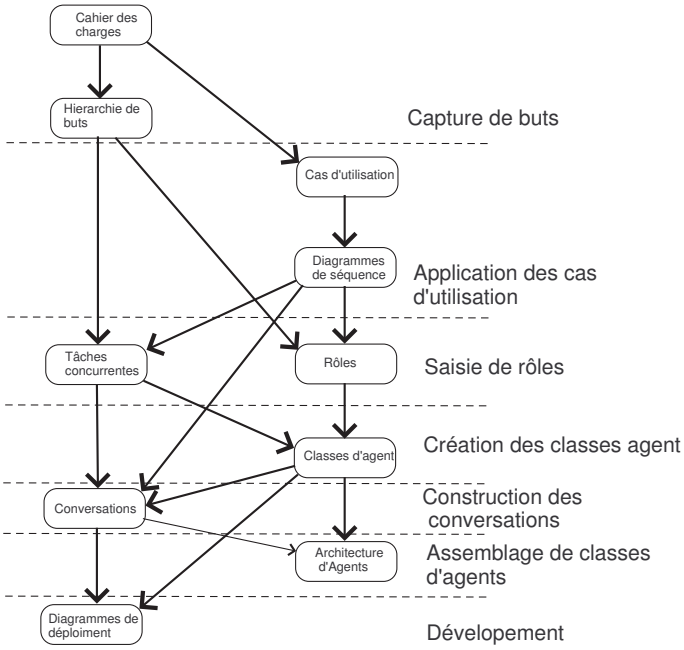
La partie d'analyse du système en Gaia est fortement centrée sur l'extraction d'une organisation, ainsi que les différents rôles qu'on va y trouver. Pour ce faire, la méthodologie propose aussi une notation particulière inspirée de la notation descriptive proposée dans la méthodologie orientée objet FUSION[3].

2.2.5 MaSE et AgentTool

MaSE et l'acronyme anglais de *Multiagent System Engineering,* et c'est le nom d'une méthodologie orienté agent créée par Deloach et al [8] pour essayer de répondre au problème du développement d'un système multiagent de façon pragmatique.

MaSE est une méthodologie en cascade itérative, fortement inspirée dans les méthodologies courantes utilisées dans le domaine du développement orienté objet, et ceci pour une raison simple: pour MaSE les *agents sont des objets* distribués et étendus. Mais dans lesquelles les séquences de événements sont clairement spécifiés, au contraire des objets simples dans lesquels par règle générale on n'a pas une notion forte de séquence. MaSE ajoute aussi quelques notions particulières au systèmes orientés agent, mais reste de façon générale comme une environnement de développement assez pragmatique, sans beaucoup de support pour des aspects plus abstraits du comportement des agents (rationalisation, par exemple).

FIG. 2.5 – *Schéma générale de la méthodologie MaSE*



Le diagramme 2.5 montre la structure générique de MaSE, avec les différentes phases du parcours du développement représentés par de boîtes arrondies, la séquence temporelle suit une convention de haut en bas, avec des liens sémantiques représentés par de flèches grasses.

MaSE possède un ensemble d'artefacts graphiques propres pour chaque phase, et qui sont représentés dans l'outil logiciel associé à la méthodologie: *agentTool*. A continuation on donnera une description simple de chaque une des étapes de MaSE:

Saisie de buts: dans cette partie du développement on doit saisir tous les buts à poursuivre pendant l'exécution du système final, en allant des plus générales aux plus spécifiques et en suivant les pistes qui donnent les premières niveaux de généralité. On peut utiliser un cahier de charges en langage naturel au début du processus, mais il n'est utile que pour générer un schéma structuré de façon hiérarchique. La hiérarchie est créée de façon assez simple, comme le montre l'exemple dans la figure 2.6

Application des cas d'utilisation: en cette phase on doit détecter les séquence des messages qui passent entre les différents acteurs du système, pendant les processus qui arrivent quand les agent essayent de satisfaire les buts du système, en se guidant du cahier de charges déjà saisie sous la forme d'un diagramme hiérarchique. La traduction n'est pas

FIG. 2.6 – *Exemple de saisie des buts*

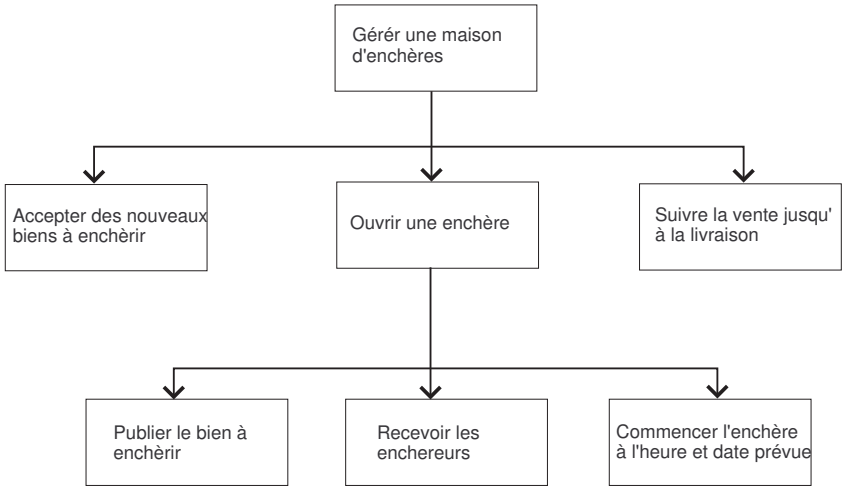
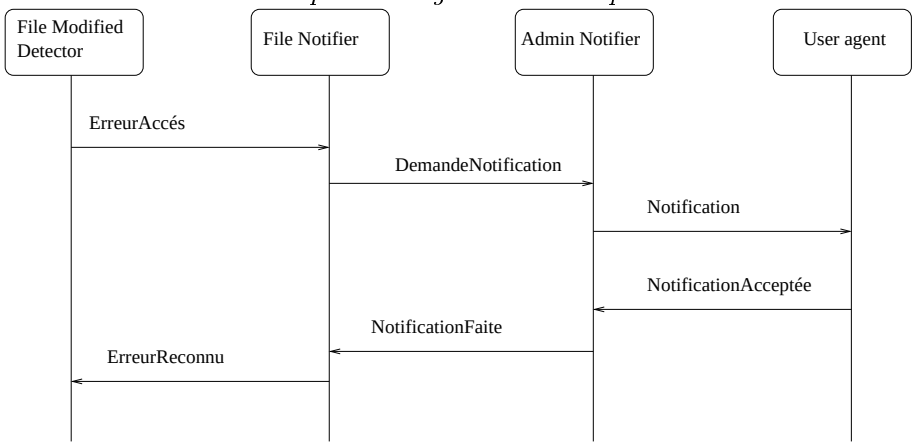


FIG. 2.7 – *Exemple de diagramme de séquence en MaSE*



directe, car pour effectuer un seul des buts prévus pour le système il se peut qu'il soit nécessaire de faire plusieurs processus d'interaction, ou bien aucun, dans le cas que un autre diagramme ait déjà satisfait les conditions d'un certain autre but. Le diagrammes utilisés sont de *chronogrammes* ou *diagrammes de séquence*, et qui sont très semblables à ceux présents dans UML 1.0[49], un exemple de diagramme de séquence peut être vu dans la figure 2.7. On utilisera un diagramme de séquence pour chaque cas différent d'interaction entre des agents.

Saisie de rôles: la prochaine étape de la méthodologie, c'est identifier tous les rôles qui doivent être pris dans le système pour que celui-ci puisse

accomplir sa fonctionnalité. Les rôles en MaSE sont des comportements *nécessaires* à la satisfaction d'un but ou pour accomplir une certaine fonction dans la structure du système, et ils sont *associés* aux agents. Ces rôles sont extraits en utilisant les diagrammes de séquence, car au même moment que proposer une solution au problème de la communication, des diagrammes suggèrent déjà des noms de modules qui seront nécessaires dans le système finale. On utilisera aussi le diagramme de saisie de buts, pour reconnaître des rôles qui doivent être pris mais qui ont été oubliés pendant la création des chronogrammes, à la fin on doit faire en sorte que chaque un des buts est poursuivi pour au moins un rôle.

2.2.6 Madkit

Madkit est un environnement de développement pour le création de systèmes multiagents proposée par Ferber et Gutknecht [18, 19]. Cette plateforme a été créée pour tester la validité de leur approche pour la modélisation de systèmes multiagent appelé Aalaadin.

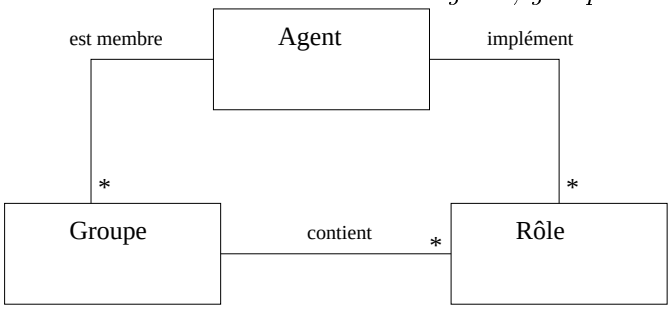
Madkit offre trois principes pour le développement:

- Une architecture d'agent légère basée sur un micro-noyau. Madkit ne force pas le développeur à utiliser un modèle particulier d'agent, pour assurer l'interaction entre des agents d'une structure interne inconnue et le reste du système, ils utilisent un *runtime* qui permet de faire cette interaction. La fonction principale du micro-noyau est d'assurer l'intégration des modules/agent dans le système, en prenant garde des aspects tels que la gestion de groupes et de rôles prises par les agents actifs dans le système, ou assurer la communication entre les entités.
- Des moyens pour donner aux services une transparence au niveau des agents dans le système("agentification"). Une particularité de Madkit est le fait que beaucoup de services (par exemple l'échange de messages, gestion de sécurité) sont considérés comme étant offerts ou représentés par des agents, ce qui permet d'introduire des changements ostensibles dans ces mécanismes et services, considérés comme atomiques par d'autres plateformes, sans affecter excessivement la cohérence fonctionnelle du système.
- Une approche pour le développement qui se sert des composants graphiques.

2.2.6.1 Aalaadin

Aalaadin ou AGR(acronyme pour Agent-Groupe-Rôle) est un modèle abstrait pour la conception des systèmes multiagents [10] centré sur l'organisation. Ce modèle qui sert de base conceptuelle à la plateforme Madkit,

FIG. 2.8 – *Modèle de la relation entre agents, groupes et rôles*



propose trois aspects qui définissent un système constitué d'agents, à savoir:

Les agents Dans AALAADIN un agent est une entité qui joue des *rôles* dans un *groupe*. Le modèle ne fait pas d'hypothèses sur la nature interne des agents, ou sur les mécanismes auxquels le développeur fait appel pour représenter leur comportement. Ce choix est délibéré, car le modèle d'agent à utiliser est laissé au gré du développeur.

Les groupes En général, un groupe est simplement un ensemble d'agents qui sont rassemblés en suivant un critère quelconque. Dans le contexte de ce modèle abstrait, les groupes peuvent exprimer aussi l'idée de fonctionnalité ou but commun. Par exemple, un système multiagent complet peut être considéré comme un groupe pendant l'accomplissement de sa tâche. Les agents sont considérés comme pouvant appartenir à un ou plusieurs groupes.

Les rôles Dans le contexte du modèle AALAADIN, les rôles sont une représentation abstraite de la fonctionnalité qui peut prendre un agent dans le système, et un attribut dynamique. Les agents peuvent prendre potentiellement plusieurs rôles, et aussi prendre des rôles différents dans différents groupes. Dans le travail de cette thèse on s'est inspiré de la notion *d'instance de rôle*, mais sans prendre en compte les aspects de contrôle d'affectation de rôle qui sont proposés par MADKIT.

La figure 2.8 montre un meta-modèle proposé par Ferber et Gutknecht dans [18, 10, 11], représentant la relation entre les trois aspects mentionnés ci-dessous.

2.2.6.2 SEdit

SEdit est un outil d'édition des notations visuelles qui vient en complément de la plateforme MADKIT. La fonctionnalité principale de SEdit est de permettre l'édition des diagrammes hautement modifiables. En étant partie de la plateforme, cet outil utilise aussi une approche orientée agent. En par-

ticulier, les artefacts graphiques individuels présents dans l'espace de travail et leur syntaxe/sémantique d'interaction sont contrôlés par des agents.

SEdit permet à l'utilisateur de définir ses propres artefacts graphiques avec leur apparence visuelle, ainsi que sa syntaxe/sémantique d'interaction associée, à l'aide d'une spécification en syntaxe XML.

La flexibilité de notation donnée par SEdit aurait été particulièrement utile dans le cours de cette thèse, malheureusement on a eu connaissance de l'outil trop tard dans le processus de développement de la thèse.

2.3 Modélisation de protocoles de communication.

Pendant la très récente révolution technologique que la disponibilité généralisée d'ordinateurs a provoqué, il a été de plus en plus évident que les fabuleuses capacités de ces machines, aurait encore plus d'utilité si elles pouvaient travailler ensemble. Pour ce faire, il a fallu développer tôt des moyens pour le faire interagir correctement.

Ce besoin de faire interagir les ordinateurs est presque aussi vieux que les ordinateurs électroniques eux-mêmes, et a produit de nombreuses solutions pour assurer cette communication désirée.

Pourtant, des problèmes particuliers furent évidents dès le début : les ordinateurs manipulent toujours des données discrets, ce qui veut dire qu'ils peuvent prendre certains valeurs seulement (par exemple, un ensemble des valeurs entiers). Dans les ordinateurs toutes les données sont fondamentalement des nombres binaires, qui sont codifiés dans la partie électronique de la machine en désignant certains niveaux de voltage comme des "zéros" et d'autres comme des "uns", donc les systèmes de communication pour ordinateurs doivent être capables d'échanger des longues chaînes de ces deux valeurs zéro et un.

Bien contrairement aux systèmes de télécommunications préexistants qui manipulaient (et manipulent) des données analogiques, lesquelles peuvent prendre un nombre indéterminé des valeurs dans un range continu(par exemple,. des nombres rationnels).

Au début les solutions étaient centrées sur les aspects plus évidents et pragmatiques du problème de l'interaction, cela dire, de trouver la façon de connecter physiquement deux ordinateurs préexistants, pour permettre l'envoi des signaux binaires sous la forme des voltages d'un point à un autre. Ce problème fut abordé en utilisant les expériences, celles-ci déjà vieilles à ce moment là, de systèmes de télécommunications préexistants (téléphone, télégraphe, radio, etc.), et en produisant des solutions naïves, mais efficaces désormais, pour assurer l'inter-communication et l'échange des données numériques binaires entre ordinateurs reliés par structures électroniques, qui étaient-elles, des modifications plus ou moins élaborées des systèmes de télécommunications pour données analogiques.

Ces structures de communication ou réseaux, ont évolué beaucoup depuis, en produisant des systèmes de plus en plus capables, fiables et rapides. Parallèlement au développement des réseaux de communications, les logiciels désignés pour les faire fonctionner(protocoles de communication) ont évolué eux aussi, en étant un des bénéficiaires du très vite développement des capacités de calcul et mémoire des ordinateurs sur lesquels ils sont exécutés.

Cette évolution des protocoles de communication, qui est aussi apparue tôt dans le procès dialectique entre hardware et software, a provoqué des effets intéressants, en produisant des nouvelles applications qui permettait(et permet encore) aux structures physiques de fonctionner au-delà parfois de ses capacités inhérentes, créant de véritables structures de communication "virtuelles" qui existent seulement comme algorithmes faisant coopérer les systèmes électroniques, ceux-ci beaucoup plus spécifiques et difficiles à changer une fois standardisées et acceptées par le marché. Mais parfois, les protocoles ont aussi provoqué de changements sur les infrastructures électroniques soujacentes, car les protocoles peuvent suggérer des changements profitables.

2.3.1 Sur les protocoles

Un protocole est l'ensemble de règles qui guide l'interaction entre deux ou plusieurs dispositifs automatiques liés par un moyen de communication. Les protocoles d'interaction permettent de décrire explicitement les enchaînements conversationnels lors des communications entre les agents. Ils représentent un schéma commun de conversation utilisé pour exécuter une tâche, une stratégie de haut niveau gouvernant les interactions entre les agents, tout en permettant de faciliter leur dialogue. Un protocole précise qui peut dire quoi à qui et quand, en représentant les réactions possibles à ce qui est dit a chaque sujet, par un langage ad hoc ou par d'autres codes. Les protocoles permettent de définir une séquence des messages communiqués entre les agents et décrivent comment les agents doivent réagir aux messages reçus durant les interactions.

Donc les protocoles sont des algorithmes qui définissent un processus de communication entre deux ou plusieurs entités. Comme tous les algorithmes, ils peuvent être spécifiés, analysés, et développés en utilisant un ensemble des méthodologies et des notations spécifiques pour eux. A travers l'histoire récente des protocoles de données numériques, il est évident que la plupart d'entre eux ont été spécifiés avec des automates à états finis, bien qu'il existent bien d'autres façons de modéliser ou développer un protocole.

Parfois, lorsqu'un problème résolu par deux ou plusieurs modules devient assez complexe, le protocole qui décrive la communication qui mène à la solution sera tout au moins aussi complexe, ce qui a mené les experts à créer une authentique panoplie de schémas plus ou moins formels de décomposition du problème en plusieurs parties plus faciles à documenter, modéliser et à implémenter.

Une fois le problème du développement d'un protocole achevé -tout au moins, dans l'apparence- la solution développée pouvais passer au statut de standard appuyé par une organisation qui déclarait le protocole raisonnablement fiable.

Cette approche a été satisfaisante pour résoudre un bon nombre des problèmes liés aux couches plus basses de la communication, cependant, la simple subdivision du problème et la spécification informelle ne suffit plus pour les applications plus modernes, car le nombre de protocoles ad hoc à développer dans les applications tend plus que jamais à augmenter, à fur et à mesure que les applications deviennent de plus en plus fédérées et capables d'interagir de façon lointaine, pour ajouter de la valeur à sa fonctionnalité.

2.3.2 Les sous-protocoles

Dans l'histoire du développement du produit logiciel, il a toujours été de l'intérêt de développeurs trouver des méthodes, stratégies et schémas pour arriver au même résultat avec le minimum d'effort. Comme on pourrait atteindre, dans le cas des protocoles de communication il arrive exactement la même chose.

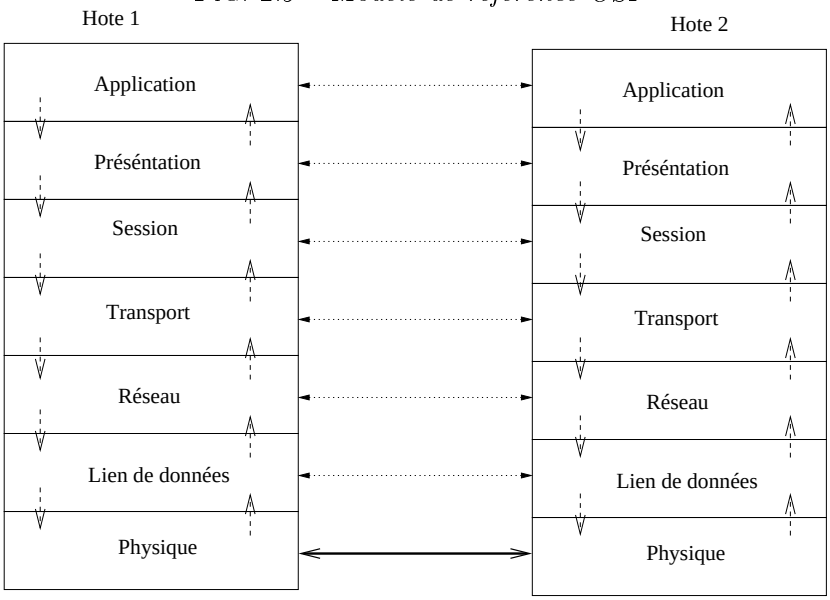
Dû au fait que pendant longtemps la fonctionnalité principale à assurer par un protocole de communication a été l'établissement, suivi et fermeture d'un lien de communication, les travaux existants sont spécialement centrés sur la manière d'assurer une telle fonctionnalité avec le minimum d'effort et le plus de clarté possible pour le développeur.

Il n'est pas étonnant donc de voir que presque tous les protocoles de niveau réseau représentent en eux mêmes un module réutilisable, car beaucoup d'entre eux sont désignés pour assurer un ensemble restreint de fonctionnalités (la connection) et pour laisser à d'autres parties de l'application finale le soin de choisir à quoi ses capacités seront utilisés.

Le modèle de référence OSI[30], qui malgré la croyance commune parmi les étudiants de réseaux informatiques, n'étais pas originalement destiné à être un modèle purement abstrait, offre cependant un bon exemple de l'approche de réutilisation habituelle chez les développeurs de protocoles de communication.

Dans la figure 2.9 une schéma représentant la structure générale du modèle de référence OSI est montrée. Le diagramme montre les sept couches ou niveaux d'abstraction dans lesquelles est divisée la fonctionnalité du protocole OSI de communication, lequel est capable de transmettre des données entre deux applications de façon fiable. L'idée principale derrière cette division en couches, est de proposer un approche modulaire dont chaque couche s'occupe d'un aspect particulier de la communication, et libère aux couches supérieures de devoir faire face à des détails répétitifs et donc superflus. En suivant cet approche, la couche inférieure par exemple (la couche physique) serait la partie du système en réseau qui fait face au problème de la

FIG. 2.9 – *Modèle de référence OSI*



traduction de donnés binaires vers signaux électriques, et l'envois *physique* de ces signaux dès l'hôte 1 vers l'hôte 2 et vice versa, en terme de *signaux* électriques (ou optiques ou radioélectriques si on veut), et ferait face aux problèmes annexes au type du milieu physique utilisé.

Toutes les couches au-dessous de la couche physique (qui est la seule avec un lien "réel"), se limitent à recevoir de l'information et des commandes à partir de la couche immédiatement supérieur, les lire et éventuellement les traduire avant de les envoyer à la couche immédiatement inférieure, de tel façon que chaque couche "croît" être en interaction avec sa couche analogue dans l'hôte lointain.

Le groupe de protocoles IP[1] est un autre exemple notoire (et celui-ci avec beaucoup d'exemples d'application) de la subdivision de fonctionnalités en couches: les protocoles de niveau application les plus courants de l'Internet (HTTP, FTP, Telnet, le X-Protocol, RealAudio, SMB et une longue liste) sont basés sur les fonctionnalités données par TCP et UDP, eux mêmes basés sur les fonctionnalités des protocoles IP (ARP, RARP, ICMP, etc.), qui à son tour, fonctionnent sur les protocoles du niveau inférieur qui interagissent avec le système réseau "réel" qui permette l'existence de la communication (Ethernet, ATM, modem, Wi-Fi).

Il est notoire donc que les concepteurs de protocoles de communication ont beaucoup utilisé l'idée de *réutilisation* du travail, et qu'ils ont privilégié pour cela la *modularisation* et la factorisation de fonctionnalités en modules spécialisés.

Cette factorisation de fonctionnalités en couches est particulièrement bien

adaptée à la façon dont fonctionnent les systèmes d'APIs qui existent pour communiquer les processus d'utilisateur avec le système de exploitation, et permettent de définir les couches inférieures comme des services "à boîte noire" offerts par le système d'exploitation, et le développement des logiciels peur s'abstraire des détails embarrassants et se centrer sur la fonctionnalité à établir.

Un des résultats les plus clairs de cet approche, est une réduction importante dans la quantité du travail associée au développement d'un protocole de communication. En termes pratiques, on peut appeler ce schéma pour économiser des efforts de développement la *réutilisation par délégation*, car on délègue à une entité ou module tiers le soin d'accomplir correctement la tâche de communiquer, et que cet module peut être invoqué par une quantité non déterminé d'applications.

Il pourrait être intéressant de savoir s'il est possible d'avoir un tout autre genre de réutilisation, qui pourrait être appelé *réutilisation par composition*. Et qui serait appliquée plutôt en temps de conception/implementation du système.

Il a été démontré que c'est possible[57], et cette idée a donné naissance au concept de sous-protocole. Les sous-protocoles [57, 36, 35, 37] sont des séquences partielles et réutilisables d'acte de communication qui définit l'ordre temporel de la séquence mentionné au-dessus avec un processus de communication spécifique entre les rôles. Les rôles représentent le comportement qu'une entité donnée porte dans un protocole, ils pourraient être utilisés et répétés plusieurs fois dans le système. Ils sont comme les composants d'une machine dont l'assemblage de ces composants construit le système.

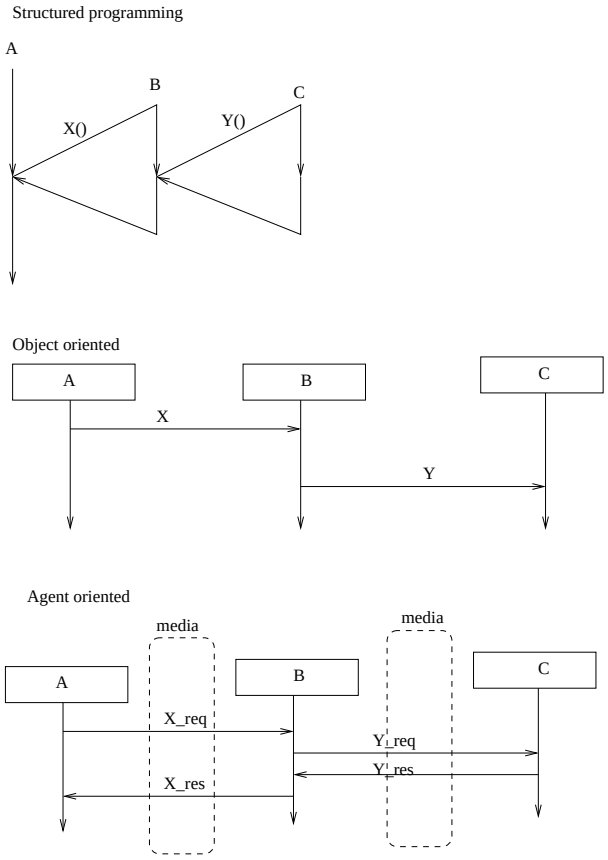
En ce cas, on suppose que chaque protocole est composé en plusieurs sous-protocoles. Un protocole peut contenir plusieurs sous-protocoles généralement différents, mais il peut contenir des sous-protocoles similaires dans les cas de comportement itératif.

2.3.3 Protocoles d'interaction

Une des caractéristiques principales du logiciel orienté objet est l'encapsulation des fonctions et des données dans une classe. Cette encapsulation est utile parce qu'en associant une fonction à un module de programmation spécifique, nous créons également un modèle mental du système, où chaque module est différencié des autres par le rôle et les attributs qu'il prend à l'intérieur de l'environnement que nous imaginons.

Un autre effet positif de l'encapsulation, c'est d'obliger le développeur à penser en termes d'interactions parmi des modules et pas en termes de fonctions et appels à fonction. Les méthodes publiques des objets définissent la manière dont ils peuvent interagir entre eux, proposant ainsi l'idée que l'interface publique de l'objet, est une sorte de *contrat* ou de protocole entre objets, ce qui permette de prendre plus loin notre dramatisation du système,

FIG. 2.10 – *Évolution simplifiée de l'interaction*



et qui permet en principe de penser en terme de dialogues entre *rôles*, ce qui se rapproche plus de la façon dont les êtres humains conçoivent le monde (histoires à la place de fonctions).

Le schéma 2.10 montre un modèle simpliste du changement subtil qui est arrivé au concept d'interaction . Dans la programmation structurée ce concept d'interaction entre modules a été brouillée par l'idée de la subdivision du problème en sous problèmes plus simples, avec une approche de division haut en bas. En dépit de ce fait, les programmeurs ont noté bientôt qu'il était plus facile de trouver une fonction et de comprendre un code source, s'ils groupaient les fonctions qui avaient un lien sémantique dans un même fichier source, que de les laisser sans une classification quelconque.

Sur le schéma 2.10 les flèches représentent le flou de l'exécution, et les boucles représentent un appel à procédure ou fonction. Nous pouvons voir qu'appeler une fonction $x()$ sur un module B, et appeler une fois à l'intérieur de $x()$ à une fonction $y()$ dans un troisième module C, est curieusement

semblable à une invocation de méthode entre classes; le subfigure au milieu montre la même situation mais cette fois en utilisant une approche plus ou moins orientée objet, où les modules sont des vrais objets appartenant à une classe; ici les appels à fonction sont considérés des invocations de méthode, et leurs occurrences s'appellent *messages* ; la notation séquentielle est complètement intentionnelle et sera expliquée après.

Certainement, en code binaire il n'y avait pas tellement de différence entre le premier et le deuxième cas, mais dans l'esprit des réalisateurs la différence est énorme: l'algorithme est maintenant légèrement plus près d'un manuscrit, d'une *histoire*, que dans le cas de la simple énumération tout à fait artificielle, des comportements atomiques faits par aucune entité différentiable quelconque.

C'est une raison cognitive profonde, celle qui nous motive à imaginer les systèmes comme un dialogue entre acteurs: il est beaucoup plus naturel pour les êtres humains de représenter les événements d'un système, une fois que le problème a été *personnifié*. La différence se trouve la plupart du temps dans le fait de pouvoir imaginer qu'une entité fait un ensemble de choses, et l'ordre temporel dont elle les fait. Ainsi, il n'étonne pas que beaucoup des diagrammes représentant la séquence temporel de messages, soient utilisés dans les méthodologies orientées objet comme un artefact standard de développement.

Chez les agents ce processus de dramatisation va une étape plus loin : les entités dans le système, appelées les objets avant, sont des agents désormais. Ce changement apparent léger d'approche nous permet, au moins mentalement, d'assigner à l'entité un ensemble de propriétés non associées précédemment au concept de l'objet. Entre ces nouvelles propriétés, il y a *l'autonomie* . La plupart des auteurs font une emphase forte sur l'autonomie comme facteur principal définissant un agent, ce concept est plus ou moins ambigu, principalement parce que les auteurs ont proposé différentes idées au sujet de ce que attribue-t-on à un agent pour qu'il soit considéré autonome, et ces attributs vont du fait de l'exécution sur différents processus de système ou sur différents systèmes physiques, au comportement intentionnel plus complexe des agents sociaux. Pour cette raison et pour notre propre avantage, nous croyons qu'il est nécessaire de définir l'agent et l'autonomie comme suit:

Comportement autonome: nous prenons un point de vue assez simple au sujet du comportement autonome, un agent autonome est l'équivalent d'un processus ou d'un fil de système d'exploitation, il s'exécute séparément des autres et la seule manière de coopérer avec l'univers d'agents dans son environnement, est l'utilisation d'un mécanisme de communication entre processus. Mais cette définition est si ouverte qui permet l'inclusion de presque n'importe quelle application distribuée ou multitâche, ainsi nous devons définir d'une manière quelconque ce

qui est un agent.

Agent: encore nous prenons un point de vue conservateur. Un agent est une entité qui a la capacité de dire "non" à une certaine demande, ou qui est capable de dire "allons-y" suivant le déclenchement d'un événement interne, ou tous les deux. Les mécanismes internes déterminant quand dire "non" et quand dire "allons-y" sont considérés comme d'un niveau supérieur à la description simpliste que nous utilisons.

Si nous maintenons dans l'esprit ces deux restrictions minimales pour définir un système en tant que 'multiagent ', nous pouvons voir clairement ses similitudes avec les systèmes distribués, ainsi la seule "colle" permettant aux entités d'un tel système de travailler ensemble est clairement, les protocoles d'interaction entre l'entités(les agents).

La partie inférieure du schéma 2.10 montre une traduction possible pour le diagramme orienté objet situé au milieu. Donc ici les modules A, B et C sont des agents, et les appels à méthode sont remplacés par l'envois de messages à travers le mécanisme de communication inter-processus utilisé par le système (habituellement un ensemble de couches de communication en réseau), ce qui reçois habituellement le nom *d'interaction*.

Dû au fait que le mécanisme de communication inter-processus employé pour répandre les interactions n'est pas spécifié, il est raisonnable de représenter la transmission de chaque message séparément, ce qui produit une simple séquence de messages ordonnés dans le temps (un protocole), ainsi le diagramme dans la partie inférieur exprime l'occurrence d'un message `X_req` de A à B, lequel une fois reçu cause l'envoi d'un message `Y_req` de B à C , provoquant une réponse `Y_res` de C , qui est reçu à son tour par B , qui ensuite envoie `X_res` comme réponse à A .

Certainement, le saut des diagrammes de séquence de messages entre objets aux protocoles d'interaction n'est pas aussi direct et simple qu'il semble dans la figure: traduire les invocations de méthode entre les objets vers des interactions entre agents d'une mode linéaire, ne crée pas vraiment un protocole. N'importe quel protocole réel doit prendre en considération des aspects importants, dont un système monolithique orienté objet peut bien s'en passer: comme l'authentification, le chiffrage, le contrôle d'erreurs de communication et la gestion de la session. Même pour des systèmes où la gestion de la session est assistée par un tableau noir ou par un courtier inclus dans la plateforme, chaque agent doit prendre soin des messages qu'il reçoit et des réponses qu'il émet, des attentes de ses partenaires, des restrictions temporelles et des paramètres, afin d'assurer le fonctionnement. La flexibilité qui vient de l'indépendance, comme tout dans la vie, a un prix.

2.3.4 Sous-protocoles d'interaction

De nos jours, l'industrie et les consommateurs exigent plus fortement que jamais un niveau élevé de qualité dans leurs produits. Dans le cas du logiciel, il est difficile de définir et prouver l'existence de cette qualité. Ceci se produit parce qu'il est extrêmement difficile de vérifier exhaustivement que le comportement du système est correcte et qu'il ne contient pas d'erreurs. La plupart des programmes sont des entités complexes, sujets à des facteurs presque aléatoires qui rendent extrêmement difficile d'assurer un comportement robuste ou fiable dans le moyen ou le long terme.

La plupart des réalisateurs suivent une approche pragmatique d'essai en corrigeant le plus exhaustivement possible leurs produits, et on accepte comme un fait de la vie que le système assemblé toujours contiendra des comportements indésirables et des erreurs. Il est possible que la perception et l'esprit humain ont une limite pour que sa capacité à traiter la complexité et pour comprendre, et que tout programmeur , même le plus soigneux et le plus expérimenté produit une longue série d'erreurs écrivant un programme au delà de certain degré de complexité.

Cependant, la pratique de la programmation a également prouvé qu'une compréhension entièrement abstraite du système (ce qu'il fait, comment il le fait) est un traitement très bon contre la dispersion mentale sur des détails faisant face à la complexité; la pratique et la vie quotidienne prouve également qu'il est plus facile que nous fassions face à seulement un problème simple à la fois, cette restriction normale affecte évidemment la manière dont nous résolvons des problèmes, ayant pour résultat des approches semblables de solution dans des domaines très différents. Chaque fois que la complexité se développe au delà de notre capacité, nous tendons à créer les modèles abstraits qui divisent la complexité dans de plus petits instances du problème toujours que c'est possible.

Dans les systèmes orienté agent, la subdivision du problème est au fond même du procédé de développement: l'autonomie est un des caractéristiques principales des agents, et elle fait que la modélisation de l'interaction entre les pièces une question plus qu'appropriée. Nous considérons que les méthodologies orientées agent devraient souligner et encourager l'utilisation des modèles basés sur l'interaction, afin de rendre les interfaces et les protocoles de système un facteur principal à l'analyse et à la conception du système.

Cependant, l'introduction des protocoles dans la conception sur n'importe quel méthodologie orientée agent est une tâche difficile, parce que la plupart des protocoles sont des entités complexes exigeant un étude soigneux par eux-mêmes. Pour réduire cette complexité, et rendre de ce fait le procédé de développement plus facile pour des réalisateurs, quelques mesures méthodologiques doivent être prises. Comme exemple de ce étapes méthodologiques, nous proposons dans cette thèse plusieurs algorithmes qui réalise la traduction automatique des protocoles d'interaction à un modèle étendu

de machine d'état fini, après la discussion de quelques problèmes ouverts de la programmation orientée d'agent.

Pour exemplifier un tel procédé de traduction automatique, nous choisissons une représentation visuelle confortable pour des interactions (diagrammes d'interaction de UML 2.0) pour appliquer au-dessus d'elle notre algorithme proposé, obtenant à l'extrémité un modèle de machine d'état fini exprimé avec Promela.

2.4 Vérification formelle des protocoles.

En informatique, la tendance normale de l'esprit humain à subdiviser les problèmes a produit différentes approches pendant la brève histoire de l'art de la programmation: de la programmation structurée surgit durant les années '70 du siècle passé, à la programmation orientée objet des années 80 et débuts des années 90. Le génie logiciel en science pragmatique a subit divers changements d'approche, ce qui a résulté en une série de méthodologies devenues obsolètes alors que leur modèle d'inspiration faisait la même chose.

Les protocoles de communication sont aujourd'hui aussi utiles qu'ils l'étaient il y a vingt ans, peut-être encore plus. Un protocole est tout simplement un ensemble de règles et de comportement suivis chaque fois qu'un système lié à un réseau communique avec les autres. A une certaine époque, la conception de protocoles visait principalement à résoudre les problèmes qui surgissent chaque fois qu'une communication avait lieu par les médias peu fiables de l'époque.

Les problèmes de communication associés aux médias peu fiables, ont été maintenant en grande partie résolus pour presque toutes les applications courantes, et ils ne sont plus une préoccupation majeur (excepté pour les applications en temps réel, comme la transmission de données multimédia, où il existe encore beaucoup d'intérêt de recherche, ou sous des problèmes bien particuliers, comme l'Internet interplanétaire). Mis à part les rares contre-exemples, la conception de protocoles et la recherche dans ce domaine est maintenant située à un niveau plus élevé d'abstraction. Toutefois la conception de protocoles n'est pas non plus un problème trivial, même en utilisant des canaux de transmission et des protocoles de communication fiables.

Une entité communiquant par un réseau fiable doit résoudre des problèmes récurrents comme créer, terminer, fixer et contrôler une session. Dans un monde où les violations de sécurité sur les systèmes gérés en réseau sont un grand souci, n'importe quel système réaliste doit assurer un ensemble minimal de dispositions de sécurité. Pour les développeurs, cette augmentation des restrictions de sécurité accroît la charge de travail nécessaire pour mettre en ouvre un protocole, même insignifiant en apparence.

Il y a des cas où des solutions générales se sont avérées très efficaces pour des problèmes de sécurité en réseau, par exemple, l'emploi des proto-

coles standardisés basés sur le chiffrage à clef publique pour cacher toutes les données transmises dans un canal (par exemple SSL pour des transactions sécurisées du protocole HTTP); mais pour beaucoup d'applications basées sur la coopération et l'enchaînement de compétences de plusieurs modules indépendants, ces solutions ne sont pas suffisantes. Pour assurer la sécurité d'un système réparti il est nécessaire de rechercher exhaustivement des erreurs dans le comportement de système, et pas seulement chiffrer chaque connexion.

La plupart des développeurs confrontés au problème de la vérification de la sécurité suivent une approche pragmatique: analyser, concevoir et programmer le système, et corriger alors chaque chemin possible d'exécution. Malheureusement la plupart des protocoles commerciaux sont extrêmement complexes, avec une surabondance de détails d'exécution qui sont très facilement ignorés dans le procédé de développement des systèmes modernes, qui est habituellement contraint dans le temps.

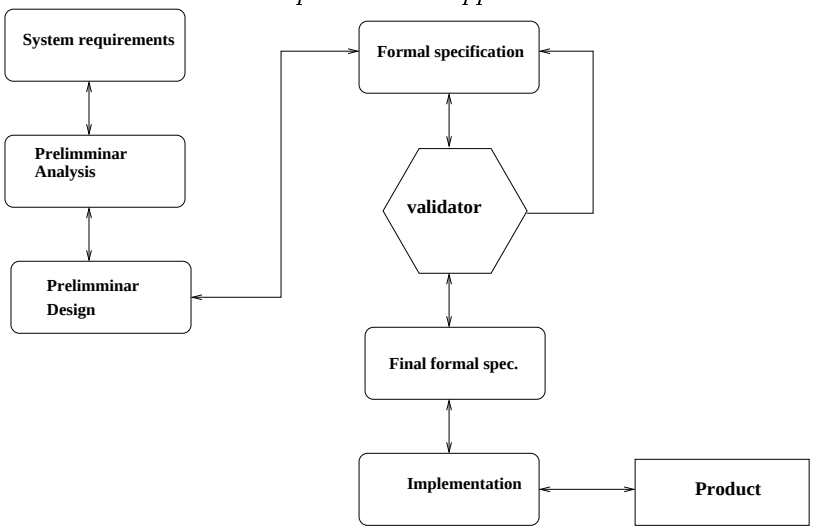
Le problème des erreurs produites par des omissions dans la vérification n'est pas nouveau. Dès le début, les réalisateurs de système distribués se sont rendus compte des difficultés pour assurer non seulement la création d'un système complètement sûr, mais également pour un système avec un fonctionnement à peine acceptable.

Parallèlement aux approches pragmatiques pour l'analyse et la conception de systèmes, des solutions de rechange sont apparues sous la forme de méthodologies qui utilisent des notations formelles. Une notation formelle est un langage lisible par une machine, conçu pour permettre la représentation et la validation automatique des propriétés d'un système, à l'aide d'un logiciel qui accepterait en entrée du code exprimé dans la dite notation. Étant donné que le domaine des méthodes formelles en informatique est plutôt large, notre intérêt sera concentré sur les méthodes et notations employées pour modéliser et valider des systèmes parallèles et des protocoles de communication, comme LOTOS, ESTELLE et Promela/SPIN.

Supposons que la spécification M' du système M soit une expression lexicalement et syntaxiquement valide en L qui représente le comportement du système M, et que V_L soit un programme qui accepte des expressions en langage L comme entrée. Si le modèle M' une fois lu par V_L satisfait un ensemble minimal de propriétés (comme la vivacité ou la sécurité), il est possible de dire que la spécification M'est valide. Il est intéressant de noter que les langages formels et leurs outils respectifs de validation peuvent répondre à la question: "la spécification M' est-elle valide ou pas?", mais ne peuvent pas répondre à " est-ce que M' un modèle précis du système M?".

L'intérêt principal conduisant à l'utilisation des méthodes formelles en général, est de déléguer la détection d'erreurs au programme qui accepte le langage formel, et de réduire ainsi l'occurrence d'erreurs fonctionnelles non détectées habituellement par les réalisateurs humains. Pour les protocoles, la détection d'inconsistances dans la séquence d'échange de messages es la dé-

FIG. 2.11 – *Étapes du développement basé en model-checking*



termination de la vivacité de tous les états possibles est un aspect crucial du développement, étant donné la quantité élevée de messages que un protocole utilise habituellement.

Le schéma 2.11 montre une vue très générale du processus d'analyse et de conception typique pour un protocole, quand on est assisté par une méthode formelle de spécification. Le processus commence toujours par créer ou bien lire une description informelle du protocole à modéliser (par exemple un document de spécification, un RFC, etc. écrit en langage naturel), suivie d'une analyse et d'une conception préliminaire dont le but est d'améliorer la compréhension du problème. Ces trois premières étapes sont toujours faites, que la documentation soit créée ou pas, parce qu'il est impossible de prendre les spécifications informelles d'un protocole et de les traduire vers une spécification formelle à la première lecture (un processus de familiarisation est toujours impliqué quand nous sommes confrontés à un problème). Une fois que le système est plus ou moins compris, alors le processus formel de spécifications peut vraiment commencer.

Évidemment les détails sur la façon dont un processus formel de spécifications se déroule changent d'une technique à l'autre, mais une grande majorité des méthodes formelles spécialisées dans les systèmes de communication, porte sur la modélisation du comportement dynamique plutôt que celui des données du système, et utilisent des machines d'état fini pour le représenter. La plupart des vérificateurs ont habituellement un comportement itératif, où l'outil qui valide prend des spécifications d'entrée et détermine si quelques propriétés appropriées sont satisfaites par elles, comme la sécurité et la vivacité. Au cas où il y aurait des propriétés qui sont non-satisfaites, les spécifications d'entrée doivent être modifiées jusqu'à ce qu'elles le de-

viennent. Une fois que des spécifications passent le processus de vérification de propriétés, appliqué par le logiciel de validation, il est possible d'accepter les spécifications comme correctes ou de les raffiner plus loin, pour arriver à un plus grand niveau de détail.

Malheureusement, les techniques formelles ajoutent une quantité de travail non négligeable au procédé déjà difficile du développement de la plupart des systèmes logiciels, limitant de ce fait leur applicabilité à quelques domaines très particuliers: chaque fois que le système à développer a besoin de la validation la plus forte possible (systèmes de sécurité critique, par exemple), où les créateurs sont disposés à investir le temps et l'effort nécessaires à un plus long processus de spécifications. Plus encore, la plupart de méthodes assistées par un outil de validation formelle sont restreintes à comprendre et vérifier le problème, mais les développeurs doivent réaliser le produit logiciel par eux-mêmes.

En dépit de la difficulté initiale plutôt élevée quand on utilise des notations formelles, les systèmes complexes comme les protocoles de communication sont des sujets habituellement très adaptés à l'utilisation d'une méthode formelle. Les protocoles de transmission sont un des exemples de logiciels les plus complexes disponibles, et un dont il est le plus difficile de détecter les comportements incorrects, particulièrement ceux qui surgissent des malentendus du document de spécifications.

Toutefois il vaut la peine de mentionner que tandis que la plupart d'articles sur la validation formelle des protocoles montrent des résultats encourageants pour ceux disposés à les employer, il n'y a pas beaucoup d'exemples de l'utilisation réelle de méthodes formelles dans des protocoles de classe industrielle [59]. Ceci est provoqué par beaucoup de facteurs, depuis la réticence toute simple des ingénieurs de protocoles à apprendre et utiliser un outil formel, jusqu'au fait qu'un protocole déjà mis en application depuis plusieurs années n'a pas un grand besoin de validation.

Pour approfondir un peu les concepts exprimés dans cette section, on parlera de l'approche le plus courant pour modéliser les systèmes faits d'entités communicantes, les automates à état fini, et de quelques exemples de langages formels qui ont été utilisés avec succès pour modéliser ces systèmes, à savoir, Estelle, LOTOS, Promela/SPIN et les réseaux de Petri.

2.4.1 Systèmes discrets communicants, états et transitions

Le travail de cette thèse on s'intéresse aux systèmes discrets communicants, en particulier, aux systèmes logiciels communicants. L'idée de système discret en générale, il est important de le noter, est très étroitement liée à celle de machine, de dispositif artificiel qui accompli une fonction, et dans presque tous les exemples de cette thèse, on supposera que les systèmes discrets dont on parle sont des machines automatiques programmables ou machines-logiciels.

Un système discret est toute entité qui peut prendre une quantité finie d'états différentiables significatifs. Un interrupteur électrique qui allume une ampoule est un exemple de système discret, car il peut prendre seulement deux états différentiables significatifs: allumé et éteint; tandis qu'un dispositif comme un baromètre, n'est pas considéré comme discret car il peut prendre un nombre indéterminé d'états (toutes les valeurs de pression atmosphérique acceptés dans le domaine opérationnel du dispositif).

Dans le cas de systèmes discrets communicants et parallèles, il est depuis longtemps démontré qu'ils peuvent être représentés avec le formalisme d'automates à état fini. Et il est très courant de voir qu'on utilise un approche pour la validation appelé *analyse de traces*[55]. Approche qui consiste à représenter et analyser d'une façon algorithmique, toutes les séquences de changements d'état possibles de l'automate qui fait du modèle du système, en essayant spécialement de retrouver celles considérées comme indésirables.

Sous cette approche, l'ensemble qui contient tous les états possibles que peut prendre un système discret donné, reçois communément le nom d'espace d'états. Dans le cas de l'ampoule citée auparavant, cet espace d'états serait un ensemble $S = \{allume, eteint\}$.

Si on accepte l'idée de représenter tous les état possibles d'un système discret dans un ensemble, il est naturel d'imaginer que le système a à tout moment un *état actuel* $e \in S$, et que cet état actuel e varie à travers le temps en suivant une progression caractéristique du système représenté. Pour représenter tous les changements d'état qui peuvent arriver à un système quelconque, on peut utiliser une notation de graphe orienté, dont les nœuds sont des états, et les arêtes représentent la possibilité de changement vers un état quelconque à partir de l'état actuel.

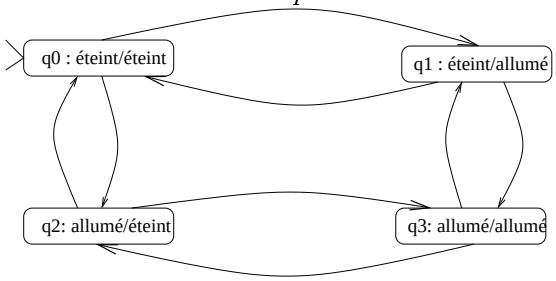
Les notions d'espace d'états, état et transition sont fondamentales pour comprendre une bonne partie des techniques de modélisation formelle de systèmes discrets communicants. Malheureusement, en modélisant de la sorte des systèmes nous nous rendons vite compte que l'espace d'états d'un système relativement simple en apparence, peut être potentiellement gigantesque (millions, voir milliards d'états différents), ce qui peut bien les rendre impossibles à manipuler de cette façon (en énonçant l'espace d'états), car nos capacités de manipulation et de stockage des états se voient tout simplement dépassées.

Le problème n'est pas facilité quand on essaye d'énoncer l'espace d'états d'un système discret composé d'autres, ou qui travaille avec d'autres en parallèle. Un exemple élémentaire de cette explosion d'états, serait d'imaginer qu'on veut générer l'espace d'états d'un système s' composé de deux interrupteurs analogues a celui nommé auparavant. Dans ce cas, on peut facilement voir que l'ensemble d'états de s' est:

$$S' = \{(allume, allume), (allume, eteint), (eteint, allume), (eteint, eteint)\}$$

Il est aussi facile de voir que toute addition de sous-éléments individuels

FIG. 2.12 – *Exemple d'automate.*



à deux états chacun, produirait une croissance dans le nombre d'états possibles de l'ordre de 2^n, où n est le nombre d'éléments individuels inclus dans le système. Tous ceux qui connaissent un peu de complexité algorithmique seront d'accord de dire que c'est extrêmement mauvais.

Heureusement, il existe des mesures palliatives à ce problème de l'explosion d'états. Une de ces mesures est de représenter seulement les états du système qui sont significatifs pour le problème qu'on veut résoudre, une autre est d'utiliser des notations plus expressives et concises que la simple énumération de tous les états possibles du système.

Traces

Un concept important dérivé de l'application d'automates pour la modélisation de systèmes discrets communicants, sont ceux de *trace* et *ensemble de traces*, appelés aussi respectivement *comportement* et *propriétés*.

Dans les aspects le plus pertinents de la modélisation par des automates à états finis, se trouve le fait que le propriétés d'un système peuvent être représentées comme des *traces*. Une trace est simplement une séquences d'états qui peuvent être visités suivant l'enchaînement des transitions de l'automate.

Soit un automate $M = \{Q,\Sigma,T,s,F\}$qui modélise un système discret m, une trace ou comportement et une séquence ordonnée d'états $t = \{s_1,...,s_k\}$ dont $s_1,...,s_k \in Q$, qui énonce une des séquences possible de changements d'état qui satisfait l'expression $s_1 \longrightarrow s_k$.

Par exemple, en reprenant l'exemple du système fait de deux interrupteurs, on peut facilement voir que son automate serait comme celui représenté dans la figure 2.12, si on prend comme hypothèse que le changement instantané des deux interrupteurs à la fois est impossible.

Dans cet exemple, une trace ou comportement seraient les séquences $\{q_0q_1q_3\}$ et $\{q_0q_1q_3q_2q_0\}$, pendant que la séquence $\{q_0q_3q_2q_1\}$n'est pas une trace admise par l'automate.

2.4.2 Propriétés désirables

Une fois qu'on a énoncé tous les états et transitions possibles d'un système discret, soit de façon explicite ou avec une notation concise quelconque, il devient nécessaire de déterminer si le modèle est valide. Cependant, il faut aussi savoir ce qu'est cette validité.

Comme on a dit auparavant dans la section 1.2.3, le processus de validation consiste a prouver que le modèle du système satisfait un ensemble minimal de propriétés désirables. Cependant la nature de ces propriétés désirables est un concept très variable en fonction du type du système qu'on est en train d'étudier ou développer. Dans [55], Rushby énonce au moins trois types d'approches différentes pour le développement et la modélisation formelle : les systèmes où la fiabilité prime avant tout, les systèmes de sécurité critique et les systèmes temps réel; qui expriment chacun ses propres définitions sur les propriétés qu'un système doit avoir pour être valide, et qui sont tous parfaitement raisonnables dans le contexte d'application où ils sont utilisés. Il faut noter aussi que les définitions de ce qui est désirable ne sont pas nécessairement disjointes, mais elle peuvent être contradictoires parfois.

Si on devait positionner notre intérêt de recherche dans un de ces trois axes, il serait prudent de dire que ce serait la fiabilité qui prime avant tout. Malgré cette approche qui délaisse intentionnellement les aspects de sécurité critique et de temps réel, on reste assurés du fait que les propriétés qui assurent la fiabilité d'un système sont assez souvent des propriétés qui améliorent la sécurité.

Dans [2], Alpern et Schneider démontrent que dans le cas des modèles basés sur l'analyse des séquences d'états (ou traces), ces propriétés désirables peuvent être réduites à deux catégories[42]:

Sécurité: le fait de dire que le modèle d'un système quelconque est sûr, implique en termes abstraits que rien de mauvais n'arrive. Quand on parle de "rien de mauvais" on veut dire qu'en suivant toutes les séquences du graphe de transitions du modèle, on ne peut jamais se trouver dans un état qu'on considère comme indésirable. Il ne faut pas confondre état indésirable avec état d'erreur détecté, car bien qu'on puisse considérer que quand le système se trouve dans un état où il ne peut plus poursuivre sa fonction normalement, on arrive à un état indésirable. Il est plus correct de dire que l'état d'erreur détecté est l'état dans lequel on souhaite que le système soit, chaque fois qu'il perçoit que quelque chose lui interdit d'accomplir sa fonction correctement. Dans la plupart de cas, les états réellement indésirables sont des états non planifiés, non prévus, où on ne sait plus désormais quel sera le comportement du système.

Vivacité: la vivacité peut être définie de façon abstraite comme le fait qu'au cours de l'exécution du système, il arrive éventuellement quelque chose

de bon. La bonté assignée à une séquence d'états du système est un terme abstrait, bien entendu, mais en règle générale on présume que si dans chaque état du système, il y a une séquence d'états qui mène vers un état désirable dans l'accomplissement de sa fonction, on peut dire que le système est vivace.

Un autre résultat intéressant du travail de Alpern et Schneider, est qu'ils démontrent aussi que chaque propriété vérifiable par l'analyse de traces est une intersection d'une propriété de sécurité et une de vivacité. Ils vont jusqu'à dire que toutes les propriétés significatives vérifiables sont une intersection de ces deux types de propriétés, considération qui a été critiquée et redéfinie pour se centrer uniquement sur les propriétés vérifiables par analyse de traces.

Une classe particulière des propriétés de sécurité est les invariants, qui sont ces propriétés qui affirment que dans chaque état du système, il est vrai qu'on n'arrive jamais à un mauvais état à partir de là. Par exemple, on peut citer le cas où l'on détecte qu'il y a dans un système des transitions qui arrivent à un état non-final où il n'est plus possible de changer d'état (un deadlock).

2.4.3 Le TDF Estelle

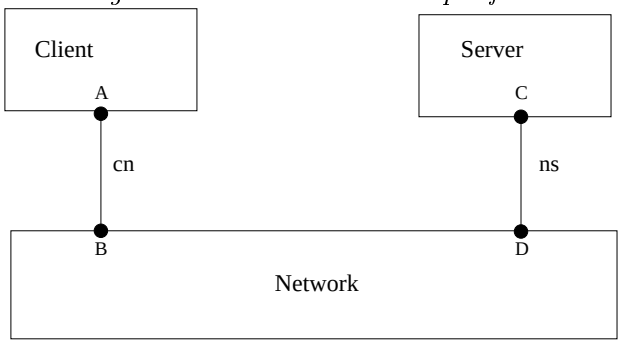
Estelle (Extended State Transition Language) est une de deux techniques de description formelle définies par l'ISO pour la description des protocoles de communication[27, 31]. La syntaxe et la sémantique du langage sont bien définis dans des documents proposés par l'ISO, mais il n'y a pas un outil logiciel standard associé avec le langage. Cependant il existe plusieurs outils développés par des tiers (EDT, PetDingo [12], ESTIM [7], XEC [58]) qui acceptent le langage Estelle ou des dialectes.

Estelle permet de modéliser le système en utilisant une structure hiérarchique de modules, où chaque module peut être décomposé en sous-modules plus simples. Les modules communiquent à travers des points d'interaction et les messages sont véhiculés par des canaux de communication. Le comportement dynamique de chaque module, où qu'il soit dans la hiérarchie, peut être décrit par un automate étendu communicant. Les principaux éléments du langage Estelle sont:

Module: Les modules dans ce langage représentent explicitement des machines à états finis. La notation permet de définir l'ensemble d'états du module, les transitions et les messages échangés parmi les différents automates qui font la structure du système.

Point d'interaction: Un point d'interaction est le point d'accès d'un module. En Estelle le seul type de communication supporté correspond au type asynchrone. Donc chaque point d'interaction a une queue or-

FIG. 2.13 – *Diagramme modulaire d'une spécification* ESTELLE



donnée associée où les messages d'entrée sont stockés jusqu'à sa récupération explicite.

Canal: Les canaux servent comme lien explicite entre deux, et seulement deux points d'interaction. Les canaux sont des entités déclarées explicitement dans la spécification. Elles n'ont pas de capacité de stockage de messages et sont indifférentes au contenu qu'elles portent. Sa fonction principale est de permettre de définir la structure du système à modéliser et de représenter les mécanismes d'échange de données.

La figure 2.13 montre un exemple d'une vue d'ensemble d'un système spécifié en ESTELLE, la notation n'est pas standardisée, mais elle est copiée de celle presentée dans les articles sur l'outil ESTIM[7]. Les modules D'ESTELLE suivent un approche du subdivision du problème semblable a celui de classes dans les langages OO, bien que ESTELLE ne soit pas du tout un langage orienté objet. Dans le modèle exprimé dans la figure 2.13, il y aurait un module définissant le comportement des entités du type *Client* par exemple, tandis que *le nombre* d'entités du type *Client* dans le système est définit ailleurs (dans le corps principal de la spécification). Dans le langage, la définition de chaque module est en quelque sorte analogue la définition d'un type de données abstrait, et le corps principal de la spécification (analogue au procédure principale d'un logiciel en Pascal[62]) représente l'environnement où les entités existent.

Dans ESTELLE, une spécification est un texte ASCII standard qui décrit le comportement observable d'entités de boîte noire. Ces entités sont liées par les canaux de la communication.

Le code 1 est un exemple incomplet de spécification ESTELLE , c'est la spécification du corps (comportement interne) pour un simple échange de messages. Comme on peut voir, la syntaxe est similaire à celle du langage de programmation Pascal.

La première déclaration définit l'ensemble d'états de l'entité (ON,OFF), la déclaration suivante spécifie quel état est l'état initial (OFF). La partie

Algorithm 1 Exemple de code en ESTELLE

```
body descrSwitch for Switch;
state OFF, ON; \{ Records ON/OFF state \}
initialize to OFF
  begin
  end;
trans \{ *** Activate/deactivate switch *** \}
      when P.TurnOn
            from OFF to ON
                begin
                    output P.TurnedOn
                end;
        when P.TurnOff
            from ON to OFF
                begin
                    output P.TurnedOff
                end;
end; \{ SwitchBody \}
}\end
```

suivante spécifie les transitions et les symboles de l'entrée qui les déclenchent. Il y a seulement deux transitions, la première transite de 'OFF' à 'ON' avec le symbole d'entrée 'P.TurnOn'. La deuxième transition de 'ON' à 'OFF' avec le symbole d'entrée 'P.TurnOff'. Les paires début-fin autorisent d'insérer du code qui va être exécuté après que l'événement précédent ait été déclenché.

Les outils qui acceptent des spécifications exprimées en ESTELLE varient beaucoup dans leurs capacités à effectuer des analyses formels d'une spécification donnée. Mais il ont en commun plusieurs choses:

- *Un approche dite "d'ingenieur"*, étant donné que la syntaxe du langage est très facile à comprendre (car inspiré du langage de programmation Pascal[62]), presque "du pseudocode", les algorithmes modélisés peuvent être spécifiés d'une façon qui est très proche de son implémentation finale.
- *Le langage est facilement interprété* ou traduit vers une représentation exécutable. Tous les outils ou bien permettent d'interpréter et d'exécuter la spécification ou bien ils gênèrent du code dans un langage de programmation quelconque (du C par exemple). Dans le cas particulier de Pet/Dingo et EDT, cette capacité est étendue vers la génération rapide de prototypes.
- *La spécification se fait par décomposition progressive du problème* en sous-problèmes plus simples. La spécification peut commencer avec des modules très grossiers, et être progressivement raffinée avec la définition de modules internes aux entités.

- Le modèle à automates étendus communicants permet d'appliquer des algorithmes de validation classiques, comme la détection de blocages, sécurité, vivacité et l'assurance de terminaison; mais malheureusement, la taille du problème joue un rôle prépondérant sur la faisabilité du test, car les automates deviennent facilement de taille trop importante.

2.4.4 LOTOS

LOTOS vient de l'acronyme anglais *Language of Temporal Ordering Specification*, et il est standardisé par l'organisation internationale de standards dans la norme 8807[26]. Tout comme ESTELLE, LOTOS est un langage conçu pour modéliser des systèmes parallèles ou distribués, mais l'approche pour la modélisation et la manipulation formelle est différent.

En termes généraux, le langage LOTOS divise la modélisation du système en deux parties séparées mais qui ensemble créent un comportement cohérent:

- Modélisation des données: pour représenter les données et les opérations qui leur sont habituellement associées, LOTOS utilise une notation basée sur le langage de représentation de types de données abstraits *ActOne*. *ActOne* est un langage spécialement conçu pour représenter à un niveau fondamentale les données, tant ses possibles valeurs que les opérations qui leur sont applicables.
- Modélisation du processus d'exécution; LOTOS utilise une sémantique opérationnelle inspirée directement du langage d'algèbre de processus CCS, proposé par Robin Milner dans [43]. L'approche de CCS (et LOTOS) est d'exprimer le comportement du système en représentant la séquence des événements visibles par un observateur externe.

2.4.5 Promela/SPIN

SPIN[21] est le nom d'un outil de validation formelle crée par les laboratoire Bell et maintenu par un groupe de développement depuis sa création en 1980. SPIN permet de modéliser et de valider des systèmes distribués et parallèles en utilisant une approche appelé "on the fly", qui consiste principalement à générer un évaluateur automatisé en langage ANSI C. Cet évaluateur une fois compilé et exécuté, produit une représentation très compacte du système basée sur des automates de Boochi, sur laquelle il est possible d'effectuer un nombre de preuves classiques dans le domaine du model-checking mais aussi de vérifier la validité de formules logiques, exprimées par le développeur en logique temporelle linéaire. Ce qui rend le *model compacte* créé par SPIN différent, est qu'il est crée "sur le champ" (dont vient le nom "on the fly"), à la place de créer tout l'espace d'états qui peut être potentiellement gigantesque dans le modèles de taille conséquente. Ceci a comme résultat

TAB. 2.1 – *Types de données primitifs dans Promela*

type	équivalent en C	intervale de valeurs
bit	champ de bits	0..1
bool	champ de bits	0..1
byte	char,byte	0..255
short	short int	$-2^{15}-1..2^{15}-1$
int	int	$-2^{31}-1..2^{31}-1$

qu'on peut manipuler des modèles plus grands et plus complexes avec moins d'espace mémoire et de temps d'exécution.

Le langage accepté par SPIN s'appelle Promela, nom qui vient de l'acronyme anglais *Process Meta Language*. Un des aspects les plus intéressants de Promela/SPIN est aussi l'approche "ingénieur". Le langage donne des constructions qui sont familières à ceux qui connaissent le langage C, sans pour autant perdre de vue qu'il s'agit d'un langage de spécification formelle et pas un langage de programmation.

En Promela il existe la notion de variable, qui a une signification équivalente a celle de C: une variable est une valeur symbolique qui existe dans un certain contexte, qui contient une valeur et qui peut être manipulé avec les opérations appropriées. Pour les variables, le langage Promela définit un ensemble assez riche de types de données primitives.

Dans le tableau 2.1 sont listés les types primitifs de Promela. Le langage accepte aussi la définition de rangées d'une ou plusieurs dimensions, en utilisant une syntaxe analogue à celle du langage de programmation C. Par exemple l'expression `byte array[20];` déclare une variable appelée `array` qui représente une rangée de 20 octets.

Dans le langage mentionné il est possible de définir des valeurs symboliques. Pour cela on a la construction `mtype`. Avec `mtype` on peut déclarer les étiquettes qui sont habituelles dans les modèles de protocoles, par exemple, `mtype = { ack, nak, err, next, accept }`.

2.4.5.1 Contrôle de l'exécution

Le langage Promela est en grande partie un langage structuré, qui contient les construction habituelles d'un langage de ce type (séquence, if-then-else, while, do). Mais il y une difference importante: dans ce langage, il n'y a pas de différence entre une condition logique et une instruction. Le flou de l'exécution de n'importe quelle séquence d'expressions est soumise à l'*exécutabilité* de chacune des sous-expressions en séquence.

L'exécutabilité d'une expression logique est soumise jusqu'à ce qu'elle soit évalué comme *vraie*. Tant qu'elle ne le sera pas, l'expression est considérée comme *pas exécutable*, et l'expression et de toutes les autres qui la suivent en séquence restent donc bloquées. L'executabilité s'applique aussi aux expres-

Algorithm 2 Expressions equivalentes

```
while (x != y)
skip /* atteindre que x soit egal à y */

(x==y) gate ! OK /* si x n'est pas egal à y l'expression reste
bloquée */
```

sions d'entrée/sortie, où l'expression reste bloquée jusqu'à ce qu'une valeur soit disponible pour être lu ou bien, que le canal de communication soit prêt à recevoir un message.

Par exemple, dans l'algorithme 2 il y a un exemple de deux expressions Promela qui font exactement la même chose. Sauf que la seconde utilise la convention de blocage des expressions logiques évaluées comme fausses.

2.4.5.2 Processus

Le *processus* est la base d'une spécification dans le langage Promela. Un processus en Promela est analogue à l'idée de fonction ou procédure dans les langages de programmation structurés, c'est-à-dire, une unité de code réutilisable qui reçoit une étiquette symbolique, et qui peut être invoquée a posteriori dans une autre partie du programme.

Les processus en Promela, tout comme les fonctions/procedures de la programmation structurée, peuvent recevoir des valeurs d'entrée (des paramètres) et générer des valeurs de sortie. Les processus de Promela comme les processus en programmation structurée définissent les fonctionnalités et la sous-divisions abstraite du système à modéliser, mais avec quelques différences.

La première différence sémantique avec un langage structuré habituel, est le fait que les processus peuvent être spécifiés de telle façon, qu'ils représentent une exécution concurrente, c'est-à-dire, un processus quelconque peut exprimer le fait de lancer un ou plusieurs fils d'exécution. La seconde différence est sans doute le fait que les processus sont capables de *communiquer* entre eux en utilisant une construction du langage appelé *canaux* (channels), les canaux sont des entités abstraites qui peuvent transporter des valeurs symboliques d'un processus à un autre.

2.4.6 Réseaux de Petri

Les réseaux de Petri sont une notation formelle et visuelle utilisée pour représenter des systèmes discrets concurrents. Ils ont été proposés originalement par Carl Adam Petri dans [51] et présentés en détail dans sa thèse doctorale [52]. Depuis ils ont été adaptés, étendus et modifiés pour modéliser des systèmes à caractéristiques particulières qui ne pouvaient pas être représentés de façon satisfaisante avec le modèle originale.

Algorithm 3 Exemple de processus

```
byte state = 3;
proctype A()
{
(state == 1) -> state = 3
}
proctype B()
{
state = state - 1
}
init {
do
:: run A();
:: run B()
od
}
```

Les réseaux de Petri ont plusieurs avantages pour la représentation des systèmes concurrents. Pour commencer, ils peuvent représenter le parallélisme de façon beaucoup plus compacte que les automates à état fini. Ils sont en réalité une généralisation des automates, avec le même pouvoir expressif. Il y a cependant des extensions de la notation originale qui sont encore plus puissantes, et qui sont en fait des constructions formelles Turing complètes (c'est à dire, elles peuvent représenter n'importe quel algorithme représentable dans une machine de Turing).

Un autre avantage significatif des réseaux de Petri, est le fait qu'ils disposent d'une notation visuelle qui est assez intuitive pour les modèles de taille modérée en termes de nombre d'états et de transitions, ce qui les rend attirants pour ceux qui s'intéressent à la modélisation visuelle.

2.4.7 CPDL

Dans [22, 36], Huget et Koning proposent un méthodologie pour la spécification et validation des protocoles d'interaction pour les systèmes multiagent. Cette méthodologie a des caractéristiques qui ont servi d'inspiration pour le travail de recherche présenté ici. Il y a, en particulier, deux propositions::

- *CPDL*, qui est un langage formel pour la spécification de protocoles d'interaction.
- *UAMLe et GrCPDL*, deux langages pour la modélisation visuelle des protocoles représentés avec CPDL.

2.4.7.1 Le langage CPDL

CPDL est l'acronyme anglais de *Communication Protocol Description Language*, ou *langage de description des protocoles de communication*. Ce langage est une notation semi-formelle conçue pour décrire les protocoles qui sont toujours présents dès lors que les agents dans un système multiagent quelconque interagissent. Une contribution importante de CPDL est qu'il est fait pour rendre possible la réutilisabilité du protocole. Cette réutilisation est possible via une approche composants

Dans CPDL, un protocole est une structure constituée à partir de *micro-protocoles*. Un micro-protocole est une structure composée de plusieurs champs, à savoir:

- *Nom*: une étiquette descriptive pour faire référence au micro-protocole.
- *Sémantique*: la sémantique est une description de la fonctionnalité du micro-protocole en langage naturel.
- *Sémantique de paramètres*: cette partie sert à établir la sémantique des différentes entités invoquées pendant la définition du micro-protocole.
- *Définition*: cette partie est une spécification des messages échangés entre les entités. Le langage utilisé pour représenter les messages (performatifs), est un langage ad hoc qui ressemble aux langages de description formelle CCS et LOTOS. Ce langage admets plusieurs types d'opérations sur la séquence de messages, de telle sorte qu'il est possible de représenter des cas classiques d'interaction comme les séquences ordonnées de messages, les alternatives ou choix non déterministes, la synchronisation, les échéances et les exceptions.

Il faut noter que dans sa proposition, les protocoles et les micro-protocoles sont vus comme des *artefacts semi-formels* dans un processus de développement de génie logiciel (dans ce cas particulier : le génie des protocoles). Ceci veut dire qu'il existe un processus pour générer ces artefacts, et que ce processus peut être assez formel pour être fait à l'aide d'un ordinateur.

2.4.7.2 La notation visuelle GrCPDL

Le langage CPDL est accompagné d'un ensemble de logiciels conçus pour accepter comme entrée la notation proposée, qui font en son ensemble une plate-forme de spécification et développement expérimentale assez complète. Au cours de leur travail, les auteurs de la proposition CPDL se sont vite rendu compte qu'il serait fort utile d'avoir une voie assisté pour ordinateur, pour pouvoir créer un protocole d'une façon plus intuitive que la verbosité d'une simple description textuelle. Ils ont proposé pour cela un langage visuel pour la modélisation des protocoles, et une algorithmique pour amener ces diagrammes vers une spécification en CPDL.

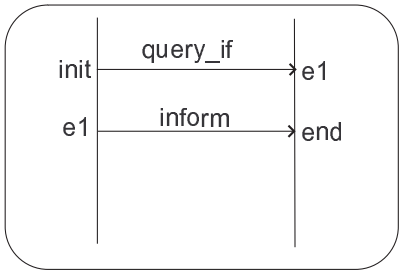


FIG. 2.14 – *Exemple de diagramme en GrCPDL*

Ce langage visuel fût appelé GrCPDL, dont la figure 2.14 est un exemple. Cette notation représente le déroulement du changement d'états dans un automate communicant, ce qui fait que les diagrammes GrCPDL ont une correspondance presque directe avec la description d'un micro-protocole de CPDL.

2.4.7.3 UAMLe

La notation visuelle UAMLe est une extension de la notation UAML (acronyme pour *Unified Agent Modeling Language*) proposée et utilisée pendant un certain temps par l'organisation FIPA [13] pour spécifier les protocoles de communication que l'organisation proposait comme standards. Cette notation est basée sur un sous-ensemble du langage visuel UML, en particulier, sur la notation graphique appelé *diagramme de séquence*. La principale différence entre la notation utilisée en UAML et son analogue d'UML est le fait d'enrichir la notation de séquence de messages, avec des artefacts-opérateurs qui affectent la sémantique de la séquence, et qui permettent de représenter le dialogue entre entités d'une manière plus naturelle.

La figure 2.15 montre l'apparence d'un diagramme UAML et UAMLe. En ce cas particulier, l'interaction représentée est une demande en provenance d'un agent vers un ensemble de n de ses partenaires. Les chiffres mis en dessous des deux côtés de la première flèche expriment le fait que c'est un seul agent qui envois un même message `query-if` vers n de ses partenaires. Les trois flèches entourées dans trois boîtes qui entrent dans le *Rôle 1* expriment le fait que les entités appartenant au *Rôle 2* vont choisir entre une de ces possibles réponses. Le crochet marqué avec une lette n qui précède les trois boîtes de choix, montre que l'entité appartenant au *Rôle 1* va attendre n réponses avant de continuer.

La notation UAMLe ne modifie pas substantiellement celle de UAML, sauf pour l'addition de certains artefacts qui augmentent l'expressivité, tels que les exceptions.

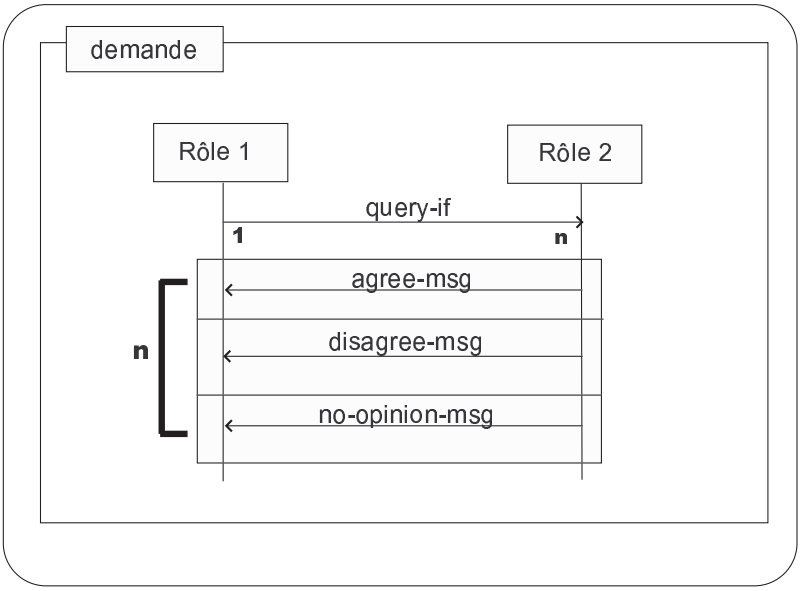


FIG. 2.15 – *Exemple de diagramme UAML*

Chapitre 3

Modélisation des protocoles d'interaction

Le but de la thèse était originalement d'étudier les approches possibles pour modéliser, d'une façon visuelle, des protocoles d'interaction entre des systèmes faits d'agents et choisir un pour l'étudier et proposer des améliorations possibles. Mais progressivement, l'approche a changé de lui même vers un étude approfondi de la notion de rôle en se centrant sur la communication, et la façon dont cette notion de rôle pourrait servir comme une aide à la modélisation avec une approche constructif.

On a commencé avec un approche simpliste pour la modélisation: en construisant un traducteur qui prenait une représentation textuelle d'une séquence de messages, et produisait une représentation formelle exprimé avec le langage Promela/SPIN.

Cependant, nôtre intérêt est dérive vers des nouvelles sources, quand on a décidé d'étendre la fonctionnalité de nos premiers travaux pratiques vers des fonctionnalités qui semblaient "simples" d'ajouter, mais qui ont montré des nouveaux problèmes intéressants -et compliqués- par eux mêmes.

3.1 Définitions

3.1.1 Sur les rôles

Les idées présentées dans cette section, ont été publiées dans le papier [38]. Dans nôtre intérêt, les rôles sont définis en termes des caractéristiques qui sont perceptibles par un observateur extérieur à l'entité, plutôt que ces disponibles par une observation directe de l'état mental propre à l'agent. C'est à dire, on se penche sur l'aspect communication de la notion de rôle.

On est conscients que l'idée de rôle est une notion beaucoup plus riche que la simple description d'un comportement visible depuis une perspective externe, mais dans nôtre intérêt qui est de modéliser la communication et la

valider, on peut se permettre d'abstraire les comportements complexes qui gênèrent les interactions que nous voyons depuis l'extérieur, et se centrer sur le déroulement de la communication elle-même.

Donc, dans ce travail de thèse un rôle est justement une étiquette qu'on donne à un patron répétitif d'interactions entre entités qui prennent une fonction pendant un processus d'interaction. Le scénario (ou groupe) est défini par l'ensemble de rôles qui interagissent pendant un processus d'interaction dès le début à la fin, et par leurs relations les uns avec les autres, qui sont à son tour définies par les messages qui sont échangés. La fonction d'un agent est donc définie par le réseau des dépendances et d'actions dans lesquelles il est inclus, pour accomplir la fonction désirée du processus d'interaction.

Une chose qui a été notée auparavant est le fait qu'il y a certaine séquences de communication qui semblent se répéter dans une même application, voir d'un cas d'implantation à un autre. Cette répétition de séquences de communication est possible de la voir comme un rôle répétitif entre applications.

En ce cas là, on retrouve la notion de sous-protocole tel qu'elle a été proposé dans [35, 36, 22], car quand un patron particulier des dépendances relationnelles et d'interactions se présente de façon régulière dans un système, il est possible de voir ce patron comme un bloque sémantiquement unifié et interchangeable.

Si on voit les rôles comme des simples séquences de communication (il faut se rappeler qu'on utilise une perspective externe ou de boites noires), ce sont justement les séquences d'actions ce qui est vraiment important. Alors le rôle devienne simplement une étiquette que nous attachons à une séquence stéréotypée de messages avec une unité sémantique, conçue pour nous permettre de le manipuler comme un artefact de spécification. S'il arrive que le même rôle apparaisse dans des scénarios multiples, et s'il incorporent le même modèle des dépendances et d'interactions, il est envisageable donc de créer un processus de génie logiciel qui puisse produire cette artefact-*Rôle* particulier, et qui puisse le manipuler avec une approche constructive.

Il existe une autre possibilité avec cette notion de rôle-artefact: si un agent contient dans sa définition un ensemble de multiples rôles qui sont eux mêmes vus de façon récurrente, il serait donc utile de définir un rôle quelconque de plus haut niveau *en composant* certains de ces rôles plus élémentaires.

3.1.2 Rôle-artefact

On peut exprimer un artefact rôle R comme un comportement stéréotypé qui exprime l'ordre temporel d'échanges de messages entre un agent et d'autres, dans un chemin analogue au concept de *rôle* dans un morceau de théâtre où chaque acteur a un dialogue bien défini et précis, ordonné dans le temps. Nous considérons le rôle comme étant un attribut assigné à une entité appelé *agent* et équivalent à un sous-protocole avec unité sémantique.

Si chaque agent a un rôle à exécuter, nous pourrions dire qu'un ensemble des rôles qui interagissant fait un *scénario*. Dont un scénario est un ensemble de deux ou plusieurs rôles qui sont reliés par leurs interactions et qui partagent une fonctionnalité commune. Les rôles sont toujours associés aux scénarios pour fournir le contexte.

Dans nôtre cas, les rôles en étant des structures de Kripke, ils peuvent parfaitement être représentés par des automates à état finis ou par des réseaux de Petri. Cependant, la notation d'automate nous a apparu beaucoup plus simple et directe pour exprimer les aspects qu'on voulait pouvoir représenter, principalement dû au fait qu'on disposait des outils plus adaptés pour le model checking basé sur automates. En particulier, on s'est inspiré d'après la définition d'automate étendu proposé par le TDF ESTELLE.

Soit $r = \{s,Q,C,\Sigma_1,T,F,L\}$ un rôle, où $Q = \{q_1,q_2,...,q_n\}$ est l'ensemble fini d'états du rôle qui contient les n états possibles du r, $s \in Q$ est un état dit *initiale*, $C = \{c_1,c_2,...,c_k\}$ est l'ensemble des canaux de communication qui relie r à ses partenaires de communication et dont $k \geq 2$ et toujours pair (les canaux sont présumés unidirectionnels, donc il faut au moins deux pour assurer la communication entre deux entités), $\Sigma_1 = \{s_1s_2,...,s_j\}$est l'ensemble de tous les symboles qui peuvent être reçus ou envoyés par r pendant son existence, $T : Q \times C \times \Sigma \longmapsto Q$ est la fonction de transition qui change l'état actuel en acceptant ou en envoyant un symbole;$F \subseteq Q$ est l'ensemble des états finals acceptables pour l'automate et L et un ensemble de propositions logiques dont l'évaluation permet ou arrête l'exécution de la fonction de transition.

Pour nous, chaque rôle r est un automate étendu qui exprime le comportement observable du système, et ce rôle est une description des toutes les séquences d'interaction observables qui peuvent arriver à l'agent. Laissez-nous exemplifier notre définition en utilisant la définition de ContractNet proposée par FIPA. Le protocole tel comme il est, représenté au moyen de la notation d'Agent UML proposé par Odell et al est illustré dans la figure 3.1

Le scénario contient deux rôles: l'*Initiateur* et le *Participant.* L'ensemble des interactions est:

I = {cfp, not understood, refuse, propose, reject proposal, accept proposal, failure, inform done, inform ref}

L'ensemble de canaux de communications C n'est pas montré explicitement, mais on peut supposer qu'une interaction a lieu entre deux rôles, donc nous pourrions dire qu'il y a deux canaux de communication c_1 et c_2 qui lient l'Initiateur et le Participant, donc $C = \{c_1,c_2\}$.

Si nous continuons avec le même protocole ContractNet, nous pourrions exprimer le comportement de rôles d'Initiateur et celui des Participants comme deux machines à état fini étendues distinctes. La figure 3.2 représente un modèle possible pour le rôle d'Initiateur.

Comme cet automate est un automate étendu, cette machine accepte un

FIG. 3.1 – *Représentation AUML du protocole FIPA ContractNet*

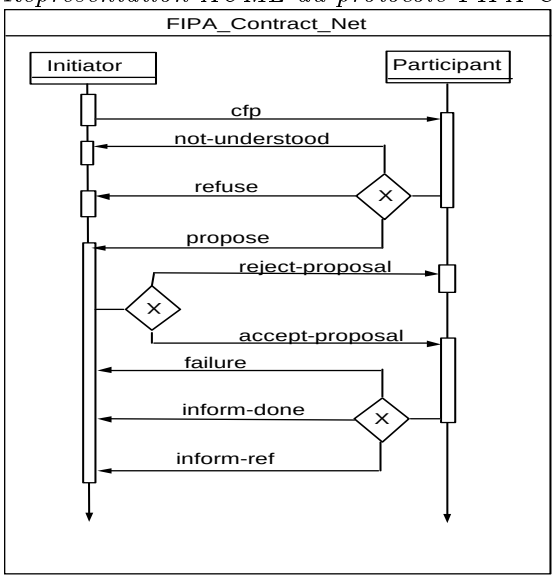
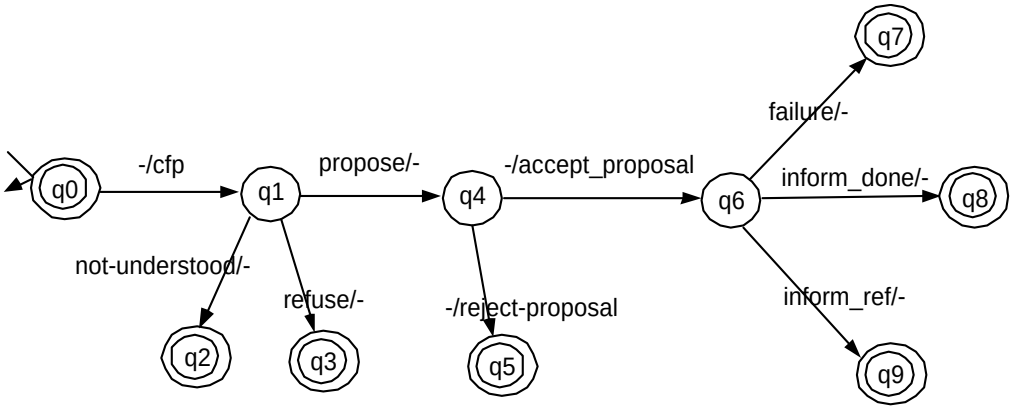


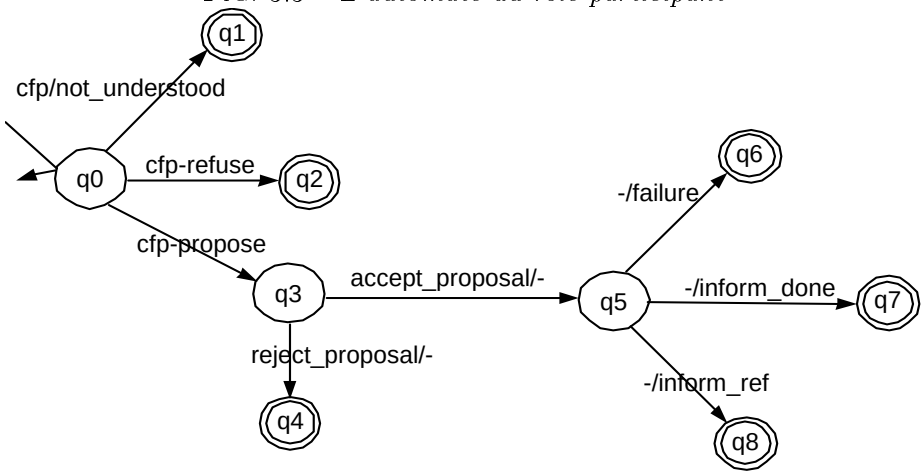
FIG. 3.2 – *L'automate du rôle initiateur*



symbole d'entrée et un symbole de sortie à chaque transition. La notation que nous utilisons ici pour représenter les deux types de symboles est `<symbole d'entrée>/<symbole de sortie>`. S'il n'y a pas aucun message de sortie ou d'entrée dans chaque transition, nous utilisons la marque de moins '-' pour représenter qu'il n'y a pas d'entrée or de sortie dans la transition.

La figure 3.3 représente le modèle d'automate correspondant au rôle de *Participant*. Les entrées/sorties sont séparées par un trait et ils sont localisés côté à côté avec leurs transitions correspondantes. La traduction d'une représentation graphique à une représentation formelle en utilisant notre dé-

FIG. 3.3 – *L'automate du rôle participant*



finition de Rôle comme un FSM étendu est très facile a codifier, comme nous verrons.

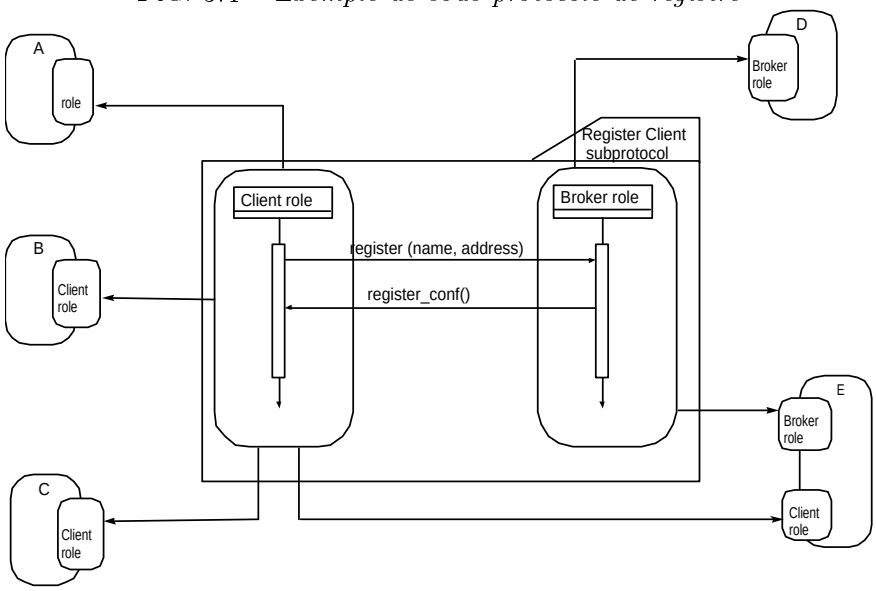
3.1.2.1 Sémantique

La raison pour définir les rôles comme machine à état étendu, et les scénarios comme des n-uplets qui contiennent des rôles, les interactions et les canaux de communications, c'est pour proposer une façon de les traduire dans le contexte d'un langage de modélisation formelle, pour en profiter des capacités de validation de l'outil associé à ce langage.

Dans [36, 35] un concept de sous-protocole est proposé. Les sous-protocoles sont des séquences partielles et réutilisables d'actes de communication qui définissent le classement temporel de messages, dans un processus de communication spécifique entre les rôles. Les rôles, comme dit auparavant, représentent le comportement que tout agent a pendant le déroulement d'un processus d'interaction. Il peut nous sembler naturel que pendant le déroulement d'un processus d'interaction, un agent ne puisse prendre qu'un seul rôle. Cependant, et il a été noté par certains spécialistes [48], si les sociétés artificielles veulent ressembler aux naturelles, il pourrait être nécessaire de considérer qu'une entité puisse prendre plusieurs rôles au même temps, ou bien prendre *et abandonner* un certain rôle avant la conclusion d'un processus d'interaction. Bien que nôtre définition de rôle soit quelque peu limité, ces réflexions sur la nature des rôles dans les systèmes multiagent ont une certain consonance avec nôtre travail.

Il est bien possible de concevoir qu'un agent puisse prendre un rôle dans un système mais aussi prendre deux ou plus des rôles dans un processus d'interaction quelconque. Si on suit l'analogie jusque ses ultimes conséquences,

FIG. 3.4 – *Exemple de sous-protocole de registre*



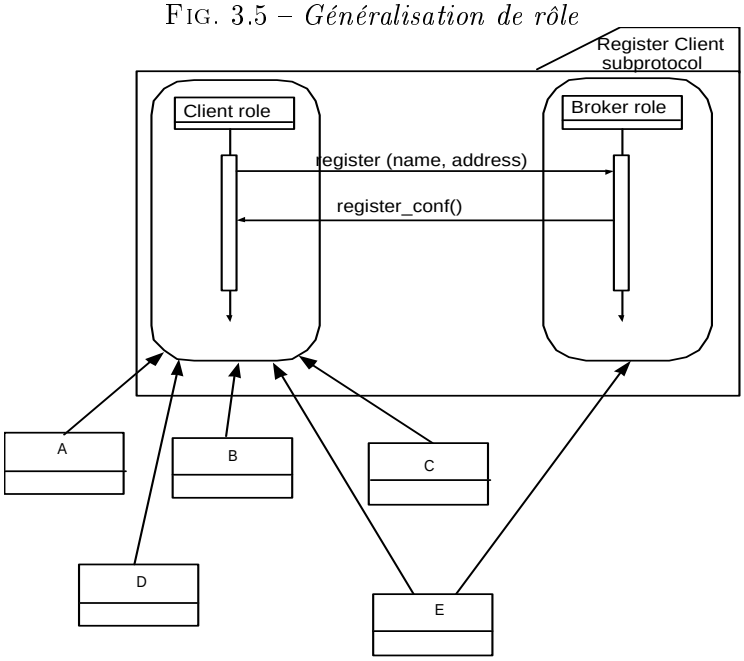
les rôles ne sont pas restreints à une seule entité-agent, du même que les sociétés humaines n'interdisent pas qu'une personne ait plus d'une fonction dans une organisation. Une entité A pourrait prendre deux rôles C_1 et C_2 , pendant qu'une entité $D \neq A$ pourrait prendre seulement le rôle C_1.

La figure 3.4 montre un paquetage AUML version 1 simple qui exprime une sous-protocole de registre vis-à-vis d'un *broker*. Ce sous-protocole a deux rôles: *Courtier* et *Client*. Les deux sont représentés avec des boîtes étiquetées. Les actes de communications (performatifs ou messages) sont représentés par des flèches étiquetées, où l'étiquette exprime le message échangé et ses paramètres.

La notation pour l'ordre temporelle est la conventionnelle aux diagrammes de séquences de UML et AUML. Les rôles de *Client* et de *Courtier* dans ce sous-protocole sont spécifiques. Mais en même temps, ils sont applicables à beaucoup d'entités différentes (nommé A,B,C,D et E). Elles sont représentés dans la figure comme des boîtes rondes qui "contiennent" à l'intérieur les rôles. La notation des boîtes à tiret autour des rôles, aussi bien que les flèches qui lient ces rôles aux agents qui les mettent en application, n'est pas une notation de AUML. Cette notation montre seulement la présence d'un lien abstrait entre les rôles et les agents qui les mettent en application.

La notation AUML d'Odell dans sa première version donne le support explicite pour les protocoles d'interaction et la spécification du rôle, mais pas pour spécifier la liaison du rôle au agent et la réutilisation du protocole. Il pourrait être utile à avoir une telle notation pour étendre le caractère expressif d'AUML. Ce lien entre rôles vers les agents est faite toujours, mais

FIG. 3.5 – *Généralisation de rôle*



implicitement.

La figure 3.5 montre le même sous-protocole d'enregistrement du client dans le système globale, mais en utilisant maintenant une exemple de notation suggérée par nous, basée sur l'artefact graphique *généralisation* de UML. D'autres artefacts UML pourraient être utilisés, mais nous pensons qu'ils n'expriment pas suffisamment la modification du comportement intrinsèque qu'un agent subit quand il met en application un rôle. C'est une bonne idée pour changer la notation pour éviter la confusion avec l'artefact de généralisation de classe de UML, mais le contexte devrait se débarrasser de cette confusion.

Quelques auteurs notent bien que les protocoles d'interaction qui sont bien conçus et les rôles qu'ils créent, pourraient être parfaitement utilisés comme un mécanisme de réutilisation [23]. AUML est suffisamment expressif pour représenter des protocoles d'interaction et des rôles, mais elle manque de notation spécifique pour lier ces rôles aux agents qui les utilisent.

Cependant, cacher le comportement interne du module autant que possible, et baser la représentation des fonctionnalités du système sur les interfaces faiblement associées ne suffit pas pour modeler complètement ou pour implémenter un système de communication modulaire basé sur la notion d'orientation agent.

Il y a toujours un processus interne impliqué pour qu'un agent sache comment réagir quand il reçoit un stimulus spécifique et quand il prend

l'initiative et commence une interaction sans aucune entrée.

Comme les rôles sont des spécifications partielles de protocole (ils spécifient seulement la perspective qui a l'agent à l'intérieur du processus de communication), nous avons besoin d'un mécanisme interne pour contrôler l'état d'un rôle, toutes les fois que nous utilisons ce rôle à l'intérieur d'un agent. Et pour ce faire, on a aussi adopté la notion de *garde logique* qui habilite l'exécution de toute interaction, ce qui justifie l'inclusion d'un ensemble d'expressions logiques L dans la définition de l'automate.

3.1.2.2 Rôles et scénarios dans la modélisation

Nous considérons des rôles comme étant des comportements stéréotypés qui pourraient être pris par une entité pour accomplir une tâche significative, où l'ordre des actes de communication qu'une entité qui envoie et reçoit par une interaction spécifique. L'aspect le plus important est de connaître quel module envoie quelque chose à qui, et exactement quand il le fait.

Ayant défini le concept de rôle comme une machine à état étendu, le scénario comme un ensemble de rôles et l'opérateur de l'héritage du rôle comme un produit entre l'automate du rôle de l'ancêtre et celle du descendant, il est possible de définir un langage de spécification simple qui permette de démontrer l'utilité réel de la notion de rôle comme artefact de modélisation.

3.1.2.3 Généralisation et composition

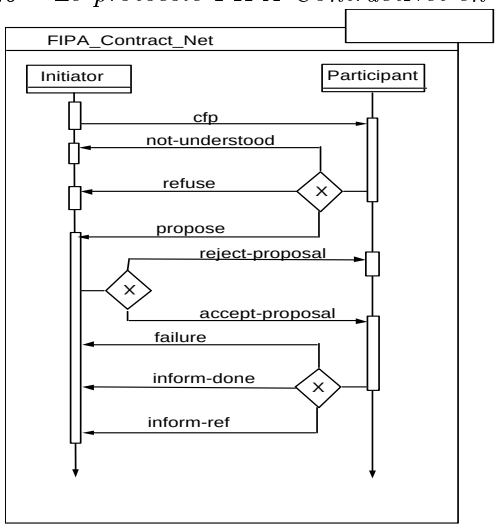
AUML fournit la notation spécifique pour modeler des protocoles d'interaction. Ces protocoles sont des meta-définitions de séquences d'interaction entre les rôles qui ont des paramètres. Dans ce contexte, les paramètres dans un protocole d'interaction sont des valeurs indéterminées, qui ont besoin d'être définis au moment de l'exécution pour permettre le fonctionnement du protocole. La motivation pour cette facilité est de créer des protocoles complètement réutilisables qui expriment un processus d'interaction stéréotypé qui utilise des entités de types inconnus.

Pour exemplifier cette idée de protocole d'interaction générique, nous pourrions prendre le *ContractNet* mentionné auparavant.

La figure 3.6 montre le même protocole FIPA Contrat Net qui était montré auparavant dans figure 3.1, seulement avec une modification légère. Dans le coin supérieur à droite de la boîte qui entoure la figure il y a une boîte à tiret qui contient l'étiquette *cfp*. La présence de cette boîte implique que le protocole Contrat Net est générique. Il a besoin d'un paramètre pour définir complètement son comportement. Dans ce cas le paramètre est le contexte dépendant du message *'cfp'*. Lequel c'est l'offre spécifique à chaque processus d'interaction de Contrat Net.

L'idée de généralisation est assez attirante pour ceux qui cherchent de mécanismes de réutilisation. Nous avons décidé de l'adopter dans notre pro-

FIG. 3.6 – *Le protocole FIPA ContractNet en AUML*



position. Nous avons aussi décidé d'ajouter un autre niveau de paramètres: les paramètres au niveau de *rôle*.

La signification de ceci n'est pas difficile à comprendre, nous pouvons le lier à la notion de composant. Les composants ont beaucoup de définitions. La plupart de ces définitions acceptent de considérer ces composants comme modules échangeables qui offrent une interface constante au monde externe. L'ensemble minimum de propriétés que toute entité dans un contexte des composants doit satisfaire est l'interface du composant.

Si on veut utiliser des rôles et des protocoles d'interaction comme des artefacts pour la réutilisation, on a besoin de mettre des restrictions dans la composition. Ces restrictions assurent que chaque partie du système pourrait réagir réciproquement l'un avec l'autre sans présenter de défauts de synchronisation ou de séquence provoqués des séquences incompatibles pendant l'échange de messages. Notre définition de rôle pourrait être réduite à un arbre de messages ordonnés dans le temps au style CCS, ou bien, représentée comme un modèle d'automate d'état fini vérifiable par *model-checking*.

La plupart des problèmes importants qui restreignent la composition de rôles, sont liés au accouplement d'échanges des messages entre un rôle quelconque et ses compagnons, et le besoin de certifier la présence de propriétés désirables, dans le sens qui lui donne le model-checking.

3.1.3 Nos rôles

Définition: Un *rôle* est le répertoire normatif du comportement d'un agent. Bien qu'ils aient une signification beaucoup plus riche dans la pratique, pour nous les rôles sont exactement équivalents à une structure abs-

traite arborescente analogue à une structure de Kripke, qui énonce chaque une des possibilités d'exécution et d'interaction d'un agent. Tandis que les rôles peuvent être définis indépendamment des groupes auxquels ils appartiennent, ils doivent être joués chez ces groupes pour avoir un sens. Les rôles peuvent se composer d'autres rôles. En outre, les rôles peuvent être *paramétrisés* pour représenter un comportement générique.

Une *instance* est une entité qui exécute un rôle formellement identifiée, et qui peut être invoquée en cas de besoin pendant le déroulement d'une interaction. Un rôle peut avoir zéro, une ou plusieurs instances de lui à un moment spécifique de l'exécution. Les agents peuvent être associés à un rôle et seulement un, à un point quelconque du temps. Cependant, les agents peuvent être associés à différents rôles avec le temps.

Une assignation de rôle est l'association dynamique (en temps d'exécution) d'un rôle avec un agent. L'assignation de rôle à un agent peut être le résultat d'un processus exogène ou endogène.

Un groupe est un ensemble de deux agents ou plus qui sont reliés par l'intermédiaire de leurs tâches ou assignation de rôle, où ces rapports doivent former un graphe connexe chez le groupe. Des agents et des rôles sont associés aux groupes pour fournir le contexte.

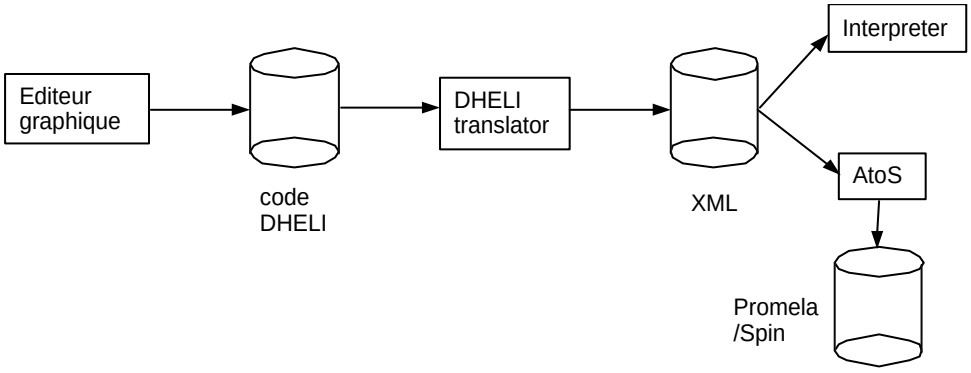
- Un système multiagent se compose d'un ensemble de comportements interdépendants de rôle, fournissant un modèle collectif dans lequel les agents jouent un rôle, ou plusieurs rôles.
- Une organisation est un groupe dont les rôles et les interactions sont typiquement prévus pour être relativement stables. Ils changent peu ou pas du tout à travers le temps. Cependant, la capacité de *changer de rôle* existe.
- L'environnement fournit les conditions dans lesquelles une entité (agent ou objet) existe. Celles-ci incluent les groupes auxquels un agent peut appartenir.

3.2 Modelisation orientée interaction

Cette thèse suit un approche multicouches. Certaines parties ou couches sont de nature théorique (par exemple, méthodologie et approches pour la spécification/programmation), et d'autres ont plus pragmatiques (des outils logiciel). Nous pourrions dire que nôtre projet a trois niveaux opérationnels principaux:

1. Une structure générique de la modélisation orientée-interaction. Cette partie du projet est une proposition d'une approche de la modélisation des systèmes multiagents centrées sur l'interaction. Il y a un grand nombre de travaux intéressants qui proposent des langages orientés-agent et qui sont basés sur des aspects spécifiques d'agence. La propo-

FIG. 3.7 – *Structure générale du système*

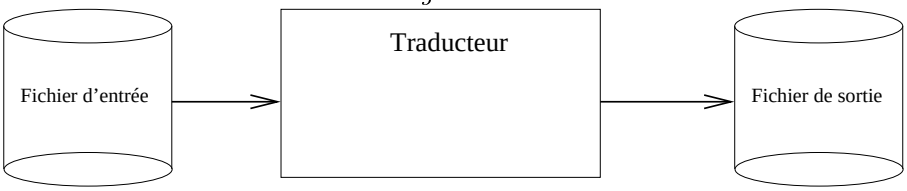


sition bien connue d'*Agent0* par Shoham[56] est un exemple d'un langage orienté-agent qui se concentre sur la modélisation de la connaissance. Il y a un deuxième genre de plates-formes de développement d'agents qui fournissent des installations et d'infrastructure pour la communication parmi les systèmes hétérogènes sous la forme de bibliothèques du langage. Cependant, il y a peu de plates-formes qui rentrent compte du problème de communication de systèmes multiagents selon une perspective de programmation interne (c.-à-d., la proposition des langages ad hoc pour ce problème). Notre proposition implique la création d'un ensemble de critères qui définissent des propriétés qui pourraient être utiles de représenter à l'intérieur d'un langage de programmation orienté-agent.

2. Une méthodologie légère qui suit la structure de la programmation sus-mentionnée. Cette proposition méthodologique sera ajustée pour résoudre le problème éventuel des nouvelles exigences d'un langage orienté-interaction, et en même temps il cherchera à fournir une intégration facile avec la proposition d'Agent-UML, pour pouvoir ainsi tester AUML dans le contexte d'une méthodologie assistée par ordinateur.
3. Un outil CASE qui permette d'implémenter la méthodologie de modélisation proposée.

Les logiciels développés sont chacun d'entre eux des projets semi-indépendants (ils pourraient être exécutés séparément des autres parties). Mais ses entrées, fonctionnalités, et ses sorties sont en rapport avec les entrées, fonctionnalités et sorties des autres sous-systèmes. Il y a une partie qui est interactive et visuelle, dans le sens qu'elle attends des instructions à travers d'une interface graphique, mais le reste du système suit une approche de *pipeline,* tel qui est présenté dans la figure 3.7.

FIG. 3.8 – *Schéma générale de l'outil initial*



3.3 Travaux préliminaires

Pour commencer avec les travaux de recherche de la thèse, on a décidé d'implémenter un système qui démontre la viabilité de la traduction automatique à partir d'une représentation visuelle, vers une représentation formelle quelconque qu'on pouvait choisir à ce moment là. Après une certaine période de discussion, on a décidé que la plus serait la notation donné par Promela, avec son outil de validation associé SPIN.

Les raisons de notre choix sont simples: Promela/SPIN est un outil pour valider des systèmes concurrents/communiquants qui a beaucoup d'exemples d'application sur des travaux pratiques, ainsi qu'une littérature abondante, et bien sûr le fait que l'outil de validation formelle soit distribué gratuitement. D'autres outils sont dans le même cas, mais aucun n'avait dans notre jugement, le niveau de maturité de SPIN tant en fonctionnalités qu'en performance tout en étant gratuit.

Le choix de la notation formelle était important, car on avait clairement besoin d'un model-checker capable d'explorer l'espace d'états des modèles potentiellement gigantesques, et de valider automatiquement le plus grand nombre de propriétés d'intérêt pour un spécialiste de systèmes distribués. En ce sens, on aurait éventuellement pu utiliser LOTOS, SDL, ou ESTELLE [29, 28, 27, 31], et obtenir des résultats finaux similaires, mais cela n'était pas sûr, car à la fin du travail le choix du langage formelle aurait peut-être affecté les capacités d'expressivité du modèle, comme on le verra plus tard.

L'idée initiale était simple: imaginer un processus de développement linéaire avec une série de phases courtes qui définiraient une méthodologie légère, et qui se centrerait sur une fonctionnalité: entrer une représentation visuelle d'un côté et sortir de l'autre une représentation formelle équivalente.

Dans la figure 3.8 on peut voir la structure générale de l'outil qu'on souhaitait avoir au départ pour commencer les travaux de recherche.

Maintenant il est possible de se poser les questions qui viennent de soi: quels sont ces deux fichiers d'entrée et sortie, et comment on les traduit l'un vers l'autre? C'est justement à ça qu'on donnera réponse par la suite.

3.3.1 Le langage d'entrée première version

Ceci représente le travail publié dans l'article [39] et une partie du travail présenté dans le chapitre [53]. En quelques mots: on voulait représenter des protocoles d'interaction dans un langage visuel simple et facilement traduisible vers une représentation formelle. A ce moment là, les diagrammes de séquence de UML attiraient fortement l'intérêt de certains chercheurs, qui les trouvaient intéressants pour leur similitude avec les chronogrammes, couramment utilisés pour développer des protocoles de communication. Ils cherchaient une définition qui visait à étendre l'expressivité du langage UML, car les promoteurs de cette approche d'amélioration-par-extension (et nous aussi) trouvaient la définition actuellement en vigueur assez limité.

Les diagrammes de séquence dans la version 1 de UML définissent les diagrammes de séquence d'une façon qui est convenable pour un approche orienté objet, mais pas nécessairement pour un approche orienté agent. Les entités-objets en étant statiques -en comparaison avec des agents- n'avaient vraiment pas besoin d'une trop riche capacité d'expression, en ce qui concerne le contrôle de l'exécution interne. D'ailleurs, cette notion de contrôle de l'exécution est plutôt inutile à ce niveau-là de la spécification.

Dans un diagramme de séquence de UML standard[49], nous ne sommes pas intéressés par ce qui se passe à l'intérieur de l'objet, mais par ce qu'il échange avec ses partenaires et dans quel ordre. Ce qui pour un modèle à base d'objets, est parfaitement logique: les objets étant des entités passives activées par le passage des messages entre eux, le déroulement de l'exécution interne pouvait être ignoré sans diminuer la validité du modèle si on se contentait de représenter l'invocation de méthodes et l'ordre dans lequel ceux-ci apparaissaient.

Cependant, les auteurs de UML reconnaissent qu'un plus grand niveau de détail pouvait être nécessaire dans quelques cas d'application particuliers, en admettant qu'on pouvait "faire des branches" sur une ligne de vie d'un objet pour représenter un choix apparu à un certain moment pendant l'activation de l'objet, ou bien le fait qu'un objet possédait plus d'un fil d'exécution.

Malheureusement la notation pour représenter cette éventualité en UML 1 était floue et ambiguë. La solution palliative proposée par les auteurs de UML se centrait sur le fait qu'on dispose aussi d'un artefact d'annotation ou commentaire, ce qui devrait permettre de diviser les deux cas possibles: ou bien l'objet a fait un choix ou bien il "lance" un fil d'exécution à un certain moment. Il faut noter que dans sa critique des extensions de UML[40], Jürgen Lind est sceptique en ce qui concerne l'enrichissement de la notation de UML avec encore plus d'artefacts, et pointait vers la possibilité d'utiliser justement cette notation de commentaire, pour représenter les cas où on a des choix ou de l'exécution parallèle. Mais nous avons décide de suivre l'approche de l'enrichissement, car il avait les plus grandes possibilités de générer des résultats intéressants pour nous.

La raison pour laquelle les diagrammes de séquence avaient -et ont- tellement d'intérêt pour nous c'est qu'ils sont une notation assez naturelle pour représenter l'échange de messages entre deux ou plusieurs entités abstraites. Ils sont d'ailleurs utilisés par FIPA pour représenter ses protocoles de base tel que ContractNet.

3.3.1.1 AUML

La notation standardisée UML dans sa première version[49] peut se révéler insuffisante pour modéliser des agents et des systèmes basés sur des agents pour trois raisons: premièrement, comparé aux objets, les agents sont *actifs* car ils peuvent prendre *l'initiative* dans le contexte d'une interaction. Deuxièmement, les agents n'agissent pas seulement d'une façon isolée mais en coopération et coordination avec d'autres agents , et troisièmement ils communiquent plus fréquemment de façon *asynchrone*.

Pour essayer de combler les possibles déficiences d'UML, il a été proposé une extension de UML aux concepts agents appelé AUML[46, 47]. AUML est justifié vraisemblablement par le fait qu'UML *pourrait* inclure des spécifications plus riches des agents, en proposant des modifications du format de quelques artefacts standard, mais en gardant un certain 'esprit' ou approche visuel facilement compréhensible par n'importe quel développeur habitué à UML.

La notation AUML est fortement centrée sur l'interaction entre les agents. Les diagrammes de séquence de AUML sont très similaires à ceux de UML, mais avec l'addition des nouveaux artefacts graphiques avec sa sémantique associée. Ces opérateurs son principalement de deux types: les opérateurs d'exécution et les opérateurs d'interaction. Tous se réduisent à trois: XOR, OR, et AND, tant pour contrôler l'exécution comme pour l'envoi de messages. Ils ont été ajoutés pour essayer de modéliser et d'exprimer d'une façon plus *complète* les interactions entre les agents.

La ligne de vie d'un agent dans les diagrammes d'interaction définit la période de temps pendant laquelle un agent est actif, cette ligne de vie est représentée graphiquement en UML par les lignes verticales pleines. Les représentation graphiques pour les opérateurs d'exécution ont une sémantique bien définie, par exemple AND (qui représente le parallélisme), XOR (un seul message parmi une liste à envoyer) et OR (choix non déterministe entre une ou plusieurs alternatives).

Le XOR peut être abrégé (ou simplifié) par une séparation d'un chemin d'exécution comme il est montré dans la figure 3.10. Le chemin d'interaction est décomposé en différents chemins d'interaction, dont un seul est choisi. Ainsi la ligne de vie d'une entité est décomposée en deux ou plusieurs branches et le chemin d'exécution définit la réaction aux types différents des messages reçus.

Le chemins d'interaction montrent le moment où une entité exécute des

FIG. 3.9 – *Diagramme AUML (XOR)*

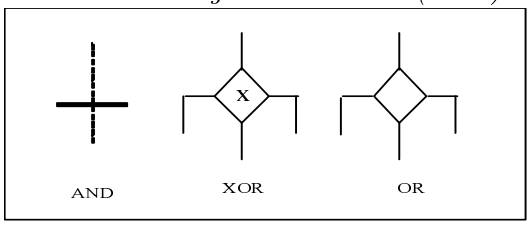
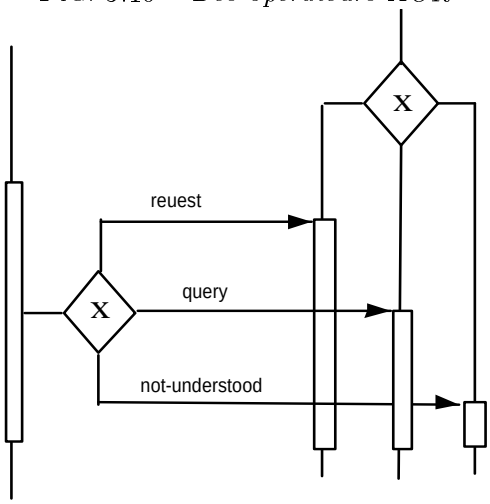


FIG. 3.10 – *Des opérateurs XOR*

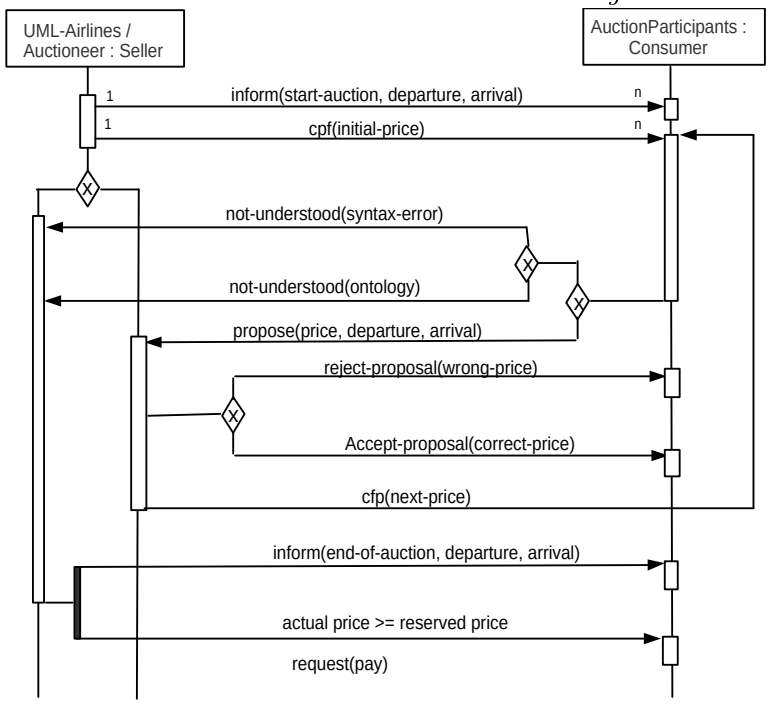


tâches comme une réaction à un message d'entrée. Il représente seulement la séquence temporelle d'une action, mais pas la relation de contrôle entre la source et le récepteur du message. Un chemin d'exécution est toujours associé avec la ligne de vie d'un agent-rôle. Le support de chemins d'interaction concurrents est une extension déjà recommandée par UML, mais pas définie clairement en tant que standard.

Nous allons présenter un exemple simple qui représente l'interaction de prendre un billet bon marché "dégriffé" dans une agence de voyages. Dans cet exemple, nous avons l'enchère de tels billets qui peut être effectué en utilisant le *FIPA English-Auction Protocol* comme la figure 3.11 nous montre. Le commissaire priseur (le *Auctioneer*) propose initialement un prix qui est moins chère que le prix prévu par le marché. Ensuite le prix augmente graduellement. Le commissaire informe tous les participants que l'enchère (auction) est commencée (il est représenté par les messages `inform(start-auction, departure, arrival)`, et il annonce les détails du voyage.

A chaque pas, lorsqu'un nouveau prix a été annoncé (représenté par

FIG. 3.11 – *Protocole FIPA d'enchère anglaise*



`cfp(initial-price) et cfp(new-price)`). Le commissaire attend jusqu'au nouveau `deadline` pour voir si un participant envoie un signal pour exprimer qu'il veut payer le prix proposé pour le billet. Si un participant n'a pas compris l'ontologie ou la syntaxe du `cfp`, il répond avec un acte de communication `not-understood` . Le symbole du diamant indique une décision qui résulte par zéro ou plusieurs communications. Lorsqu'un participant indique qu'il veut accepter le prix, le commissaire lance un nouvel appel pour augmenter le prix (`cfp(new-price)`).

L'enchère continue jusqu'à ce qu'il n'y ait plus de participants qui soient prêts à payer le prix proposé. Dans ce cas, l'enchère se termine. Si le dernier prix accepté par l'acheteur est plus grand ou égale au prix de réservation du commissaire, alors la billet est vendu à ce participant avec le prix consenti (dans les autres cas, l'enchère a failli). Les participants sont informés de la fin de l'enchère et l'acheteur doit payer le prix du billet.

3.3.1.2 Notre langage

Pour essayer de combler les défauts de notation d'AUML que nous percevions, nous avons fait des propositions supplémentaires dans le même sens, au travers des améliorations et modifications de la notation standard d'UML. Cela a résulté en la création d'un langage simple et d'un outil appelé ATOS.

Algorithm 4 exemple de code

```
(init,null,null,null)->
(out, initiator,participant,[null,true,cfp])->
(
(in, participant,initiator,[refuse,true,null])
|
(in, participant,initiator,[not_undestood,true,null])
|
(in, participant,initiator,[propose,true,null])->
(
(out, initiator,participant,[null,true,reject_proposal])
|
(out, initiator,participant,[null,true,accept_proposal])->
((in, participant,initiator,[failure,true,null]) |
(in, participant,initiator,[inform_done,true,null]) |
(in, participant,initiator,[inform_ref,true,null]))
)
)
```

AtoS viens de l'acronyme anglais *AUML to SPIN translator*.

Le langage d'entrée est fait pour représenter des structures qui sont équivalentes (dans le sens de l'expressivité) aux diagrammes de séquence d'AUML version 1. Comme la plupart des techniques et notations de la description formelle basées sur des automates étendus, l'outil est basé sur la notion de module ou d'entité autonome communiquante. Le langage propose un ensemble de déclarations pour spécifier la séquence temporelle de messages échangés entre ces entités.

Dans l'outil qui accepte le nouveau langage, les entités sont nommées *rôles*, dont le comportement interne est représenté comme défini dans la section 3.1.1. Chaque Rôle existe dans un contexte spécifique, appelé par nous un *Scénario,* qui contient toujours au moins deux rôles. La raison de cette restriction vient du fait que nous avons besoin d'au moins deux entités qui communiquent l'une avec l'autre pour avoir un système *multi*-agent.

Le programme 4 montre un exemple de déclaration d'un Rôle appelé *initiator* qui exprime le comportement visible d'un *Initiator* du protocole FIPA ContractNet. Une spécification peut contenir une quantité ouverte de rôles, mais chacun d'eux doit impérativement avoir un nom unique, et avec la restriction que chaque rôle doit être défini dans un fichier de texte séparé, nommé `<nom_du-rôle>.auml`. Cette restriction est importante car dans le code on fait une référence explicite aux rôles qui envoient ou reçoivent les interactions, et ces noms sont supposés faire référence aux rôles définis dans les fichiers du même nom. Par exemple, le code contenu dans l'algorithme 4 doit être dans un fichier appelé `initiator.auml` car dans les interactions on

invoque explicitement `initiator` comme la source des interaction de sortie ou la destination des interactions d'entrée tout au long du code.

3.3.1.3 Les fichiers d'entrée

Les fichiers d'entrée sont du texte en ASCII standard, chaque fichier a un nom qui doit concorder avec les noms de rôle invoqués dans ce même code. En particulier, toutes les interactions de type IN (entrée) doivent avoir comme destination le rôle actuel, et les noms doivent concorder en conséquence, même pour les interaction du type OUT, dont la source doit être le rôle courant. L'outil est sensible aux majuscules dans le noms de rôles et de variables. Une interaction de type IN avec un destinataire différent du rôle défini dans le ficher courant est signalé comme une erreur sémantique (car un rôle ne peut pas recevoir des messages qui ne lui soient pas directement adressés). La seule exception à cette règle est l'interaction du type *triggering*

Quand on veut commencer le processus de traduction, il faut donner dans la ligne de commande tous les noms des fichiers qui font partie de la spécification/scénario, ou le logiciel retourne une erreur. Le logiciel est assez flexible pour savoir qu'il manque un ou plusieurs fichiers (s'il y a, par exemple, un rôle mentionné comme source ou destination d'interaction dans un des fichiers, mais le logiciel n'a pas reçu explicitement la localisation de son fichier source).

3.3.1.4 Syntaxe du langage d'entrée

Une fois qu'on a accepté que la notation Agent UML en apparence satisfaisait nos besoins, on a continué par définir un langage qui puisse le représenter de la façon la plus complète possible.

Dans [60], Wei et al proposent un langage simple inspiré sur le standard Z.120 [32, 33], et qui pouvait réprésenter de façon complète les opérateurs d'exécution et de communication présents dans AUML tel qu'il était défini dans [46], mais avec quelques extensions spécifiques à leur travail de recherche.

En s'inspirant de la notation proposé dans[60] , on a crée un langage simple qui permet de représenter tous les opérateurs d'exécution et les actes d'interaction proposés dans AUML version 1, et qui en ajoute quelques autres.

AUML propose quatre artefacts pour représenter le contrôle d'exécution dans l'entité, dont trois sont acceptes dans notre outil:

- Composition séquentielle: soient deux expressions E_1et E_2, l'expression $E_1 \rightarrow E_2$, indique E_1puis E_2.
- Choix du type XOR: $E_1!E_2$, exprime le choix entre E_1 ou bien E_2.
- Choix du type AND (parallélisme) : $E_1\&E_2$, cette expression veut dire que E_1et E_2se produisent.

L'opérateur de contrôle de l'exécution appelé OR proposé par Odell et al, n'est pas supporté par l'outil car la sémantique opérationnelle de cet opérateur n'est pas bien définie, et son utilité est excessivement restreinte. Il suffit d'avoir les opérateurs XOR (choix non déterministe) et AND (parallélisme) pour avoir une expressivité équivalente à celle donnée par d'autres notations bien prouvées comme CCS/LOTOS[29, 26], Promela[21] et ESTELLE [27].

Les interactions individuelles acceptées par l'outil sont:

- Envoi de message : $(out, \{src\}, \{tgt\}, (\emptyset, true, msg_{out}))$, dans ce cas on représente l'envoi d'un message msg_{out} du rôle $\{src\}$ vers le rôle $\{tgt\}$, le symbole $\emptyset$ représente une valeur nulle à l'entrée (c'est une interaction d'envoi de message et on n'attend pas de recevoir un symbole) et le symbole *true* spécifie que cette interaction est perpétuellement *habilité*. On peut choisir à souhait l'expression booléenne qu'on utilisera pour habiliter l'interaction.
- Réception de message : $(in, \{src\}, \{tgt\}, (msg_{in}, true, \emptyset))$, cette expression est analogue à la dernière, sauf qu'ici on exprime une interaction pendant laquelle le rôle $\{tgt\}$reçoit de $\{src\}$ un message msg_{in}.
- Envoi de message habilité par une condition interne: $(out, \{src\}, \{tgt\}, (\emptyset, P, msg_{out}))$, le symbole P est une expression booléene exprimée dans la syntaxe de Promela/SPIN (très proche de celle du langage C).
- Envoi de message activé par entrée (ou *triggering*) : $(out, \{src\}, \{tgt\}, (msg_{in}, true, msg_{out}))$, dans ce cas on exprime qu'un message msg_{in} en provenance de $\{src\}$ est suivi immédiatement par un message msg_{out} vers le rôle $\{tgt\}$; à la différence des autres actes d'interaction, dont l'entrée et la sortie sont toujours en concordance avec le *rôle actuel* qui reçoit ou envois le message. Dans le cas du *triggering* le rôle actuel qui exécute l'action n'est pas explicitement mentionné -sauf si le message provient de lui même ou va vers lui même-.
- Envoi de message en XOR : $(xorout, \{src\}, \{tgt_1, ..., tgt_k\}, (\emptyset, P, msg_{out_1}))$, cette expression est équivalente à dire$(out, \{src\}, \{tgt_1\}, (\emptyset, P, msg_{out_1}))$!..! $(out, \{src\}, \{tgt_k\}, (\emptyset, P, msg_{out_k})) \rightarrow exp$.

  > Il faut noter que si on utilise les statuts `xor_out`, `and_out` ou `or_out` il n'est pas possible de définir individuellement l'expression logique qui habilite l'activation de l'interaction, car comme on peut voir, l'équivalence que nous avons défini fait que chacune des interactions individuelles ait la même expression de garde logique.

- Envoi de message en AND : $(andout, \{src\}, \{tgt_1, ..., tgt_k\}, (\emptyset, true, msg_{out_1}))$, expression qui est équivalente à $(out, \{src\}, \{tgt_1\}, (\emptyset, true, msg_{out_1}))$ &..& $(out, \{src\}, \{tgt_k\}, (\emptyset, true, msg_{out_k})) \rightarrow exp$.

- Message en multicast : $(out, \{src\}, \{tgt_1,tgt_2,...,tgt_k\}, (\emptyset,true,msg_{out}))$, cette notation permets d'envoyer le message msg_{out} vers chacune des entités spécifiées dans l'ensemble $\{tgt_1,tgt_2,...,tgt_k\}$.
- Synchronisation : $(in, \{src_1,src_2,...,src_k\}, \{tgt\}, (msg_{in},true,\emptyset))$, cette notation exprime le fait que le rôle tgt attends de recevoir un message msg_{in} en provenance de chaque entité dans l'ensemble$\{src_1,src_2,...,src_k\}$.
- Itération: le langage est capable de représenter d'une façon simple le fait qu'un rôle retourne dans un point antérieur de son exécution. On représente cela avec l'auto-invocation du nom du rôle à la fin d'une des lignes de vie. Par définition, l'auto-invocation est la dernière interaction dans la ligne de vie actuelle, et toute interaction qui soit supposée arriver après cette auto-invocation est interprétée comme une erreur sémantique. L'état dans lequel se trouve le rôle immédiatement après l'exécution de l'auto-invocation est l'état initial, mais avec la différence que les variables manipulées dans l'interaction antérieure ne sont pas réinitialisées et gardent leurs anciennes valeurs.
- Fin: il n'y a pas de notation explicite pour représenter la fin dans l'exécution d'un rôle. Chaque fois que dans une ligne de vie on arrive à un point où il n'y a plus d'interactions spécifiées ensuite, on considère que c'est un état normal de fin.

3.3.1.5 Processus de traduction

Pour effectuer la traduction on s'est servi d'un générateur de compilateurs LARL (Berkeley yacc) et d'un générateur d'analyseur lexiques (jflex) qui ont un nombre acceptable de fonctionnalités et sont amplement documentés.

Le processus de traduction est divisé en deux parties:

1. Génération de l'arbre syntaxique de la spécification.
2. Traduction de l'arbre syntaxique vers une représentation en Promela.

La première partie est faite en utilisant un processus de *parsing* simple avec l'aide d'un analyseur à pile *yacc*, qui crée un arbre binaire dont les nœds feuille sont des interactions individuelles et les nœuds internes sont des opérateurs de contrôle de l'exécution. Le deuxième partie est faite en utilisant une méthode de traduction dirigée par la syntaxe.

3.3.1.6 Génération de l'arbre syntaxique

Le fragment de code 5 montre la syntaxe simplifiée en notation BNF du langage d'entrée. Cette syntaxe est un extrait de celle qu'on a donné comme entrée au générateur d'analyseurs syntaxiques *byacc*. L'outil *byacc* produit automatiquement du code -en ce cas particulier, du Java- qui fait l'implémentation de l'analyseur syntaxique pour la grammaire spécifiée à l'entrée. Dans le fragment de code 5, les symboles en majuscules sont des

Algorithm 5 Grammaire BNF du langage AtoS

$input \Longrightarrow \{\varepsilon \mid spinEntete\ initDec\ SEQCONCATOK\ sentence\}$
$spinEntete \Longrightarrow \{VERBATIM\ any*\ VERBATIM\}$
$initDec \Longrightarrow \{LPARAM\ INIT\ COMMA\ varDecl\ COMMA\ msgDecl\ COMMA\ NULL\ RPARAM\}$
$sentence \Longrightarrow \left\{ \begin{array}{c} declaration\ SEQCONCATOK\ sentence \mid \\ declaration\ XOR\ sentence \mid \\ declaration\ AND\ sentence \mid \\ LPARAM\ sentence\ RPARAM \end{array} \right\}$
$declaration \Longrightarrow \{outDec \mid inDec \mid triggDec \mid brdcstDec \mid synDec\}$
$outDec \Longrightarrow \{LPARAM\ OUT\ COMMA\ source\ COMMA\ target\ COMMA\ ecaDef\ RPARAM\}$
$inDec \Longrightarrow \{LPARAM\ IN\ COMMA\ source\ COMMA\ target\ COMMA\ ecaDef\ RPARAM\}$
$triggDec \Longrightarrow \{LPARAM\ TRIGG\ COMMA\ source\ COMMA\ target\ COMMA\ ecaDef\ RPARAM\}$
$brdcstDec \Longrightarrow \{LPARAM\ BRDCST\ COMMA\ source\ COMMA\ idListe\ COMMA\ ecaDef\ RPARAM\}$
$synDec \Longrightarrow \{LPARAM\ SYN\ COMMA\ idListe\ COMMA\ target\ COMMA\ ecaDef\ RPARAM\}$

symboles terminaux, qui sont traités par un parseur généré par l'outil *jflex*, qui est un dérivé du *GNU flex*.

La partie sémantique de la spécification du langage a comme tâche de construire une structure de données arborescente binaire, qui contient comme feuilles les interactions individuelles, et comme nœds internes ou branches, les opérateurs d'exécution acceptés par le langage.

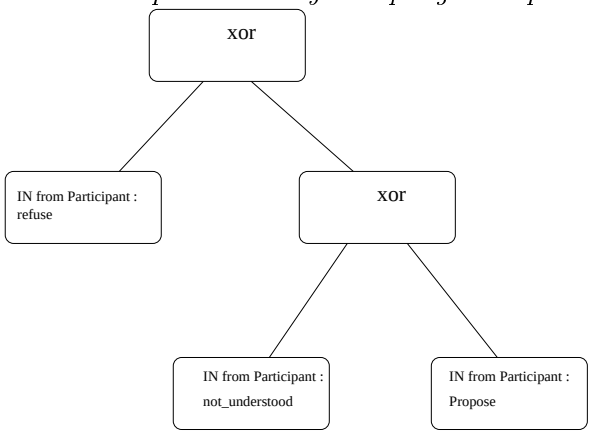
Par exemple, dans la représentation interne du langage accepté par AtoS, l'expression `(in, participant, initiator, [refuse,true,null]) | (in, participant, initiator, [not_undestood,true,null])| (in, participant, initiator, [propose,true,null])` est traduite dans une structure de données telle qu'on la voit dans la figure 3.12.

L'analyseur syntaxique est à une seule phase. Pendant la lecture du code source, la parti sémantique de l'analyseur crée une table de symboles où sont mis les noms d'entités, les valeurs symboliques (étiquettes) et les variables échangées dans la spécification du rôle.

L'analyseur sémantique fait aussi une liste de canaux, information nécessaire surtout au moment de la traduction vers Promela, car ce langage utilise la notion de canal de communication abstrait. Comme notre langage ne déclare pas explicitement les canaux de communication reliant les entités, il fallait les générer en collant les noms des entités invoquées explicitement dans les opérations d'entrée sortie.

Donc à la fin de la première lecture du fichier d'entrée, le logiciel obtient

FIG. 3.12 – *Exemple d'arbre syntaxique généré para l'outil*



comme résultat un arbre d'opérations, une table de symboles et une table de tous les canaux de communication qui sortent on envoient information dans le module.

3.3.1.7 Traduction vers Promela

La traduction du fichier source vers une représentation formelle arrive une fois que le fichier d'entrée est lu et traité. La traduction se fait en utilisant un algorithme récursif classique de traversée de structure arborescente, en suivant un ordre de "fils-à-gauche en premier". Dans chaque instance de la fonction, celle-ci vérifie d'abord si le fils gauche du nœud actuel est une *interaction atomique* (n'importe quelle opération d'entrée sortie), si c'est le cas, la fonction produit le code Promela nécessaire pour représenter l'occurrence d'un acte d'interaction. La génération du code est plutôt trivial, bien qu'elle doit se faire avec beaucoup de soin, pour en assurer la correction syntaxique et sémantique des expressions générées dans le langage final. La première phase assure que dans toute structure de données nœud-feuille contenant une interaction atomique, il y a déjà toutes les valeurs nécessaires pour construire une opération d'entrée ou de sortie correcte dans le langage final (le type d'opération, les symboles à échanger, les sources ou les destinations de l'interaction, etc.)

Le fragment de pseudo-code dans l'algorithme 6 montre le processus de traduction d'une façon simplifiée. Dans une première instance on avait décidé de générer une représentation finale qui suivait rigoureusement la forme d'une machine à état finis, mais la traduction vers un automate pur en Promela s'est vite révélée être beaucoup plus coûteuse, qu'une traduction qui profite des caractéristiques de Promela. La raison de cela est que les opérateurs de séquence et de choix non-déterministe entre expressions augmentent la taille de l'espace d'états de l'automate de façon polynômial, pendant que

la représentation du parallélisme en termes d'états et transitions dans un automate, l'augmente de façon exponentielle.

En particulier, la séquence et le choix non déterministe sont traduits vers une notation Promela qui représente la transitions d'états. Pour ce faire nous produisons dans le logiciel une étiquette numérique interne au processus, étiquette qui en prenant certaines valeurs choisies, habilite l'exécution de certains fragments de code. Chaque occurrence d'un opérateur de séquence augmente exactement d'un le nombre d'états du système, pendant que le choix non déterministe l'augmente d'au moins deux états.

Le parallélisme par contre, représente une "rupture" de la notation d'états, car on utilise simplement l'opérateur de parallélisme donné par le langage final, pour commencer les deux expressions résultantes de la traduction du fils gauche et droit, à partir du nœud contenant l'opérateur de parallélisme.

Il est relativement facile de générer du code Promela pour représenter l'entrée et la sortie de valeurs. Le langage donne des constructions linguistiques explicites pour ça. Par exemple, une entrée d'une valeur symbolique $sym0$ par un canal $c0$ est représentée `c0?sym0` en Promela, en ce qui concerne la sortie d'une valeur symbolique $sym1$ par un canal $c1$, tout est représentée par `c1!sym1`.

Malheureusement il y a des restrictions de langage qui rendent le travail de génération d'une spécification correcte un peu plus compliqué. Par exemple, il existe une restriction dans le langage Promela, qui fait que tous les canaux doivent être déclarés comme convoyeurs d'un seul type de données, ce qui oblige en l'occurrence, à ce que toutes les valeurs qui passent à travers un canal soient du même type. Si on déclare un canal comme transportant des étiquettes symboliques, on ne peut envoyer que des étiquettes à travers lui, du même si on le déclare comme transporteur de valeurs entières, ou d'octets. La solution la plus courante à cet inconvénient, est de déclarer des canaux qui transportent des valeurs composées ou structures (Promela permet de définir des types de données complexes), structures qui une fois définies avec un ensemble assez riche de valeurs internes, nous permettent de représenter d'une manière approchée tous les différents types de paquetages envoyés dans un système réel en réseau.

Cependant on a utilisé une approche différente pour la solution adoptée dans l'outil. Comme on a définit que le langage d'entrée permet l'envoi d'ensembles de valeurs symboliques (c'est à dire, des étiquettes) et numériques (des valeurs entières) dans une même interaction, dans la spécification finale on définit des canaux qui transportent des structures, mais qui doivent être seulement capables de contenir *une seule* instance de toutes les valeurs trouvées en traversant le canal en question dans la spécification. L'envoi de toute une séquence est simulé par l'envoi en séquence de chacune des valeurs contenues dans l'ensemble définit dans la spécification d'entrée.

Par exemple, une interaction de sortie qui envoie `un` ensemble de valeurs $\{e1,2,e2\}$ d'une entité A vers une entité B, serait représentée en langage

Algorithm 6 Algorithme de traduction

fonction traduction (noeudActuel: noeud, etatActuel: état) : code
variables
code de sortie : code;
commence
si noeudActuel=racine **alors**
code de sortie := l'entête du module;
si filsGauche(noeudActuel) est une interaction atomique **alors**
commence
code de sortie:= code de sortie + le code de sortie ou entrée approprié, à travers le/les canaux désignés et avec le/les symboles choisis;

fin;
sinon si typeOperateur(noeudActuel) = séquence **alors**
commence
code de sortie :=traduction(filsGauche(noeudActuel)) en séquence avec traduction(filsDroîte(noeudActuel));
fin;
sinon si typeOperateur(noeudActuel) = choix non déterministe (XOR) **alors**
commence
code de sortie := établir le choix entre traduction(filsGauche(noeudActuel)) et traduction(filsDroîte(noeudActuel));
fin;
sinon si typeOperateur(noeudActuel) = choix non déterministe (XOR) **alors**
commence
code de sortie := établir le parallélisme entre traduction(filsGauche(noeudActuel)) et traduction(filsDroîte(noeudActuel));
fin;
si noeudActuel=racine **alors**
code de sortie:= code de sortie + la clôture du module imposée par le langage;
retourne code de sortie;
fin;

d'entrée comme `(out, A, B, [null,true,{e1,2,e2}])`, et aurait comme traduction finale une expression en Promela comme celle-ci:

```
A_B! label, e1, 0;
A_B! number, none, 2;
A_B! label, e2, 0;
```

Ce qui exprime le même comportement, mais en envoyant une seule structure générique, contenant une seule valeur à la fois à travers le canal A_B. Cette solution n'est sans doute pas la plus élégante, mais elle est dans nôtre expérience pratique, la plus facile à implémenter et aussi celle qui produisait le code final le plus lisible.

3.3.1.8 Traduction des diagrammes de rôles vers modèles formels

Étant donné que le langage que nous avons défini est équivalent à un diagramme de séquence d'AUML (avec toutefois quelques modifications et extensions), il s'avère nécessaire de prendre des décisions sur la manière d'amener d'une sémantique de diagramme de rôle, vers une d'entités-agents concrets.

Après tout, les rôles sont des descriptions génériques du comportement qu'une entité concrète doit suivre pour accomplir une fonction, donc ils ne sont pas exécutables par eux-mêmes, sinon est par une entité que va prendre ce rôle dans une communication.

Les diagrammes de rôle proposés par AUML et acceptés par la FIPA sont on ne peut plus explicites à ce sujet: les rôles représentent potentiellement le comportement d'un ensemble d'entités, donc dans un diagramme de rôles on n'a pas la totalité de la description de l'architecture d'un système.

Bien que le traducteur défini dans cette première étape se soit révélé utile pour prendre une description proche de la notation visuelle utilisée pour modéliser les protocoles, et pour générer en suite une description formelle de ce diagramme dans un langage formelle bien connu; Il reste un inconvénient majeur: le langage ne propose pas une notation explicite pour ce qu'on appelle "la création d'instances" de rôle.

Pour couvrir cette lacune fonctionnelle, le langage prend comme hypothèse de base que pour chaque rôle déclaré dans la spécification, il y a une et seulement une entité qui prend ce rôle. Dans cette première version de l'outil, la description faite du rôle était en même temps la déclaration de l'agent que l'implémente.

En reprenant l'exemple de ContractNet proposé dans le fragment de code 4, il faudrait augmenter le nombre d'entités *Participant* pour pouvoir représenter réellement une instance de tel protocole. Il faut noter qu'une telle manipulation modifierait aussi sensiblement le modèle du rôle Initiateur.

3.3.1.9 Fonctionnalités supplémentaires

Une de fonctionnalités les plus intéressantes proposées par Promela/SPIN, est sa capacité de déterminer la vivacité d'un modèle, tout en permettant au développeur de choisir les points dans l'exécution qu'on considère comme une progression dans l'exécution. L'outil peut déterminer s'il y a des cycles infinis dans le modèle qui ne passent jamais par ce point de l'exécution, et déterminer ainsi un manque de vivacité. De la même façon, Promela/SPIN peut aussi être utilisé pour savoir si le modèle arrive éventuellement à un état final correct.

L'outil AtoS est assez flexible pour permettre de générer une traduction adaptée qui utilise ces deux capacités. Par défaut, la traduction obtenue marque les état finaux du modèle avec la notation requise par SPIN (des étiquettes explicites). Il est facile de connaître les états finaux car dans le langage d'entrée toutes les fins de ligne de vie sans itération sont considérées comme des états finaux. Donc avec la traduction par défaut il est possible de vérifier si le rôle a bien un état final en utilisant la validation *on the fly* proposée par SPIN.

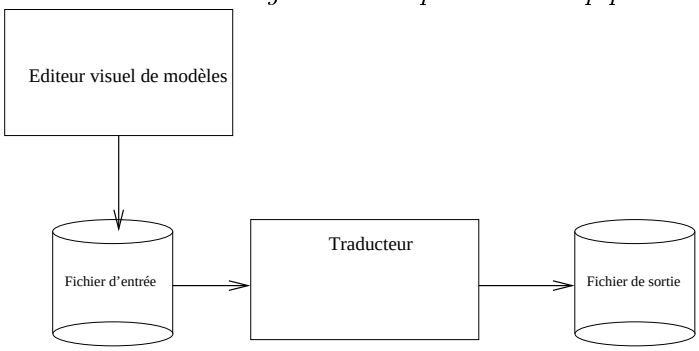
Si on est intéressé par un point particulier de l'exécution du système, il faut l'exprimer explicitement dans le texte du modèle d'entrée. Pour représenter que nous désirons savoir si le système est vivace selon un critère qui nous est propre, nous marquons avec un caractère spécial la ou les interactions atomiques qui soient considérées comme une progression dans l'exécution du système. La traduction prend en compte ce marquage, et produit une spécification avec la notation adéquate (des étiquettes progress).

3.4 Modélisation visuelle

Cette partie correspond aux travaux présentés dans le papier [54]. Une fois que nous avons développé un traducteur capable de représenter notre proposition de rôle, nous avons décidé d'étendre les capacités de modélisation vers une approche plus visuelle. La raison principale de ce choix était de tester "sur le terrain" une notation visuelle inspirée par AUML. Les spécifications textuelles de notre langage sont confrontées au même problème de manque d'intelligibilité, qui affecte les notations formelles qui lui servent d'inspiration: à partir d'une certaine taille du problème traité, la signification du modèle devient de plus en plus difficile à saisir, jusqu'à ce qu'elle soit pratiquement inintelligible.

La figure 3.13 montre la structure modulaire du système à cette phase du développement. La fonction principale de l'interface est de recevoir des commandes de l'utilisateur au travers d'une interface visuelle d'utilisateur simple. La conception des commandes recevables par l'outil vise la construction d'un modèle visuel basé sur la notation AUML, en incluant les extensions (et omissions) proposées par nous; car finalement la fonctionnalité de l'outil

FIG. 3.13 – *Schéma générale du processus en pipe-line*



est de générer les fichiers d'entrée pour le traducteur AtoS.

3.4.1 Artefacts visuels acceptés

A ce stade du développement, on a repris une partie des artefacts graphiques des diagramme de séquence proposés par Odell et al dans [46, 47], c'est-à-dire, les opérateurs de choix non déterministe (XOR), et de parallélisme (AND) tant pour la ligne de vie des entités-rôle comme pour les interactions. Et on en a ajouté quelques-unes, à savoir:

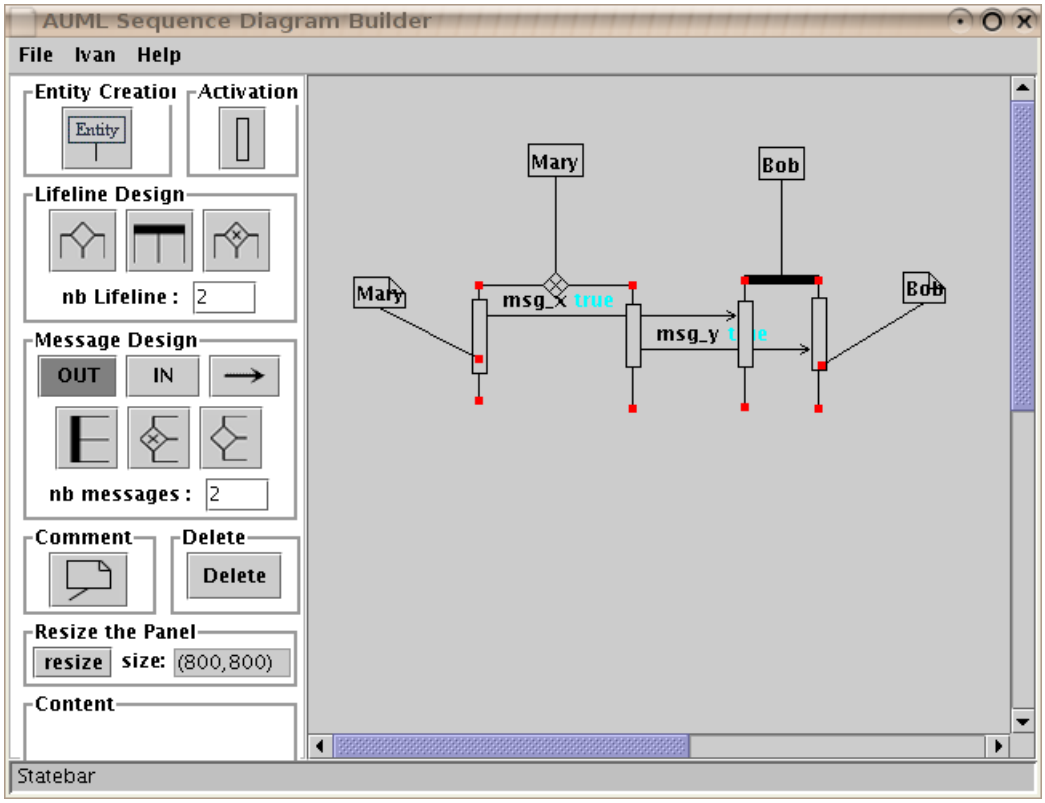
Synchronisation Cette opération permet de représenter les cas où il est nécessaire de recevoir un message de notification à partir d'un ensemble de rôles différents, pour pouvoir continuer l'exécution. Le symbole utilisé pour représenter la synchronisation est analogue a celui de l'opérateur de parallélisme, sauf que la direction des flèches est inversée.

Itération Pour représenter l'occurrence d'une itération on a utilisé l'opérateur de commentaire standard en UML, avec inclusion de l'étiquette du rôle.

Dans la figure 3.14 on peut voir l'apparence de l'interface. Le modèle d'utilisation est relativement simple, avec un écran divisé en trois zones, la première zone est un menu d'utilisateur avec des options indispensables (sauvegarder, ouvrir, sortir, etc.), la deuxième zone est dédiée à une barre d'outils contenant les artefacts visuels acceptés sous la forme d'icônes, et la troisième zone est dédiée à la création et la manipulation des schémas.

Le processus de création du schéma suit une approche constructive classique pour ce type d'interface, dans laquelle l'utilisateur sélectionne avec le pointeur l'icône dans la barre d'outils représentant l'artefact graphique qu'il désire ajouter au diagramme, et qu'il positionne ensuite avec la souris dans l'espace de travail. Les éléments graphiques sont capables de reconnaître ceux avec qui ils peuvent se lier et ceux avec qui se lier est dépourvu de sens.

FIG. 3.14 – *Interface d'utilisateur*



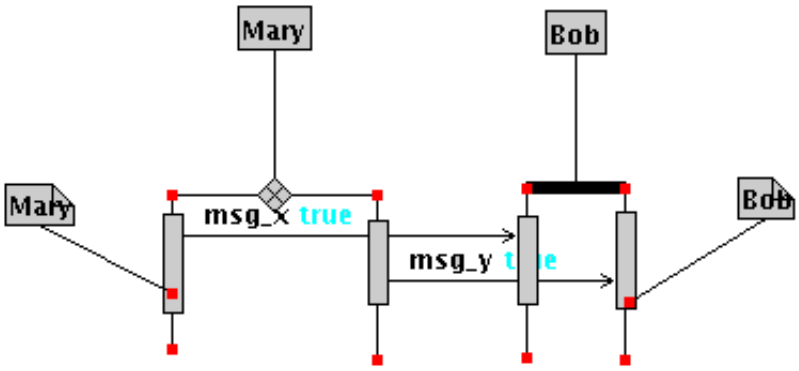
Cette interface bien que conçue pour satisfaire nos besoins, est plutôt limitée dans ses capacités d'édition et en fonctionnalités. Par exemple, les entités superflues peuvent être éliminées à n'importe quel moment, mais à la condition qu'elles ne soient pas reliées avec d'autres. Une fois qu'elles le sont, les entités peuvent être effacées dans l'ordre dans lequel ont été ajoutées dans le diagramme. Aussi le logiciel ne donne pas de fonctionnalités élémentaires comme l'impression du diagramme, bien que dans le cas d'un outil pour créer de schémas, une telle fonctionnalité aurait été utile.

Avec des fonctionnalités ainsi restreintes, nous pourrions nous demander à quoi exactement nous sert cet outil. C'est ce à quoi nous allons donner une réponse dans la suite.

3.4.1.1 Détection d'inconsistances sémantiques

Le fait d'utiliser une notation graphique pour modéliser un système complexe comme un protocole de communication, s'est révélé être à la fois convenable et source de problème particuliers. L'adéquation de la notation gra-

FIG. 3.15 – *Exemple de diagramme inconsistent*



phique découle naturellement de la manière de représenter l'exécution d'une entité en termes des séquences d'événement dans le temps. La difficulté vient de l'expressivité de la notation elle-même.

On s'est vite aperçu qu'il est très facile de construire des diagrammes qui semblent avoir un sens et être sémantiquement correctes, mais qu'au moment de la traduction et validation le model-checker détectait des problèmes importants.

Par exemple, dans la figure 3.15 on a un schéma syntaxiquement valable créé avec l'interface d'utilisateur, dans lequel plusieurs inconsistances sémantiques flagrantes sont représentées. La première est le fait que la sémantique de l'opérateur de choix non déterministe (XOR) dans la ligne de vie de l'entité *Mary* spécifie qu'une seule de deux alternatives est choisie, donc cette entité peut ou bien envoyer le message msg_x ou le message msg_y, pendant que l'opérateur de parallélisme dans la ligne de vie de l'entité *Bob* indique que cette entité attend de recevoir les deux. Un autre problème vient de l'utilisation des opérateurs d'interaction dans deux sous-lignes de vie.

Dans l'entité *Bob*, l'occurrence d'une itération dans une des branches exprime le fait qu'une fois que l'entité a reçu le message msg_y, cette entité retourne à son état initial. Cependant, l'entité *Mary* choisi de façon non déterministe d'envoyer seulement un des deux messages msg_x et msg_y, de telle sorte que si elle choisi msg_x, elle retourne en suite à son état initial, provoquant la fin de l'exécution de l'entité *Bob*, malgré le fait que cette entité sera à nouveau prête à envoyer un des deux messages mentionnés auparavant. Évidemment, ce diagramme présente plusieurs possibilités de génération de deadlocks, et la validation de la traduction finale en Promela preuve ce fait sans trop de difficultés.

Cette exemple est certes, très simple, mais il représente aussi une instance d'une série d'autres problèmes fins associés à la notation utilisée (en ce cas

AUML modifié). Cependant, on croit que la notation proposée par la version 1 de UML est déjà assez riche pour être au moins équivalente à une notation comme CCS.

3.4.1.2 Types d'erreurs communes

Et faisant des diagrammes avec la notation graphique proposée, on a constaté qu'il y a quelques types d'erreurs sémantiques plus récurrents, à savoir:

Non rendez-vous dans les diagrammes Il est très fréquent de retrouver ce problème, bien que l'interface graphique assure qu'il y a bien une interaction d'entrée correspondant à chaque interaction de sortie qui part d'une entité donnée vers une autre. On doit s'assurer que chaque fois qu'il y a un processus d'interaction, celui-ci peut être exécuté correctement. En validant le modèle final en langage Promela, ce problème se manifeste le plus souvent comme un deadlock. Les causes qui provoquent le fait qu'une interaction quelconque ne peut pas être exécutée sont principalement:

- Finalisation prématurée. Ceci arrive quand une entité finit son exécution pendant qu'une entité externe attend encore d'interagir avec elle. Dans les diagrammes AUML il est très facile de provoquer la fin de l'exécution d'une entité sans le vouloir.
- Incompatibilité opérationnelle. C'est ce qui arrive quand la définition de la ligne de vie ne prend pas correctement en compte la séquence possible d'interactions entre les partenaires. L'exemple le plus fréquent est l'utilisation incorrecte des opérateurs qui divisent la ligne de vie, ce qui peut provoquer qu'une certaine interaction d'entrée ou de sortie ne puisse pas se produire.

Non finalisation Par définition, tous les rôles doivent avoir au moins un état final. Ceci est dû au fait que les rôles sont supposés être des séquences réutilisables des messages, et pour avoir une telle possibilité, il est nécessaire que le comportement décris ait un état final. Il est possible avec la notation que nos avons proposé de construire des rôles qui ne finissent pas, mais leur utilité comme unités ou bloques réutilisables pourrait être mise en question à ce moment-là.

3.4.1.3 Assistance au développeur

Bien que l'interface graphique ait comme utilité principale la manipulation du diagramme de rôles qui fait la spécification, ainsi que la génération de code pour le traducteur; elle propose aussi quelques fonctionnalités ajoutées pour économiser du travail au développeur. Car il y a du travail qui est très répétitif et qui peut être effectué par l'ordinateur lui-même.

La première de ces fonctionnalités est la détection de la condition de non finition. L'outil peut détecter directement les cas les plus évidents de non finition, et générer une notification au développeur relative à ce fait. Ces cas dits évidents sont tous ceux dont on a un diagramme qui a à la fin de toutes ses sous-lignes de vie un opérateur d'itération.

Cependant, l'interface graphique peut être aussi utilisée pour détecter les cas qui ne sont pas directement détectables. Pour effectuer cette détection on utilise les capacités que l'outil de traduction propose par défaut.

3.5 AUML version 2

Comme il a été dit auparavant, la notation AUML telle qu'Odell et al l'avaient défini [46, 47] n'était qu'une proposition sans objectif de standard. Comme telle elle était sujet aux commentaires, modifications et extensions par de tiers, et aussi aux actualisations à la vue des progrès du génie logiciel dont elle s'inspirait. Car il faut noter que UML est aussi un standard qui change.

3.5.1 UML 2

L'organisation qui propose UML (le *Object Model Group*) a récemment lancé la version UML 2 [14], nouveau standard qui est supposé améliorer certains des artefacts déjà présents, et qui en ajoute d'autres. Certains de ces artefacts ont des caractéristiques qui les rapprochent de certaines idées déjà présentes dans la proposition d'AUML, et d'autres sont complètement nouvelles. Cet avancement notoire du standard de base, a fait qu'une nouvelle proposition d'AUML a été créée.

UML 2 propose des changements pour les artefacts suivants déjà présents dans UML 1:

- Les diagrammes de paquetages.
- Les diagrammes de classes.
- Les diagrammes de cas d'utilisation.
- Les diagrammes d'objets.
- Les diagrammes d'implantation.
- Les diagrammes de charte d'états.
- Les diagrammes de séquence.
- Les diagrammes d'activités.

Et aussi propose des nouveaux artefacts pour le développeur. À savoir:

- Les diagrammes de structure composée.
- Les diagrammes de composants.
- Les diagrammes de résumé d'interaction.
- Les diagrammes de temps.

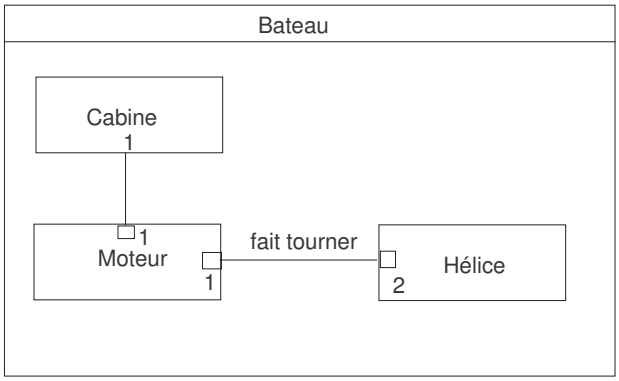


FIG. 3.16 – *Diagramme de classe contenant des parties, des ports et des connections*

En général, la nouvelle version d'UML amène des changements dans les aspects structurelles du modèle à objets du système. Par exemple, dans les diagrammes de classe il y a addition des concepts tels que:

Partie: Une partie est un attribut ou une instance d'une classe quelconque dont l'intégralité de sa durée de vie est incluse dans la durée d'une autre classe. Les parties sont éléments qui composent une instance d'une classe majeur. C'est une amélioration de la notion *d'agrégation* déjà présente dans les *diagrammes de classe*.

Port: Les ports sont un artefact qui est associé aux *Parties*, et qui exprime un point d'interaction à travers lequel le monde extérieur peut s'adresser à une entité d'une certain classe. Le port est un attribut de classe qui peut recevoir des messages. Un *port* relie exclusivement les *Parties* et pourrait être vu comme l'équivalent d'un point d'interaction, dans les techniques de description formelle pour les systèmes communicants, telles que LOTOS et ESTELLE.

Connecteur: Un connecteur est le lien qui connecte les Ports et qui expriment la présence d'un canal de communication entre les deux *Parties*. Les connections ont une cardinalité à chaque extrémité, donc elle peuvent exprimer la relation qui existe entre une partie et les autres.

Le diagramme 3.16 montre un exemple de diagramme de classe d'UML 2 qui représente la structure d'une classe *Bateau* qui a un *Moteur* et deux *Hélices*. Les boîtes étiquetées représentent les parties qui composent la classe *Bateau*. Les ports apparaissent comme des petits cubes dessinées à l'intérieur de la boîte représentant la classe.

Une autre amélioration importante proposée par UML 2 se situe dans la notation des *chartes d'états*. La notation de charte ou machine à états dans UML 1 et 2 sert à représenter la séquence des appels de méthode qu'une

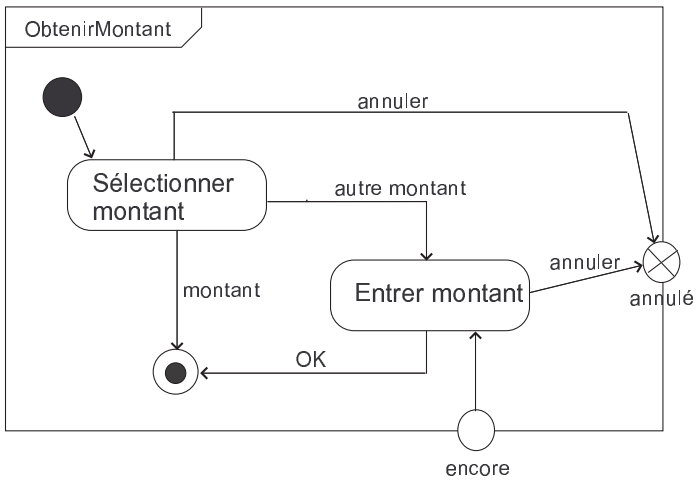


FIG. 3.17 – *Exemple de diagramme d'états*

classe quelconque fait ou reçois pour accomplir sa fonction. Dans la nouvelle version de la notation, les machines à états peuvent être *assemblées*, *héritées* et *surchargées*. Parmi les nouvelles caractéristiques admises par UML 2 se trouvent:

- Des machines composés avec points d'entrée et sortie, qui permettent construire une machine à états complexe avec des sou-machines plus simples, ainsi qu'une certaine réutilisation notationnelle.
- Généralisation des machines à états. La généralisation de classes est maintenant aussi applicable aux machines à états. Les sous-machines peuvent être modifiées si nécessaire.

Les figures 3.17 et 3.18 sont des exemples de diagrammes d'états dans la nouvelle version d'UML. Le diagramme 3.17 montre l'automate qui pourrait modéliser une interaction simple entre l'utilisateur et un distributeur automatique d'argent. L'état initial normal est marqué par la flèche qui part du point noir en haut à gauche. Le cercle contenant deux lignes croisées (étiqueté *annulé*) exprime un état final de type erroné, tandis que le cercle vide qui mène vers l'état étiqueté *Entrer montant* est un autre point d'entrée qui sera activé dans une autre partie du diagramme.

Le diagramme 3.18 représente l'activité globale du distributeur. Dans les activités du distributeur, les sous-activités sont modélisées par des sous-machines à états, représentées dans le diagramme par des boîtes étiquetées. On peut voir qu'une des boîtes a le même nom que celui de la figure 3.17, ceci montre une composition, et comment utiliser les points d'entrée et sortie définis dans le diagramme 3.17 (comme *annulé* et *encore*).

Un autre changement significatif est l'inclusion de la notion de *généra-*

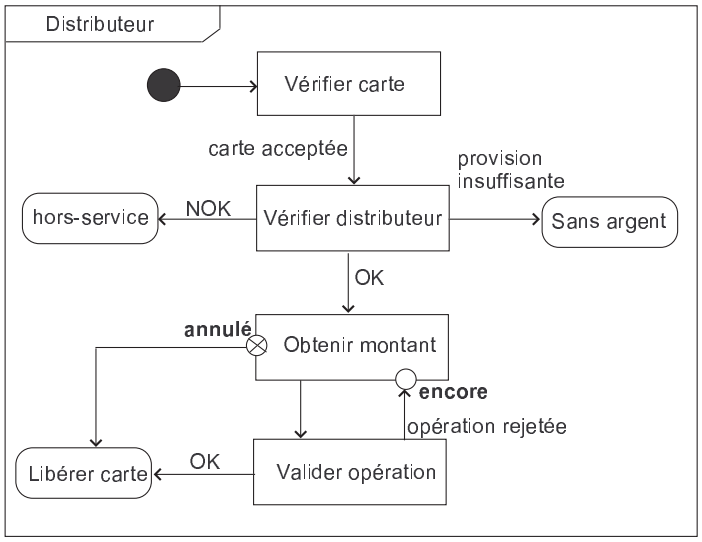


FIG. 3.18 – *Diagramme d'états composé dans UML 2*

lisation et de *surcharge* de la charte d'états. Ceci implique que chaque fois qu'une relation de généralisation est définie, la charte d'états de la classe descendante est copiée à partir de celle de la classe d'origine. Dans UML 1 on pouvait supposer que la même chose arrivait, mais la sémantique des effets de la généralisation sur la charte d'états était laissée au soin du développeur. Dans la figure 3.19 on peut voir la relation explicite existante entre la généralisation de classes et la généralisation de sa charte d'états. Par défaut, la machine d'états exprimée dans *charte d'états 2* et la même que la *charte d'états 1*.

Il faut dire que la sémantique de généralisation s'applique aussi à la surcharge du comportement de la classe: la charte d'états et ses sous-chartes peuvent être modifiées individuellement, mais en essayant de respecter les points d'entrée et de sortie déjà spécifiés dans les diagrammes ancêtres. Nous croyons que ces changements en tout sont très significatifs pour les capacités expressives de la notation UML, et dans le cas particulier de la notation d'extension d'états, se rapprochent à l'esprit à ce qu'on avait mentionné dans [53].

Les modifications et extensions liées aux parties et aux ports rapprochent plus UML des notations formelles dédiées à la spécification des systèmes communicants tel que SDL, Estelle, Promela et LOTOS.

Bien que ces changements soient significatifs, les modifications les plus importantes pour ceux qui s'intéressent à la modélisation des systèmes multiagents sont celles qui touchent la *représentation de l'interaction*.

Dans UML 2 il y a quatre artefacts graphiques qui traitent de l'interac-

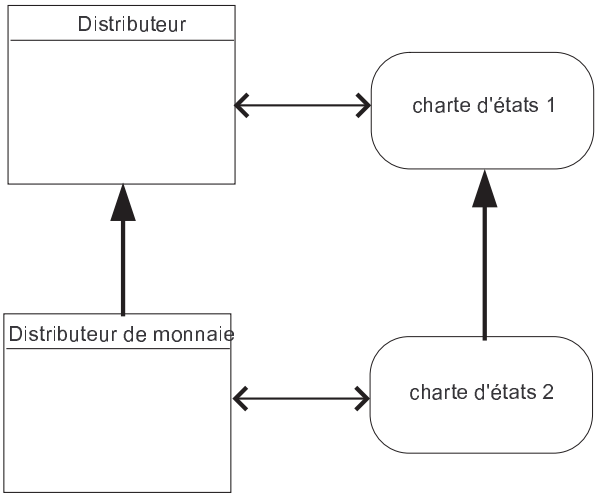


FIG. 3.19 – *Généralisation de la charte d'états en pseudo-UML 2*

tion, à savoir:

Diagrammes d'interaction: appelés aussi *diagrammes de séquence*. Ceux-ci expriment l'interaction en se centrant sur l'ordre ou séquence des messages échangés par des entités appartenant à deux ou plusieurs classes, dans une instance d'interaction.

Vue générale d'interaction: ces diagrammes sont une variante des diagrammes d'activités mais qui à la place de représenter l'ordre *d'exécution* d'un groupe d'actions ponctuelles, représentent l'ordre *d'interaction* entre groupes d'entités et dont les nœuds peuvent être des diagrammes de séquence complets.

Diagrammes de communication: aussi appelés *diagrammes de collaboration*, ces artefacts représentent les relations entre entités en se centrant sur l'ordre ou la séquence des messages que ces entités échangent. A la différence des diagrammes de séquence, où l'ordre est spécifié avec une convention spatiale (la représentation visuelle de haut en bas), dans les diagrammes de communication on utilise des étiquettes numériques sur les messages échangés. Ces diagrammes donnent la même information que les diagrammes de séquence mais d'une manière différente.

Diagrammes de temps: ces diagrammes spécifient le changement dans le temps de l'état interne d'une ligne de vie donnée. Utilisés pour représenter les réactions des entités face aux stimulus.

Les diagrammes de séquence d'UML 1 étaient inspirés de la notation de *chronogrammes* déjà présente dans plusieurs méthodologies et ses notations associées. La notation était assez simple, et n'admettait qu'une quantité

limitée de types de communication et d'exécution. UML 2 ajoute une plus grande quantité de types de communication et aussi permet de modéliser un plus grand ensemble de cas d'exécution dans l'envoi de messages.

Les diagrammes d'interaction sont faits avec les sous-artefacts suivants:

Cadre: la notation de *cadre* est utilisée pour donner un nom et une unité sémantique de façon visuelle au processus d'interaction.

Ligne de vie: la ligne de vie représente l'exécution interne d'une entité présente dans le processus d'interaction. Elle est représentée par une boîte étiquetée avec une ligne qui part d'en bas. Cette ligne représente l'avancement du temps.

Occurrence d'exécution: cet artefact exprime le fait que l'entité est active pendant une certaine durée. La notation pour cet artefact est une boîte en trait plein sur la ligne de vie de l'entité.

Occurrence d'interaction: cet artefact représente le fait qu'une certaine séquence d'interaction stéréotypée arrive, et qu'elle est définie ailleurs. Les séquences stéréotypées peuvent être définies une seule fois dans le modèle, économisant ainsi du travail et permettant une certaine réutilisation notationelle.

Fragment combiné: la notation de fragment combiné permet de modéliser un ensemble de cas d'exécution sur une ligne de vie, par exemple, le parallélisme ou le choix non déterministe.

État invariant/continuation: avec cette notation il est possible d'exprimer des conditions qui doivent être satisfaites par une entité pendant le déroulement de l'interaction.

Co-région: la notation de co-région est une alternative visuelle pour représenter le parallélisme, elle est utilisée à la place d'un fragment combiné.

Arrêt: la notation d'arrêt montre la fin d'une entité de façon explicite.

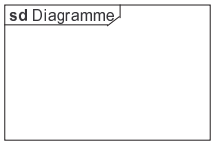
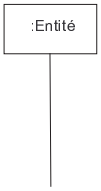
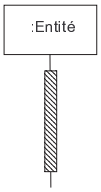
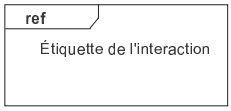
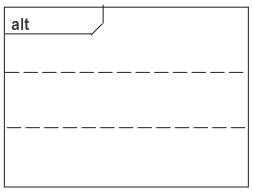
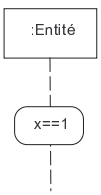
Contrainte de durée: avec UML 1 il était déjà possible de représenter la durée entre un message et un autre, mais UML 2 améliore la sémantique de cette notation. En particulier, il est possible d'utiliser des variables de temps obtenues sous plusieurs formes et les utiliser comme contrainte quelque part dans l'interaction.

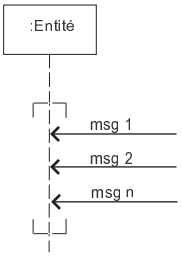
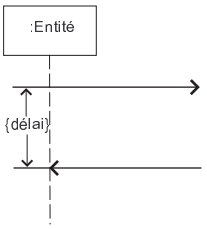
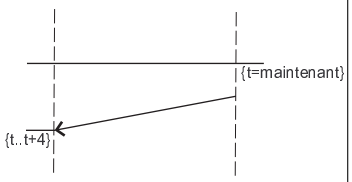
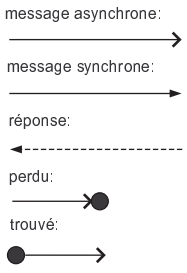
Message: les types de messages supportés par UML 2 sont les mêmes qu'en UML 1: les messages asynchrones, les messages synchrones et les réponses.

Message perdu: cette notation représente la perte d'un message.

Message trouvé: cette notation sert à représenter les cas où il est nécessaire de montrer l'arrivée d'un message inattendu ou avec une origine inconnue.

Ordre générique: la notation d'ordre générique sert à restreindre les possibles séquences de messages présentes dans une interaction. Cette artefact peut lier deux messages ou plus et implique que ces messages arrivent avec un ordre séquentiel stricte.

Nom	Symbole	Explication
Cadre		La notation pour représenter un cadre est une boîte étiquetée avec le stéréotype **sd** suivi du nom de la séquence.
Ligne de vie		La ligne de vie est représentée par une ligne qui part d'une boîte étiquetée avec le nom d'un rôle ou d'une classe, elle est toujours unique, bien qu'avec la notation de fragment combiné il est possible d'exprimer le fait que l'exécution est divisé en chemins parallèles.
Occurrence d'exécution		Cette notation montre la présence d'activité dans une entité, elle est représentée par un carré plein sur la ligne de vie. Cette notation est identique à celle déjà présente dans UML 1.
Occurrence d'interaction		Cet artefact est représenté par une boîte étiquetée avec le stéréotype **ref** et un nom déjà utilisé dans un autre diagramme. Il montre que l'intégralité des actions modélisées par un autre diagramme arrive à un certain point de l'interaction actuelle et que cette séquence est définie ailleurs par économie.
Fragment combiné		La notation de fragment combiné exprime des opérations sur une ou plusieurs sous-lignes de vie. La notation est appliquée sur une ligne de vie, laquelle est divisée en deux ou plusieurs morceaux sur lesquels l'opération signalée visuellement avec un stéréotype est appliquée. Ces opérations sont: choix d'une seule alternative (stéréotype **alt**) entre plusieurs, parallélisme (stéréotype **par**), *opération optionnelle* (stéréotype **op**) et *séquence faibles* (**seq**).
Invariant d'état/ Continuation		Les invariants sont représentées par une boîte arrondie contenant une contrainte exprimée en OCL.

Nom	Symbole	Explication
Co-région		La notation de co-région est une abréviation de celle de parallélisme proposée dans l'artefact de fragment combiné.
Arrêt		La notation d'arrêt est représentée avec une croix sur la ligne de vie. Aucune opération ne peut être effectuée sur la ligne de vie à partir de ce point
Contrainte de durée		Cette notation sert à représenter des contraintes de durée entre deux interactions, les unités de temps peuvent être choisies à souhait.
Contrainte de temps		Il est possible aussi de déclarer des variables qui sont initialisées avec des opérations de temporisation telles que *temps actuel*, ou bien balisées dans les messages échangés, et les utiliser quelque part dans le diagramme.
Messages		Les types de messages dans UML 2 sont un sur-ensemble de ceux proposés dans UML 1, avec l'ajout d'une notation spéciale pour représenter des cas particuliers, comme la perte de messages ou les messages inattendus.

Nom	Symbole	Explication

3.5.2 Extensions proposées dans AUML 2

La première proposition de AUML proposait de nouveaux artefacts dont la fonctionnalité était d'étendre l'expressivité de la notation UML, surtout en ce qui concerne la représentation de l'interaction entre entités. Aspect qui était relativement faible (du point de vue de la modélisation de *protocoles*, et non des *messages* entre objets).

Cependant, ce état de faits a considérablement changé depuis, car les acteurs industriels derrière UML 2 ont demandé l'inclusion de plusieurs nouveaux concepts et modifications dans la notation, et bien évidement, aussi pour celle qui modélise l'interaction. Ces modifications vont précisément dans le même sens déjà prévu par les créateurs de AUML et par nous: l'enrichissement de l'expressivité en termes d'opérateurs d'exécution.

En particulier, on a la nouvelle définition du diagramme d'interaction dans UML 2 avec l'artefact appelé *Fragment Combiné*, qui fait tout ce qui était fait par les anciens opérateurs de contrôle d'exécution et ceux de communication dans AUML.

Ceci étant, dans la nouvelle proposition d'AUML [25, 24] on n'a plus besoin d'enrichir avec des opérateurs d'exécution ou de communication la notation de ligne de vie et les messages déjà présents dans UML 2. Les extensions se centrent maintenant sur d'autres aspects plus particuliers aux systèmes multiagent, à savoir:

- L'organisation: la proposition actuelle d'AUML enrichi les diagrammes de classe d'UML 2 et retrouve des concepts assez proches de ceux présents dans des cadres méthodologiques comme MadKIT [18].
- L'interaction: à nouveau les diagrammes d'interaction sont considérés comme l'artefact adéquate pour représenter les protocoles d'interaction, cependant Odell, Huget et al [24] prennent la notation UML 2 et ajoutent quelques nouveaux stéréotypes à l'artefact *fragment combiné*. Les extensions sont plus centrées sur la représentation de la notion de rôle, le comportement générique et la réutilisation. Aussi ils proposent plusieurs modifications de l'artefact appelé *vue générale d'interaction*, notation qui était absente dans UML 1.

3.6 Une approche constructive pour la spécification

Bien que les résultats obtenus dans la première partie de notre travail montraient qu'il était possible de faire la traduction des diagrammes de séquence inspirés de la notation AUML vers une notation formelle, cette solution restait incomplète pour une raison fondamentale: son incapacité à

représenter multiples instances d'un seul rôle. En effet, la traduction automatique produisait comme résultat un modèle finale, qui présumait que dans le système spécifié il y avait une seule instance de chaque rôle décrit dans la spécification textuelle donnée en entrée, présomption naïve tout au moins, quand la notation AUML étant basée sur l'idée de rôle, par elle même suggère la possibilité d'avoir plusieurs entités qui prennent un même rôle.

Cependant, admettre une telle occurrence a des implications certaines sur la sémantique d'une spécification, ainsi que sur la spécification des entités. La raison à cela est que les diagrammes de rôle proposés dans AUML version 1 et les nôtres n'ont pas de notation explicite pour représenter la cardinalité des relations entre les rôles et les instances qui les adoptent. La raison est que les diagrammes de séquence n'avaient pas une approche faite pour l'implantation, et les aspects relatifs au nombre d'entités qui prennent un certain rôle étaient vus comme secondaires dans le contexte de cet artefact en particulier. Cette information se devait d'être spécifiée avec une nouvelle notation o bien quelque part ailleurs, dans un équivalent de l'artefact diagramme d'objets déjà présent dans UML. Cependant, cette notation n'a jamais été proposée dans la proposition originale de la notation AUML.

Avec UML 2 et la nouvelle proposition du groupe de travail derrière AUML, ce n'est plus le cas. Dans les diagrammes il est maintenant parfaitement possible de spécifier la cardinalité des relations existantes entre les entités qui prennent les différents rôles. Tout ceci nous a amené a produire une nouvelle version étendue de nôtre langage et de l'outil qui l'accepte.

3.6.1 Extensions de la notation

Avec les changements importants arrivés dans la notation AUML, il est devenu évident pour nous qu'il faudrait modifier et/ou étendre la notation proposée pour faire face à des cas plus complexes et intéressants que ceux qu'il était possible de représenter auparavant, et aussi que nous aurions besoin de plus de *flexibilité* au moment de spécifier un système de façon visuelle.

La direction à suivre pour spécifier plus convenablement un système multiagent était évident: étendre la notation déjà présente avec l'idée de cardinalité entre les relations de communication. Cette cardinalité est tout simplement, la quantité d'entités qui interviennent dans un processus de communication donné. La figure 3.20 montre comment représenter la cardinalité dans un processus d'interaction reliant des entités appartenant à deux rôles différents en AUML 2, notation qui était absente dans la première version du langage. Il faut noter que si la cardinalité associée à un des rôles présents est égal à un, il devient possible de spécifier une étiquette pour cette entité, mais si la cardinalité est majeur à un, les entités ne peuvent pas être différenciées de cette façon, dans un certain sens elles deviennent *un ensemble*.

Si on prenait en compte la possibilité qu'il y ait plusieurs instances d'un rôle dans une spécification donnée, le langage accepté par l'outil AtoS (lan-

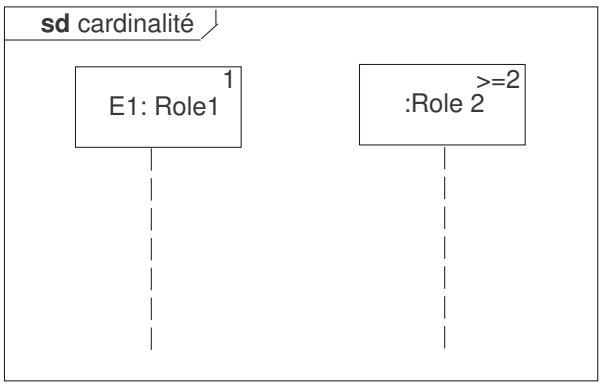


FIG. 3.20 – *Exemple de cardinalité dans une relation de communication en AUML 2*

gage AtoS dorénavant) était incapable de pouvoir représenter ce cas particuliers, car dépourvu d'une notation pour les spécifier.

La raison de cela est que le langage AtoS était conçu de tel façon que chaque entité est spécifiée séparément du reste (chaque entité était modélisée suivant la perspective locale que cette entité avait de l'interaction globale), et que l'analyseur lexique/syntaxique/sémantique du langage AtoS devait faire le lien entre les différentes parties, et former avec le modèle globale du système, qui était représente finalement sous la forme d'une spécification en langage formel. Ce qui était suffisante pour prouver qu'une spécification basée sur un diagramme de rôles pouvait être traduit vers une représentation formelle, mais pas pour générer la multiplicité de cas qu'un diagramme de rôles pouvait générer.

Pour cette raison ont a décidé de changer le langage pour admettre des cas plus complexes d'interaction.

3.6.2 Scénarios

Bien qu'on s'avait rendu compte déjà du fait que les rôles ne sont jamais seuls, on n'avait pas donné assez d'importance au concept d'ensemble de rôles (scénario para analogie) jusqu'à ce qu'on ait besoin d'implémenter la cardinalité dans les relations.

On avait besoin de pouvoir spécifier tout le comportement d'un ensemble de rôles dans une unité sémantique, dont tous les comportements seraient spécifies et où on pourrait spécifier aussi combien d'instances d'un rôle quelconque on pouvait avoir. On a pensé dès le début à une solution semblable à celle proposé par les langages structurés et orienté objet: l'idée d'instances d'un type de donnés abstrait tel quel. Si on considérait que les rôles étaient

des analogues de classes ou des TDA avec des attributs en plus (la séquence de messages), on pouvait imaginer construire une spécification du système avec autant d'instances qu'on le souhaitait, seulement en déclarant plus d'instances de rôle.

Imaginons que nous voulons modéliser le protocole `ContractNet` avec un langage semblable a celui proposé dans la section 3.3.1.2 (voir algorithme 7), mais avec une extension importante: on donne la possibilité de déclarer des *instances de rôle*.

Pour pouvoir y arriver, il dévient nécessaire d'ajouter une notation spéciale pour pouvoir différentier les rôles par nom, les donner une cardinalité et les associer à une au plusieurs instances. Dans l'algorithme 7 on montre un exemple de ce code mentionné auparavant, qui exprime le protocole `ContractNet`. Dans le code on peut voir que les rôles ont un nom explicite (préfixé par le mot `role`).

Dans cette autre notation on exprime avec un `(1)` juste après le nom du rôle, le fait qu'un rôle est présent q'une seule fois dans le processus d'interaction modélisé. De façon analogue, on utilise `(+)` pour exprimer le fait qu'un rôle donné est présent un nombre indéterminé de fois, mais avec au moins une instance. La notation en-bas nous permet d'exprimer combien d'instances d'un certain rôle nous voulons réellement avoir. Cette notation est prise complètement de celle déjà présente dans la majorité des langages de programmation structurés/imperatifs.

Avec ceci, on peut imaginer un schéma de traduction vers une notation formelle comme Promela/SPIN, qui nous permettrai d'avoir multiples copies du comportement spécifié par un rôle (si nous parlons d'un rôle avec une cardinalité plus grand qu'un), ou bien une seule, dans le cas d'un rôle avec une seule instance.

Cependant, on s'est vite aperçu qu'ajouter une simple étiquette de cardinalité et énoncer les instances possibles d'un rôle dans une interaction, n'était pas suffisant pour pouvoir spécifier correctement un protocole d'interaction: il fallait aussi prendre des provisions sur l'identité des instances du rôle.

Dans le cas du protocole ContractNet, il est facile de voir qu'il y a un point de l'exécution dans lequel un certain agent est privilégie sur les autres (celui qui fait le meilleur offre à l'initiateur). Différence importante car une instance de rôle (c'est à dire, un agent) doit savoir avec quelle instance d'un autre rôle complémentaire est en train d'interagir. Avec une notation complètement orientée aux rôles, dans laquelle il n'y a pas de façon de différentier une instance de rôle d'une autre, on a vu qu'il était *très difficile* (voir impossible) de générer un modèle valable du système.

3.6.3 Langage d'entrée, deuxième version

Pour permettre la définition des rôles en différenciant les phénomènes partagés de ceux qui concernent exclusivement les entités individuelles, on

Algorithm 7 Exemple de code, reprise

```
role initiator (1)(
(init,null,null,null)->
(out, initiator,participant,[null,true,cfp])->
(
(in, participant,initiator,[refuse,true,null])
!
(in, participant,initiator,[not_undestood,true,null])
!
(in, participant(p1),initiator,[propose,true,null])->
(
(out, initiator,p1,[null,true,reject_proposal])
!
(out, initiator,p1,[null,true,accept_proposal])->
((in, p1,initiator,[failure,true,null]) !
(in, p1,initiator,[inform_done,true,null]) !
(in, p1,initiator,[inform_ref,true,null]))
)
)
)


role participant (+)(
(init,null,null,null)->
(in, initiator, participant, [cfp, true, null])->
((out, participant , initiator,[null,true,refuse])
 !
 (out, participant , initiator,[null,true,not_understood])
 !
 (
  (out, participant , initiator,[null,true, propose])->
  (
   (in, initiator, participant, [reject_proposal, true, null])
   !
   (
    (in, initiator, participant, [accept_proposal, true, null])->
    (
     (out, participant , initiator,[null,true,failure]) !
     (out, participant , initiator,[null,true,inform_done]) !
     (out, participant , initiator,[null,true,inform_ref])
    )
   )
  )
 )
)
)

participant (agent1,agent2,agent3)

initiator(agent4)
```

a étendu le langage de spécification qu'on avait crée auparavant avec les propriétés suivantes:

- Rôles comme ensembles d'entités avec un comportement stéréotype.
- Agents comme instances des rôles.
- Une sémantique de la communication rôle-à-rôle, rôle-à-agent et agent-vers-agent.
- Le scénario comme la traduction qui laisse au développeur la possibilité de définir la architecture du modèle.

A continuation, on donne une description de chaque un des points annoncés auparavant.

3.6.4 Rôles comme ensembles d'entités

Depuis le début de ce travail, il étais devenu clair pour nous, que les rôles représentaient des ensembles d'entités avec un comportement commun, et que une notation basée sur rôles dans l'idéal décrirait ce qui fait chaque entité appartenant au rôle donné. Ce qui n'était pas si évident était les conséquences que cet approche a sur la sémantique d'un modèle. Dans le langage que nous avons proposé en première instance, on considérait que chaque comportement était un rôle, mais que chaque rôle avait une seule instance. Ce qui est évidemment un cas particulier des rôles avec un cardinalité multiple, mais aussi et il faut bien le noter, l'équivalent notationnel d'un langage impératif plus ou moins orienté objet, et un plutôt limité dans ses capacités expressives. Donc la prochaine étape qui semblait la plus naturelle, était de considérer que le langage devait de considérer les rôles comme représentant le comportement d'un ensemble d'entités, tel qu'on l'avait mentionné dans la section 3.1.1.

3.6.5 Agents comme instances des rôles

Au moment de faire le saut vers des spécifications plus génériques qui prennent en compte la potentialité de la cardinalité dans les relations de communication, le problème de la sémantique du modèle devient plus complexe, et mérite une analyse plus approfondi. Le premier problème vient du fait que quand on parle des rôles avec une cardinalité, on parle en fait d'une architecture du système où interagissent une ou plusieurs entités appartenant à un même rôle, avec une ou plusieurs entités appartenant aux rôles complémentaires du rôle mentionné auparavant. Qu'est-ce la signification d'un message de la part d'un ensemble d'entités vers un autre ensemble d'entités?

Chapitre 4

Conclusion

Dans ce chapitre nous présentons une synthèse de divers problèmes abordés pendant la durée de cette thèse ainsi que les solutions que nous avons proposées pour les résoudre. Ensuite nous présentons des perspectives de recherche futures. Dans notre cas particulier, une révision plus profonde de la relation de rôle, agent et groupe, rapprochant notre travail plus vers un cadre méthodologique plus mûr tel que Madkit[18].

4.1 Problèmes abordés

Le propos initial de cette thèse était d'étudier les avantages que pouvait avoir la modélisation de protocoles d'interaction de systèmes multiagent, en utilisant une notation conçue spécifiquement pour les protocoles d'interaction, et en s'appuyant sur un ensemble d'outils logiciels. Pour arriver à nos fins, nous avons suivi l'approche qui consiste à prendre directement des diagrammes représentant des protocoles de communication sous la forme d'une notation graphique adaptée pour la représentation de séquences de messages, et finalement générer à l'aide d'outils ad hoc des équivalents formels de ces diagrammes. Ces équivalents formels devaient faire en sorte qu'ils puissent être facilement validés avec un outil dédié spécifiquement à la validation de protocoles de communication.

Pour ce faire on avait besoin de prendre une notation à la fois flexible tout en étant intéressante et facile pour des personnes familiarisés avec la modélisation de protocoles de communication et celle des systèmes modulaires (par exemple, des systèmes orienté objet). Nous avons choisi comme notation graphique celle proposée par James Odell, car dans sa version 1 elle était à la fois proche de UML (notation avec laquelle nous étions bien familiarisés) et suffisamment expressive pour pouvoir représenter un ensemble important de protocoles d'interaction. Nous avons modifié la notation pour rendre le processus de traduction plus facile et nous avons fait quelques essais pour estimer l'applicabilité de la nouvelle notation. Comme notation formelle nous

avons choisi le langage Promela, qui est un de plus utilisés pour modéliser des systèmes parallèles et/ou communicants.Nous l'avons choisie parce qu'il était à la fois bien documenté, avec beaucoup d'exemples d'application et gratuit. Il s'est avéré que Promela était une notation riche et capable de représenter tous les aspects que nous désirions représenter.

On a trouvé que la notation originale d'AUML était améliorable, dans le sens qu'elle définissait des opérateurs avec une fonctionnalité redondante ou sémantiquement confuse (comme le OR) et qu'elle manquait des capacités d'expression pour représenter des besoins récurrents dans les protocoles de communication, tels que la synchronisation par rendez-vous et l'itération.

4.2 Perspectives de recherche

Le processus de traduction défini ne prend pas en compte la différence existante entre les diagrammes de rôles et les instances de rôle. Différence qui est bien présente dans le travaux récents concernant AUML, ainsi que dans les méthodologies qui nous ont servi comme source d'inspiration (principalement GAIA et Madkit). La solution prise dans les outils développés est la plus facile à implémenter, mais aussi la plus limitée: pour chaque rôle défini il y a exactement une entité qui agit comme instance de ce rôle. Ce qui oblige le développeur à construire un nouveau modèle pour chaque instance particulière d'un même scénario avec les mêmes rôles.

On considère que cela n'est pas du tout nécessaire, car un processus de traduction qui prend en compte la relation entre rôles et instances, peut générer tous les exemples nécessaires de façon automatique. Nous croyons que la traduction faite par un développeur humain chaque fois qu'il prend un diagramme de rôles et le traduit vers une spécification avec des instances de rôle, peut être automatisée.

On a prévu pour cela des notations plus expressives et des schémas de traduction plus perfectionnés, qui permettraient éventuellement d'avoir une vraie dynamique de rôles dans la spécification. En effet, les rôles ont été identifiés par certains (ainsi que par nous-mêmes) comme des schémas du comportement qui pourraient habiliter l'idée de réutilisation dans le contexte de systèmes multiagent. Si on accepte que les rôles sont des comportements stéréotypés qui peuvent être pris ou abandonnés par les entités qui constituent un système fait d'agents de façon dynamique, il est envisageable de créer des environnements de développement qui mettent en application effectivement cette instantiation de rôle, et pas seulement lors de la compilation, mais d'une façon dynamique. Cela pourrait être aperçu comme un signe d'intelligence, si l'on considère que la capacité de prendre un rôle différent dans une organisation faite d'agents est un attribut propre d'un agent intelligent.

Une autre perspective qui nous est apparue comme évidente est de positionner le travail présent (ou plutôt une extension) dans le contexte d'un

outil capable de représenter la dynamique des rôles. Nos discussions nous ont amené spécialement vers Madkit, qui est à notre connaissance l'outil qui a les capacités les plus intéressants en ce qui concerne la définition et l'implémentation de rôles. Il est envisageable de proposer des extensions fonctionnelles pour des environnements de développement de ce genre, qui permettrait d'effectuer la validation de l'interaction d'une instance particulière d'un des scénarios définit par les rôles.

Annexes

Chapitre A

Dévéloppement

Le fichier d'entrée en XML a pour but à la fois de permettre la traduction du diagramme vers PROMELA/SPIN et de reconstruire le diagramme AUML fait par l'utilisateur. Il y a donc une partie *commune*, la partie qui permet de traduire en PROMELA/SPIN (à peu de chose près) et un partie rajoutée qui permettra de repositionner les différents éléments du diagramme.

Partie commune

Nous avons dans un premier temps définit la structure du fichier XML dont la un schéma représentatif est montré dans la figure A.1.

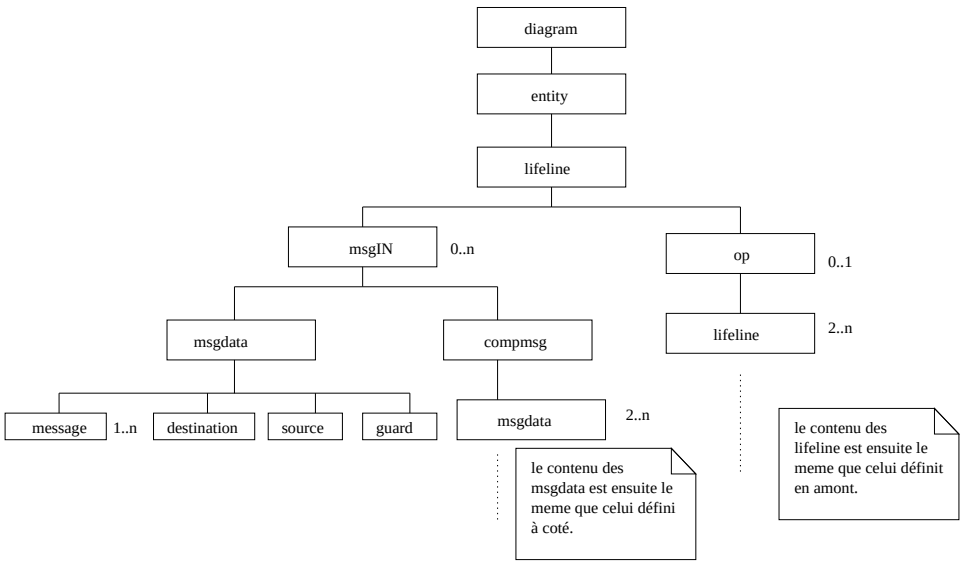


FIG. A.1 – *Diagramme de la structure du fichier XML*

Ce schéma ne fait que définir les éléments contenus par chaque éléments. Les attributs sont définis ci-dessous:

1. diagram : nom -> chaîne de caractère
2. entity : nom -> chaîne de caractère
3. lifeline : rien
4. msg : msgtype (c'est à dire message entrant ou sortant) -> énumération
5. msgdata : rien
6. compmsg : optype (type d'opérateur, OR, AND, XOR) -> énumération
7. op : type (type d'opérateur aussi) -> énumération
8. message, destination, source, guard n'ont pas d'attributs.

Avec le schéma qui donne la hiérarchie et les attributs donnés au-dessus, la DTD est complètement définie.

Son implémentation est défini dans la figure A.2

```
<!ELEMENT diagram (entity*)>
<!ATTLIST diagram name CDATA #REQUIRED>

<!ELEMENT entity (lifeline)>
<!ATTLIST entity name CDATA #REQUIRED>

<!ELEMENT lifeline (msg*, op?)>

<!ELEMENT msg (msgdata | compmsg)>
<!ATTLIST msg msgtype (OUT | IN) #REQUIRED>

<!ELEMENT msgdata (message, destination, source, guard)>

<!ELEMENT compmsg (msgdata, msgdata+)>
<!ATTLIST compmsg optype (OR | AND | XOR) #REQUIRED>

<!ELEMENT op (lifeline, lifeline+)>
<!ATTLIST op type (OR | AND | XOR) #REQUIRED>

<!ELEMENT message (#PCDATA)>
<!ELEMENT destination (#PCDATA)>
<!ELEMENT source (#PCDATA)>
<!ELEMENT guard (#PCDATA)>
```

FIG. A.2 –

Interface visuelle

Ce document avait pour objectif au départ de préparer la réalisation, maintenant sa mise à jour (très nécessaire) va permettre de faciliter la compréhension du code. Le document de spécification ainsi qu'une connaissance des diagrammes AUML sont chaudement recommandés pour une meilleure compréhension du présent document.

Les outils utilisés sont le SDK 1.3.1 de Sun pour le codage pur et l'API SAX 2.0 avec une partie de Xerces de la Fondation Apache pour la partie chargement d'un diagramme de séquence AUML.

Architecture générale

Architecture très générale de l'interface

Elle correspond au diagramme UML suivant :

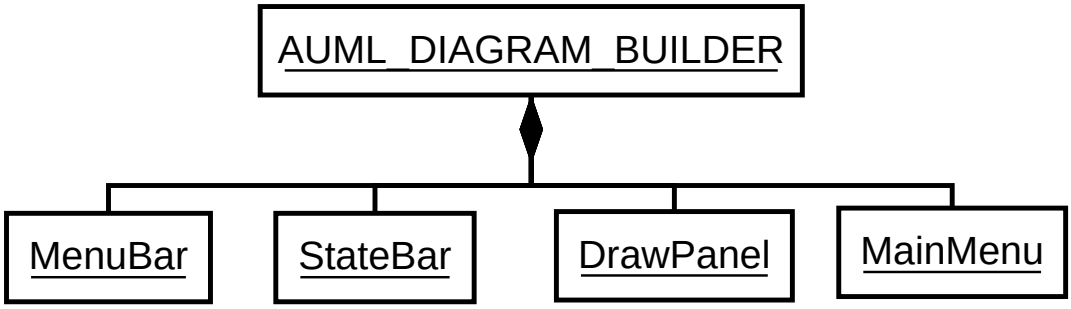


FIG. A.3 – *Architecture très générale de l'interface*

L'AUML_DIAGRAM_BUILDER est la frame de l'interface, les autres éléments sont tous contenus dans celle-ci. Le MenuBar est le menu contenant les sous-menus File, Ivan et Help. La StateBar est la barre permettant à l'utilisateur de comprendre où il se trouve et ce qu'il doit faire. Le DrawPanel contient tout les éléments du diagramme, aussi bien graphiquement, c'est un panel, que structurellement par l'intermédiaire d'un tableau d'entités. Le MainMenu contient tous les boutons qui permettent de construire et dessiner les éléments sur le plan de travail.

Architecture des données du diagramme

On reprend la structure du document XML que l'on retranscrit, quasiment à l'identique, en modèle objet. Le résultat pourra servir de base pour concevoir une architecture plus précise et complète.

En fait on peut surtout se servir de ce schéma pour concevoir la structure de donnée associée. On peut greffer ensuite les attributs graphiques sur ces

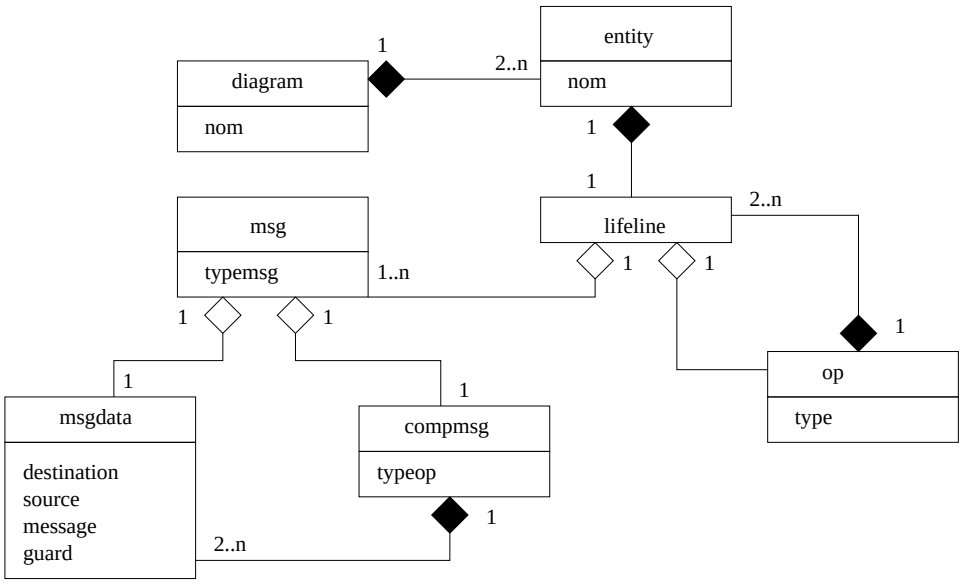


FIG. A.4 – *Architecture de la structure de donnée*

classes. Les attributs présents ne sont là que pour montrer l'emplacement des informations minimales à la complète compréhension de l'emplacement des données servant à la construction du fichier xml voulu.

La collection utilisée pour le stockage des messages sera une *HashMap* car elle permet d'utiliser une clé pour retrouver un élément. La clé utilisée sera la coordonnée y du point origine du message. Pour les autres collections le *Vector* suffit amplement. Dans les lifeline les messages seront stockés dans 2 collections différentes, l'une pour les messages sortant l'autre pour les entrants. Sur la figure-2, le message peut être soit entrant ou sortant.

Hierarchie des éléments de diagramme

Une fois que l'on a l'organisation, il reste à voir comment construire et hiérarchiser des objets pouvant refléter l'architecture. C'est l'objectif de la figure-A.5.

Remarques :

1. Les attributs héritées ne sont pas réécrits dans les classes filles.
2. Seules les attributs impliqués dans l'interprétation du diagramme (pour le futur fichier xml) sont représentés ici.
3. L'attribut *isOut* de *MsgSimple* est bloqué à VRAI car cela n'apporte rien de lui donner la possibilité d'être entrant, puisqu'on peut déjà obtenir n'importe quel message avec un message sortant. Je m'explique : si on veut faire entrer un message dans une ligne de vie il suffit de le

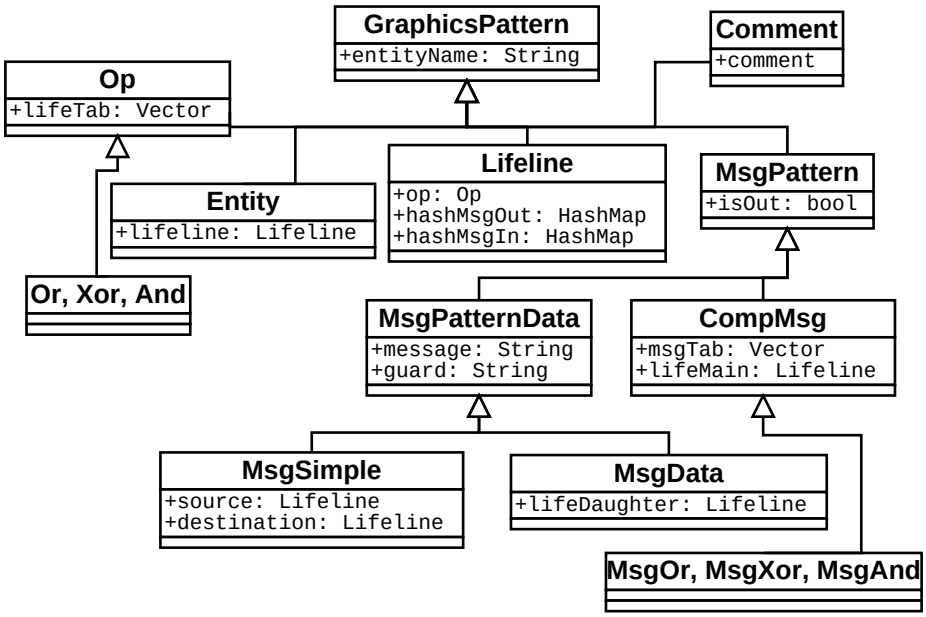


FIG. A.5 – *Organisation des éléments du diagramme AUML*

faire sortir de la ligne de vie source... Alors que pour les messages composés ce n'est pas si simple lorsqu'on veut que la branche principale soit entrante.

Sur ce schéma, on voit bien que les messages forment une famille à part, comme sur le schéma de la figure-A.4. Les messages simples (*MsgSimple*) se trouvent au même niveau que les données de messages composés (*MsgData*) comme dans la figure-A.4 et dérivent logiquement d'une même classe (*MsgPatternData*) du fait d'un contenu et d'un comportement proche.

Remarque : *MsgData* ne peut avoir comme *MsgSimple* une source et une destination puisque *isOut* n'est pas bloqué à VRAI. On a donc l'attribut *lifeDaughter* qui se retrouve soit source soit destination suivant la valeur de *isOut*. Même chose pour *lifeMain* de *CompMsg*.

Pour ce qui est de l'entité on retrouve la structure de plus haut avec la ligne de vie en tant qu'attribut. La ligne de vie contient un opérateur (évidemment seulement si un opérateur y est accroché). Cet opérateur contient lui-même un tableau de ligne de vie (*lifeTab*) et la boucle est bouclée.

On alors les messages qui se greffent sur les lignes de vie avec une *HashMap* pour les entrants et une autre pour les sortants, un *MsgData* étant considéré comme un message à part entière par la *HashMap* de la ligne de vie. Pour ceux qui n'ont pas tout suivi (moi-même je ne me suis pas toujours je sais donc que ça peut arriver), chaque message est contenu à la fois dans deux lignes de vie différentes, la source et la destination (en gros, parce que pour les messages composés, ce n'est pas si simple, le message n'est pas

le même à la source(arrivée si entrant) et à l'arrivée(source) vue qu'il y a plusieurs arrivées(sources) pour une source (arrivée)).

Mode de fonctionnement des objets graphiques et de l'espace de travail

L'espace de travail ou plus exactement son objet *Graphics* accueillera toute la représentation graphique du diagramme. Un second objet *Graphics* est utilisé pour faire buffer, c'est celui-ci qui recevra par chacun des éléments le dessin de ces derniers. Le *Graphics* buffer est ensuite affecté à celui de l'espace de travail (drawPanel). A chaque élément ajouté, l'espace de travail effacera puis redessinera tout le diagramme. Ce sera la méthode *PaintComponent,* ou une méthode équivalente qui reconstruira le diagramme en parcourant la structure où tous les objets sont stockés. Chaque objet stockera donc son type, sa méthode de dessin, ses coordonnées, ses coordonnées "actives" (points rouges), ainsi que ses coordonnées de suppression.

Conception

On peut diviser cette conception en 3, voire 4 parties importantes : la première concerne la construction du diagramme par l'utilisateur, la deuxième concerne la partie édition de ce diagramme, à priori pour l'instant il n'y a que la suppression mais un zoom pourrait être ajouté par exemple, la troisième concerne la sauvegarde et la récupération du diagramme et la dernière serait l'aide... qui sera à priori minimale!

Classes impliquées dans la construction du diagramme par l'utilisateur

Base de l'interface graphique

Le premier diagramme de classe (figure-A.6) représente une vue plus proche de la réalité du code de la frame principale, le contenu de la figure-A.3 étant plus abstrait.

DrawPanel et Content sont en fait tous les deux attributs de FramePrinc, mais ce qu'ils ont en plus des attributs déjà présents dans FramePrinc est le fait que ce sont des classes utilisateur et non de l'API Java.

Pour les attributs de FramePrinc, on ne voit qu'un seul attribut représentant les boutons et leur actions, mais en fait il y en a peu près une dizaine et tous les représenter aurait alourdi le schéma.

Voici la liste de ces boutons :

1. Créations : Entité, opérateurs OR, XOR, AND pour diviser les lignes de vie, message simple, messages composés OR, XOR, AND, commentaire.

2. Modifications : Activation des lignes de vie, activation des messages entrant et sortant, commande de suppression.

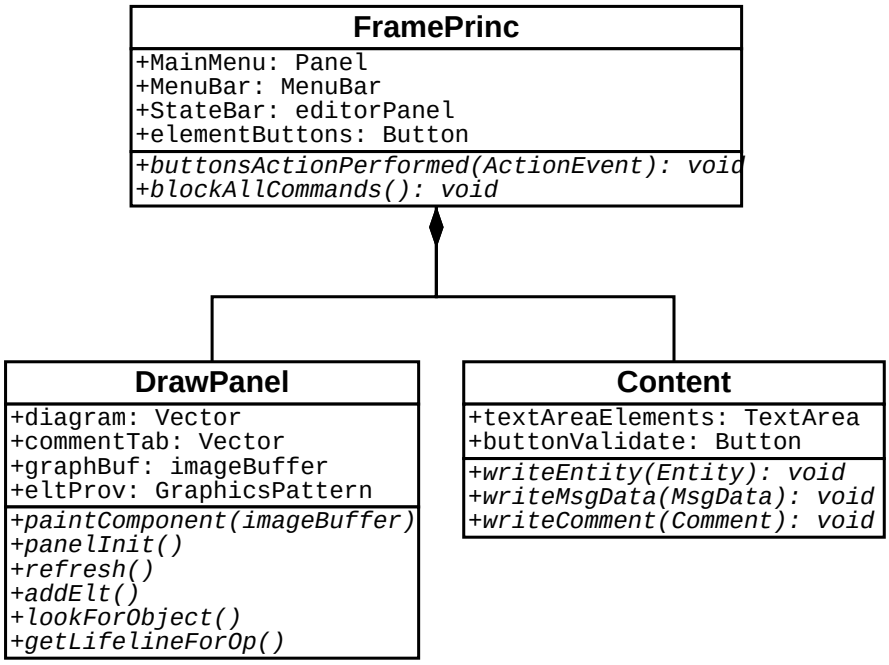


FIG. A.6 –

Principe de l'appel et de la construction des classes par le DrawPanel : C'est l'écouteur du bouton de construction d'objet qui construit l'objet et l'envoie au drawPanel avec *addElt()*. Le drawPanel donne le buffer d'image en paramètre et active les mouseListeners de l'objet qui sera construit par ses méthodes internes. L'objet gère donc complètement sa propre construction. Ce sont les écouteurs de souris qui demandent ensuite le rafraîchissement du drawPanel, *refresh()* (qui efface tout le contenu du drawPanel) accompagné de *repaint()* qui appelle *paintComponent()* (qui reconstruit tout le diagramme) automatiquement. Une fois la construction de l'objet fini, celui-ci est ajouté à la collection d'entités (qui contiennent tous les autres objets) du drawPanel avec toutes les informations nécessaires à sa reconstruction.

Détails des classes gérant les éléments du diagramme

Les *MouseListeners* : Tous les éléments du diagramme gèrent leur construction, sauf les lignes de vies qui sont construites par les autre éléments, par l'intermédiaire d'une classe interne *MouseListener*Object implémentes les écouteurs de souris de java. DrawPanel possède aussi une classe interne de

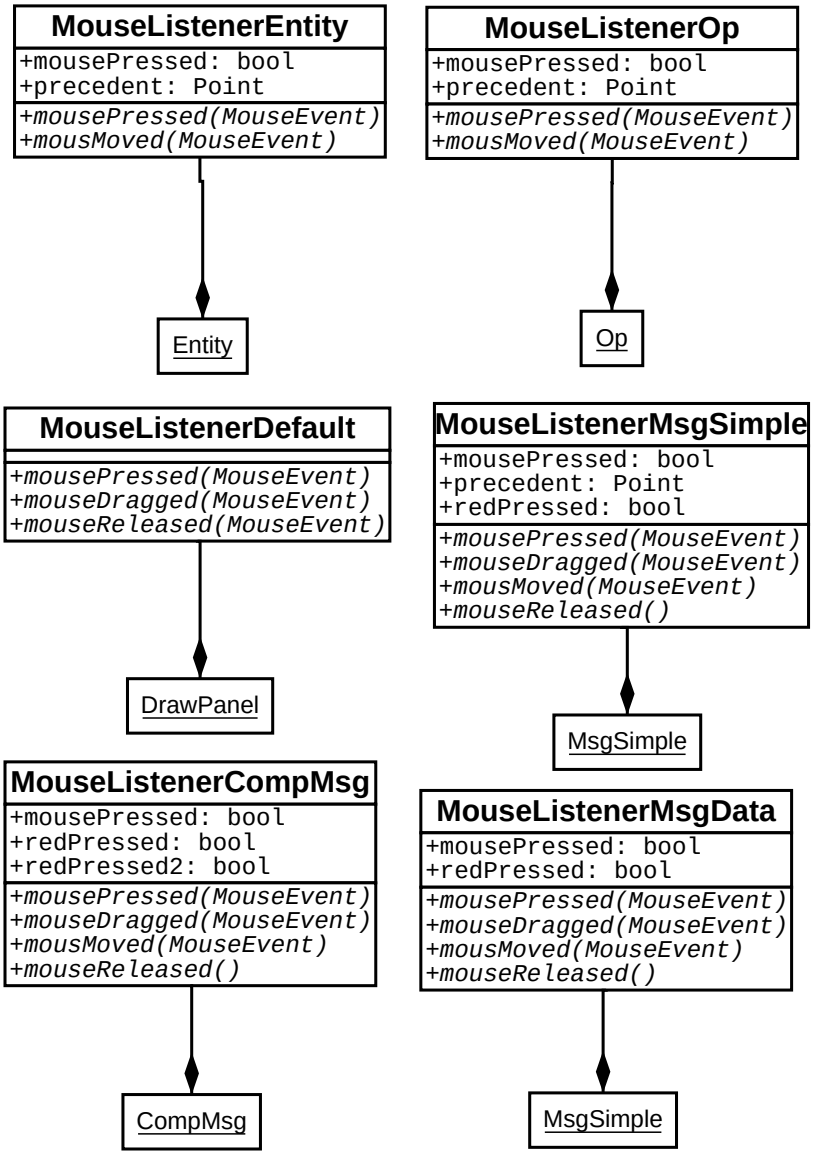


FIG. A.7 – *MouseListeners des éléments*

ce modèle pour gérer les événements souris par défaut lorsqu'aucun élément est en construction. Par rapport la figure-A.7, il y a quelques remarques à faire si on veut espérer comprendre le sens attributs et ce qu'il y a dans les méthodes.

Pour toutes les classes, *mouseMoved* appelle une fonction qui dessine la représentation de l'élément à l'endroit où se trouve le curseur. Il y a également la variable *MousePressed* qui est présente dans toutes les classes. Ce booléen devient vrai une fois que le bouton gauche a été pressé une fois, cela évite de construire plusieurs éléments avant que le premier n'ai été fini. Cela évite aussi que le dessin de la figure continue à suivre le curseur. L'attribut *precedent* est également souvent présent, il sert lorsque l'emplacement de l'élément n'a pas encore été choisi et que la figure "suit" le curseur; *precedent* est alors l'origine du dessin provisoire dessiné à l'émplacement précédent de la souris. Ce point sert à n'effacer et à redessiner que la partie du dessin qui a été modifié par la présence de la figure. Ca n'a pas vraiment d'utilité en fait puisque tout effacer et tout redessiner est aussi rapide, c'est pour cette raison que la 2ème solution a été adoptée par la suite. *redPressed* représente, pour *MsgSimple* et *MsgData,* le fait que l'utilisateur a cliqué sur l'extrémité rouge du message en construction.

Pour ce qui est des méthodes, celles-ci appellent d'autres méthodes qui sont dans la classe parente, par exemple à chaque *mousePressed* correspond un *actionMousePressed*.

1. Pour l'entité, presser le bouton de la souris détermine l'origine de l'entité et appelle la méthode qui va activer la saisie du nom de l'entité dans le contentPanel.
2. Pour l'opérateur, c'est la même chose sauf que si une ligne de vie ayant l'extrémité libre ne se trouve pas proche du clic, le clic a une action d'annulation de la construction de l'élément et l'objet est supprimé.
3. Pour le message simple, le premier clic joue le même rôle que pour Op. Le second permet de choisir le côté vers lequel le message est dirigé. Le *mouseDragged* permet, une fois le côté choisi (contrôlé par un booléen) , de tirer le bout de message jusqu'à une ligne de vie, donc de dessiner l'étirement du message. *mouseReleased* vérifie si une ligne de vie est proche et si oui déclare le message fait et active la saisie du contenu sinon le bout rouge est toujours actif et *mouseDragged* est toujours valide.
4. Pour *CompMsg*, c'est la même chose sauf qu'au lieu d'un bout rouge c'est autant de bouts que de messages demandés. Il faut aussi noter un différence de gestion des événements, ce sont les petits messages (MsgData) qui se gèrent eux-mêmes. C'est à dire CompMsg choisit le côté du message puis ce sont les *MouseListeners* des MsgDatas qui gèrent l'étirement et la détection d'un ligne de vie.

Lorsque le dessin de la figure est fini, il faut toujours penser à enlever les *MouseListeners* du DrawPanel, sinon il peut arriver des évènements quelque peu surnaturels...

Les classes des éléments : Pour gérer la construction et les modifications (qui seront vues dans la partie suivante), il faut un certain nombre de méthodes et d'attributs, ceux-ci sont présentés dans la figure-A.8. Seules les fonctions non évidentes ou nécessaire à la compréhension de la conception sont présentes. Cela pour ne pas trop surcharger le schémas et les explications... Une autre remarque peut être faite sur cette figure, les classes de spécialisation d'Op (Or, Xor, And) ne sont pas représentées ainsi que celles de CompMsg(MsgOr, MsgXor, MsgAnd), ceci est dû au fait qu'il n'y a quasiment rien dans ces classes presque tout étant dans la classe parente.

Méthodes et attributs de *GraphicsPattern* :

- *origine* : C'est le point d'origine de l'élément, il sert pour le dessin de la figure.
- *delSurface* : Ce vecteur contient les informations nécessaires pour la suppression de l'élément : origine et forme de la surface de suppression.
- *entityName* : C'est le nom de l'entité à laquelle l'élément appartient et le nom de lui-même pour l'objet entité.
- *graphBuf* : C'est le contexte graphique de l'image buffer, il permet de dessiner sur cette image buffer. La référence de ce contexte graphique est la même pour tous les éléments, même le *DrawPanel,* ce qui permet de construire le diagramme entier sur le contexte avant de l'afficher.
- *drawPanel* : C'est l'espace de travail, sa présence est nécessaire dans les attributs pour pouvoir ajouter les *mouseListener* et demander le rafraîchissement de l'espace de travail.
- *changeLength(Point)* : Son utilité change selon l'élément qui l'implémente. La raison de sa présence dans *GraphicsPattern* est que *DrawPanel* s'en sert dans son *MouseListenerDefault* pour changer ce qui est éditable dans le diagramme par défaut, suivant l'objet cliqué ou "draggé", changeLength() déplace ou étire l'objet en question. Les fonctions du *MouseListenerDefaut* seront vues dans la partie de l'édition.
- *draw()* : Cette méthode est implémentée dans chaque élément, elle dessine l'élément dans le *graphBuf*.
- *drawMove(Point)* : Elle dessine l'élément en mouvement en prenant comme origine le point passé en paramètre.
- *belongsToDelSurface(Point)* : Elle renvoie vrai si le point appartient au vecteur delSurface.
- *reloadDelSurface()* : Elle rafraîchit le vecteur delSurface par exemple après un déplacement.

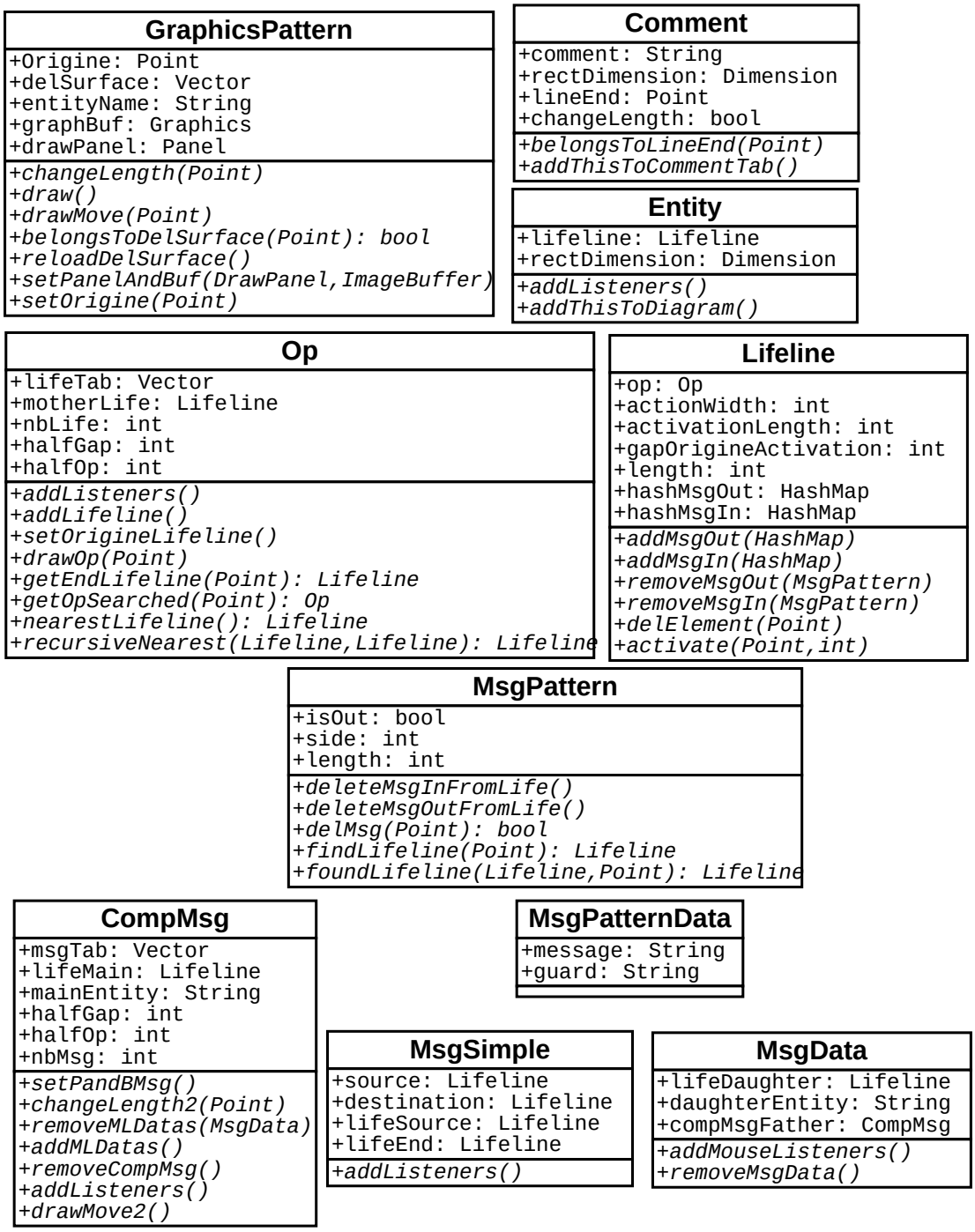


FIG. A.8 – *Les éléments avec leurs méthodes et attributs*

- *setPanelAndBuf(DrawPanel, Graphics)* : Grâce à cette fonction, l'élément possède dans ses attributs une référence sur le *DrawPanel* et l'objet *Graphics* du buffer image.
- *addListeners()* :
- *setOrigine()* : Rôle évident mais indispensable. Un changement d'origine implique souvent celui de beaucoup d'autres éléments, l'implémentation n'est donc pas toujours évidente.

Méthodes et attributs d'*Entity* :

- *lifeline* : C'est la ligne de vie directement accrochée à l'entité. Cela fait partie de la structure de donnée.
- *rectDimension* : Ce sont les dimensions du rectangle représentant l'entité.
- *addListeners* : Ajoute le *MouseListenerEntity* au drawPanel.
- *addThisToDiagram* : Ajoute l'entité au *diagram* du *DrawPanel*.

Méthodes et attributs d'*Op* :

- *lifeTab* : Ce vecteur contient toutes les lignes de vie qui sortent de l'opérateur. Cela fait partie de la structure de donnée.
- *motherLife* : C'est la ligne de vie qui est au-dessus de l'opérateur.
- *nbLife* : Nombre de ligne de vie de lifeTab.
- *halfGap* : C'est le demi-espace entre 2 lignes de vie. Lorsque la largeur est modifiée, c'est cet attribut qui est modifié.
- *halfOp* : C'est la demi-hauteur de l'opérateur, cet attribut permet de trouver le milieu de l'opérateur.
- *addLifeline(Lifeline)* : Cette méthode ne sert que pour le chargement d'après un fichier, sinon les lignes de vie sont créées et ajoutées automatiquement par le constructeur.
- *setOrigineLifeline()* : Met à jour l'origine des lignes de vie du tableau.
- *drawOp()* : Cette méthode permet de ne spécialiser que le dessin de l'opérateur en lui-même (le losange, le losange avec une croix ou le trait épais) et de partager le dessin des branches et des lignes de vie. En fait c'est la seule méthode implémentée par les classes filles d'Op, ce qui permet une duplication minimale du code.
- *getEndLifeline(Point)* : Cette méthode cherche une ligne de vie dont l'extrémité est proche du point passé en paramètre. La méthode cherche dans tous les éléments en aval de l'opérateur.
- *getOpSearched(Point)* : Cette méthode cherche un opérateur dont une extrémité rouge sensible est proche du point en paramètre.
- *nearestLifeline()* : Cette méthode, importante pour l'affichage du diagramme mais inutile sinon, permet de trouver la ligne de vie (d'une

autre entité) la plus proche de l'opérateur selon l'axe des x. Cela permet de ne pas faire chevaucher les lignes de vie, ce qui permet d'éviter des bugs d'affichage qui peuvent être très gênant.
- *recursiveNearest()* : Elle complète nearestLifeline() pour pouvoir parcourir récursivement tout le diagramme.

Méthodes et attributs de *Lifeline* :
- *op* : C'est l'opérateur accroché au bout de la ligne de vie, s'il n'y en a pas c'est le pointeur *null*. Cela fait partie de la structure de donnée.
- *actionWidth* : C'est la largeur de la bande d'activation.
- *activationLength* : C'est la longueur de la bande d'activation.
- *gapOrigineActivation* : C'est l'espace qui sépare l'origine de la ligne de vie du début de l'activation.
- *length* : C'est la longueur de la ligne de vie.
- *hashMsgOut* : C'est la collection qui renferme tous les messages sortant.
- *hashMsgIn* : C'est la collection qui renferme tous les messages entrant.
- *delElement(Point)* : C'est le complément de la méthode *mousePressedAction()* de *Delete*. Elle fait le travail de parcours des collections et de suppression de l'élément concerné.
- *activate(Point, int)* : Ajuste les valeurs des attributs *activationLength* et *gapOrigineActivation*. Le point est l'origine de la sélection et l'entier la longueur.

Méthodes et attributs de *MsgPattern* :
- *isOut* : Ce booléen permet de savoir si le message composé ou un MsgData est sortant ou non. Un message composé est sortant si sa branche principale est sortante. Un MsgData est donc sortant quand sa flèche est dirigée vers la ligne de vie. Un message simple est obligatoirement sortant (en fait il faut obligatoirement en choisir un et c'est sortant qui a été choisi).
- *side* : C'est un entier qui est soit 1 soit -1 (1 à droite et -1 à gauche), il permet de savoir vers quel côté est dirigé un message.
- *length* : C'est la longueur du message. Pour le CompMsg il s'agit de la branche principale.
- *deleteMsgIn(Out)FromLife()* : Enlève le message concerné de la ligne de vie où il est entrant (sortant).
- *delMsg(Point)* : La méthode vérifie d'abord que le message est concerné par le point en paramètre (*belongsToDelSurface()*) puis agit différemment suivant le type de message. Pour le message simple et le message composé, la méthode le supprime de ses lignes de vie où il est entrant et sortant. Pour le *MsgData*, la méthode ne supprime que le message

concerné, c'est à dire que s'il reste, après la suppression, au moins 2 messages au message composé, le message composé reste.
- *findLifeline(Point)* : Cette méthode permet de trouver la ligne de vie activée (s'il y en a une) qui est proche du point en paramètre.
- *foundLifeline(Lifeline, Point)* : Elle complète *findLifeline* pour pouvoir parcourir récursivement toute la collection.

Méthodes et attributs de *CompMsg* :
- *msgTab* : C'est le vecteur qui contient tout les *MsgData*. Cela fait partie de la structure de donnée.
- *lifeMain* : C'est la ligne de vie à laquelle est connectée la branche principale du *CompMsg*.
- *mainEntity* : C'est le nom de l'entité à laquelle appartient la branche principale du message.
- *halfGap* : C'est le demi-espace entre 2 messages. Lorsque la largeur est modifiée, c'est cet attribut qui est modifié.
- *halfOp* : C'est la demi-hauteur de l'opérateur, cet attribut permet de trouver le milieu de l'opérateur.
- *nbMsg* : C'est le nombre de message.
- *setPAndBMsg()* : Cette fonction joue le même rôle que *setPanelAndBuf()*, mais pour tous les messages du *CompMsg*.
- *changeLength2(Point)* : Cette méthode change l'attribut *halfGap*, donc la largeur de l'opérateur.
- *removeMLDatas(MsgData)* : Cette méthode enlève tous les *mouseListeners* des messages de *msgTab*.
- *addMLDatas()* : Cette méthode remet tous les *mouseListeners* des messages du vecteur.
- *drawMove2()* : Cette méthode dessine le message en cours de construction lorsque le côté n'a pas encore été choisi.

Méthodes et attributs de MsgPatternData :
- *message* : Cette chaîne de caractère correspond au ... message. Cela fait partie de la structure de donnée.
- *guard* : Cette chaîne de caractère correspond à la garde du message. Cela fait partie de la structure de donnée.

Méthodes et attributs de *MsgSimple* :
- *source* : Cette chaîne de caractère correspond à l'entité source du message. Cela fait partie de la structure de donnée.
- *destination* : Cette chaine de caractère correspond à l'entité d'arrivée du message. Cela fait partie de la structure de donnée.

- *lifeSource* : C'est la ligne de vie source du message.
- *lifeEnd* : C'est la ligne de vie d'arrivée du message.

Méthodes et attributs de MsgData :

- *lifeDaugther* : C'est la ligne de vie sur laquelle est connectée le message, elle peut aussi bien être destination que source, c'est pour cette raison qu'on ne peut lui donner comme nom source ou end.
- *daughterEntity* : C'est le nom de l'entité de *lifeDaughter*.
- *compMsgFather* : C'est le *compMsg* auquel appartient le *msgData*.

Diagrammes de séquence

Pour mieux comprendre les liaisons entre les objets, le mieux est de traiter un exemple de construction d'un élément sous la forme de diagrammes de séquence. L'exemple que je vais prendre est la construction d'un message composé OR. Toutes les actions sont incluses dans la constuction du message composé :

1. Création après clic sur l'icône. figure-A.9.
2. Déplacement d'un dessin en même temps que le curseur. figure-A.10.
3. Définition d'une origine avec condition d'activation de la ligne de vie cliquée. figure-A.11.
4. Changement de côté du dessin suivant le côté du curseur. figure-A.12.
5. Définition d'un côté du message. Important : Dans ce diagramme on fait pour la première fois dans la série de diagramme un drawPanel.refresh() - drawPanel.repaint() avec appel des fonctions draw(), dans la suite cette séquence sera simplifiée par un drawPanel.refresh() seul. figure-A.13.
6. Définition de la largeur ou de la longueur du message (optionnel). Pour le diagramme on suppose que l'on veut changer la largeur. Si on veut le diagramme pour la longueur, on enlève le 2 aux fonctions. figure-A.14.
7. Etirement du message élémentaire jusqu'à une ligne de vie activée. figure-A.15.
8. Relâchement du bouton (après l'étirement de 7.). figure-A.16.
9. Définition du message et de la guarde du message élémentaire. figure-A.17.

Remarque : Les trois dernières opérations sont répétées autant de fois qu'il y a de message élémentaire.

Chaque numéro correspond à une action donc à un diagramme de séquence.

L'exemple précédent est ce qu'il y a de plus complexe comme construction d'élément, par exemple pour les entités c'est beaucoup plus simple. Cela

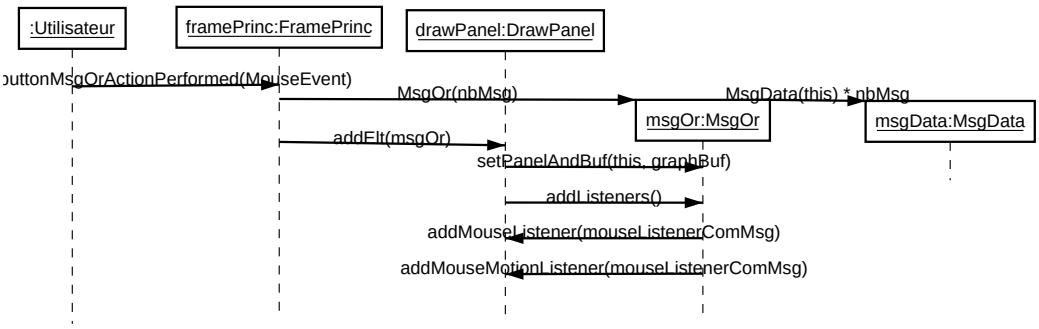


FIG. A.9 – *Séquence de création du message composé*

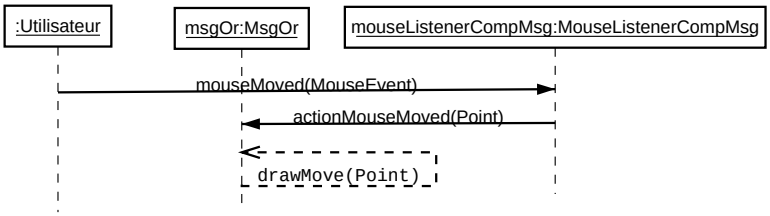


FIG. A.10 – *Séquence du mouvement de souris*

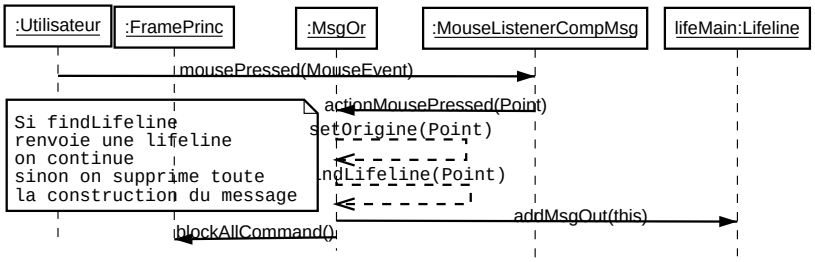


FIG. A.11 – *Séquence de définition de l'origine*

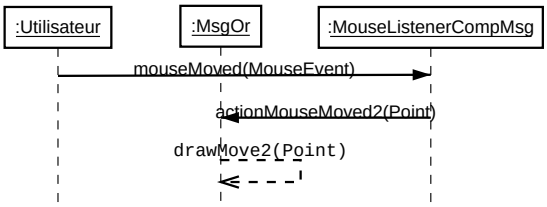


FIG. A.12 – *Séquence de mouvement de la souris avant le choix du côté du message*

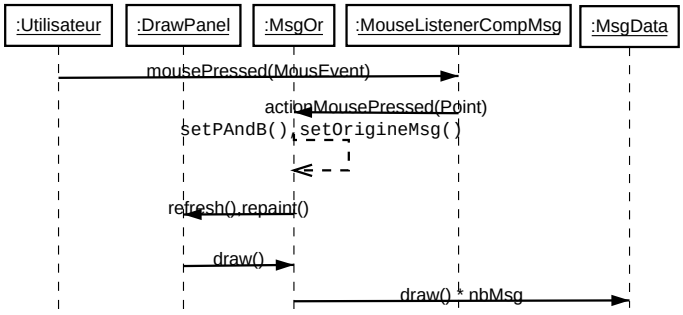


FIG. A.13 – *Séquence du clic pour le choix du côté*

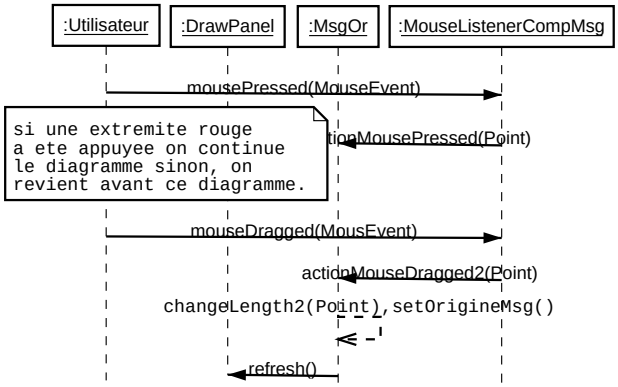


FIG. A.14 – *Séquence pour le choix de la taille du message composé*

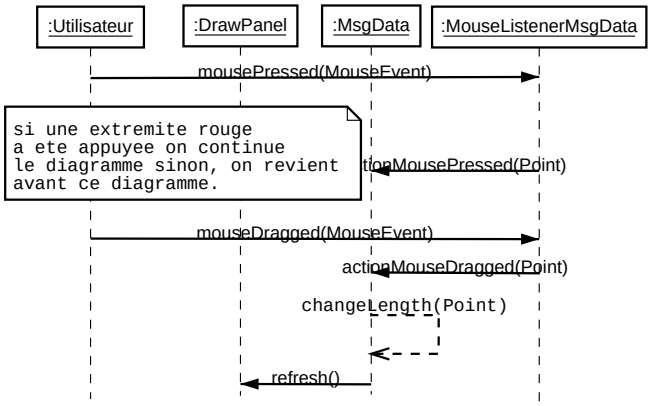


FIG. A.15 – *Séquence d'étirement d'un message élémentaire*

correspondrait à peu de chose près aux étapes 1, 2, 3 et 9. Si on comprend le principe des appels de méthodes de cet exemple, on peut comprendre le principe de construction de tout le diagramme.

Classes impliquées dans l'édition du diagramme

Activate Lifeline

Comme la plupart des classes agissant sur le diagramme, ActivateLife possède son propre *MouseListenerActivate* qui lui permet de détecter la sélection de l'utilisateur.

La classe *ActivateLife* n'hérite d'aucune autre classe, elle est complètement indépendante et ne sert qu'à héberger les méthodes nécessaires à l'activation. Cette classe repose principalement sur les méthodes *findAndActivateLifeline()* qui renvoie un booléen et son alter-ego *foundLifeline(Lifeline)* qui permet la récursivité. Ces méthodes cherchent dans un premier temps la ligne de vie qui pourrait correspondre à la sélection tout en vérifiant que la sélection est valide. Si une ligne de vie est trouvée elle est activée et *foudLifeline(Lifeline)* renvoie *true* ainsi que *findAndActivateLifeline()*.

La ligne de vie est activée grâce à la méthode *activate(Point,int)* de Lifeline, le point étant l'origine de la sélection (en fait seuls l'axe des y compte) et l'entier la hauteur de la sélection. C'est cette méthode qui donne les bonnes valeurs à *gapOrigineActivation* et *activationLength* de la *lifeline*.

Delete

De la même façon que *ActivateLife*, *Delete* est une classe indépendante avec aussi un *MouseListenerDelete*. Le principe est exactement le même que pour *ActivateLife* avec une différence que l'équivalent de *foundLifeline(Lifeline)* est *delElement(Point, GraphicsPattern)* de Lifeline. Son placement dans *Lifeline* correspond à une volonté d'optimisation de la recherche étant donné le rôle central de ligne de vie. Le paramètre *GraphicsPattern* correspond à l'élément en aval de la ligne de vie concerné (soit une entité soit un opérateur), il sert justement à différencier les comportements suivant le fait que ce soit un opérateur ou une entité. Tout le traitement est donc effectuer dans *delElement()*. Le rôle de *delElement()* est de chercher un élément correspondant au point et de supprimer l'élément tout en gardant le diagramme cohérent non seulement graphiquement mais aussi dans sa structure. Le traitement à effectuer étant parfois assez lourd, il était mieux d'éviter les redondances de code. Par exemple lorsqu'un message est supprimé en même temps que la ligne de vie où il arrivait (ou partait), il faut également le supprimer dans la collection de l'autre ligne de vie d'où il partait (ou arrivait).

Classes impliquées dans la sauvegarde du diagramme

Sauvegarde de la structure

Pour ce qui est de la structure et de la DTD, tout est dans le document de spécification joint à celui-ci. Au niveau de la conception de cette sauvegarde, c'est assez simple : Au niveau des entrées-sorties, les classes de l'API sont utilisées au maximum et pour ce qui est des données il suffit de parcourir la structure en écrivant ce qui est demandé par la DTD. Par défaut lorsque le constructeur de *Saver* est appelé par *FramePrinc*, la classe se comporte comme un "saveAs". Si on veut juste "save", il faut appeler la méthode *save()* de l'objet déjà construit. Il n'y a qu'un seul objet *Saver* stocké dans *FramePrinc*.

Sauvegarde du diagramme

Pour la sauvegarde du diagramme le principe est exactement le même à la différence près qu'il y a bien plus d'informations à stocker. La DTD est donc bien différente, celle-ci est représentée dans la figure A.18. L'ensemble de ses données sont nécessaires et suffisantes pour reconstruire un diagramme.

Remarques :

- Des éléments *comment* et *size* ont été ajoutées pour pouvoir sauvegarder les commentaires et la taille de l'espace de travail.
- Lorsqu'on fait un nouveau diagramme, un autre objet *saverDiagram* doit être reconstruit, autrement dit, le "save" est de nouveau équivalent au "saveAs".

Classes impliquées dans le chargement du diagramme

Pour charger le diagramme, l'API SAX 2.0 et le parseur Xerces de Apache sont utilisés. Le choix du fichier, l'initialisation du parseur et le lancement du parsing sont fait dans la classe *LoaderDiagram*. Ensuite c'est la classe *LoaderDefaultHandler* qui est utilisée pour le traitement des données.

La classe LoaderDefaultHandler Cette classe hérite de *DefaultHandler* de l'API SAX, les méthodes *endDocument*(), *startElement()*, *endElement()* et *characters()* sont utilisées. Lors du parsing, le parseur exécute tout seul ces méthodes avec comme paramètre les éléments lus dans le fichier xml. *startElement()* permet de savoir quand commence un élément, par exemple l'élément *entity,* et *endElement()* quand il finit, ainsi on peut savoir ce que contient cette entité. A chacun des éléments correspond une méthode *addElement()* qui permet de construire et d'ajouter l'élément au diagramme. Plusieur booléens permettent de garder en mémoire le fait que tel ou tel élément est le conteneur. Par exemple lorsque l'on a un élément *msgdata*, il

faut savoir s'il est inclus dans un message composé, un booléen garde donc en mémoire le fait qu'une balise *compmsg* est en amont.

Remarques sur la réalisation

Ce que le logiciel ne peut pas faire (dans sa dernière version)

- On ne peut pas construire de message composé tel qu'une ou plusieurs branches partent dans un sens (droite ou gauche) et une ou plusieurs branches partent dans l'autre sens (gauche ou droite). C'est une restriction par rapport aux spécifications d'un diagramme de séquence AUML.

Bugs restant

- Il y a un bug gênant dans la suppression, non pas graphiquement mais dans la structure : lorsqu'on supprime une ligne de vie ayant un operateur avec des lignes de vie ayant des messages accrochés, les messages restent dans la collection de l'autre ligne de vie, celle qui est jointe par les messages. Cela occasionne un erreur dans la sauvegarde. Il faudrait passer un peu de temps (que je n'ai pas) sur le code pour faire une méthode qui s'occuperait des messages en aval de l'objet supprimé.
- Normalement le non-chevauchement de ligne de vie empêche les bugs de message, mais si un message va d'une ligne de vie à une autre ligne de vie de la même entité, l'affichage des messages n'est plus protégé par l'anti-chevauchement des lignes de vie (qui n'est valable que pour les lignes de vie des autres entités).

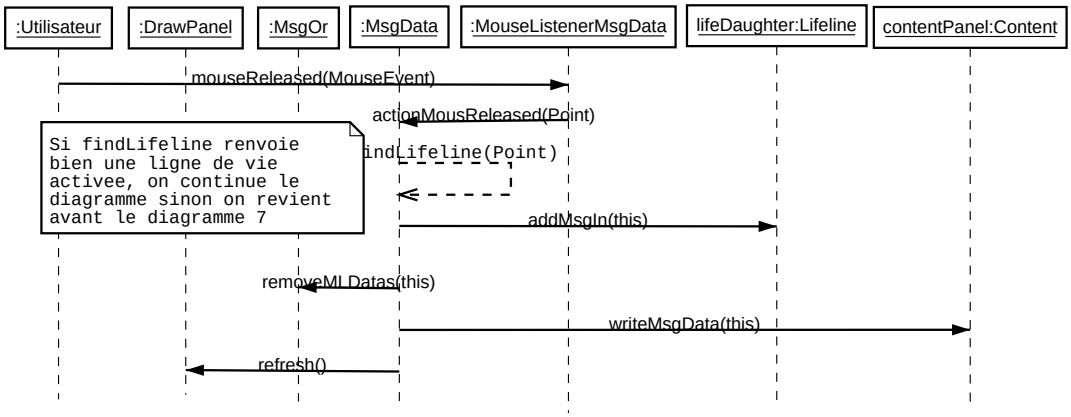


FIG. A.16 – *Séquence du relâchement du du bouton après l'étirement*

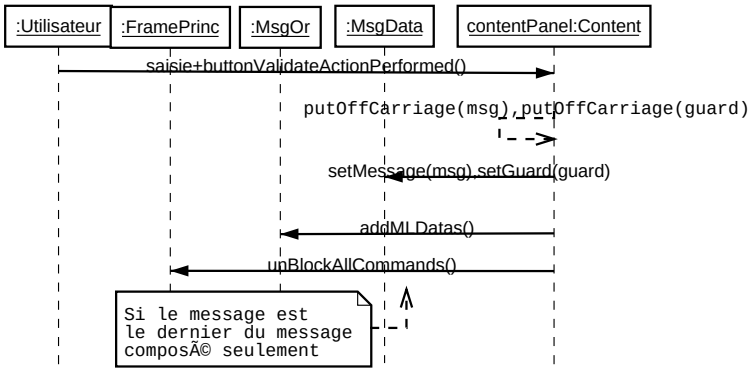


FIG. A.17 – *Séquence de la saisie du message et de la garde*

```
<!ELEMENT diagram  (entity*,comment*,size)>
<!ATTLIST diagram  name CDATA #REQUIRED>

<!ELEMENT entity   (lifeline)>
<!ATTLIST entity   name CDATA #REQUIRED>
<!ATTLIST entity   origine CDATA #REQUIRED>
<!-- origine pattern: "(x,y)" -->

<!ELEMENT lifeline (msg*, op?)>
<!ATTLIST lifeline  origine CDATA #REQUIRED>
<!ATTLIST lifeline  length CDATA #REQUIRED>
<!-- length pattern: "x" -->
<!ATTLIST lifeline  gapOrigineActivation CDATA #IMPLIED>
<!ATTLIST lifeline  activationLength CDATA #IMPLIED>

<!ELEMENT msg      (msgdata | compmsg)>
<!ATTLIST msg      msgtype (OUT | IN) #REQUIRED>
<!ATTLIST msg      side (-1 | 1) #REQUIRED>


<!ELEMENT msgdata  (message, destination, source, guard)>
<!ATTLIST msgdata  ismsgsimple (true|false) #REQUIRED>
<!ATTLIST msgdata  origine CDATA #REQUIRED>
<!ATTLIST msgdata  length CDATA #REQUIRED>

<!ELEMENT message (#PCDATA)>
<!ELEMENT destination (#PCDATA)>
<!ELEMENT source (#PCDATA)>
<!ELEMENT guard (#PCDATA)>

<!ELEMENT compmsg  (msgdata, msgdata+)>
<!ATTLIST compmsg  optype (OR | AND | XOR) #REQUIRED>
<!ATTLIST compmsg  origine CDATA #REQUIRED>
<!ATTLIST compmsg  halfGap CDATA #REQUIRED>
<!ATTLIST compmsg  length CDATA #REQUIRED>
<!ATTLIST compmsg  nbMsg CDATA #REQUIRED>


<!ELEMENT op       (lifeline, lifeline+)>
<!ATTLIST op       type (OR | AND | XOR) #REQUIRED>
<!ATTLIST op       origine CDATA #REQUIRED>
<!ATTLIST op       halfGap CDATA #REQUIRED>
<!ATTLIST op       nbLife CDATA #REQUIRED>

<!ELEMENT comment  (#PCDATA)>
<!ATTLIST comment  origine CDATA #REQUIRED>
<!ATTLIST comment  comment CDATA #REQUIRED>
<!ATTLIST comment  lineend CDATA #REQUIRED>

<!ELEMENT size  (#PCDATA)>
```

FIG. A.18 –

Bibliographie

[1] Information Sciences Institute . « *Internet Protocol, DARPA Internet Program Protocol Specification* ». Defense Advanced Research Projects Agency, Information Processing Techniques Office, University of Southern California 4676 Admiralty Way, Marina del Rey, California 90291, 1 édition, septembre 1981.

[2] Bowen Alpern et Fred B. Schneider. « Defining liveness ». *j-IPL*, 21:181–185, 1985.

[3] B. W. Bates, J. M. Bruel, R. B. France et M. M. Larrondo-Petrie. « Guidelines for Formalizing Fusion Object-Oriented Analysis Models ». Dans P. Constantopoulos, J. Mylopoulos et Y. Vassiliou, éditeurs, *Advanced Information Systems Engineering*, volume 1080, pages 222–233. Springer-Verlag, 1996.

[4] E. T. Bell. *Men of Mathematics*. Simon and Schuster, New York, 1986. publié originalement en 1937.

[5] J. K. Bennett. « The Design and Implementation of Distributed Smalltalk ». Dans Norman Meyrowitz, éditeur, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 22, pages 318–330, New York, NY, 1987. ACM Press.

[6] Frederick P. Brooks, Jr.. « No silver bullet: essence and accidents of software engineering ». *Computer*, 20(4):10–19, 1987.

[7] Jean-Pierre Courtia et Pierre de Saqui-Sannes. « ESTIM: An Integrated Environment for the Simulation and Verification of OSI Protocols Specified in Estelle ». *Computer Networks and ISDN Systems*, 1(25):83–98, 1992.

[8] S. Deloach. « Analysis and Design using MaSE and agentTool ». Dans *12th Midwest Artificial Intelligence and Cognitive Science Conference (MAICS)*, Miami University, Oxford, Ohio, mars 2001.

[9] Yves Demazeau. « From interactions to collective behaviour in agent-based systems ». Dans *Proceedings of the European Conference on Cognitive Science*, Saint-Malo, France, avril 1995.

[10] Jacques FERBER et Olivier GUTKNECHT. « Aalaadin: a meta-model for the analysis and design of organizations in multi-agent systems ». Dans *ICMAS'98*, juillet 1998.

[11] Jacques FERBER et Olivier GUTKNECHT. « A meta-model for the analysis and design of organizations in multi-agent systems ». Dans *Proceedings of the Third International Conference on Multi-Agent Systems (ICMAS98)*, pages 128–135, Paris, France, 1998.

[12] Stefan FISCHER et Bernd HOFMANN. « An Estelle Compiler for Multiprocessor Platforms ». Rapport Technique TR-93-005, 1, 1993.

[13] Foundation for Intelligent Physical Agents . « FIPA 97 Specification Part 2: Agent Communication Language », octobre 1997. Version 2.0.

[14] UML Superstructure FTF. « UML 2.0 Superstructure specification ». Rapport Technique, OMG, Object Management Group, 250 First Avenue, Ste 100 Needham, MA 02494, 2003. Draft adopted specification.

[15] U. GLÄSSER et R. KARGES. « Abstract State Machine Semantics of SDL ». *Journal of Universal Computer Science*, 3(12):1382–1414, 1997.

[16] G. GOTTLOB, G. KAPPEL et M. SCHREFL. Semantics of Object-Oriented Data Models – The Evolving Algebra Approach. Dans J. W. SCHMIDT et A. A. STOGNY, éditeurs, *Next Generation Information Technology*, volume 504 de *LNCS*, pages 144–160. Springer, 1991.

[17] OMG The Object Management GROUP. « Common Object Request Broker Architecture, CORBA 3.0.3 specification ». online specification, mars 2004.

[18] Olivier GUTKNECHT et Jacques FERBER. « The MADKIT Agent Platform Architecture ». Dans *Proceedings of the Agents Workshop on Infrastructure for Multi-Agent Systems*, pages 48–55, 2000.

[19] Olivier GUTKNECHT, Jacques FERBER et Fabien MICHEL. « Integrating tools and infrastructures for generic multi-agent systems ». Dans Jörg P. MÜLLER, Elisabeth ANDRE, Sandip SEN et Claude FRASSON, éditeurs, *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 441–448, Montréal, Canada, 2001. ACM Press.

[20] E. HAYWOOD et P. DART. « How well do modelling notations support the production of a "good" SRS? ». Rapport Technique, 1999.

[21] Gerard J. HOLZMANN. « The Model Checker SPIN ». *Software Engineering*, 23(5):279–295, 1997.

[22] Marc-Philippe HUGET. « *Une ingénierie des protocoles d'interaction pour les systèmes multiagent* ». PhD thesis, Université Paris IX Dauphiné, juin 2001.

[23] Marc-Philippe HUGET. « Desiderata for Agent Oriented Programming Languages », 2002.

[24] Marc-Phillipe HUGET et James ODELL. « FIPA Modeling: Agent Class Diagrams ». Rapport Technique, FIPA: Foundation for Intelligent Phy-

sical Agents, Genève, Suisse, 2003. FIPA TC Modeling prelimminary specification.

[25] Marc-Phillipe HUGET et James ODELL. « FIPA Modeling: Interaction Diagrams ». Rapport Technique, FIPA: Foundation for Intelligent Physical Agents, Genève, Suisse, 2003. FIPA TC Modeling prelimminary specification.

[26] ISO. « ISO 8807: Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour ». Standard, International Standards Organization, Genève, Suisse, 15 février 1987. First edition.

[27] ISO. « Information processing systems — Open systems interconnection — Estelle — A formal description technique based on an extended state transition model ». IS 9074, Genève, Suisse, 1989.

[28] ISO. « Information technology — Programming languages — Pascal ». IS 7158, Genève, Suisse, 1990.

[29] ISO. « Information processing systems — Open systems interconnection — Guidelines for the Application of Estelle, LOTOS, and SDL ». TR 10167, Genève, Suisse, 1991.

[30] ISO. « *Information technology—Open Systems Interconnection—Basic Reference Model: The Basic Model* ». ISO/IEC, 7498-1 édition, 1994.

[31] ISO. « Information processing systems — Open systems interconnection — Estelle — A formal description technique based on an extended state transition model ». IS 9074, Genève, Suisse, 1997.

[32] ITU-T. « Recommendation Z.120. Message Sequence Charts ». Rapport Technique Z-120, International Telecommunication Union – Standardization Sector, Genève, Suisse, 2000.

[33] ITU-TS. « ITU-TS Recommendation Z.120: Message Sequence Chart 1996 (MSC96) ». Rapport Technique, ITU-TS, Genève, Suisse, 1996.

[34] Eric JUL, Henry LEVY, Norman HUTCHINSON et Andrew BLACK. « Fine-Grained Mobility in the Emerald System ». *ACM Transactions on Computer Systems*, 6(1):109–133, février 1988.

[35] Jean-Luc KONING et Marc-Philippe HUGET. « Validating Reusable Interaction Protocols ». Dans Hamid ARABNIA, éditeur, *The 2000 International Conference on Artificial Intelligence (ICAI-00)*, Las Vegas, NV, June 2000. CSREA Press.

[36] Jean-Luc KONING, Marc-Philippe HUGET, Jun WEI et X. WANG. « Extended Modeling Languages for Interaction Protocol Design ». Dans M. WOOLDRIDGE, P. CIANCARINI et G. WEISS, éditeurs, *Second International Workshop on Agent-Oriented Software Engineering (AOSE-2001)*, Montréal, Canada, mai 2001.

[37] Jean-Luc KONING et Pierre-Yves OUDEYER. « Introduction to POS: A Protocol Operational Semantics ». *International Journal of Cooperative Information Systems*, 10(1-2):101–123, 2001.

[38] Jean-Luc KONING et Ivan ROMERO HERNÁNDEZ. « Thoughts on an Agent Oriented Language Centered on Interaction Modeling ». Dans *Second IEEE International Symposium on Advanced Distributed Systems (ISADS-2002)*, Guadalajara, Mexique, November 2002. IEEE, Springer-Verlag.

[39] Jean-Luc KONING et Ivan ROMERO HERNÁNDEZ. « Generating Machine Processable Representations of Textual Representations of AUML ». Dans Fausto GIUNCHIGLIA, James ODELL et Gerhard WEISS, éditeurs, *Agent-Oriented Software Engineering III, Third International Workshop, AOSE 2002, Bologna, Italy, July 15, 2002, Revised Papers and Invited Contributions*, volume 2585 de *Lecture Notes in Computer Science*. Springer, 2003.

[40] Jürgen LIND. « Specifying Agent Interaction Protocols with Standard UML ». Dans *Agent Oriented Software Engineering (AOSE01)*, Montréal, Canada, 2001.

[41] Michael LUCK et Mark D'INVERNO. « A Conceptual Framework for Agent Definition and Development ». *The Computer Journal*, 44(1):1–20, 2001.

[42] J. MCLEAN. « A General Theory of Composition for Trace Sets Closed Under Selective Interleaving Functions ». Dans *Proc. IEEE Symposium on Research in Security and Privacy*, pages 79–93, 1994.

[43] Robin MILNER. *A Calculus of Communicating Systems*, volume 92 de *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[44] NASA Office of Safety and Mission Assurance, Washington, DC. « *Formal Methods Specification and Verification Guidebook for Software and Computer Systems, Volume I: Planning and Technology Insertion* », juillet 1995. NASA-GB-002-95, Release 1.0.

[45] NASA Office of Safety and Mission Assurance, Washington, DC. « *Formal Methods Specification and Analysis Guidebook for the Verification of Software and Computer Systems, Volume II: A Practitioner's Companion* », mai 1997. NASA-GB-001-97, Release 1.0. Available at `http://eis.jpl.nasa.gov/quality/Formal_Methods/`.

[46] James ODELL, H. Van Dyke PARUNAK et Bernhard BAUER. « Extending UML for Agents ». *AOIS Worshop at AAAI 2000*, 2000. *in press*.

[47] James ODELL, H. Van Dyke PARUNAK et Bernhard BAUER. « Representing Agent Interaction Protocols in UML ». *AAAI Agents*, juin 2000.

[48] James ODELL, H. Van Dyke PARUNAK et Mitch FLEISCHER. « The Role of Roles in Designing Effective Agent Organizations ». *Lecture Notes*

on Computer Science, volume 2603: Software Engineering for Large-Scale Multi-Agent Systems:22–28, 2003. Garcia A. and Lucena, C. and Zambonelli F. and Omicini A. and Castro J. editors.

[49] OMG. « *Unified Modeling Language Specification* ». Object Management Group, 250 First Avenue, Needham, MA 02494 Etats Unis, 1.4 édition, septembre 2001.

[50] Derek PARTRIDGE et Antony GALTON. « The Specification of 'Specification' ». *Minds and Machines*, 5(2):243–225, mai 1995.

[51] Carl Adam PETRI. *Kommunikation mit Automaten.* Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.

[52] Carl Adam PETRI. « Kommunikation mit Automaten. ». *New York: Griffiss Air Force Base, Technical Report RADC-TR-65–377*, 1:1–Suppl. 1, 1966. version anglaise.

[53] Ivan ROMERO HERNÁNDEZ et Jean-Luc KONING. « *Intelligent Knowledge-Based Systems: Business and Technology in the New Millennium* », volume Knowledge Based Systems, Chapitre From Roles to Agents: Considerations on Formal Agent Modeling and Implementation. Kluwer Academic Press, juillet 2004. Dans ce chapitre nous discoutons de la notion de rôle, et celle de rôle comme artefact pour le développement des protocoles de communication entre agents.

[54] Ivan ROMERO HERNÁNDEZ et Jean-Luc KONING. « A Visual Tool for the Modeling of AUML Specifications ». Dans John DEBENHAM, éditeur, *Symposium on Professional Practice in AI*, Toulouse, France, August 22–27 2004. IFIP World Computer Congress, WCC 2004, Kluwer.

[55] John RUSHBY. « Critical System Properties: Survey and Taxonomy ». *Reliability Engineering and System Safety*, 43(2):189–219, 1994.

[56] Yoav SHOHAM. « AGENT0: A Simple Agent Language and Its Interpreter ». Dans *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI-91)*, volume 2, pages 704–709, Anaheim, California, USA, juillet 1991. AAAI Press/MIT Press.

[57] Munindar P. SINGH. « Toward Interaction-Oriented Programming ». Rapport Technique TR-96-15, juin 1996.

[58] J. THEES. « Protocol implementation with Estelle – from prototypes to efficient implementations », 1998.

[59] Xu WANG, Shing-Chi CHEUNG et Jun WEI. « On the Modeling of Exchanging Processes in E-commerce Protocols ». *unknown*, août 2001.

[60] Jun WEI, Cheung SHING-CHI et Wang XU. « Towards a Methodology for formal design and analysis of Agent Interaction Protocols ». *Proceedings of International Software Engineering Symposium (ISES '2001), Wuhan, Hubei, China*, mars 2001.

[61] J. E. WHITE. « A High-Level Framework for Network-Based Resource Sharing ». online request for comments, RFC 707, décembre 1975. La première mention de la notion d'appel à fonction lointane.

[62] N. WIRTH. « The programming language Pascal (Revised report) ». Rapport technique 5, Dept. Informatik, Inst. Für Computersysteme, ETH Zürich, Zürich, Suisse, juillet 1973.

[63] Michael WOOLDRIDGE, Nicholas R. JENNINGS et David KINNY. « The Gaia Methodology for Agent-Oriented Analysis and Design ». *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, 2000.

[64] Michael J. WOOLDRIDGE et Nicholas R. JENNINGS. « Agent Theories, Architectures, and Languages: A Survey ». Dans Michael J. WOOLDRIDGE et Nicholas R. JENNINGS, éditeurs, *Workshop on Agent Theories, Architectures & Languages (ECAI'94)*, volume 890 de *Lecture Notes in Artificial Intelligence*, pages 1–22, Amsterdam, Pays Bas, janvier 1995. Springer-Verlag.

[65] Michael J. WOOLDRIDGE et Nicholas R. JENNINGS. « Intelligent Agents: Theory and Practice ». *Knowledge Engineering Review*, 10(2):115–152, juin 1995.