Communication Agent et Interprétation Scheme pour l'apprentissage au méta-niveau

Clement Jonquet

ACADEMIE DE MONTPELLIER

UNIVERSITE MONTPELLIER II

- Sciences et Techniques du Languedoc -

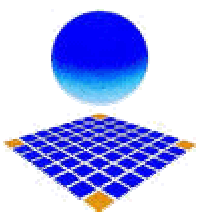
DEA Informatique

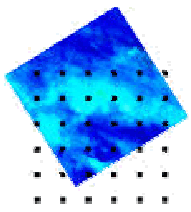
Communication Agent et Interprétation Scheme pour l'apprentissage au méta-niveau

Clément Jonquet

***Encadrant* : Stefano A. Cerri**

Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier (LIRMM)





Juin 2003

Résumé

La communication entre agents cognitifs est un domaine de recherche en pleine effervescence. Notre travail consiste ici à proposer un modèle, basé sur le modèle STROBE, qui considère les agents comme des interpréteurs Scheme. Ces agents sont capables d'interpréter des messages dans un environnement donné incluant un interpréteur qui apprend par les conversations. Ces interpréteurs peuvent, en outre, évoluer dynamiquement au fur et à mesure des conversations et représentent la connaissance de ces agents au niveau méta. Nous illustrons ce modèle théorique par une expérimentation de dialogue de type « professeur-élève » où un agent apprend un nouveau performatif à l'issue de la conversation.

Ainsi, ce mémoire présente avant tout les deux domaines dont notre travail s'inspire, c'est à dire l'évaluation en Scheme et la communication agent. Puis, il présente notre modèle et l'illustre par une expérimentation. Et enfin, il termine en regardant en quoi ce modèle peut être effectif dans des domaines comme le Web, le Grid ou les dialogues de type e-commerce (grâce à un point de vue contraintes). Ce dernier point est particulièrement développé pour montrer comment réaliser la spécification dynamique d'un problème en développant des agents communicants. Les détails de l'implémentation sont également fournis.

Mots clés : **Scheme, réflexivité, réification, communication agent, STROBE, interprétation de message, méta-évaluation, apprentissage par communication, apprentissage au méta-niveau, spécification dynamique.**

Abstract

Cognitive agent communication is a research field in full development. We propose here an extension and a partial implementation of the STROBE model, which regards the agents as Scheme interpreters. These agents are able to interpret messages in a dedicated environment including an interpreter that learns from the current conversation. These interpreters evolve dynamically, progressively with the conversations, and thus represent evolving meta-level agent's knowledge. We illustrate this theoretical model by a "teacher-student" dialogue experimentation, where an agent learns a new performative at the completion of the conversation.

This report is organized as follows. First, we survey the two research domains inspiring our experiment: Scheme evaluation and agent communication. Then we present our model and illustrate it by an experimentation. Finally, we show how this model is effective for domains such as the Web, Grid Computing and e-commerce dialogue (with a constraint point of view). This last idea is particularly developed to show how to enable the dynamic specification of a problem by designing communicating agents. The implementation details are also available.

Keywords: **Scheme, reflexivity, reification, agent communication, STROBE model, message interpretation, meta-evaluation, learning-by-being-told, learning-by-communicating, meta-level learning, dynamic specification.**

Sommaire

Index des figures

Introduction

Dans le cadre de ce stage de DEA, nous[1] nous sommes mis au travail sans trop savoir où nous allions. C'est à dire que le sujet n'était pas défini avant le début du stage, il s'est construit petit à petit au fur et à mesure de notre travail et nous ne savons pas s'il est réellement défini aujourd'hui[2]. Cependant nous pensons que ce stage a rempli son rôle pédagogique et formateur et nous allons tenter de le montrer dans ce rapport. Il est le résultat d'une formation à (et par) la recherche rassemblant un enseignant, un étudiant et une équipe dans un travail commun. Notre idée était d'arriver à bien maîtriser une série d'outils pour faire émerger des potentialités et donc des solutions. En effet, l'informatique n'est pas toujours « spécifier et après coder »[3] mais cela peut être aussi « expérimenter et reconnaître de nouvelles potentialités architecturales ». C'est exactement ce qu'il s'est passé. Nous avons commencé notre travail en étudiant les outils qui nous paraissaient puissants et intéressants pour les domaines qui nous intriguaient. L'outil principal était **Scheme** et le domaine intrigant, **les interactions entre agents**.

Ce rapport de stage va donc essayer de vous expliciter notre démarche. Dans un premier temps nous avons donc travaillé sur Scheme pour comprendre en quoi ce petit dialecte issu de Lisp possède un potentiel très puissant[4] et pourquoi cet outil simple et pédagogique peut nous être plus utile qu'un « super langage » moderne et industriellement approuvé. Etudier le langage Scheme implique d'utiliser ce langage mais surtout de comprendre comment il est construit et quels sont les mécanismes qui le définissent. Pour réaliser ce point, nous avons décidé de faire ce que l'enseignement prône à juste raison : travailler sur un méta-évaluateur Scheme. En parallèle, nous étudions une série d'articles sur des alternatives possibles à la modélisation des interactions entre agents. Ainsi, nous avons travaillé sur le modèle STROBE [CER 99b] & [CER 97] qui, lui-même, utilise Scheme. Ce modèle nous a inspiré des idées qui ont pu être, par la suite, expérimentées.

Nous allons donc expliciter notre modèle de communication agents, basé sur le modèle STROBE, qui considère les agents comme des interpréteurs Scheme. **Ces agents sont capables d'interpréter des messages dans un environnement donné incluant un interpréteur qui apprend par les conversations. Ces environnements sont dédiés à une conversation et les interpréteurs peuvent, en outre, évoluer dynamiquement au fur et à mesure des conversations. Ils représentent la connaissance de ces agents au niveau méta. Nous proposons un mécanisme d'apprentissage à ce méta-niveau par modification dynamique des interpréteurs.** De plus, nous illustrons ce modèle théorique par une expérimentation de dialogue de type « professeur-élève », où un agent apprend un nouveau performatif à l'issue de la conversation.

Ainsi, la suite de ce rapport se compose de la manière suivante : Le chapitre 1, introduisant Scheme, explique pourquoi ce langage est intéressant et explicite son mécanisme d'évaluation. Nous présentons également dans ce chapitre notre méta-évaluateur, support de tout notre travail. Le chapitre 2 a pour but de rappeler les problématiques de la communication agent et recadre notre travail dans ce domaine. Notre modèle, les agents le supportant et le mécanisme que nous proposons sont explicités dans le chapitre 3. Ici est également présentée l'expérimentation qui appuie notre travail. Finalement, le chapitre 4 se consacre à l'étude de ce modèle et de ses intérêts pour des domaines tels que le Web, le Grid Computing ou les dialogues de type e-commerce (grâce à un point de vue contraintes). Ce dernier point est particulièrement développé pour montrer comment réaliser la spécification dynamique d'un problème en développant des agents communicants. Ce chapitre propose également d'autres extensions possibles à notre travail. Les annexes présentent la légère implémentation de notre modèle et les articles soumis dont il a été l'objet.

---

[1] Nous entendons par « nous » le binôme de travail composé par l'étudiant et l'encadrant.

[2] Voir *Le Monde* du 03/06/03, ou Jean Pierre Serre, Prix Abel 2003 de mathématiques est interviewé : « La tradition voulait qu'on ne donne pas de sujet de thèse. Il fallait trouver tout seul. Pendant un an et demi, je n'ai rien trouvé. Mais lorsque ça marchait, c'était original. »

[3] Comme nous le verrons, notre modèle tente de changer cette idée reçue grâce à la spécification dynamique de problème.

[4] Le standard R^{5}RS [R5R 98] ne fait que 50 pages et nous le verrons un tout petit nombre de formes spéciales permettent de réécrire presque tout le langage.

Chapitre 1. Scheme dans tous ses états

1.1. Présentation

La première partie de notre travail a consisté à étudier le langage Scheme sous le plus de formes possibles. Scheme est un langage de la famille Lisp / Algol60 crée par Sussman et Steele au MIT[5] [SUS 75] dans la fin des années 70. C'est un langage de programmation applicative, dialecte de Lisp, dont le but est plus universitaire qu'industriel. Scheme est utilisé aux quatre coins du monde pour enseigner aux étudiants les subtilités de la programmation sous toutes ses formes. Sa syntaxe peut être intégrée par un étudiant, n'ayant jamais fait de programmation, en quelques heures. C'est un des plus petit langage de haut niveau existant. Il permet, en outre, de représenter facilement certains concepts mathématiques ce qui facilite la compréhension de la programmation en utilisant un domaine souvent déjà bien connu des futurs programmeurs. Par ailleurs, Scheme est énormément utilisé dans le monde de la recherche car sa simplicité est un atout pour faire comprendre des concepts très compliqués. Nous devrions plutôt dire que Scheme possède la grande faculté de rendre les choses simples. Il sert par exemple de support aux célèbres livres [ABE 96], [QUE 94] et [CHA 96]. Moreau et al. expliquent comment construire un cours de programmation avec Scheme dans [MOR 98]. Ils présentent les différentes propriétés du langage et la façon de les illustrer avec Scheme : l'abstraction, via des données ou des procédures, la récursivité, l'évaluation par substitution, l'évaluation par environnement, le typage dynamique, les flots, les continuations, les macros, l'évaluation paresseuse... Tous ces concepts sont autant de points que l'on peut aborder dans un cours de programmation avec Scheme. Nous en détaillerons quelques-uns ici, ceux qui seront utilisés plus tard pour le sujet qui nous intéresse.

Ainsi, nous distinguerons dans un premier temps les caractéristiques de Scheme qui en font un langage si puissant, en particulier l'abstraction. Nous verrons également ses différences avec Lisp qui justifient notre choix. Une section sera également consacrée à l'implémentation des objets avec Scheme. Nous aborderons aussi les avantages qu'il peut avoir en tant que langage Web. En seconde partie, nous nous intéresserons au concept d'évaluation qui permet de comprendre comment un langage est construit. Nous détaillerons ici le modèle d'évaluation par environnement et le principe d'évaluation paresseuse qui, nous le verrons, peut s'avérer très utile. Ensuite, nous présenterons le méta-évaluateur sur lequel nous avons travaillé et qui sert de support à tout notre travail et en particulier à l'expérimentation présentée au Chapitre 3. Finalement, les outils que nous allons utiliser seront explicités, en particulier le principe de réflexivité et le mécanisme des *reifying procedures* qui permettent de modifier dynamiquement un interpréteur.

1.2. Pourquoi Scheme ?

Tout l'art de la programmation est d'écrire des programmes dont le but est de résoudre des problèmes. Pour résoudre des problèmes, il faut un outil à la fois formel et qui ne soit pas plus compliqué à utiliser que ce que le problème est à résoudre. Si l'on considère la formule Langage = Syntaxique + Sémantique + Pragmatique alors Scheme convient tout à fait car il est basé sur la sémantique du lambda calcul, ce qui en fait un très bon outil théorique, et c'est un dialecte de Lisp dont la syntaxe peut s'apprendre en quelques heures.

[5] Massachusetts Institute of Technology

1.2.1. Caractéristiques du langage

1.2.1.1. L'abstraction par l'expression Scheme

Scheme est un langage d'une rigueur implacable. Il possède le grand atout d'avoir une seule façon de représenter les procédures, les données et l'application des premières sur les secondes : l'expression Scheme. Ce formalisme est tiré des S-expressions[6]. En fait, on dit que les langages Lisp ont la possibilité de représenter les procédures comme des données (des listes). Après tout, n'oublions pas que Lisp est l'acronyme pour LISt Processing ! Ceci est très bien expliqué dans [ABE 96] qui présente dans ses deux premiers chapitres les subtilités de l'abstraction : Abstraire avec des procédures ou comment, à partir de procédures simples (ou primitives), construire des procédures plus compliquées (ou composées) pour obtenir le résultat attendu. Mais aussi, abstraire avec des données ou comment à partir de données simples, obtenir des données plus complexes représentant des concepts voulus.

Donc, quasiment tout le langage peut être défini par 5 formes spéciales :

- Pour les procédures : `(lambda paramètres corps)`
- Un opérateur de définition et d'affectation : `(define nom exp)` et `(set! nom exp)`
- Une structure de contrôle : `(if cond exp1 exp2)`
- Un opérateur de citation : `(quote exp)`[7]
- Et aussi, pour les données : `(cons a b)` `(car exp)` `(cdr exp)`, la liste vide `()`[8]

A partir de là, une chose consiste à apprendre des concepts (des primitives) d'un langage, donc savoir qu'elle est leur sémantique et leur syntaxe ; et une autre chose est d'apprendre à résoudre des problèmes en utilisant des combinaisons de ces primitives pour abstraire. La programmation consiste en un dialogue entre le programmeur et l'interpréteur Scheme qui associe des valeurs à ses programmes. La clef est que tout ceci se fait dynamiquement !

1.2.1.2. Propriétés du langage

Normark [NOR 91] cite quatre points caractérisant Scheme :

- Scheme lie les variables libres de manière statique dans les procédures.
- Les procédures Scheme sont des objets de première classe[9]. Elles permettent d'encapsuler un environnement de définition associant variable et valeur au moment de la définition d'une procédure. Ceci augmente considérablement le potentiel expressif du langage. Le couple formé par la lambda expression définissant la fonction et son environnement de définition s'appelle une fermeture.
- Tous les éléments d'une expression Scheme sont évalués de la même manière, en particulier le premier dont l'évaluation peut renvoyer la définition d'une procédure.
- Un programme n'est pas de la donnée. C'est à dire qu'un programme n'est pas facilement accessible comme une structure de liste. La procédure `eval` ne fait pas partie non plus du langage de base[10].

De plus, nous avons aussi les autres caractéristiques :

- L'abstraction est possible via les procédures et les données (cf. plus haut).
- Les données Scheme sont typées dynamiquement au niveau langage mais peuvent être explicitement typées au niveau du programme utilisateur.

---

[6] Une S-expression est soit un nombre, soit une chaîne, soit un identificateur, soit une liste de S-expression.

[7] Avec `unquote, quasiquote, et unquote-splicing` la macrologie devient possible.

[8] Cependant `cons`, `car` et `cdr` peuvent également être réécrites par des procédures [ABE 96] donc ne font pas partie des 5.

[9] Un objet de première classe peut être nommé par une variable, stocké dans des structures de données, être passé en paramètre et être retourné comme résultat d'une autre procédure.

[10] Ce point la n'est plus exact aujourd'hui puisque la fonction `eval` d'évaluation d'une expression Scheme est disponible dans les distributions de Scheme (norme R5RS).

- Les processus peuvent être itératifs ou récursifs.
- Une mémoire existe grâce au concept d'environnement (cf. §1.3.1).

1.2.1.3. Différences avec Lisp

Scheme lie les variables libres de manière statique dans les procédures (portée statique). C'est à dire que les variables libres sont liées dans le contexte (environnement lexical) de définition de la procédure. En Lisp, les variables sont liées dynamiquement c'est à dire que les variables libres dans une procédure reçoivent leur valeur de l'environnement d'appel et non de l'environnement de définition. Les variables libres peuvent avoir plusieurs valeurs en fonction du déroulement du processus d'appel.

En outre, Scheme est un langage à typage dynamique. Lisp l'est aussi mais pas réellement car il différencie données `(defvar...)` et procédures `(defun...)`. En effet, en Lisp, le premier argument d'un appel de procédure doit être un nom de procédure ce qui contredit la caractéristique de Scheme qui dit que l'évaluation d'un élément d'une expression Scheme est la même quelle que soit la position de l'élément.

Exemple :

```
(let ((fun (compute-a-function)))
    (fun x y))
```

est possible en Scheme…

```
(let ((fun (compute-a-function)))
    (funcall fun x y))
```

…en Lisp il faut `funcall`

1.2.2. Implémentation des objets en Scheme

Nous reprenons ici une citation de Saint-James dans [MAS 90] :

> *(...) chaque nouveau concept introduit en programmation peut facilement être introduit en Lisp, grâce à la totale liberté qu'il offre, suscitant de nouvelles perspectives d'utilisation et d'implémentation.*

La possibilité de mémoriser l'environnement de définition d'une procédure dans sa fermeture et l'opérateur d'affectation `set!` permettent de considérer les procédures comme des entités encapsulant des états locaux et des méthodes c'est à dire comme des objets. Aujourd'hui, la programmation objet est à son apogée mais nous pouvons toutefois nous demander ce qu'est un objet ou du moins d'où cela vient. A l'heure où le programmeur lambda sait créer une classe Java qui se pose les questions : Comment est crée la classe `Object` que je viens d'étendre ? Comment est crée la méta-classe `Class` que je viens d'instancier ? En fait, la fameuse formule : « tout est objet » n'est qu'une dérivée partielle de la formule « tout est donnée et procédure » !

En effet, comme le montre Normark dans [NOR 91], on peut représenter des objets en utilisant seulement des procédures donc avec Scheme. Nous allons voir comment les méthodes d'une classe peuvent être appelées via un passage de message. L'architecture d'une classe en Scheme peut être schématisée par le module de la Figure 1.

Chaque objet peut avoir ses propres variables locales (ces attributs). Il doit cependant obligatoirement fournir des accesseurs (get et set) dessus puisque le seul moyen d'accéder à l'objet est l'appel de méthode. La fonction `self` sert à « aiguiller » selon le type du message vers telle ou telle fonction correspondante à une méthode de la classe. Si `self` ne trouve pas la méthode correspondante dans la classe elle-même, elle regarde dans la classe juste au-dessus dans la hiérarchie. Une instance de la classe est définie par `(define nom_instance (class-name))`. La procédure `send` est le seul moyen d'interaction avec les objets. Ainsi, tout ce fait par passage de messages :

`(send instance message parameter)` est équivalent à l'expression `((instance 'message) parameter)`

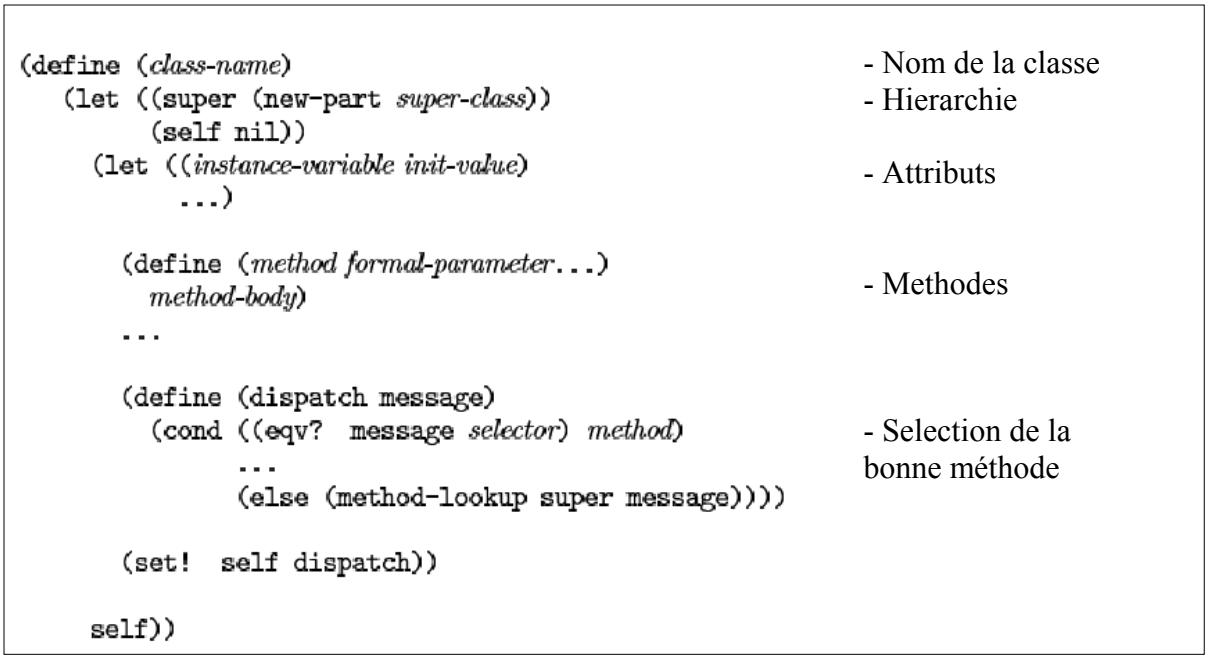


Figure 1. Représentation d'une classe d'objets par une procédure Scheme

Le modèle présenté ci-contre permet de construire des hiérarchies de classe en héritage simple. Cependant, si nous supposons que `super` ne réfère plus à un seul objet mais à une liste d'objet, alors nous pouvons mettre en place un système à héritage multiple. [MAS 90] propose d'autres approches des objets en Lisp, ainsi qu'une implémentation du célèbre modèle ObjVlisp. Il existe également CLOS (Common Lisp Object System) qui est un langage objet complet basé sur Common Lisp.

Remarque : Aujourd'hui, l'intérêt pour les langages à objet étant complètement avéré, la communauté Lisp et Scheme propose des langages comme KAWA ou SISC qui permettent de générer du Java bytecode[11] à partir de programme Scheme. Bigloo, langage proposé par Serrano [SER 02], permet également de rapprocher Scheme de C ou C++.

1.2.3. Scheme comme langage Web

Le concept d'environnement permet aux programmes Scheme d'avoir une mémoire. Mais qu'en serait-il si cette mémoire était le Web ? [CER 00a]. C'est à dire, si le Web était l'environnement d'évaluation des programmes. Un environnement qui associerait des URL à des documents Web, comme il associe des variables à leur valeur. Actuellement, les documents du Web sont en majorité écrits en HTML ou génèrent du HTML mais la nécessité de séparer la syntaxe et la sémantique fait s'imposer un nouveau *markup* langage approuvé aussi bien dans le mode de la recherche que dans le monde industriel : XML (eXtensible Markup Language). Par ailleurs, les documents XML sont des arbres et Scheme un des meilleurs langage de représentation et de traitements des arbres. Des langages liant Scheme (les S-expressions) et XML apparaissent de plus en plus : SXML, SXPath, SXSLT cf. [KIS 02]. Normark travaille également depuis plusieurs années sur LAML [NOR 03].

En outre, les caractéristiques de Scheme et, en particulier, le typage dynamique et l'évaluation paresseuse que nous verrons plus bas, semblent être de bons atouts pour les langages du Web [CER 99a]. Imaginons que des pages Web contiennent les données qui intéressent un client. Leur temps d'accès peut ne pas être prévisible, leur type ne peut être anticipé. Donc, il faut des fonctions génériques et donc un langage à typage dynamique. Si l'association entre un type donné et les procédures qui lui sont associées (constructeur, sélecteur...) n'est pas connue par un client, alors les documents Web doivent devenir des objets répondant à des messages suivant un protocole donné. Ils doivent pouvoir fournir cette association. On en arrive donc à une communication où le client et le

[11] Le code interprété par la JVM (Java Virtual Machine).

document doivent s'entendre avant tout sur des conventions puis s'échanger pour de bon les données. Notre travail s'inscrit complètement dans cette logique car s'entendre signifie dialoguer et dialoguer signifie communiquer ! Etant donné que le Web devient le domaine de prédilection des agents, la boucle est bouclée.

Scheme possède intrinsèquement des concepts faisant de lui un langage pour le Web et les agents. [CER 99b] montre comment utiliser les concepts de flots, d'objets et surtout d'environnement (STReams OBject Environments) pour modéliser les communications. [QUE 00] montre comment utiliser le concept de continuation pour implémenter des serveurs Web. [GOU 02] propose d'utiliser le concept de flot pour mémoriser l'intégralité d'une conversation.

1.3. L'évaluation d'une expression Scheme

Un interpréteur Scheme fonctionne suivant la boucle classique : *Read - Eval – Print –Listen* (REPL). Celle-ci fait toujours la même chose : lire l'expression, l'évaluer, afficher le résultat et attendre la prochaine expression. Dans les paragraphes suivants, nous ne détaillerons que le procédé d'évaluation des combinations[12] c'est à dire des primitives et des fonctions composées définies via `lambda`. Les formes spéciales (`if`, `quote`…) n'obéissent pas forcément au même schéma.

1.3.1. Evaluation par environnement

Un environnement est une structure de donnée dont le but est d'associer des variables avec des valeurs. Traditionnellement, cette structure est nécessaire au moment de l'évaluation ou de la compilation d'un programme pour éditer les liens entre les variables et les valeurs. Un environnement est constitué d'une « série » de cadres (ou blocs) qui sont eux-mêmes des tables de liaisons variable-valeur. Une variable a pour valeur la valeur donnée par sa liaison dans l'environnement. L'environnement est très important pour l'évaluation d'une expression car il définit le premier élément du contexte d'exécution[13]. Le modèle classique d'évaluation par environnement se caractérise par deux étapes :

- Pour évaluer une combination, il faut évaluer ses sous-expressions et appliquer la valeur de l'opérateur aux valeurs des arguments.
- Pour appliquer une procédure composée à un ensemble d'arguments, il faut évaluer le corps de la procédure dans un nouvel environnement construit de la manière suivante : Etendre l'environnement de définition de la procédure en lui ajoutant le cadre liant les paramètres formels de cette procédure aux valeurs des arguments de l'appel.

Remarque : Ce mode d'encapsulation est dit environnement lexical. Une procédure n'a accès qu'aux variables qu'elle définit elle-même et celles déjà présentes dans l'environnement dans lequel la fonction a été évaluée. En opposition, il existe des environnements dynamiques où une procédure a accès à l'ensemble des variables définies, non plus lors de son évaluation mais lors de son appel.

La Figure 2 [ABE 96] montre comment, dans l'environnement global (4), le couple (2) définit la fermeture de la procédure `square` avec comme environement de définition, ici et comme souvent, l'environnement global. Lors de l'appel (1), l'environnement de définition est étendu par le cadre E1 (3) ce qui permet de lier le paramêtre formel `x` avec l'argument `5`.

[12] Le terme combination est un mot anglais qui représente un appel à une procédure (primitive ou composée) de la forme `(opérateur opérandes)`. Il peut etre traduit en francais par « application » ou « forme » voir même « combinaison » mais pour éviter toute ambiguïté, nous continuerons dans la suite de ce rapport à utiliser le mot anglais.

[13] Le contexte d'exécution d'une expression est constitué de l'environnement d'évaluation de cette expression et de la continuation qui correspond à la prochaine expression à évaluer avec le résultat de cette l'expression. Une continuation est donc une procédure d'arité un, qui définie ce qu'il y a à faire du résultat d'une évaluation. En anglais on parle de "execution context" ou de "state of the computation".

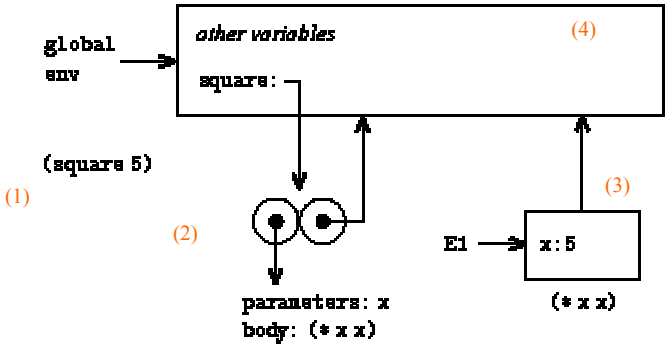


Figure 2. Exemple d'évaluation par environnement

C'est grâce à cette encapsulation que l'on peut dire que les procédures Scheme sont des objets de première classe. Le modèle d'évaluation par environnement, nous l'avons vu dans le cas des objets, permet un gain important de modularité des programmes. Il donne aussi aux programmes la possibilité d'avoir une mémoire accessible qu'ils peuvent modifier.

1.3.2. Evaluation paresseuse

Un évaluateur paresseux évalue les combinations de manière particulière. Il retarde l'évaluation de tous les arguments d'une procédure et ne les force que lorsqu'ils sont explicitement appelés (par exemple, appelés par une primitive). Pour un évaluateur Scheme classique, les arguments des procédures sont dits stricts s'ils sont évalués avant l'application de la procédure. Ceci supposant qu'ils seront tous nécessaires. Ce mécanisme s'appelle le « call-by-value » c'est à dire que l'appel de la méthode est fait avec la valeur des arguments. Ce sont ces valeurs qui sont empilées dans la pile au moment de l'appel. A moins de pouvoir prévoir quels arguments sont stricts et lesquels sont non stricts, il est préférable d'utiliser un mode d'évaluation dit paresseux car il n'évalue pas les arguments d'une procédure avant qu'elle en ait réellement besoin. Ce mécanisme s'appelle le « call-by-name » c'est à dire que l'appel de la méthode est fait avec le nom des arguments de façon à pouvoir les retrouver dans l'environnement le moment voulu.

Remarque : Même si l'évaluation paresseuse n'est pas implémentée par défaut dans les interpréteurs Scheme, certaines expressions sont tout de même évaluées de façon paresseuse. Par exemple `(if cond exp1 exp2)`, où `exp1` (respectivement `exp2`) n'est évaluée que si et seulement si `cond` est vrai (respectivement faux). Le fait d'utiliser un évaluateur paresseux permet alors d'écrire `if` non plus comme une forme spéciale mais comme une fonction. Ce qui fait passer le nombre minimum de primitives du langage à 4 ! (cf. §1.2.1.1)

Lorsqu'on travaille avec des fonctions génériques ou avec des objets en Scheme, il est important de s'assurer de l'évaluation paresseuse des expressions. En effet, il ne sert à rien d'évaluer les paramètres d'une procédure avant d'avoir trouvé (si elle existe) et évalué la fonction adéquate qui leur correspond.

L'évaluation paresseuse permet d'utiliser le concept de flot (*stream*). Les flots sont une structure de donnée proche de la liste. Ils permettent de modéliser l'état d'un système sans utiliser les assignations ou les données mutables en conservant son évolution dans le temps. L'idée est de construire une liste partielle en fournissant la méthode pour en obtenir la suite. D'un point de vue abstraction de donnée, les flots sont équivalents aux listes. La différence se situe au niveau du moment de l'évaluation. En effet, les éléments d'une liste (`car` et `cdr`) sont évalués au moment de la construction de celle-ci, tandis que pour un flot, le `car` est évalué de la même manière mais le `cdr` n'est évalué que lorsqu'il est appelé. Les flots sont construits à l'aide des procédures `delay` et `force` qui sont respectivement capables de retarder l'évaluation d'une expression et de forcer l'évaluation d'une expression retardée.

Remarque : Il est possible de proposer une structure d'environnement conservant l'histoire. Au lieu d'avoir un cadre de la forme (var val), il serait de la forme d'une liste (var valn valn-1 ... val1). Voir,

encore mieux, de la forme d'un flot (var val1 val2…valn…) où les trois derniers petits points correspondraient à la promesse de la prochaine valeur de var.

1.4. Notre méta-évaluateur Scheme

La caractéristique fondamentale des langages Lisp, nous l'avons vu, est la possibilité de représenter les procédures comme des données. Cette caractéristique fait de Lisp, et encore plus de Scheme (troisième caractéristiques de Normark), un excellent langage pour écrire des programmes qui doivent manipuler d'autres programmes comme des évaluateurs et des compilateurs.

1.4.1. Langage reconnu par notre méta-évaluateur

Le méta-évaluateur[14] que nous proposons est issu du célèbre article de Jefferson et Friedman, *A Simple Reflective Interpreter* [FRI 92]. Cet évaluateur a l'avantage d'être facile à prendre en main et possède déjà la propriété de réflexivité dont nous avons besoin. Il propose en outre le mécanisme des *reifying procedures* qui, comme nous le verrons plus loin, permet à un programme utilisateur d'accéder à son contexte d'exécution et éventuellement de le modifier. Cet évaluateur a une signature de la forme `(evaluate expression environnement continuation)`.

L'évaluateur de [FRI 92] étant très minimal, nous y avons rajouté au fur et à mesure de notre travail, différentes méthodes selon ce que nous désirions faire. Définir un évaluateur est équivalent à définir un sous-langage correspondant aux expressions que cet évaluateur est capable de traiter. La Figure 3 illustre les expressions reconnues par notre méta-évaluateur donc, le sous langage de Scheme qu'il définit.

```
exp  ::=   (exp {exp}*)
       |   ((lambda ({identifier}*)) {exp}+)
       |   (define identifier exp)
       |   (let localenv {exp}+)
       |   (if exp exp exp)
       |   (set! identifier exp)
       |   (begin {exp}+)
       |   (quote exp)
       |   (quasiquote exp)
       |   (unquote exp)
       |   (unquote-splicing exp)
       |   identifier
       |   constant
```

Figure 3. Langage reconnu par notre méta-évaluateur (fonction `evaluate`)

En effet, les quelques formes spéciales de la Figure 3 permettent d'écrire bon nombre de procédures et donc de programmes. Il y manque cependant des opérateurs de structure de donnée comme `cons, car, cdr` ou comme les prédicats `pair?, null?` ; Ceux-ci seront donc considérés comme des primitives qui seront interprétées par l'interpréteur Scheme traditionnel.

[14] Un évaluateur écrit dans le même langage que celui qu'il évalue est dit évaluateur méta-circulaire ou méta-évaluateur.

1.4.2. Implémentation

Notre méta-évaluateur est défini principalement par deux procédures : `evaluate` et `apply-procedure`[15] correspondantes aux deux étapes de l'évaluation par environnement présentées au §1.3.1. Nous détaillerons ici uniquement les fonctions `evaluate-combination` et `apply-procedure` qui sont nécessaires à la compréhension du mécanisme des *reifying procedures*. L'implémentation complète est disponible en Annexe A.

La fonction principale du programme est la fonction `evaluate` qui sélectionne suivant l'expression à évaluer la méthode adéquate. En effet, les variables, les constantes ou les formes spéciales nécessitent une évaluation particulière. Dans le cas général, soit pour une combination, cette méthode appelle `evaluate-combination`. Cette fonction réalise la première des deux étapes de l'évaluation par environnement. Elle évalue l'opérateur `(operator-part)` ainsi que les opérandes `(operands-part)` et passe les résultats à la fonction `apply-procédure`. Ceci, dans le cas où l'opérateur ne correspond pas à une *reifying procedure* (cf. §1.5.3).

```
(define evaluate-combination
      (lambda (e r k)
        (evaluate (operator-part e) r
            (lambda (proc)
            (if (reifier? proc)
                ((reifier-to-compound proc) (operands-part e) r k)
                (evaluate-operands (operands-part e) r
                    (lambda (args) (apply-procedure proc args k))))))))
```

La fonction `apply-procedure` réalise la deuxième étape. Si la procédure est composée (`compound`) elle évalue son corps dans une extension de l'environnement de définition de la procédure où les paramètres formels sont liés aux arguments d'appel (cette extension est construite avec la méthode `extend`). Si c'est une primitive, elle transmet le calcul à la fonction `apply-primitive` qui appelle directement la fonction via l'interpréteur traditionnel.

```
(define apply-procedure
      (lambda (proc args k)
        (if (compound? proc)
            (evaluate-sequence
                  (procedure-body proc)
                        (extend (procedure-environment proc)
                                (procedure-parameters proc)
                                args)
                  k)
            (k (apply-primitive (procedure-name proc) args)))))
```

Regardons maintenant la structure d'environnement de notre méta-évaluateur. Celle-ci est une liste de liaisons de la forme `(var val)` où `val` varie selon le type de variable dont il s'agit :

- Constante ou variable simple, la liaison est du type `(variable valeur)`
- Procédure, la liaison est du type `(nomproc type param corps defenv)` qui caractérise la fermeture de la procédure où `nomproc` est le nom de la procédure crée, `type` est le type de cette procédure (`compound, reifier, thunk…`), `param` est la liste des paramètres formels de la procédure, `corps` est l'expression correspondante à ce qu'elle effectue et `defenv` est l'environnement de définition de la procédure
- Primitive, la liaison est du type `(nom primitive nom)`

Les fonctions `extend` et `get-pair` permettent respectivement d'étendre un environnement et de trouver la valeur correspondante à une variable dans un environnement. A l'évaluation, une variable est avant tout recherchée dans environnement local de la procédure s'il existe (`let…`) puis dans l'environnement de définition puis dans l'environnement passé en paramètre à la fonction d'évaluation

[15] Correspondantes aux fonctions `eval` et `apply` de [ABE 96] et `evaluate` et `invoke` de [QUE 94].

et à défaut dans l'environnement global. Les primitives, par exemple, sont liées dans l'environnement global.

1.4.3. Module implémentant l'évaluation paresseuse

Une partie de notre travail a consisté à rajouter la propriété d'évaluation paresseuse à notre méta-évaluateur. Nous lui avons donc inséré un module redéfinissant certaines fonctions en particulier `evaluate-combination` et `apply-procedure` de façon à l'implémenter. Nous nous sommes inspiré pour cela du chapitre 4 de [ABE 96]. Dorénavant, nous typons les données au moment de l'évaluation d'une combination. Les arguments, dont l'évaluation est retardée, sont transformés en « thunks ». Un thunk verra son évaluation forcée au moment ou cela sera nécessaire. Les liaisons dans l'environnement deviennent `(var thunk exp env)`. Vous trouverez en Annexe A (fichier lazysimple.scm) tous les détails de cette implémentation.

1.5. Réflexivité et interprétation

En informatique, la réflexion[16] est le domaine où les programmes peuvent se décrire ou se manipuler eux-mêmes. Un langage réflexif a pour caractéristique principale le fait d'être extensible. En effet, les programmes écrits à l'aide d'un langage réflexif ont la possibilité d'étendre ce langage lui-même en accédant à son contexte d'évaluation.

1.5.1. Architecture à tour réflexive

En ce qui concerne la réflexivité, l'évaluateur proposé dans [FRI 92] est déjà implémenté de manière réflexive. C'est à dire que le code de l'évaluateur est écrit dans le sous-langage de Scheme que celui-ci reconnaît. Mais Jefferson et Friedman proposent également, dans leur article, une architecture dite à tour réflexive où à chaque niveau correspond un évaluateur qui interprète le langage du dessus. Le niveau supérieur, quant à lui, interprète le code de l'utilisateur. Deux mécanismes permettent de « se déplacer » dans la tour de façon à évaluer le code à n'importe quel niveau : la réflexivité permet de passer au niveau supérieur et la réification permet de redescendre. Ces deux points nous intéressent tout particulièrement.

1.5.2. Modification dynamique d'un interpréteur

Les langages fournissent souvent des méthodes de modification du code utilisateur à l'exécution mais celles-ci sont très indisciplinées et dangereuses. En outre, ces méthodes ne permettent pas à l'utilisateur d'accéder entièrement au contexte d'exécution. L'architecture de tour réflexive le permet ; Elle donne accès au contexte d'exécution et permet d'étendre l'interpréteur. [SIM 92] propose de considérer les interpréteurs comme des objets de première classe. Cet article présente Recfi *(Reflective extension by first class interpreters*), un langage réflexif et extensible basé sur le concept des meta-continuation. L'idée est d'exécuter à la fois le code système (définissant l'interpréteur) et le code utilisateur au même niveau. L'expression, l'environnement et la continuation constituent le contexte d'exécution, ils sont généralement les arguments d'un interpréteur (`evaluate`).

Notre proposition se base sur ces principes, elle utilise le mécanisme des *reifying procedure* décrit au paragraphe suivant, pour modifier des interpréteurs stockés dans des environnements. Ces modifications se font, nous le verrons, à l'issue d'une conversation entre agents.

[16] Etymologiquement la réflexion vient du fait de retourner une image d'un objet (phénomène optique des miroirs). Plus tard, le terme signifia aussi l'état mental qui consistait à réfléchir sur soi-même. Les Anglais utilisent le mot « reflexion » pour le phénomène physique et le mot « reflection » pour l'informatique.

1.5.3. Mécanisme des reifying procedures

La réification consiste à rendre accessible dans le langage de programmation un objet du langage. Les *reifying procedures* fournissent le mécanisme nécessaire pour avoir accès au contexte d'exécution de l'interpréteur du niveau d'en dessous. Elles sont construites à l'aide de la fonction `make-reifier` qui rajoute dans l'environnement une liaison du type `(nomproc reifier param corps defenv)` :

```
(define nomproc
      (make-reifier param corps defenv))
```

L'expérimentation présentée au §3.6 illustre le mécanisme des *reifying procedures* par un exemple. Lorsqu'elles sont évaluées, leur corps est traité comme si il était une partie du code de l'interpréteur qui évalue cette *reifying procedure*. Exactement, le corps est évalué dans une extension de l'environnement de définition de la *reifying procedure* où le premier argument formel est lié à la liste des opérandes non encore évalués (arguments d'appel de la *reifying procedure*), le second est lié à l'argument correspondant à l'environnement de l'évaluateur évaluant le corps de la *reifying procedure* et le troisième est lié à l'argument correspondant à la continuation de l'évaluateur évaluant ce corps. Il est courant de trouver dans le corps des *reifying procedures* des appels du genre (`evaluate (cXr e) r k`) `X={a, ad, add, addd…}` qui permettent d'accéder aux arguments d'appels. L'idée principale est que les programmes des utilisateurs puissent avoir les mêmes accès que l'interpréteur lui-même. Grâce à cela, **les procédures implémentant l'évaluateur peuvent être accédées et modifiées par les programmes utilisateurs** de la même façon que l'environnement et les continuations. Cette propriété fait des *reifying procedures* l'outil idéal pour modifier dynamiquement nos interpréteurs. En effet, accéder au contexte d'exécution veut dire accéder à l'environnement dans lequel sont stockées les procédures définissant l'évaluateur et donc pouvoir les modifier.

Deux procédures sont également importantes, elles permettent de transformer une liaison dans l'environnement de type `compound` en `reifier` et vice et versa :

```
(define compound-to-reifier
      (lambda (compound) (cons 'reifier (cdr compound))))

(define reifier-to-compound
      (lambda (reifier) (cons 'compound (cdr reifier))))
```

Chapitre 2. Communication agent et apprentissage au meta-niveau

2.1. Présentation

La communication est à la base de notions telles que la coopération, la coordination, l'organisation sociale. Ceci, aussi bien dans les sociétés humaines que dans les systèmes informatiques et en particulier dans les SMA (Système Multi-agents). Définir et modéliser la communication a toujours été difficile. Des principes de base de la théorie de la communication de Shannon aux langages de communication agent modernes, en passant par les théories de la philosophie du langage, la communication n'a jamais été facile à formaliser. Aujourd'hui, il existe de nombreux modèles et langages de communication mais peut-on dire qu'ils sont adaptés au monde agent ou à de nouvelles formes de communication comme celles que propose le Web ? Il ne s'agit pas de prendre les langages traditionnels de communication ou même les paradigmes actuels de programmation et de les adapter au Web et aux agents. **Il s'agit de développer de nouvelles architectures et de nouveaux langages conçus pour les agents sur le Web.** En effet, les langages traditionnels sont ficelés et il est souvent très difficile de les faire évoluer. Pour être efficace, la communication doit être intrinsèque à un langage. Par exemple, en ce qui concerne l'apprentissage, il ne suffit pas simplement d'apprendre au niveau *donnée*, il faut aussi apprendre au niveau *contrôle* et au niveau *interpréteur* de ces programmes (données + contrôle). Un objet Java est capable de stocker des informations mais il ne peut difficilement modifier sa propre structure pour en intégrer de nouvelles. Notre but est de fournir un modèle qui le puisse tout en restant aussi simple que possible.

Dans un premier temps, nous tenterons de donner des éléments permettant de définir une communication, ceci en présentant promptement la communication dans les SMA, les actes de langages et certaines problématiques de la communication. Ensuite, nous regarderons l'émergence des langages de communication et un petit mot sera dit sur leur sémantique. Nous verrons également dans ce chapitre les langages et les modèles de communication existants. Un petit paragraphe explicitera aussi ce que l'on considère comme apprentissage au méta-niveau. Finalement, ce chapitre sera terminé par un scénario « idéal » de ce que pourrait être un SMA et la communication agent dans quelques années.

2.2. Qu'est-ce qu'une communication ?

2.2.1. La communication n'est pas une simple transmission

Le modèle traditionnel est la théorie de la communication de Shannon (théorie statistique mais non sémantique). Il considère quatre entités : Un *émetteur* émet un *message* sur un *canal de communication* qui est reçu par un *destinataire*. Ce message est codé via un langage particulier. La communication peut être point à point ou par diffusion [FER 95]. Ce modèle simple est le fondement de tous les formalismes de communication. Cependant, il est limité car on ne peut considérer le dialogue comme une simple transmission d'information. Par exemple, [CER 99b] nous montre que la communication sert de pilier à l'éducation. En effet, il faut considérer toutes les problématiques du dialogue et ceci quelle que soit la communication considérée : humain/agent ou agent/agent voir même humain/humain mais ceci n'est pas le sujet de notre étude. Le dialogue implique beaucoup plus de choses si on prend en compte toute sa dimension (ses finalités, ses contraintes…). De plus, il faut considérer l'état des interlocuteurs et les changements que la communication engendre chez eux. Il faut aussi considérer le sens des choses : Comment l'émetteur a-t-il l'intention de signifier quelque chose ? Comment ce qu'il émet signifie-t-il quelque chose ? Et comment le destinataire comprend ce qu'il à voulu lui signifier ? [SEA 71]

2.2.2. Communication dans les SMA

La simple mise en commun de plusieurs agents ne suffit pas à former un SMA, c'est le fait que ces agents communiquent qui le permet. C'est la communication qui permet la coopération et la coordination entre agents [FER 95]. Cette communication se fait de façon directe par envoi de message ou indirecte via l'environnement[17]. Seule la communication directe nous intéresse ici étant donné que la communication via l'environnement ne correspond pas à un dialogue. C'est donc à cette communication, intentionnelle, que nous nous référerons maintenant. Historiquement, les SMA étaient construits avec un langage de communication intégré fonctionnant de manière ad-hoc. Aujourd'hui, la communauté SMA tend à fournir des ACLs (*Agent Communication Languages*) applicables à un maximum d'interactions entre agents. En effet, fournir un ACL fort d'un point de vue sémantique donne un gros avantage pour la création et l'évolution d'un SMA [DIG 00]. Ces ACLs sont basés sur la théorie des actes de langage.

Classiquement, une conversation agent peut être modélisée par un automate à états finis où chaque état correspond à un triplet (état de l'agent 1, état de l'agent 2, acte de langage disponible) et les transitions représentent les messages échangés. Les réseaux de Pétri servent également à modéliser les conversations. Ces deux méthodes sont présentées dans [FER 95]. Cependant, ces méthodes ne se détachent pas d'une structure fixe c'est à dire que toutes les interactions sont prévisibles ie. sont inclus dans le langage reconnu, par exemple, par l'automate[18]. Les conversations ne peuvent pas évoluer de manière dynamique ce qui gène la faculté d'adaptabilité que doivent avoir les agents.

2.2.3. Communication et actes de langages

La théorie des actes de langage est issue de la philosophie du langage. Austin [AUS 70] et Searle [SEA 71] furent les premiers à la caractériser. Pour ces auteurs, parler une langue c'est accomplir des actes de langage régis par des règles qui permettent de les réaliser. Mais qu'est ce qu'un acte de langage ? C'est un moyen de structurer un dialogue. Un acte de langage est contenu dans une énonciation et vise à accomplir quelque chose. Le mot, la phrase ou le symbole qui permet de réaliser cet acte est considéré comme l'unité de communication linguistique. L'acte possède plusieurs composantes (locutoire, illocutoire, perlocutoire), il possède un but, respecte certaines règles d'expression et reflète un état psychologique particulier. Le principe d'exprimabilité [SEA 71] dit que tout ce que l'on veut signifier peut être dit. Aujourd'hui, ils sont énormément utilisés dans la communication agent sous forme de performatif représentant l'acte de langage.

2.2.4. Les problématiques de la communication agent

L'étude de plusieurs articles sur les interactions entre agents nous a permis de soulever certaines des caractéristiques qu'un langage ou un modèle de communication orienté agent doit posséder. Nous avons essayé d'en regrouper quelques-unes ici :

- Il existe plusieurs types de communications, chacun d'eux est caractérisé par deux propriétés : **les initiatives prises** et **les types d'actes de langage impliqués** [CER 99b]. Il faut donc avoir un modèle de communication qui permette de préciser ces deux propriétés.
- Si nous considèrerons le fait qu'une communication puisse avoir des effets sur les interlocuteurs (acte perlocutoire), alors il nous faut obligatoirement considérer que les agents peuvent changer de but ou de point de vue au milieu de cette communication. Ils doivent **être autonomes et doivent pouvoir s'adapte**r pendant la communication [CER 99b]. Un modèle de communication doit pouvoir **gérer les accès imprévus à l'information** [CAS 03] pour conserver l'autonomie des agents et surtout leur adaptabilité. Cette notion peut se rapprocher de **la notion d'initiative mixte** qui dit qu'à n'importe quel moment de la communication, un agent doit pouvoir intervenir pour obtenir une information, demander un service ou quoi que se soit d'autre.

[17] Environnement ici dans le sens SMA non dans le sens langage.
[18] C'est à dire que les conversations possibles sont équivalentes à un mot reconnu par l'automate à états finis.

- Ensuite, il faut considérer que des agents ou des programmes peuvent interagir entre eux ou avec des humains suivant les mêmes principes [MCC 89], [CER 00], [CAS 03]. Ce qui compte c'est **la représentation qu'un agent se fait de son interlocuteur**. Se pose aussi le problème de **l'interprétation de ses messages dans un environnement**[19] donné [CER 99b].
- Un élément très important est **la notion d'historique ou de référence au passé**. Mc Carthy [MCC 89] insiste sur le fait que les structures de données deviennent obsolètes si on se réfère directement au passé. Par exemple, un agent peut déduire qu'un passager d'avion à une réservation pour un vol si celui-ci en avait pris une dans le passé et qu'il ne l'a pas annulée au moment présent. Dans ce cas, deux informations datées évitent l'utilisation d'une structure qui contiendrait les passagers qui ont des réservations. En outre, le passé ne peut jamais être oublié, ainsi même si les effets d'un dialogue sont annulés, ils ne sont jamais oubliés !
- Comme dans toutes les communications, **le problème de la sémantique des données** que l'on échange se pose. On peut, pour cela, utiliser la notion très importante d'ontologie. Cependant, une ontologie ne doit pas être considérée par un agent comme une vérité universelle et partagée qu'il faut accepter et apprendre. Les agents communicants doivent s'entendre et construire une ontologie ensemble [CER 00]. Actuellement, la notion d'ontologie est la réponse principale à cette question mais il en existe d'autres.

2.3. Evolution des langages et sémantique

2.3.1. Emergence de la programmation des communications agents

[EIJ 02] présente le passage des paradigmes traditionnels de programmation orientés calculs à de nouveaux concepts orientés interaction et communication. L'auteur met en avant une évolution des langages vers la communication. Cette évolution est explicitée par la

Figure 4. Elle explique comment nous sommes passés, tout d'abord, de la programmation séquentielle à la programmation multi-processus, puis, de simples variables partagées (sémaphores, moniteurs) aux objets (communication par appel de méthode) ou à la programmation par contraintes, puis à la distribution de ces différents processus et enfin à la communication agent.

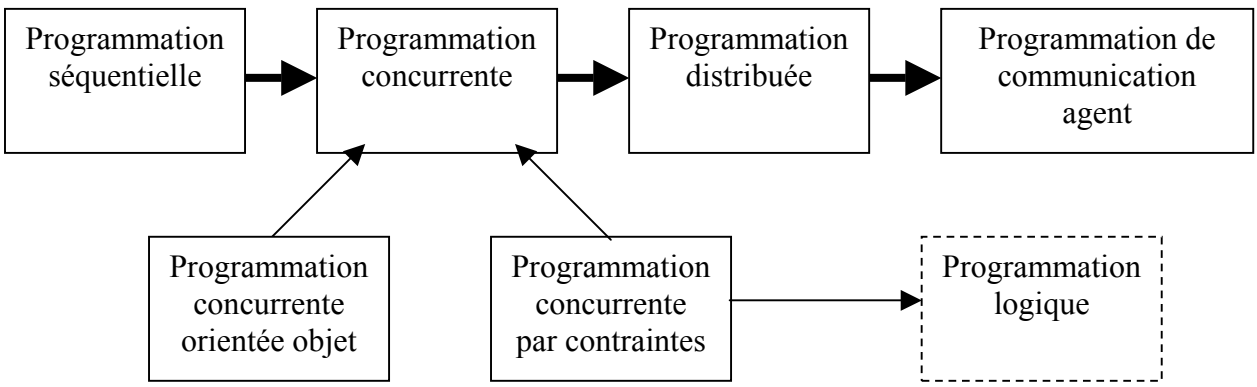


Figure 4. Emergence de la programmation orientée communication agent

Ce que tente de démontrer [EIJ 02], ce à quoi nous adhérons, c'est que les langages orientés communication agent sont plus que la suite logique des autres paradigmes, ils en sont la synthèse.

[19] A ne pas confondre avec l'environnement, notion qui correspond au monde extérieur dans un SMA.

2.3.2. Niveaux sémantiques

Les langages de communications agents sont traditionnellement divisés en trois niveaux qui possèdent chacun une sémantique différente. Le niveau *contenu*, qui correspond à l'information transmise. Ce niveau est exprimé dans un langage (KIF, Prolog, FIPA-SL, etc…) et la sémantique de ce niveau est fournie par la sémantique du langage dans lequel il est exprimé. Pour nous, il s'agit de Scheme dont la sémantique a été abordée au Chapitre 1. Les deux autres niveaux correspondent aux méta-données sur le message. Le deuxième niveau est le niveau *message*, qui correspond à ce que doit comprendre ou faire l'interlocuteur de l'information transmise. La sémantique de ce niveau est donnée par les actes de langages, ce qui sera aussi le cas pour nous. Finalement, le troisième niveau est le niveau *communication* c'est à dire les mécanismes de la communication qui est établie. Sa sémantique est donnée par les principes caractérisant la communication : synchrone/asynchrone[20], un-à-un/un-à-tous, type du canal de communication etc…

2.4. Langages et modèles de communication agent existants

KQML [FIN 97] est l'ACL le plus répandu actuellement, mais il tend à être remplacé par son successeur logique FIPA-ACL [FIP 03] qui fournit une sémantique plus forte basée sur une logique multimodale [DIG 00]. Les messages KQML ou FIPA-ACL représentent un acte de langage, ils sont indexés par un performatif et un certains nombres d'attributs (sender, receiver, language, content, ontology...). Traditionnellement, les messages KQML ou FIPA-ACL fournissent un élément qui correspond à l'ontologie utilisée dans la communication. Cela permet de rendre les ACL indépendants de n'importe quel vocabulaire et donne à l'interlocuteur un moyen d'établir la correspondance concept / signification des éléments du contenu d'un message. Dorénavant, on ne spécifie plus un ACL ad-hoc pour l'incorporer à un SMA mais on construit une ontologie qui sera passée en paramètre des messages. Notre travail propose une alternative à cet état de fait en construisant l'ontologie ainsi que l'ensemble des performatifs par le dialogue. Les ACLs reçoivent souvent la critique du manque de performatif. Notre expérimentation propose un exemple de solution à ce problème. Elle illustre une technique pour diffuser des nouveaux performatifs dans un SMA, par apprentissage direct d'agent à agent.

Face à ces langages, de nombreux modèles de communication ont été proposés. Une alternative sur la façon de considérer les agents est présentée dans [MAR 01]. Entre autres, STROBE [CER 99b] s'intéresse à de nombreux principes importants pour une communication et se base sur les trois primitives Scheme : *STReam, OBject,* et *Environment*. Il met en avant des points comme la représentation de l'interlocuteur, la conservation de l'historique d'une conversation, l'apprentissage issu de la communication, etc. Nous reviendrons souvent aux propositions de STROBE car le notre modèle s'en inspire.

2.5. Apprentissage au méta-niveau

En ce qui concerne les niveaux d'apprentissage, Scheme est très pratique car il explicite très bien les trois niveaux d'apprentissage ou de représentation des connaissances possibles que l'on retrouve dans tous les langages :

- L'apprentissage au niveau *données* consiste à affecter des valeurs à des variables déjà existantes, ou à définir de nouvelles données. Exemple : `(set! a 3)` si `a` est une variable existante dans l'environnement global ou `(define b 4)` sinon.
- L'apprentissage au niveau *contrôle* consiste à définir de nouvelles fonctions par abstractions sur celles existantes. Exemple : (`define (square x) (* x x))`.

[20] En communication synchrone, les interlocuteurs s'entendent sur le moment où s'échanger l'information. En asynchrone, ils ne s'entendent pas et l'information est provisoirement bufférisée.

- L'apprentissage au niveau *interpréteur* ou méta-niveau consiste à faire évoluer l'interpréteur Scheme. Exemple : rajouter la forme spéciale `case` ou `cond` à un interpréteur.

Notre travail propose un mécanisme d'apprentissage réalisant ces trois niveaux. En particulier le dernier car en faisant évoluer son interpréteur - un agent apprend plus qu'une simple information - il change complètement sa façon de percevoir ces informations. C'est la différence entre apprendre une donnée et apprendre à traiter une classe de données.

2.6. Scénario idéal

Imaginons maintenant un scénario « idéal », tel ceux décrits dans [IST 01], de ce que pourrait être un SMA et la communication agent dans quelques années. Il considère toutes les entités du Web comme des agents d'une même société qui peuvent communiquer les uns avec les autres naturellement et se transmettre des connaissances. Cette société pouvant s'étendre sans aucune limite. Dans ce SMA, chaque agent est initialisé avec un minimum requis de connaissances (pour interagir) ainsi qu'avec une spécialité qui le caractérise et qu'il peut transmettre aux autres. Cet agent possède un ensemble d'interpréteurs qui représentent sa connaissance et son évolution dans le temps. Il apprend tout le reste au fur et à mesure de ses communications. Il apprend même à apprendre et à enseigner ! Pour chaque agent avec qui il communique, il a une représentation spécifique de celui-ci, ce qui lui permet de tenir compte de ce qu'il apprend tout en gardant son comportement et ses croyances d'origine intactes[21]. Bien sur, les agents peuvent analyser toutes ces informations pour décider de changer ou non leur propre connaissance. D'un point de vue coopération / coordination, un agent peut demander à un autre d'évaluer pour lui tel ou tel programme et de lui renvoyer le résultat. Voir même, comme dans des architectures Grid [DER 01] [CER 03] où il est plus intéressant de déplacer le processus que les données, un agent peut transmettre à un autre un interpréteur qui lui permet d'effectuer sa tâche. Ces interpréteurs peuvent même être transmis avant une conversation comme l'on transmet aujourd'hui une ontologie. Mais vu qu'aucun interpréteur n'évolue de la même manière, puisque deux conversations ne sont jamais les mêmes, cette société d'agents évolutifs va petit à petit acquérir une pluralité de connaissances extraordinaire ! Son évolution devient totalement imprévisible et autonome. L'intégration des nouveaux agents se fait naturellement et petit à petit. Il n'est plus possible alors de prouver de manière théorique que tel ou tel agent sait accomplir une tâche ; le seul moyen est de regarder les solutions émergentes qui apparaissent lorsqu'un problème se pose.

Notre travail tente de proposer des idées pour rendre réalisable ce scénario encore utopiste aujourd'hui. Entre autres, nous allons voir comment un agent qui possède plusieurs interpréteurs de messages peut les modifier dynamiquement pour évoluer au fur et à mesure des conversations.

[21] Ou ne les changer que lorsqu'il est sûr de la validité de ce changement.

Chapitre 3. Apprentissage issu de la communication pour des agents cognitifs

3.1. Présentation

Apres le chapitre 1 qui à introduit Scheme et notre méta-évaluateur, après le chapitre 2 qui a soulevé des questions au sujet de la communication entre agents, nous pouvons maintenant décrire plus précisément notre modèle et les solutions qu'il propose. En effet, notre travail est issu de la mise en commun des deux domaines de l'informatique présentés dans les chapitres précédents. La réflexivité et les subtilités de l'évaluation sont étudiées depuis bien longtemps, par contre les interactions entre agents, domaine beaucoup plus récent, suscitent de plus en plus d'intérêts dans le monde de la recherche.

Nous verrons comment considérer nos agents comme des interpréteurs Scheme et comment gérer leurs représentations des autres. Une fois ceci présenté, nous introduirons un petit protocole de communication permettant de faire interagir ces agents. La façon dont ils sont implémentés sera aussi abordée. Enfin, nous illustrerons ce modèle par une petite expérimentation (« *toy example* ») dont le rôle consiste essentiellement à montrer que ce travail est concrètement réalisable dans des applications réelles.

3.2. Les agents comme des interpréteurs Scheme

Le modèle que nous proposons a pour caractéristique principale le fait qu'il considère les agents comme des interpréteurs Scheme[22]. Cette idée est issue de STROBE qui s'inspire de la boucle classique d'évaluation et considère les agents comme des interpréteurs de type REPL (*Read - Eval – Print - Listen*). Lors d'une communication, chaque agent exécute une boucle REPL et celles-ci sont imbriquées les unes dans les autres (cf. Figure 6). Ce principe est important car il permet de considérer les agents comme des entités autonomes dont les interactions sont dirigées par une procédure d'évaluation des messages. Notre modèle utilise Scheme aussi bien pour le contenu des messages que pour leur représentation. De cette façon, nous pouvons utiliser le même évaluateur (notre méta-évaluateur) pour évaluer le message et son contenu. Exemple d'expression Scheme représentée par des messages :

```
> (define x 2)          ⇔          > (assertion (define x 2))
: x                     ⇔          : (ack x)
> x                     ⇔          > (request x)
: 2                     ⇔          : (answer 2)
```

Ce point de vue est très intéressant car les agents bénéficient de tous les avantages liés à Scheme pour la représentation de connaissances, en particulier le modèle de contrôle fourni par les procédures et les continuations de premières classes ainsi que le modèle de mémoire fourni par les environnements de premières classes. Par exemple, STROBE propose une structure d'environnement conservant l'historique. Cette structure devient accessible à des agents représentés par des interpréteurs.

3.3. Représentation des autres

Les performatifs de STROBE se traduisent par une expression Scheme et engendrent sur l'agent récepteur un certain comportement. En particulier, la mise à jour de son « modèle du partenaire ». En effet, pour ce qui est de la représentation de l'interlocuteur, STROBE met en avant le fait qu'il faut

[22] Comme nous le verrons au paragraphe suivant nous considérons, en fait, les agents comme un ensemble d'interpréteurs.

avoir un modèle du partenaire pour pouvoir reconstruire son état interne. Ce modèle propose que chaque dialogue soit interprété dans une paire d'environnement : le premier privé, appartenant à l'agent, et le deuxième représentant le modèle du partenaire courant. C'est la notion d'Environnements Cognitifs [CER 96]. Cela permet à un agent de tenir compte ou non (selon certains critères) d'une information provenant de l'extérieur. Notre travail exploite cette notion. En fait, il se greffe dessus car, **comme le concept d'Environnement Cognitif fournit aux agents un environnement global (ou privé) et plusieurs environnements locaux de représentation de l'autre, notre proposition fournit aux agents non pas un évaluateur mais plusieurs évaluateurs dont un global (ou privé) et un pour chaque agent dont ils ont une représentation**. Ainsi, l'évaluation des messages d'une conversation se fait avec un évaluateur donné dans un environnement donné, tous les deux dédiés à la conversation. Notre travail se greffe sur ce concept d'environnement dans le sens où ces évaluateurs, pour être accessibles, doivent eux-mêmes être stockés dans ces environnements. Donc nos agents possèdent les trois attributs suivants :

- **GlobalEnv** leur environnement global
- **GlobalInter** leur interpréteur global
- **Other** = {(name, interpreter, environment)} un ensemble de triplets correspondant aux représentations des autres

Leur environnement global est privé et ne change pas (cf. note 21, page 20 et §3.7). C'est cet environnement qui est dupliqué[23] lors de l'arrivée d'une nouvelle conversation et c'est son clone, stocké dans un élément de Other, qui est modifié au fur et à mesure de la conversation. Le principe est le même pour les interpréteurs puisqu'ils sont inclus dans ces environnements. La Figure 5 illustre ces représentations.

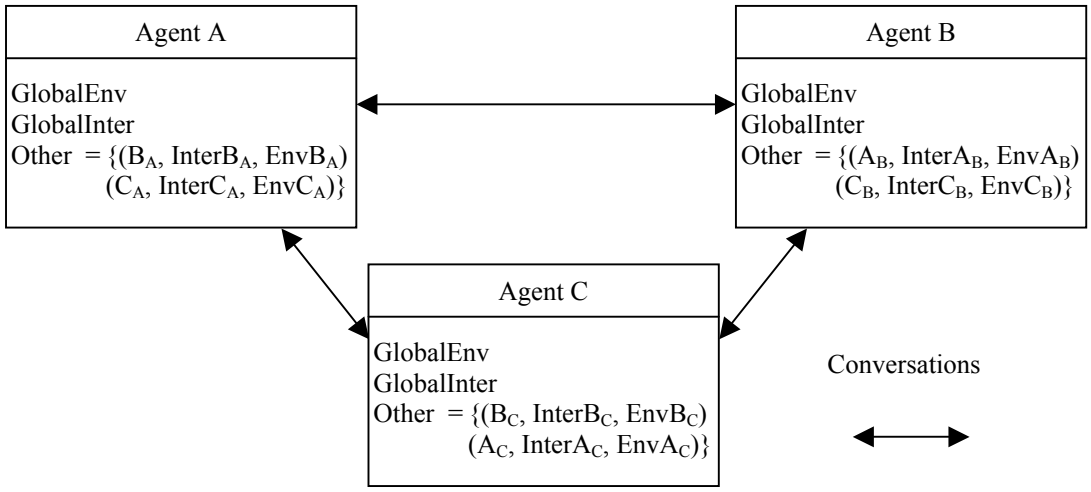


Figure 5. Les trois attributs caractérisant un agent et ses représentations

[23] Pas forcément si on considère un agent qui voudrait reprendre une conversation dans le contexte d'une autre déjà existante ou ayant existée (avec son environnement et son interpréteur). Par exemple si un agent veut continuer une conversation interrompue pour quelque raison que ce soit.

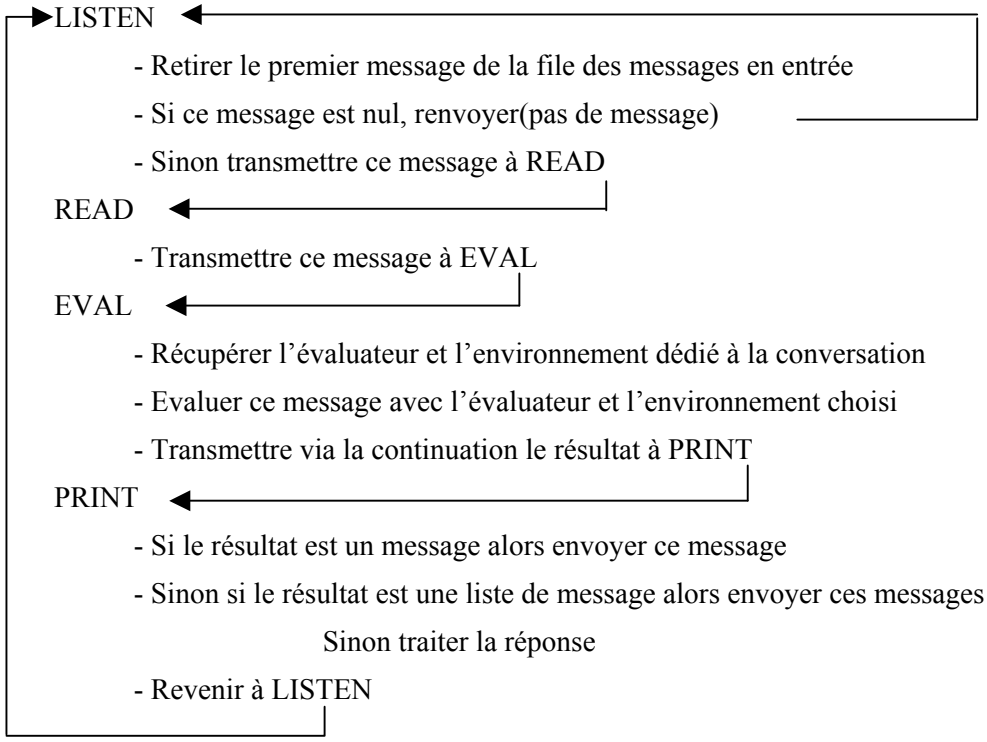


Figure 6. Boucle REPL de nos agents

3.4. Protocole de communication agent

3.4.1. Langage de communication tiré de STROBE

Les messages que nous considérons sont inspirés de la structure des messages KQML ou FIPA-ACL[24]. Ils seront notés par la suite `kqmlmsg`, ils sont de la forme :

(kqmlmsg performative sender receiver content)

Les performatifs reconnus sont : *assertion, order, request, ack, answer, executed* (cf. Tableau 2 [CER 99b]) et, nous le verrons aussi, *broadcast* qui fait l'objet de l'expérimentation. Lorsqu'un agent reçoit un message indexé par un performatif qu'il ne reconnaît pas il renvoie la liste `(no-such-performative performative)` :

- Les assertions (*assertion*) ont pour but de modifier le comportement ou les représentations de l'interlocuteur. C'est donc des messages avec pour contenu `(define…)` ou `(set!…)`. Leurs réponses sont des messages de type accusé de réception (acknowledgement (*ack*)) signifiant un succès ou une erreur.
- Les requêtes (*request*) ont pour but de connaître une représentation de l'interlocuteur, comme par exemple la valeur d'une variable ou la fermeture d'une fonction. Leur réponse (*answer*) renvoie une valeur ou une erreur.
- Les ordres (*order*) demandent à l'interlocuteur d'exécuter une procédure. Le résultat est envoyé par celui-ci par un message de type exécuté (*executed*).
- Les messages diffusion (*broadcast*) consistent à envoyer en contenu un couple `(perform, content)` qui signifie que l'interlocuteur doit envoyer un message avec comme performatif `perform` et comme contenu `content` à tous ses interlocuteurs en cours. Il n'y a pas de réponse définie pour les messages broadcast[25].

[24] Notre protocole étant énormément simplifié, nos messages ne précisent que quatre paramètres. On aurait pu cependant en implémenter plus : language, ontology, in-reply-to, reply-with, etc.
[25] Le performatif `broadcast` est en fait un méta-performatif puisque il fait appel à un autre performatif.

3.4.2. Module d'interprétation des messages

Pour que notre méta-évaluateur puisse interpréter les messages `kqmlmsg`, il faut lui ajouter un test qui les reconnaît dans la fonction `evaluate` ainsi qu'une fonction qui les traite en évaluant leur contenu `evaluate-kqmlmsg`. Ainsi la fonction `evaluate` devient :

```
(define evaluate
      (lambda (e r k)
      …
      (if (kqmlmsg? e)
           evaluate-kqmlmsg
      …
           evaluate-combination))))))))))
        e r k)))
```

Et la fonction `evaluate-kqmlmsg` contient des tests comme par exemple :

```
(define evaluate-kqmlmsg
      (lambda (e r k)
        (let ((performative (performative e))
              (sender (sender e))
              (receiver (receiver e))
              (content (content e))
              (evaluate *selection de la bonne fonction*[26]))

   (if (eq? performative 'assertion)
       (k (make-kqmlmsg 'ack receiver sender (evaluate content r return)))
    …
```

3.5. Implémentation de nos agents

3.5.1. Les objets représentants nos agents

Nous avons développé, pour les besoins de notre expérimentation, des agents capables de communiquer c'est à dire de s'échanger des messages les uns avec les autres et d'y produire des réponses significatives (ceci suivant le protocole défini plus haut). Ils ne font rien lorsqu'ils ne communiquent pas et leur autonomie se caractérise par le fait qu'ils apprennent seuls. Ils ont comme attributs name, globalEnv, globalInter, other, ainsi que deux files de messages (une en sortie et une en entrée) et une structure stockant les conversations courantes. Leur comportement consiste uniquement à appliquer la boucle REPL (Figure 6) dont les procédures sont définies plus bas. Nos agents sont en fait des objets programmés en Scheme d'après les techniques de Normark dans *Simulation of Object-Oriented and Mechanisms in Scheme* [NOR 91] présentées au §1.2.2.

La Figure 7 présente la structure exacte de nos objets, leurs attributs et leurs méthodes (constructeur, accesseur, …). On rappelle que le seul moyen de réaliser un appel de méthode est `(send instance message parameter_list)`. Le seul moyen également de changer un attribut est d'appeler son accesseur en écriture.

[26] Ici se trouve un accesseur à l'objet `simpleobj` que nous présenterons plus loin. Celui-ci renvoie l'évaluateur du récepteur du message dédié à la conversation en cours.

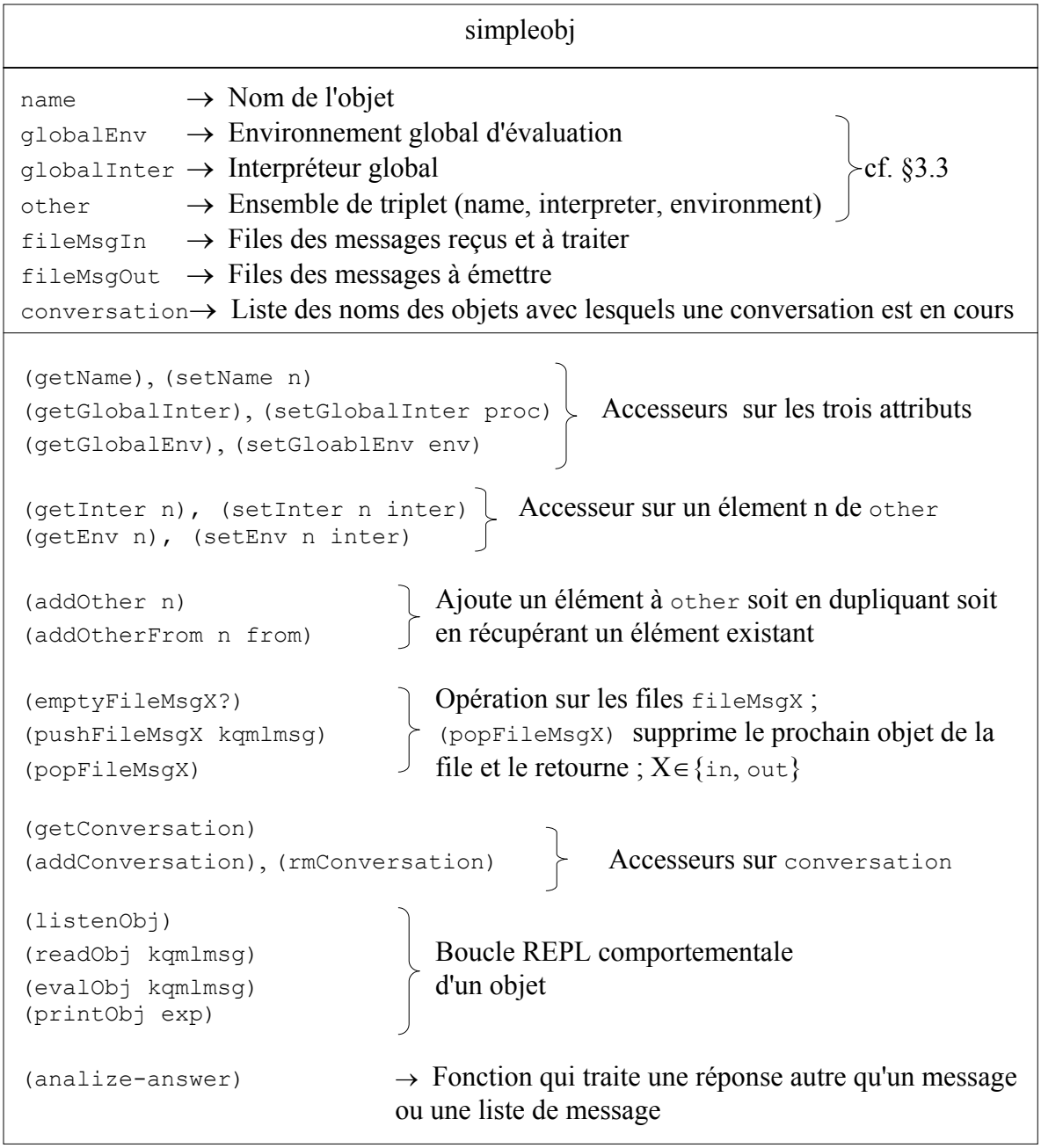


Figure 7. L'objet `simpleobj` qui définit nos agents

Remarque : Les fonctions `addOther` et `addOtherFrom` permettent respectivement de dupliquer `globalEnv` et `globalInter` au début d'une conversation ou de reprendre une conversation dans le contexte d'une autre déjà existante ou ayant existée (avec son environnement et son interpréteur). (cf. note 21, page 20)

3.5.2. Boucle d'interprétation des message

Après avoir présenté la structure de nos agents, nous allons un peu plus présenter leur comportement en détaillant les fonctions définissant leur boucle d'interprétation des messages (Figure 6). Un agent `simpleobj` ne fait rien lorsqu'il ne communique pas, il surveille (`listenObj`) en permanence l'arrivée de nouveaux messages :

```
(define listenObj
      (lambda ()
         (let ((kqmlmsg (popFileMsgIn)))
            (if (null? kqmlmsg)
                (display "No message....")
                (readObj kqmlmsg)))))
```

A l'arrivée d'un message, celui-ci est récupéré (`readObjet`) et tout de suite transféré à la fonction `evalObj` qui se charge de l'évaluer avec l'interpréteur adéquat dans l'environnement adéquat (c'est à dire ceux correspondants à cette conversation) :

```
(define readObj
      (lambda (kqmlmsg)
         (evalObj kqmlmsg)))

(define evalObj
      (lambda (kqmlmsg)
         ((getInter (send (sender kqmlmsg) 'getName ()))
          (parser kqmlmsg)[27]
          (getEnv (send (sender kqmlmsg) 'getName ()))
          printObj))))
```

La continuation de la fonction `evalObj` transmet le résultat de l'évaluation à `printObj` qui se charge d'envoyer la réponse si le résultat de l'évaluation renvoie directement de(s) message(s) ou, dans le cas où c'est un autre type de résultat, le transmet à `analyze-answer` qui le traite :

```
(define printObj
      (lambda (exp)
         (if (kqmlmsg? exp)
             (sendmessage exp)
             (if (pair? exp)
                 (if (pair? (car exp))
                     (if (eq? (car (car exp)) 'kqmlmsg)
                         (map (lambda (x)(mapper sendmessage exp)) exp)
                         (analize-answer exp))
                     (analize-answer exp))
                 (analize-answer exp)))))
```

Enfin, la fonction `analyze-answer` « traite »[28] le résultat. En particulier, si ce résultat est une liste du type `(no-such-performative performatif)`, elle regarde si elle sait comment apprendre ce performatif à son interlocuteur. Si oui, elle réalise cet apprentissage qui est préemptif sur le reste de la communication. Sinon, elle regarde si d'autres messages sont en attente dans `fileMsgOut`, qu'elle envoie.

3.6. L'expérimentation, un dialogue « professeur - élève »

3.6.1. Présentation de l'expérimentation

L'idée de notre travail est de profiter de l'apprentissage comme effet secondaire de la communication. En effet, le but de l'éducation est de faire changer d'état son interlocuteur. **Ce changement se fait après l'évaluation des nouveaux éléments apportés par la communication**. Pour cela nous utilisons les propriétés de réflexivité et de réification vues au §1.5. Grâce à notre évaluateur réflexif, nous montrons qu'un agent peut modifier sa façon de voir les choses (c'est à dire

[27] La fonction `parseur` parse le message de façon à pouvoir traiter les symbole `goodenv` et `goodeval` que l'on peut trouver dans un message et qui correspondent respectivement à l'environnement et à l'interpréteur dédiés à la conversation courante.

[28] Entre guillemet dans le sens ou il n'y à pas de traitement réel des résultats renvoyés par un agent. Par exemple si un agent nous renvoi un nombre comme résultat celui-ci n'est ni stocké ni traité. Dans notre expérimentation les agents ne font rien d'autre que communiquer.

d'évaluer des messages) en « ré-évaluant » son propre évaluateur à l'issue d'une conversation. L'expérimentation que nous présentons ici est un dialogue type « professeur-élève ». Un agent *teacher* demande à un autre agent *student* de diffuser (`kqmlmsg` de type broadcast) pour lui un message à tous ces correspondants mais le *student* ne connaît pas le performatif utilisé par le *teacher*. Par conséquent, le *teacher* transmet au *student* une série de messages (de type *assertion* et *order*) explicitant comment prendre en compte ce performatif. Il lui re-transmet alors son message d'origine qui obtient satisfaction. Le dialogue exact de l'expérimentation est décrit par la Figure 8.

TEACHER	STUDENT
Voici la définition de la procédure `square` : **(kqmlmsg 'assertion teacher student '(define (square x) (* x x)))**	Ok, je connais maintenant cette procédure : **(kqmlmsg 'ack student teacher '(*.*))**
Diffuse à tous tes correspondants : **(kqmlmsg 'broadcast teacher student '(order (square 3)))**	Désolé, je ne connais pas ce performatif : **(kqmlmsg 'answer student teacher `'(no-such-performative broadcast))**
Ok, voilà comment ajouter `broadcast` à la liste des performatifs que tu reconnais : Voilà le code que tu devras générer et ajouter à ta procédure `evaluate-kqmlmsg` : **(kqmlmsg 'assertion teacher student learn-broadcast-code-msg)**	Ok, j'ai rajouté ce code dans une variable de mon environnement : **(kqmlmsg 'ack student teacher '(*.*))**
Ensuite voila la *reifying procedure* qui te permet de changer ce code : **(kqmlmsg 'assertion teacher student learn-broadcast-msg)**	Ok, je connais maintenant cette procédure : **(kqmlmsg 'ack student teacher '(*.*))**
Exécute cette procédure : **(kqmlmsg 'order teacher student call-learn-broadcast)**	Ok, je viens de modifier mon évaluateur : **(kqmlmsg 'executed student teacher '(*.*))**
Diffuse à tous tes correspondants : **(kqmlmsg 'broadcast teacher student '(order (square 3)))**	Ok, je diffuse…

Figure 8. Dialogue « teacher - student » pour l'enseignement de *broadcast*

Remarque : Nous avons vu que la réification était la méthode pour redescendre dans les niveaux d'évaluation. Pour appliquer une *reifying procedure*, il faut donc avoir deux niveaux d'évaluation. Cela implique que toute l'expérimentation soit évaluée par notre méta-évaluateur pour réaliser le premier appel à `evaluate`, le deuxième étant effectué par l'agent lui-même lorsqu'il reçoit un message. Ainsi, la réflexivité de celui-ci devient obligatoire : notre méta-évaluateur doit pouvoir s'évaluer lui-même.

3.6.2. Détails des messages du teacher

Après toutes ces explications, nous pouvons maintenant décrire précisément comment est réalisé l'apprentissage d'un nouveau performatif. Nous allons détailler ici les messages que l'agent *teacher* envoie à l'agent *student* pour qu'il modifie son interpréteur. Le premier message est `learn-`

`broadcast-code-msg`. Cette variable correspond dans l'environnement du *teacher* au code de la nouvelle fonction `evaluate-kqmlmsg` que le *student* doit générer pour, plus tard, l'affecter à cette fonction. Ce code est construit par le *student*[29] en récupérant le corps de sa fonction `evaluate-kqmlmsg` et en lui ajoutant le code `(if (eq? performative 'broadcast)` et le traitement associé. C'est une vision constructiviste de l'apprentissage. Ainsi, l'environnement du *teacher* possède la liaison suivante :

learn-broadcast-code-msg :

```
`(define learn-broadcast-code
     (let ((newproc
        (let ((oldproc *récupération du code de evaluate-kqmlmsg*))
           *modification de oldproc pour lui ajouter
           le nouveau code (if (eq? performative…*)
        newproc))
```

Le deuxième message envoyé par le *teacher* est le plus important des trois. C'est celui qui définit la *reifying procedure* `learn-broadcast` qui va modifier l'évaluateur du *student*. L'environnement de *teacher* possède aussi la liaison :

learn-broadcast-msg :

```
`(define learn-broadcast (compound-to-reifier
                           (lambda (e r k) (evaluate (car e) r k))))
```

Lors de son évaluation `learn-broadcast` est transformée en appel d'une procédure composée `(compound)` avec comme argument : la liste de ses arguments, l'environnement dans lequel elle doit être évaluée et la continuation qu'il faut donner à cette évaluation. Le contexte d'exécution de `learn-broadcast` est alors accessible et donc modifiable ! Dans notre cas, `learn-broadcast` consiste à évaluer le `car` de la liste de ses arguments. Finalement, le *teacher* invite alors le *student* à appliquer cette fonction avec pour paramètre un appel à `set!` qui modifie `evaluate-kqmlmsg` par le code généré au préalable. La dernière liaison est donc :

call-learn-broadcast :

```
'(learn-broadcast (set! evaluate-kqmlmsg learn-broadcast-code))
```

A l'issue du traitement du dernier message, le *student* a modifié sa fonction `evaluate-kqmlmsg` et donc son interpréteur de message. Le code correspondant à cette fonction dans son environnement dédié à cette conversation est changé. Il est donc maintenant apte à traiter les messages indexés par le performatif *broadcast*. La Figure 9 donne le détail des étapes de l'évaluation de la *reifying procedure*.

[29] Il est important de remarquer que c'est le *student* qui reconstruit sa fonction à partir de celle qui existe déjà dans son environnement et non le *teacher* qui lui envoie le code de sa fonction `evaluate-kqmlmsg`. Cela permet au *student* de conserver d'autres modifications antérieures qu'il a pu avoir de sa fonction.

Quand l'appel à `learn-broadcast` est évalué...

```
(evaluate
   (learn-broadcast (set! evaluate-kqmlmsg learn-broadcast-code))
   *good env*
   *good cont*)
```
[30]

...`learn-broadcast` est changé par sa valeur dans l'environnement du *student*...

```
(evaluate
   ((reifier (e r k) (evaluate (car e) r k) *good env*)
    (set! evaluate-kqmlmsg learn-broadcast-code))
   *good env*)
```
[31]

...alors la *reifying procedure* est transformée en une `compound` procédure...

```
(evaluate
   ((compound (e r k) (evaluate (car e) r k) *good env*)
    (set! evaluate-kqmlmsg learn-broadcast-code))
   *good env*)
```

...donc `evaluate` appelle `apply-procedure`...

```
(apply-procedure
   (evaluate (car e) r k)
   ((set! evaluate-kqmlmsg learn-broadcast-code))
   *good cont*)
```

...qui rappelle `evaluate` rendant `*good env*` et `*good cont*` accessibles.

```
(evaluate
   (evaluate (car e) r k)
   ((e ((set! evaluate-kqmlmsg learn-broadcast-code)))
    (r *good env*) (k *good cont*)
    *good env*)
```

Figure 9. Etapes de l'évaluation de la *reifying procedure* `learn-broadcast`

3.7. Autonomie, raisonnement et apprentissage de nos agents

Nos agents, développés pour cette expérimentation, sont considérés comme autonomes seulement par le fait qu'ils apprennent tous seuls (sans intervention extérieure) grâce à la communication. Ils ne possèdent pas de réelle autonomie de prise de décision. C'est à dire qu'ils n'ont pas de but propre. Ils ont été simplifiés de façon à pouvoir être implémentés mais il va de soi que ce modèle s'inscrit dans un travail plus global de la communauté SMA considérant les agents comme des entités autonomes et intelligentes pouvant prendre des décisions. En outre, ce modèle doit être également complété par un modèle classique d'apprentissage et de représentation des connaissances. C'est à dire que si on apprend a un agent que "A est vrai" et que "A->B" alors il doit déduire que "B est vrai". Nous n'avons pas travaillé sur ces domaines mais là aussi nous nous inscrivons dans une logique globale de la communauté IA.

[30] *good env* and *good cont* correspondent à l'environnement et à l'interpréteur dédiés à la conversation.

[31] Comme vu précédemment une fermeture est mémorisée dans l'environnement comme `(name type parameter body defenv)`.

Par exemple, il va de soi qu'il faut ajouter à notre modèle un moyen de raisonner sur ses connaissances. Imaginons que notre *student* attaque une conversation avec un autre agent, alors il faut qu'il ait accès au nouveau performatif qu'il vient d'apprendre avec le *teacher*. Cette connaissance doit être exportée de l'environnement local (représentant le *teacher* pour le *student*) à l'environnement global (privé) du *student*. Pour ce faire, il faut une méthode "évaluant" la valeur d'une information et qui décide, ou pas, de modifier l'agent lui même (c'est-à-dire, qui décide de ré-appliquer la modification de l'interpréteur local sur l'interpréteur global). Cette méthode peut par exemple tenir compte d'un certains nombre de méta-données sur une information comme par exemple : la date, l'agent qui nous l'a donné, le degré de pertinence de cet agent, etc.

Exemple :

- Si votre ami vous donne aujourd'hui la météo de demain et que Météo France vous donne aujourd'hui la météo de demain, vous êtes plus tenté de croire Météo France.
- Par contre, si votre ami vous donne aujourd'hui la météo de demain et que Météo France vous a donné il y a une semaine la météo de demain, vous êtes plus tenté de croire votre ami.

Dans le cas de notre exemple, on pourrait considérer que tout ce qui est enseigné par le *teacher* est vrai de façon absolue et donc automatiquement le valider dans l'environnement global. Pour mettre en place ces méta données, nous pouvons imaginer avoir des environnements avec des liaisons du type : `(var val meta-data-list)` ou `meta-data-list` peut être une liste du type `(from_who, when, pertinence_degree, etc.)`. Alors, nos agents pourraient posséder des procédures traitant ces méta-données (faisant des moyennes ou d'autres calculs dessus) et ce sont ces procédures qui représenteraient leur méthodes d'apprentissage (nous restons dans un cas d'apprentissage par procédure et non par inférence). Elles illustreraient comment, à partir de ses représentations des autres, un agent peut modifier ses croyances personnelles. Soit comment, en fonction de tous ses environnements locaux, un agent peut modifier son environnement global. Ce genre de méthode préviendrait également des fausses informations provenant d'agents frauduleux : par exemple si un agent possède sur 100 environnements locaux, 90 dans lequel `plus` correspond à la procédure « + » et 10 où `plus` correspond à « – », alors il peut être capable de décider de garder la première définition et de la valider dans son environnement global.

Chapitre 4. Intérêts et extension de ces principes

4.1. Présentation

Nous avons présenté un modèle qui considère les agents comme des interpréteurs. Ensuite, nous avons illustré, par un exemple, comment une telle considération peut permettre d'apprendre au méta-niveau ou de remplacer (pourquoi pas) les traditionnelles ontologies. Nous avons étayé ceci par une expérimentation. Dans ce dernier chapitre, nous essayons de voir en quoi le modèle présenté ici peut être utile et quels sont les intérêts mis en avant par ces principes. Ce chapitre présente des travaux qui peuvent être réalisés dans le cadre de futures recherches.

Ainsi, nous étendrons dans un premier temps l'exemple déjà présenté. Puis nous analyserons également ce modèle pour voir ses avantages et ses intérêts pour le Web. Ensuite, nous regarderons comment un évaluateur non déterministe peut aider à modéliser des dialogues types *e-commerce* ou résoudre des problèmes à contraintes en réalisant la spécification dynamique d'un probleme. Finalement, nous verrons également comment rapprocher le modèle présenté dans ce mémoire à d'autres domaines tels que le *Grid Computing* par exemple.

4.2. Autres évolutions de l'interpréteur

Notre expérimentation montre comment, à l'issue d'une conversation, prendre en compte un nouveau performatif donc comment modifier la fonction d'interprétation des messages (`evaluate-kqmlmsg`) d'un agent. Mais les mêmes principes peuvent être utilisés pour modifier n'importe qu'elle partie de l'interpréteur d'un agent. Par exemple, nous aurions pu faire un dialogue qui rajoute `cond` ou `let*` au langage reconnu par notre méta-évaluateur. Voir même un agent qui enseigne à un autre comment transformer son évaluateur en évaluateur paresseux en changeant ses fonctions `evaluate` et `apply-procedure`. Grâce à ce protocole, nos agents possèdent en fait un ensemble d'interpréteurs qui représentent leurs connaissances. En effet, ils correspondent aux sous-langages que nos agents reconnaissent et donc à leurs facultés à effectuer une tâche. Comme vu dans le scénario « idéal », les agents peuvent effectuer des tâches pour d'autres ou même s'échanger leurs interpréteurs. Ceci s'inspire de l'idée énoncée dans [ABE 96] :

> *If we wish to discuss some aspect of a proposed modification to Lisp with another member of the Lisp community, we can supply an evaluator that embodies the change. The recipient can then experiment with the new evaluator and send back comments as further modifications.*

Leurs interpréteurs peuvent même être transmis avant une conversation. Le concept d'ontologie des ACLs est étendu par notre modèle par une abstraction sur l'ACL lui-même, c'est à dire que l'ontologie et l'ACL ne sont plus deux éléments distincts mais un seul et même élément. L'ontologie devient intrinsèque au langage de communication.

4.3. Intérêts pour le Web

Imaginons une société d'agents ou un SMA qui suit ce genre de modèle, alors n'importe quel agent peut apprendre quelque chose d'un autre. Si on construit un agent avec une connaissance et une spécialité le caractérisant alors cette spécialité peut se diffuser au fur et à mesure de ses communications. Pour le Web, ce genre de principe est très intéressant. Considérons un agent qui joue le rôle de serveur pour une nouvelle application orientée Web et qui utilise une série de performatifs correspondant exactement à ce qu'il doit faire. S'il est construit avec le potentiel pour enseigner ces performatifs, alors il s'intégrera très bien à une société d'agents en les enseignant au début des

communications qu'il peut avoir. La section suivante montre également comment modéliser des dialogues de type *e-commerce* qui sont très fréquents sur le Web.

En outre, un autre lien avec le Web peut être fait via le langage XML (eXtensible Markup Language). Comme nous le disions au §1.2.3, Scheme est idéal pour représenter les arbres et la structure des documents XML sont des arbres. Par exemple, [KIS 02] ou [NOR 03] présentent de nouveaux langages liant Scheme (les S-expressions) et XML. Exactement, notre idée est de considérer une DTD ou un XML-Schema et surtout un programme XSL (XSLT, XPath…)[32]comme un « interpréteur » de données XML (cf.

Figure 10). En effet, XML est un langage de construction, de sélection et surtout d'abstraction et de représentation de données composées. Un agent sur le Web représenté par un interpréteur possède alors une fonction d'interprétation des documents XML (correspondant à une DTD ou un XML-Schema ou un programme XSL) comme il possède une fonction d'interprétation des messages et il peut dynamiquement la modifier à l'issue d'une conversation. On peut imaginer alors la pluralité et l'adaptabilité de l'information sur un Web Sémantique où ces « interpréteurs » évoluent de manière dynamique et font donc évoluer également les documents XML associés.

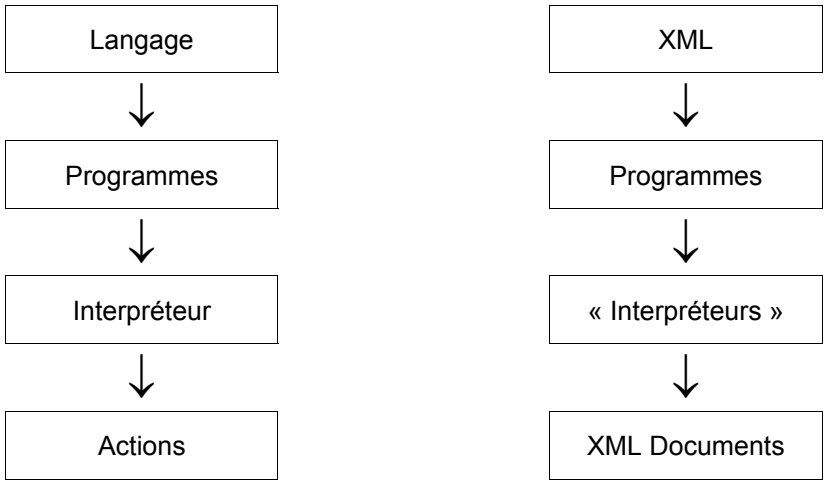


Figure 10. Analogie entre langage de programmation et XML

4.4. Dialogue et calcul non déterministe

Avant de voir comment l'évaluation non déterministe permet la spécification dynamique d'un problème, nous devons introduire la notion d'évaluateur non déterministe. C'est ce que fait le prochain paragraphe, qui s'inspire du chapitre 4 du célèbre livre de Abelson et Al. *Structure and Interpretation of Computer Programs* [ABE 96].

4.4.1. Présentation d'un évaluateur non déterministe

Donc, [ABE 96] présente les intérêts qu'un évaluateur non déterministe peut avoir pour la résolution d'un certain type de problèmes. L'idée principale est qu'avec un langage non déterministe, une expression peut avoir plusieurs valeurs possibles. Une expression représente en fait un ensemble de « mondes » possibles, chacun déterminé par un ensemble de choix. Un programme peut avoir, en évaluation non déterministe, plusieurs exécutions différentes. L'évaluation non déterministe repose sur la forme spéciale `amb`. L'expression `(amb exp1 exp2…expn)` retourne une des n expressions `expi`.

[32] Les DTD (Document Type Definition) et aujourd'hui les XML-Schema servent de grammaire de définition à un document XML. Ceci réalise la séparation entre la syntaxe et la sémantique d'un document. Les feuilles de style XSL permettent de mettre en page et de reformater un document XML. Celles-ci sont typiquement des interpréteurs.

Exemple : `(amb 1 2 3) (amb 'a 'b))` peut renvoyer six valeurs possibles `(1 a) (1 b) (2 a) (2 b) (3 a) (3 b)`. La forme `amb` donne donc plusieurs évaluations possibles à une expression.

L'intérêt d'un tel évaluateur est qu'ensuite des fonctions peuvent appeler la forme `amb` en rajoutant des contraintes (sous forme de prédicat) sur les valeurs renvoyées par `amb`. Ces contraintes s'expriment avec la forme spéciale `require` définie comme ceci :

```
(define (require p)
      (if (not p) (amb)))
```
[33]

Remarque : L'expression `amb` sans argument, `(amb)`, correspond à un échec.

L'évaluation d'une expression `amb` peut être vue comme l'exploration d'un arbre de solutions où le traitement de la fonction continue jusqu'à trouver une solution respectant toutes les contraintes et ceci tant que l'arbre complet n'a pas été parcouru. Lorsque la forme `(amb)` est évaluée cela correspond à une feuille de cet arbre, une autre branche est donc explorée. Pour reconnaître la forme spéciale `amb`, il faut modifier l'évaluateur classique pour qu'il la traite de façon particulière. Un peu comme en Prolog, on peut demander l'ensemble des solutions d'une expression logique une à une avec un évaluateur non déterministe, la forme `try-again` permet de voir la prochaine évaluation à succès d'une fonction appelant `amb`. La boucle classique d'interprétation est modifiée en évaluation non déterministe pour tenir compte de ces retours en arrière (*backtracks*).

Exemple : Considérons la fonction `(un-element-de liste)` qui renvoie la valeur d'un élément d'une liste donnée. Alors son évaluation est :

```
> (un-element-de '(a b c))
: b
> try-again
: a
> try-again
: c
> try-again
: no more values
```

4.4.2. Exemple de programme non déterministe

Avant tout, regardons comment est écrit le corps de la fonction `un-element-de` de la section précédente :

```
(define (un-element-de liste)
      (require (not (null? liste)))
      (amb (car liste) (un-element-de (cdr liste))))
```

La contrainte dans ce cas est que la liste ne soit pas vide. Lorsqu'elle est `null?` alors `(amb)` est évaluée et l'évaluateur non déterministe passe à la branche suivante de l'arbre des solutions.

Regardons maintenant un exemple de programme non déterministe un peu plus compliqué [ABE 96]. Il s'agit d'un problème type puzzle logique :

Baker, Cooper, Fletcher, Miller, et Smith vivent à différents niveaux d'une maison de cinq étages. Baker n'habite pas tout en haut de la maison. Cooper n'habite pas tout en bas. Fletcher n'habite ni tout en haut, ni tout en bas de la maison. Miller habite dans un des étages au-dessus de Cooper. Smith n'habite pas à un étage adjacent à Fletcher. Fletcher n'habite pas à un étage adjacent à Cooper. Où habite chacun d'eux ? Nous pouvons déterminer qui habite à chaque étage en énumérant toutes les possibilités et en leur appliquant les différentes contraintes ; cela donne la fonction suivante :

[33] La forme `(if cond exp)` renvoie la valeur de `exp` si `cond` est vrai. Elle ne renvoie pas de valeur sinon.

```
(define (multiple-dwelling)
  (let ((baker (amb 1 2 3 4 5))
        (cooper (amb 1 2 3 4 5))
        (fletcher (amb 1 2 3 4 5))
        (miller (amb 1 2 3 4 5))
        (smith (amb 1 2 3 4 5)))
    (require (distinct? (list baker cooper fletcher miller smith)))
    (require (not (= baker 5)))
    (require (not (= cooper 1)))
    (require (not (= fletcher 5)))
    (require (not (= fletcher 1)))
    (require (> miller cooper))
    (require (not (= (abs (- smith fletcher)) 1)))
    (require (not (= (abs (- fletcher cooper)) 1)))
    (list (list 'baker baker) (list 'cooper cooper)
     (list 'fletcher fletcher) (list 'miller miller) (list 'smith smith))))
```

Cette fonction renvoie, avec un évaluateur non déterministe :

```
((baker 3) (cooper 2) (fletcher 4) (miller 5) (smith 1))
```

4.4.3. Permettre la spécification dynamique par le dialogue

Voyons en quoi de tels principes sont intéressants pour nos agents. Nos agents sont considérés comme des interpréteurs Scheme donc ils peuvent très bien être vus comme des interpréteurs non déterministes ; C'est à dire reconnaître et traiter les formes `amb`, `require`, `try-again`... Ceci est, en effet, intéressant pour eux car ils pourraient alors résoudre des programmes comme ceux vus dans la section précédente. Mais ce n'est pas le seul intérêt. Pour faire le lien avec notre travail orienté communication agent, le point le plus intéressant est que nos agents peuvent construire de tels programmes au fur et à mesure d'un dialogue et appliquer ensuite ces programmes pour donner une réponse ou accomplir une tâche pour un autre agent. Les contraintes, définissant un programme non déterministe, peuvent être explicitées au fur et à mesure du dialogue en utilisant les outils présentés plus haut, de modification dynamique des corps des fonctions et de la façon dont elles sont interprétées.

Considérons par exemple un dialogue type *e-commerce*, de recherche de billet de train comme celui présenté dans [MAR 01]. Un billet est caractérisé par une ville de départ, une ville de destination, un prix, une date. Un agent *SNCF* est sollicité par un agent *client* pour lui faire des propositions de billets. Le dialogue en situation réelle pourrait être :

a. *Client* : Je voudrais un billet de Montpellier à Paris.
b. *SNCF* : Pour quand ?
c. *Client* : Demain avant 10H du matin. Pourriez vous me faire une proposition ?
d. *SNCF* : Voilà, train 34170, départ demain 9H30, Montpellier en direction de Paris, 150€.
e. *Client* : Vous n'auriez pas à moins de 100 € ?
f. *SNCF* : Voilà, train 34730, départ demain 8H41, Montpellier en direction de Paris, 95€.
g. *Client* : Une autre proposition s'il vous plait ?
h. *SNCF* : Voilà, train 34392, départ demain 9H15, Montpellier en direction de Paris, 98€.
i. *Client* : Ok, celui-ci me va.

Nous pouvons constater que les interactions **a**, **b**, **c** et **e** consistent à établir les contraintes sur la sélection du billet. Les interactions **d**, **f** et **h** sont des applications de la fonction de recherche de billet avec différentes contraintes. L'interaction g correspond à une demande du *client* pour avoir une autre réponse, soit, pour explorer une autre branche de l'arbre des solutions. La Figure 11 illustre la façon dont ce dialogue peut être traduit en expression Scheme pour être réalisé par nos agents. L'agent *client* transmet ses requêtes sous forme de `require` et de `try-again`. L'agent *SNCF* commence, au début de la conversation, la construction d'une nouvelle fonction `find-ticket` qu'il modifie et exécute au fur et à mesure de cette conversation.

CLIENT	SNCF
Je voudrais un billet de Montpellier à Paris **(require (eq? depart montpellier))** **(require (eq? dest paris))**	Début de la construction de `find-ticket` : **(define (find-ticket)** **(let** **((depart(amb *ens-ville*))** **(dest (amb *ens-ville*))** **(prix (amb *ens-prix*)** **(date (amb *ens-date*)** **(require** **(not (eq? depart dest)))** **(require** **(eq? depart montpellier))** **(require** **(eq? dest paris))** **(list (list 'depart depart)** **(list 'destination dest)** **(list 'prix prix)** **(list 'date date))))** *Pour quand ?*
Demain avant 10H du matin **(require (< date *demain10H*))** *Pourriez vous me faire une proposition ?* **(find-ticket)**	Modification de la fonction `find-ticket` en lui ajoutant la nouvelle contrainte. Puis exécution de cette fonction. **((depart montpellier) (destination paris) (prix 150) (date *dem9H30*))** *Voilà, train 34170, départ demain 9H30, Montpellier en direction de Paris, 150€.*
Vous n'auriez pas à moins de 100 € ? **(require (< prix 100))** **(find-ticket)**	idem. **((depart montpellier) (destination paris) (prix 95) (date *dem8H41*))** *Voilà, train 34730, départ demain 8H41, Montpellier en direction de Paris, 95€.*
Une autre proposition s'il vous plait ? **(try-again)**	Execution de `find-ticket` : *Voilà, train 34392, départ demain 9H15, Montpellier en direction de Paris, 98€* **((depart montpellier) (destination paris) (prix 98) (date *dem9H15*))**
Ok, celui-ci me va	

Figure 11. Dialogue entre l'agent *client* et l'agent *SNCF* pour la recherche de billet

Remarque : On peut imaginer qu'avant un dialogue de ce genre, nos deux agents se sont entendus sur un ensemble de performatifs à utiliser (et aient modifié leurs interpréteurs respectifs) pour cette conversation exemple `perform-require` pour exprimer les contraintes, `perform-try-again` pour exprimer la recherche d'une autre solution, `perform-proposition` pour exprimer une proposition et `order` pour exécuter la fonction `find-ticket`.

Cette idée est très intéressante car c'est le dialogue qui construit le calcul à effectuer et non le contraire. C'est un scénario qu'on pourrait retrouver dans de nombreuses applications *e-commerce* ou d'autres du même genre où des agents doivent construire un programme ensemble pour trouver une solution. Ce principe gère en outre les interventions mixtes dans la discussion qui, comme nous l'avons vu au §2.2.4, est une des problématiques de la communication agent. Ici, à tout moment, l'agent *client* peut demander une exécution de `find-ticket` même si toutes les contraintes ne sont pas établies et l'agent *SNCF* peut demander à tout moment l'expression d'une contrainte qui pour lui serait obligatoire (ex : ville de départ et d'arrivée).

L'approche classique de construction d'un programme (la plus fréquemment utilisée dans le génie logiciel), qui consiste à spécifier un programme avant de le coder, est changée par une approche de spécification dynamique pendant la construction d'un programme. C'est-à-dire que la spécification et la réalisation sont faits en même temps.

4.5. Rapprochement avec d'autres modèles et domaines

4.5.1. Scheduling Algorithm

Le travail présenté ici s'inscrit dans la même lignée que le langage C+C [CER 00] ou que le travail proposé dans [GUE 99] qui propose de fournir aux agents un algorithme d'organisation temporel (S*cheduling Algorithm*) dynamique. Un S*cheduling Algorithm* a pour rôle de traiter les messages reçus par l'agent de manière asynchrone et d'y produire une réponse adéquate. Il évolue en même temps que le comportement de l'agent. C'est lui qui, utilisant le maximum d'éléments de l'environnement d'un agent, produit la réponse à un message. Là où un objet répond à un appel de méthode sans réfléchir ou contester, un agent a un comportement qui dépend de son S*cheduling Algorithm* qui lui permet d'être autonome et de décider si et quand il consacre du temps aux autres. Là où l'objet est implémenté de manière fixe, l'agent possède un S*cheduling Algorithm* qui varie avec le temps. Nos interpréteurs réflexifs peuvent inclure un *Scheduling Algorithm* et modifier leur comportement REPL (cf. Figure 12).

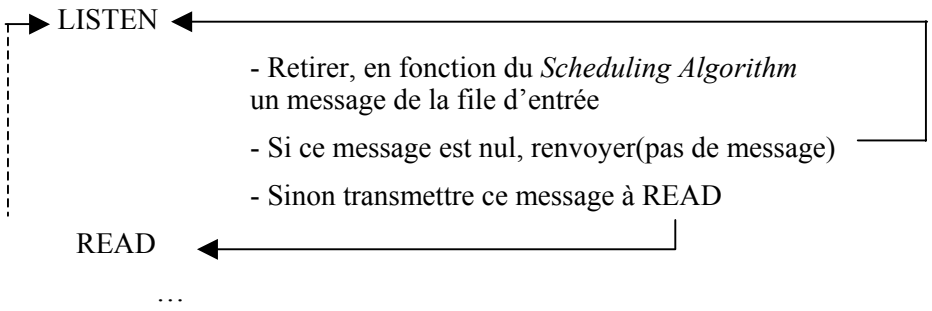


Figure 12. Intégration d'un *Scheduling Algorithm* à la boucle REPL de nos agents

4.5.2. Grid Computing

Les principes présentés dans ce mémoire peuvent être également utiles pour d'autres types de scénarios. Ils peuvent être rapprochés du Grid de deux façons. La première est le fait que des agents peuvent s'échanger des interpréteurs, ce qui revient à déplacer des programmes sur un réseau plutôt que des données, ce qui est un des principes du Grid. Le deuxième rapprochement peut être fait de la façon suivante : Imaginons qu'au lieu de transférer la définition de la procédure `square`, l'agent *teacher* transfére à l'agent *student* la définition d'une procédure implémentant un algorithme optimisé de résolution d'un problème. Par exemple, la fonction calculant la factorielle d'un nombre de façon

itérative[34] ou la fonction calculant le nombre de Fibonacci en utilisant la mémoization[35] (cette fonction, `memo-fib`, transforme le calcul exponentiel d'un nombre de Fibonacci en un calcul linéaire). Vous trouverez ces fonctions en Annexe B. Dans ce cas, l'agent *student*, après avoir appris le performatif `broadcast`, peut jouer le rôle de serveur de grille de calcul (*Grid Computing*) en procédant à une sélection des agents participants à un lourd calcul utilisant par exemple `memo-fib` de la manière suivante : Il peut demander à tous ses correspondants (interlocuteurs courants) de réaliser pour lui un calcul simple d'un nombre de Fibonacci et de lui renvoyer le résultat. Après la réception des résultats il peut classer ses correspondants dans trois catégories :

- Ceux qui n'ont pas réussi à réaliser le calcul n'ont pas de fonction de calcul d'un nombre de Fibonacci.
- Ceux qui ont répondu aussi vite que ce que lui-même l'aurait fait possèdent sans doute une bonne fonction de calcul d'un nombre de Fibonnaci (`memo-fib`).
- Ceux qui ont répondu mais lentement ne possèdent probablement pas une bonne fonction de calcul d'un nombre de Fibonnaci (`memo-fib`).

Dès lors, l'agent est capable de sélectionner[36] un ensemble d'agents et de leur demander de participer à un gros calcul, utilisant les nombres de Fibonacci. Il peut même enseigner aux autres agents la bonne version de la fonction implémentant cet algorithme. Cette idée est effectivement particulièrement intéressante pour le *Grid Computing* mais peut être également utilisée dans des protocoles de communication type *contract net*.

4.5.3. MadKit

Dans le cadre de futurs travaux, il serait intéressant d'intégrer notre modèle à la plate-forme multi-agents MadKit [MAD 00]. En effet, cette plate-forme écrite en Java, accepte la définition d'agents sous forme de scripts écrits dans d'autres langages, dont Scheme[37]. Le lien entre Java et Scheme est accompli grâce au langage Kawa [BOT 98]. Nos agents pourraient donc être implémentés en tant qu'agent MadKit et donc profiter de l'architecture AGR (Agent/Groupe/Rôle) sur laquelle la plate-forme est basée. MadKit propose également une classe de messages (ActMessage) qui peut être spécialisée pour réaliser les `kqmlmsg` que nous utilisons. Notre modèle pourrait alors être plus développé et serait facilement représentable dans une plate-forme agent consistante. Et donc facilement exportable. Le « toy example » présenté plus haut illustre la viabilité de notre méthode. Une exportation de celle-ci dans MAdKit illustrerait sa « scalabilité »[38] en augmentant ses performances.

4.5.4. Alternative à la reflexivité

Nos travaux, ont également consistés à chercher un moyen de rendre plus utilisable ce modèle implémenté. Le paragraphe précédent illustrait comment le faire avec MadKit, ce paragraphe propose d'utiliser, pour modifier dynamiquement des environnements lexicaux plutôt que la réflexivité, les environnements de premières classes proposés par Queinnec et De Roure [QUE 96]. La notion d'environnement de première classe permet de réifier des environnements lexicaux en des valeurs stockables, échangeables voire analysables. Cette technique permet d'expliquer un certain nombre de phénomènes comme la notion d'environnement extensible qu'il serait intéressant de relier à nos travaux pour en améliorer la vitesse.

---

[34] Nous l'avons dit plus haut, Scheme permet d'utiliser des processus récursifs et des processus itératifs. La version itérative du calcul de la factorielle d'un nombre est beaucoup moins coûteuse en temps et espace que la version récursive. C'est pourquoi nous pouvons parler d'implémentation optimisée.

[35] La mémoization (également appelé tabulation) est une technique qui permet à une procédure d'enregistrer, dans une table locale, les valeurs qui ont été précédemment calculées. Cette technique peut faire une grosse différence lors de l'exécution d'un programme. Lorsqu'une procédure mémoizée est invitée à calculer une valeur, elle contrôle d'abord cette table pour voir si la valeur y est déjà, si oui, elle la retourne sinon, elle la calcule normalement et la rajoute à cette table.

[36] On peut imaginer également d'autres critères de sélection comme des statistiques, des tests sur différentes données etc…

[37] Un agent Scheme dans MadKit est implémenté par les trois procédures `(activate)`,`(live)` et `(end)`

[38] C'est-à-dire son potentiel pour être étendu à des applications plus importantes.

Conclusion

Nous avons essayé, à travers ce mémoire, de récapituler et synthétiser notre travail. Nous espérons avoir réussi à rallier le lecteur à notre point de vue pour qu'il voit Scheme comme un langage puissant et utile pour des domaines complexes et en vogue comme les interactions entre agents. Nous avons essayé de montrer que la mise en commun de différents domaines (langage + communication agents) pouvait faire émerger de nouvelles idées. Notre travail a consisté à présenter une méthode d'apprentissage pour les agents cognitifs issue de la communication, cet apprentissage pouvant se faire par communication simple (niveau *donnée* et *contrôle*) ou par modification interne de l'agent (niveau *interpréteur*). Cette méthode est basée sur le fait que nos agents sont considérés comme des interpréteurs. Nous avons essayé, en dernière partie, de montrer comment une telle considération peut s'étendre à d'autres domaines et en quoi elle est consistante pour de futurs travaux. La spécification dynamique obtenue en utilisant notre modèle montre que celui-ci est centré sur le concept de dialogue. Ceci s'inscrit dans une logique, que nous partageons, qui positionne le dialogue au centre des intérêts de l'informatique de demain. Tel qu'il l'est dans les sociétés humaines.

Il est très intéressant de voir que si les agents interprètent de façon dynamique les messages qu'ils reçoivent, ils deviennent adaptables et, sans aucune intervention extérieure, peuvent communiquer avec des entités qu'ils n'ont jamais rencontrées auparavant. En outre, comme leur évaluateur est modifié pour acquérir une connaissance, il pourrait l'être aussi pour apprendre à enseigner une connaissance. Dans ce cas, le transfert du savoir dans une société d'agents deviendrait exponentiel au fur et à mesure des communications.

Ce travail n'a pas simplement pour vocation de proposer un artéfact de plus de programmation à ajouter aux agents mais l'idée est plutôt de montrer une architecture faite de manière simple et utilisable (comme le montre les exemples), d'évolution autonome des agents dans une société. Le dialogue permet la construction de programme et la résolution de problème, mais surtout le dialogue permet de définir comment se comporter dans la suite du dialogue !

Bibliographie

[ABE 96] Abelson H., Sussman G.J., Sussman J., *Structure and Interpretation of Computer Programs*, Second Edition, MIT Press, Cambridge, Massachusetts, 1996.

[AUS 70] Austin J.L., *Quand dire c'est faire*, Edition du seuil, Paris, 1970.

[BOS 96] Bosch F., *Mémoire de DEA*, LIRMM – Université Montpellier II, Montpellier, 1996.

[BOT 98] Bothner P., "Kawa Compiling Dynamic Languages to the Java VM", Cygnus Solutions, *Proceedings of the USENIX 1998 Technical Conference, FREENIX Track*, www.gnu.org/software/kawa/, 1998.

[CER 96] Cerri S.A., "Cognitive Environments in the STROBE Model", Presented at *EuroAIED: the European Conference in Artificial Intelligence and Education*, Lisbon, Portugal, 1996.

[CER 97] Cerri S.A., "A simple language for generic dialogues: "speech acts" for communication", in the Proceedings of JFLA97, Journées francophones des langages applicatifs, Collection Didactique de l'INRIA, Marc Gengler et Christian Queinnec (eds.), pages 145-168, 1997.

[CER 99a] Cerri S.A., "Dynamic Typing and Lazy Evaluation as Necessary Requirements for Web Languages", *European Lisp User Group Meeting*, Amsterdam, June 1999.

[CER 99b] Cerri S.A., "Shifiting the Focus from Control to communication: The STReams OBject Environments (STROBE) model of communicating agents", In Padget, J.A. (ed.) *Collaboration between Human and Artificial Societies, Coordination and agent-Based Distributed Computing*, Berlin, Heidelberg, New York: Springer-Verlag, Lecture Notes in Artificial Intelligence, pp 71-101, 1999.

[CER 00a] Cerri S.A., “The Computer is the Web”, *Advanced School WITREC2000*, Montpellier, Mai 2000.

[CER 00b] Cerri S.A., Sallantin J., Castro E., Maraschi D., “Steps towards C+C: a Language for Interactions”, In Cerri, S. A., Dochev, D. (eds), *AIMSA2000: Artificial Intelligence: Methodology, Systems, Applications*, Berlin, Heidelberg, New York: Springer Verlag, Lecture Notes in Artificial Intelligence, pp 33-46, 2000.

[CER 02] Cerri S.A., “Human an Artificial agent’s Conversations on the Grid”, *Electronic Workshops in Computing (eWiC)*, 1st LEGE-WG International Workshop on Educational Models for Grid Based Services, Lausanne, Switzerland, 16 September 2002.

[CHA 96] Chazarain J., *Programmer avec Scheme*, International Thomson Publishing France, Paris, 1996.

[DER 01] De Roure D., Jennings N., Shadbolt N., “Research Agenda for the Semantic Grid: A Future e-Science Infrastructure”, In Report commissioned for *EPSRC/DTI Core e-Science Programme,* University of Southampton, UK, 2001.

[DIG 00] Dignum F., Greaves M., “Issues in agent Communication: An introduction”, Dignum F. and Greaves M. (Eds.): *agent Communication, LNAI 1916*, pp1-16, Springer-Verlag Berlin Heidelberg, 2000.

[EIJ 02] Van Eijk R.M., “Semantics of agent Communication: An Introduction”, M. d’Inverno et al. (Eds.): UKMAS 1996-2000, LNAI 2403, pp.152-168, Springer-Verlag Berlin Heidelberg, 2002.

[FER 95] Ferber J., *Les Systemes Multi-agents, vers une intelligence collective*, InterEditions, Paris, 1995.

[FIN 97] Finin T., Labrou Y., “A Proposal for a new KQML Specification”, *Computer Science and Electrical Engineering Department*, University of Maryland, Baltimore, February 1997.

[FIP 03] *FIPA-ACL Specification*, Foundation for Intelligent Physical agents - Agent Communication Language, http://www.fipa.org/specs/fipa00061/

[FRI 92] Friedman D.P., Jefferson S., “A Simple Reflective Interpreter”, *IMSA'92: International Workshop on Reflection and Meta-Level Architecture*, Tokyo, 1992.

[GUE 02] Guerin F., “Specifying agent Communication Languages”, *Ph.D. Thesis*, Dept. of Electrical and Electronic Engineering, Imperial College, 2002.

[GUE 99] Guessoum Z., Briot JP., “From Active Objects to Autonomous agents”, IEEE Concurrency, vol. 7-3, pp. 68-76, 1999.

[GOU 02] Gouaich A., Guirad Y., “{Movement, Interaction, Calculus}*: an algebraic environment for distributed and mobile calculus”, In *NAISO Congress on Autonomous Intelligent Systems,* Deakin University, Waterfront Campus, Geelong, Australia, February 2002.

[IST 01] ISTAG, “Scenarios for Ambient Intelligence in 2010”, Final Report Compiled by K. Ducatel, M. Bogdanowicz, F. Scapolo, J. Leijten & J-C. Burgelman, IPTS-Seville, February 2001.

[KIS 02] Kiselyov O., “XML, Xpath, XSLT implementations as SXML, SXPath, and SXSLT”, *International Lisp Conference: ILC2002*, San Francisco, CA, October 2002.

[MAD 00] Gutknecht O. Ferber J. "Madkit: a generic multi−agent platform" *4th International Conference on Autonomous Agents*, Barcelona, Spain, June 2000, http://www.madkit.org

[MAR 01] Maraschi D., Cerri S.A., “The relations between Technologies for Human Learning and agents”, In procedings of the *AFIA 2001 Atelier: Methodologies et Environments pour les Systèmes Multi-agents,* Grenoble: Leibniz-Imag, pp. 61-73, 2001.

[MAS 90] Masini G., Napoli A., Colnet D., Léonard D., Tombre K., *Les langages à objets*, InterEditions, Paris, 1990.

[MCC 89] McCarthy J., “Elephant 2000: A Programming Language Based on Speech Acts”, *Unpublished draft* Stanford University, http://www-formal.stanford.edu/jmc/elephant.pdf, 1989.

[MIT 02] Chris H. and the MIT Scheme Team, *MIT Scheme Reference Manual*, MIT, Cambridge, Massachusetts, 2002.

[MOR 98] Moreau L., Ribbens D., Gribomont P., “Advanced Programming Techniques Using Scheme”, *JFLA98: Journées Francophones des Langages Applicatifs*, 1998.

[NOR 91] Normark K., “Simulation of Object-Oriented and Mechanisms in Scheme”, *Institute of Electronic Systems*, Aalborg University, Denmark, 1991.

[NOR 03] Normark K., “Web Programming in Scheme with LAML”, Submit *to J. Functional Programming*, April 2003.

[QUE 94] Queinnec C.,*Les langages LISP*, InterEditions, Paris, 1996.

[QUE 96] Queinnec C., De Roure D., “Sharing Code through First-class Environments”, Proceedings of *ICFP'96-- ACM SIGPLAN International Conference on Functional Programming*, Pages 251-261, Philadelphia, Pennsylvania, USA, May 1996.

[QUE 00] Queinnec C., “The Influence of Browsers on Evaluators or Continuations to Program Web Servers”, *ICFP'00,* Montreal, Canada, 2000.

[R5R 98] Kelsey R., Clinger W., Reers J., (Eds) Abelson and Al., Revised 5 Report on the Algorithmic Language Scheme, http://www.swiss.ai.mit.edu/~jaffer/r5rs_toc.html, February 1998.

[SEA 71] Searle J., *Les actes de langages, essai de philosophie du langage*, Herman Editeur, Paris, 1971.

[SER 02] Serrano M., *Bigloo homepage*, http://www-sop.inria.fr/mimosa/fp/Bigloo/, 2002.

[SIM 92] Simmons II J.W., Jefferson S., Friedman D.P., “Language Extension via First-class Interpreters”, *Computer Science Department*, Indiana University, September 1992.

[SUS 75] Sussman G.J., Steele G.L. Jr., “Scheme: An Interpreter for Extended Lambda Calculus”, *MIT AI Lab,* AI Lab Memo AIM-349, December 1975.

Remerciements

Je tiens à remercier, bien sur, mon encadrant, Stefano Cerri (LIRMM), pour son soutien et pour la façon dont il m’a motivé. Egalement, l’équipe E-dialogue du LIRMM et Nailah pour l’organisation du séminaire. Mes amis et camarades de DEA avec qui j’ai co-habité pendant ce stage et qui m’ont aidé en me donnant leurs points de vue. Ainsi que toutes les personnes étudiants ou professeurs avec qui j’ai discuté.

J’adresse également un remerciement particulier à Christian Queinnec (LIP6) qui a bien voulu prendre un peu de son temps pour relire mon mémoire et me proposer son aide.

Et bien sur, un merci tout particulier à Isabelle et Olivier qui m’ont aidé dans les relectures francophones et anglophones. Ma famille qui a fait de moi ce que je suis.

Annexes

Annexe A. Implémentation de l'expérimentation

Le modèle présenté ici a été sujet à une petite implémentation, non compète encore à ce jour, mais d'ores et déjà fonctionnelle. Elle est disponible en ligne sur http://www.lirmm.fr/~jonquet. Son intérêt principal est de montrer que la technique présentée ici est viable, c'est à dire qu'elle marche. Ce n'est qu'une version simple qui nécessite encore beaucoup de travail, soyez donc indulgent avec elle. Cette expérimentation est facilement compréhensible grâce à Scheme qui nous permet d'expliciter tous ces concepts compliqués de manière assez simple. Elle fut réalisée avec MIT Scheme 7.7.1, norme R5RS [R5R 98] disponible sur http://www.swiss.ai.mit.edu/projects/scheme/mit/.

Les fichiers

simple.scm :

Ce fichier correspond au code du méta-évaluateur que nous utilisons pour toute l'expérimentation. Cet évaluateur est issu de l'article de Jefferson et Friedman « *A Simple Reflective Interpreter* » que nous avons modifié au fur et à mesure de nos besoins.

```
;;; simple.scm - Clement JONQUET - DEA Informatique LIRMM - Mai 2003 ;;;

;;; evaluate Procedure and sub-procedures ;;;

(define  evaluate
  (lambda (e r k)
    ;(display "evaluate de : ")(display e)(newline)
         ((if (constant? e)
        evaluate-constant
        (if (variable? e)
            evaluate-variable
        (if (quasiquote? e)
            evaluate-quasiquote
        (if (if? e)
            evaluate-if
        (if (assignment? e)
            evaluate-assignment
        (if (define? e)
            evaluate-define
          (if (let? e)
            evaluate-let
        (if (begin? e)
            evaluate-begin
        (if (abstraction? e)
            evaluate-abstraction
            evaluate-combination)))))))))
     e r k)))

(define evaluate-constant
  (lambda (e r k)
    (k (constant-part e))))

(define evaluate-variable
  (lambda (e r k)
    (get-pair e r
              (lambda (success-pair) (k (cdr success-pair)))
              (lambda () (wrong "symbol not bound: " e)))))

(define evaluate-quasiquote
  (lambda (e r k)
    (define (quasifoo exp)
      (if (null? exp)
```

```
         ()
         (if (pair? exp)
             (if (unquote? (car exp))
                (cons (evaluate (car (cdr (car exp))) r return) (quasifoo (cdr
exp)))
                (if (unquote-splicing? (car exp))
                    ; Be carefull...it's not good
                    (append (evaluate (car (cdr (car exp))) r return) (quasifoo (cdr
exp)))
                    (cons (quasifoo (car exp)) (quasifoo (cdr exp)))))
             exp)))
    (k (quasifoo (car (cdr e))))))

(define  evaluate-if
  (lambda (e r k)
    (evaluate (test-part e) r
              (lambda (v) (if v (evaluate (then-part e) r k) (evaluate (else-part e)
r k))))))

(define evaluate-assignment
  (lambda (e r k)
    (evaluate (value-part e) r
              (lambda (v)
               (get-pair (id-part e) r
                        (lambda (success-pair)
                          (set-cdr! success-pair v) (k (void)))
                        (lambda ()
                          (set-cdr! global-env (cons (cons (id-part e) v) (cdr global-
env)))
                               (k (void))))))))

(define evaluate-define
  (lambda (e r k)
    (if (pair? (car (cdr e)))
        ; for call like (define (namefunc param) (body))
      (evaluate (list 'lambda (cdr (car (cdr e))) (value-part e)) r
                (lambda (v)
                  (get-pair (car (car (cdr e))) r
                            ; If the symbol already exists is value is changed
                            (lambda (success-pair)
                             (set-cdr! success-pair v) (k (void)))
                            ; Else a new bindings is added to r
                            (lambda ()
                             (set-cdr! r (cons (cons (car (car (cdr e))) v) (cdr r)))
                             (k (void))))))
      ; for a call like (define var val) or (define func (lambda param body))
      (evaluate (value-part e) r
                (lambda (v)
                  (get-pair (id-part e) r
                            ; If the symbol already exists is value is changed
                            (lambda (success-pair)
                             (set-cdr! success-pair v) (k (void)))
                            ; Else a new bindings is added to r
                            (lambda ()
                             (set-cdr! r (cons (cons (id-part e) v) (cdr r)))
                             (k (void)))))))))

(define evaluate-let
  (lambda (e r k)
    (evaluate-sequence (let-body e) (extend r (let-var e) (let-init e r)) k)))

(define evaluate-begin
  (lambda (e r k)
    (evaluate-sequence (cdr e) r k)))

(define evaluate-abstraction
  (lambda (e r k)
    (k (make-compound (formals-part e) (body-part e) r))))
```

```
(define evaluate-combination
  (lambda (e r k)
    (evaluate (operator-part e)
              r
              (lambda (proc)
               (if (reifier? proc)
                   ((reifier-to-compound proc) (operands-part e) r k)
                   (evaluate-operands (operands-part e) r
                                      (lambda (args) (apply-procedure proc args
k))))))))

(define evaluate-operands
  (lambda (operands r k)
    (if (null? operands)
       (k '())
       (evaluate (car operands) r
                 (lambda (v) (evaluate-operands (cdr operands) r (lambda (w) (k (cons
v w)))))))))

(define evaluate-sequence
  (lambda (body r k)
    (if (null? (cdr body))
       (evaluate (car body) r k)
       (evaluate (car body) r (lambda (v) (evaluate-sequence (cdr body) r k))))))

;;; apply-procedure and apply-primitive functions ;;;

(define apply-procedure
  (lambda (proc args k)
    (if (compound? proc)
       (evaluate-sequence (procedure-body proc)
                          (extend (procedure-environment proc) (procedure-parameters
proc) args) k)
       (k (apply-primitive (procedure-name proc) args)))))

(define apply-primitive
  (lambda (name args)
        (if (eq? name 'car)
           (car (1st args))
       (if (eq? name 'cdr)
           (cdr (1st args))
       (if (eq? name 'cons)
           (cons (1st args) (2nd args))
       (if (eq? name 'set-car!)
           (set-car! (1st args) (2nd args))
       (if (eq? name 'set-cdr!)
           (set-cdr! (1st args) (2nd args))
       (if (eq? name 'assq)
           (assq (1st args) (2nd args))
       (if (eq? name 'memq)
           (memq (1st args) (2nd args))
       (if (eq? name 'null?)
           (null? (1st args))
       (if (eq? name '=)
           (= (1st args) (2nd args))
       (if (eq? name 'eq?)
           (eq? (1st args) (2nd args))
       (if (eq? name 'equal?)
           (equal? (1st args) (2nd args))
       (if (eq? name 'newline)
           (newline)
       (if (eq? name 'write)
           (write (1st args))
       (if (eq? name 'display)
           (display (1st args))
       (if (eq? name 'read)
           (if (null? args) (read) (read (1st args)))
```

```
       (if (eq? name '+)
           (+ (1st args) (2nd args))
       (if (eq? name '-)
           (- (1st args) (2nd args))
       (if (eq? name '*)
           (* (1st args) (2nd args))
       (if (eq? name '/)
           (/ (1st args) (2nd args))
       (if (eq? name '<)
           (< (1st args) (2nd args))
       (if (eq? name '>)
           (> (1st args) (2nd args))
       (if (eq? name '<=)
           (<= (1st args) (2nd args))
       (if (eq? name '>=)
           (>= (1st args) (2nd args))
       (if (eq? name 'symbol?)
           (symbol? (1st args))
       (if (eq? name 'list)
           args
       (if (eq? name 'append)
           (append (1st args) (cdr args))
       (if (eq? name 'pair?)
           (pair? (1st args))
       (if (eq? name 'list?)
           (list? (1st args))
       (if (eq? name 'length)
           (length (1st args))
       (if (eq? name 'reverse)
           (reverse (1st args))
       (if (eq? name 'procedure?)
           (procedure? (1st args))
       (if (eq? name 'eof-object?)
           (eof-object? (1st args))
       (if (eq? name 'close-input-port)
           (close-input-port (1st args))
       (if (eq? name 'open-input-file)
           (open-input-file (1st args))
       (if (eq? name 'void)
           (void)
           "Shouldn't Happen!"
           ))))))))))))))))))))))))))))))))))))))))))))

;;; Environment's traitement ;;;

(define extend
  (lambda (r ids vals)
    (if (null? ids)
       r
       (extend (cons (cons (car ids) (car vals)) r) (cdr ids) (cdr vals)))))

(define get-pair
  (lambda (id r success failure)
    (find-pair id r success (lambda () (find-pair id global-env success
failure)))))

(define find-pair
  (lambda (elt alist success failure)
    ((lambda (assq-result)
       (if (null? assq-result)
          (failure)
          (success assq-result))) (assq elt alist))))

(define empty-env (list (cons 'zeo 'zeo)))

(define global-env (list (cons 'zeo 'zeo)))

(define primitive-identifiers
```

```
  (lambda ()
    '(car cdr cons set-car! set-cdr! assq memq null? = eq? equal? newline write
display read + - * / < > <= >= symbol? list append pair? list? length reverse eof-
object? close-input-port open-input-file void)))

(define initialize-global-env
  (lambda ()
    (set! global-env
         (extend empty-env (primitive-identifiers) (mapper make-primitive
(primitive-identifiers))))
    (void)))

;;; List utilities and other usefull procedures ;;;

(define 1st (lambda (l) (car l)))
(define 2nd (lambda (l) (car (cdr l))))
(define 3rd (lambda (l) (car (cdr (cdr l)))))
(define 4th (lambda (l) (car (cdr (cdr (cdr l))))))
(define 5th (lambda (l) (car (cdr (cdr (cdr (cdr l)))))))
(define test-tag (lambda (tag) (lambda (e) (if (pair? e) (eq? (car e) tag) #f))))

(define void (lambda () (cons '* '*)))

(define  wrong
  (lambda (message object)
    (display "Error: ") (display message) (display object) (newline)))

(define mapper
  (lambda (f l)
    (if (null? l)
       '()
       (cons (f (car l)) (mapper f (cdr l))))))

(define return (lambda (x) x))

;;; Procedures use by the evaluating procedures ;;;

(define let-body (lambda (exp) (cdr (cdr exp))))

(define let-var (lambda (e) (mapper (lambda (x) (car x)) (car (cdr e)))))

(define let-init
  (lambda (e r)
    (mapper (lambda (x) (evaluate (car (cdr x)) r return)) (car (cdr e)))))

(define compound-to-reifier (lambda (compound) (cons 'reifier (cdr compound))))

(define reifier-to-compound (lambda (reifier) (cons 'compound (cdr reifier))))

(define make-compound (lambda (formals body r) (list 'compound formals body r)))

(define make-primitive (lambda (op) (list 'primitive op)))

(define make-reifier (lambda (formals body r) (list 'reifier formals body r)))

(define procedure-parameters 2nd)
(define procedure-body 3rd)
(define procedure-environment 4th)
(define procedure-name 2nd)

(define constant-part (lambda (e) (if (quote? e) (2nd e) e)))
(define test-part 2nd)
(define then-part 3rd)
(define else-part 4th)
(define id-part 2nd)
(define value-part 3rd)
(define formals-part 2nd)
(define body-part (lambda (e) (cdr (cdr e))))
```

```
(define operator-part 1st)
(define operands-part cdr)

;;; Syntax's procedures ;;;

(define compound? (test-tag 'compound))

(define primitive? (test-tag 'primitive))

(define reifier? (test-tag 'reifier))

(define variable? (lambda (e) (if ((test-tag 'define) e) #f (symbol? e))))

(define if? (test-tag 'if))

(define assignment? (test-tag 'set!))

(define define? (test-tag 'define))

(define let? (test-tag 'let))

(define begin? (test-tag 'begin))

(define abstraction? (test-tag 'lambda))

(define constant? (lambda (e) (if (pair? e) (quote? e) (if (symbol? e) #f #t))))

(define quote? (test-tag 'quote))

(define quasiquote? (test-tag 'quasiquote))

(define unquote? (test-tag 'unquote))

(define unquote-splicing? (test-tag 'unquote-splicing))

;;; Read-Eval-Print loop and how load a file ;;;

(define openloop
  (lambda (read-prompt write-prompt)
    (newline)(display read-prompt)
    (evaluate (read) global-env (lambda (v)
                              (newline)(display write-prompt)
                              (if (eq? v (void))
                                  "Nothing will be displayed"
                                  (write v)) (newline) (openloop read-prompt write-
prompt)))))

(define loadfile
  (lambda (file)
    ((lambda (port)
       ((lambda (loop)
         (set! loop
              (lambda (v)
                (if (eof-object? v)
                    (close-input-port port)
                    (evaluate v global-env (lambda (ignore) (loop (read port)))))))
         (loop (read port)))
       '*))
     (open-input-file file))))

;;; The booting process ;;;

(define boot-flat (lambda () (initialize-global-env) (openloop "0< " "0> ")))

(define boot-tower
  (lambda ()
    (initialize-global-env)
    (loadfile this-file-name)
```

```
    (set-cdr! global-env (cons (cons 'global-env global-env) (cdr global-env)))
    (openloop "0> " "0: ")))

(define my-boot-tower
  (lambda ()
    (initialize-global-env)
    (loadfile this-file-name)
    (set-cdr! global-env (cons (cons 'global-env global-env) (cdr global-env)))))

(define this-file-name "simple.scm")

;;; Printing procedures ;;;

(define print-env
  (lambda (env)
    (mapper
     (lambda (x)
       (if (list? x)
          (begin (display "-- ")(display (1st x))(display " : ")
               (if (eq? 'global-env (1st x))
                   (display "...")
                   (if (eq? 'compound (2nd x))
                      (begin (display "compound ")(display (3rd x))(display " ")
                           (display (4th x))(display " ")(display "..."))
                   (if (eq? 'primitive (2nd x))
                      (begin (display "primitive ")(display (3rd x)))
                   (if (eq? 'reifier (2nd x))
                      (begin (display "reifier ")(display (3rd x))(display " ")
                           (display (4th x))(display " ")(display "..."))
                      (display (cdr x))))))
               (newline))
          (begin (display "-- ")(display (car x))(display " ")(display (cdr
x))(newline))))
     env)
    (display "-----------------Environement Printed")))

(define print-proc
  (lambda (proc)
    (begin (display (1st proc))(display " ")
          (display (2nd proc))(display " ")
          (display (3rd proc))(display " ")(display "...")(newline))))
```

lazysimple.scm :

Ce fichier présente une partie de notre travail qui a consisté à écrire un module pour rendre notre méta-évaluateur paresseux. Ceci n'entre pas directement en relation avec nos agents mais constitue une partie de notre travail sur l'évaluation en Scheme. De plus, c'est utilisé pour notre expérimentation. Le chargement de ce fichier redéfinit une série de méthodes du fichier simple.scm.

```
;;; lazysimple.scm - Clement JONQUET - DEA Informatique LIRMM - Mai 2003

;;; Procedure's re-definitions ;;;

(set! evaluate-if
      (lambda (e r k)
       (if (evaluate (test-part e) r force-it)
           (evaluate (then-part e) r k)
           (evaluate (else-part e) r k))))

(set! evaluate-combination
      (lambda (e r k)
       (evaluate (operator-part e)
                 r
                 (lambda (proc)
                   (if (reifier? proc)
                      ((reifier-to-compound proc) (operands-part e) r k)
```

```
                    (apply-procedure (force-it proc) (operands-part e) r k))))))

(set! apply-procedure
      (lambda (proc args r k)
       (if (compound? proc)
           (force-it (evaluate-sequence
                     (procedure-body proc)
                     (extend (procedure-environment proc)
                            (procedure-parameters proc)
                            (delayed-args args r))
                     k))
           (k (apply-primitive (procedure-name proc) (arg-values args r))))))

(set! print-env
      (lambda (env)
       (mapper
        (lambda (x)
          (if (list? x)
              ; Si c'est une liste
              (begin (display "-- ")(display (1st x))(display " : ")
                    (if (eq? 'global-env (1st x))
                       (display "...")
                    (if (eq? 'compound (2nd x))
                       (begin (display "compound ")(display (3rd x))(display " ")
                             (display (4th x))(display " ")(display "..."))
                    (if (eq? 'primitive (2nd x))
                       (begin (display "primitive ")(display (3rd x)))
                    (if (eq? 'reifier (2nd x))
                       (begin (display "reifier ")(display (3rd x))(display " ")
                             (display (4th x))(display " ")(display "..."))
                    (if (eq? 'thunk (2nd x))
                       (begin (display "thunk ")(display (3rd x))(display " ")
                             (display (4th x))(display " ")(display "..."))
                       (display (cdr x)))))))
                    (newline))
              ; Sinon c'est une paire pointée
              (begin (display "-- ")(display (car x))(display " ")
                    (display (cdr x))(newline))))
        env)
       (display "-----------------Environement Printed")))

(set! my-boot-tower
      (lambda ()
       (begin (initialize-global-env)
              (loadfile "simple.scm")
              (loadfile this-file-name)
              (set-cdr! global-env (cons (cons 'global-env global-env) (cdr global-
env)))
              (void))))

(set! this-file-name "lazysimple.scm")

;;; New procedures and thunk implementation ;;;

(define delayed-args
  (lambda (args r)
    (mapper (lambda (x) (delay-it x r)) args)))

(define arg-values
  (lambda (args r)
    (mapper (lambda (x) (evaluate x r force-it)) args)))

(define force-it
  (lambda (obj)
    (if (thunk? obj)
      ;(evaluate (thunk-exp obj) (thunk-env obj) force-it)
      ;obj)))
       (let ((result  (evaluate (thunk-exp obj) (thunk-env obj) force-it)))
```

```
         (set-car! obj 'evaluated-thunk)  ; if it's a thunk it's become an en
evaluated-thunk
         (set-car! (cdr obj) result)      ; replace exp with its value
         (set-car! (cdr (cdr obj)) '())   ; forget unneeded env
         result)
       (if (evaluated-thunk? obj)
           (thunk-value obj)
           obj))))

(define delay-it
  (lambda (e r)
    (list 'thunk e r)))

(define thunk? (test-tag 'thunk))

(define thunk-exp (lambda (thunk) (car (cdr thunk))))
(define thunk-env (lambda (thunk) (car (cdr (cdr thunk)))))
(define thunk-cont (lambda  (thunk) (car (cdr (cdr (cdr thunk))))))

(define evaluated-thunk? (test-tag 'evaluated-thunk))
(define thunk-value (lambda (evaluated-thunk) (car (cdr evaluated-thunk))))
```

evaluators.scm :

Ce fichier contient le module d'interprétation des kqmlmsg ainsi que la fonction d'évaluation utilisée par le *teacher*.

```
;;; evaluators.scm - Clement JONQUET - DEA Informatique LIRMM - Mai 2003

;;; evaluate procedure ;;;

(set! evaluate
      (lambda (e r k)
      ;(display "evaluate de : ")(display e)
      ;(newline)
      ((if (constant? e)
           evaluate-constant
       (if (variable? e)
           evaluate-variable
       (if (quasiquote? e)
           evaluate-quasiquote
       (if (if? e)
           evaluate-if
       (if (assignment? e)
           evaluate-assignment
       (if (define? e)
           evaluate-define
       (if (let? e)
           evaluate-let
       (if (begin? e)
           evaluate-begin
       (if (kqmlmsg? e)
           evaluate-kqmlmsg
       (if (abstraction? e)
           evaluate-abstraction
           evaluate-combination))))))))))
       e r k)))

;;; kqmlmsg's evaluation ;;;
;;; kqmlmsg's type is (kqmlmsg performative sender receiver content)

(define evaluate-kqmlmsg ())
(set! evaluate-kqmlmsg
      (lambda (e r k)
       (let ((performative (performative e))
             (sender (sender e))
             (receiver (receiver e))
             (content (content e))
```

```
            (evaluate (send (receiver e) 'getInter (list (send (sender e) 'getName
()))))))

         (if (eq? performative 'assertion)
             (k (make-kqmlmsg 'ack receiver sender (evaluate content r return)))
         (if (eq? performative 'request)
             (k (make-kqmlmsg 'answer receiver sender (evaluate content r return)))
         (if (eq? performative 'order)
             (k (make-kqmlmsg 'executed receiver sender (evaluate content r
return)))
         (if (eq? performative 'ack)
             'ok
         (if (eq? performative 'answer)
             (if (equal? content (void)) 'ok (k (list (evaluate content r return)
performative sender receiver)))
         (if (eq? performative 'executed)
             (if (equal? content (void)) 'ok (k (list (evaluate content r return)
performative sender receiver)))
             ;Sinon
              (k (make-kqmlmsg 'answer receiver sender `'(no-such-performative
,performative)))))))))))

; New evaluate procedure which can evaluate broadcast typed kqmlmsg ;;;

(define  evaluate-with-kqmlmsg-and-broadcast ())
(set! evaluate-with-kqmlmsg-and-broadcast
      (lambda (e r k)
       ((if (constant? e)
            evaluate-constant
        (if (variable? e)
            evaluate-variable
        (if (quasiquote? e)
            evaluate-quasiquote
         (if (if? e)
            evaluate-if
        (if (assignment? e)
            evaluate-assignment
        (if (define? e)
            evaluate-define
        (if (let? e)
            evaluate-let
        (if (begin? e)
            evaluate-begin
        (if (kqmlmsg? e)
            evaluate-kqmlmsg-and-broadcast
        (if (abstraction? e)
            evaluate-abstraction
            evaluate-combination)))))))))
        e r k)))

(define evaluate-kqmlmsg-and-broadcast ())
(set! evaluate-kqmlmsg-and-broadcast
      (lambda (e r k)
       (let ((performative (performative e))
             (sender (sender e))
             (receiver (receiver e))
             (content (content e))
             (evaluate (send (receiver e) 'getInter (list (send (sender e) 'getName
()))))))

         (if (eq? performative 'assertion)
             (k (make-kqmlmsg 'ack receiver sender (evaluate content r return)))
         (if (eq? performative 'request)
             (k (make-kqmlmsg 'answer receiver sender (evaluate content r return)))
         (if (eq? performative 'order)
             (k (make-kqmlmsg 'executed receiver sender (evaluate content r
return)))
         (if (eq? performative 'ack)
```

```
            'ok
        (if (eq? performative 'answer)
            (if (equal? content (void)) 'ok (k (list (evaluate content r return)
performative sender receiver)))
        (if (eq? performative 'executed)
            (if (equal? content (void)) 'ok (k (list (evaluate content r return)
performative sender receiver)))
        (if (eq? performative 'broadcast)
            ;Renvoie une liste de kqmlmsg
            (k (mapper (lambda (x) (make-kqmlmsg (car content) receiver x (car (cdr
content)))) (send receiver 'getConversation ())))
            ;Sinon :
            (k (make-kqmlmsg 'answer receiver sender `('no-such-performative
,performative))))))))))))
```

simpleobj.scm :

Ce fichier définit les agents que nous avons présentés dans l'article. Ce sont en fait des objets programmés en scheme à l'aide des techniques présentées dans l'article de Normark, « *Simulation of Object-Oriented and Mechanisms in Scheme »* [NOR 91]. Ces agents sont assez simples mais peuvent être modifiés par la suite pour les agrémenter un peu.

```
;;; simpleobj.scm - Clement JONQUET - DEA Informatique LIRMM - Mai 2003 ;;;

;;; The simpleobj obect definition's procedure ;;;

(define (simpleobj)
  (let ((super (new-part object))
        (self ()))

    ;Attributs
    (let ((name 'n)                    ; name
          (globalEnv global-env)       ; global environment
          (globalInter evaluate)       ; global interpreter (frame in the env)
          (other ())                   ; set of frame (name, interpreter, environment)
          (fileMsgIn ())               ; reception Message's file (FIFO)
          (fileMsgOut ())              ; outgoing Message's file (FILO and FIFO)
          (conversation ()))           ; list of current conversation

      ;Constructor
      (define (new1 n)
       (set! name n))

      ;Accessor
      (define (getName) name)
      (define (setName n) (set! name n))
      (define (getGlobalInter) globalInter)
      (define (setGlobalInter function) (set! globalInter function))
      (define (getGlobalEnv) globalEnv)
      (define (setGlobalEnv env) (set! globalEnv env))
      (define (getInter n) (car (cdr (assq n other))))
      (define (getEnv n) (car (cdr (cdr (assq n other)))))
      (define (setInter n inter)
       (define (foo list)
         (if (eq? n (car (car list)))
             (set-car! list ('n inter (getEnv n)))
             (foo (cdr list)))
         (foo other)))
      (define (setEnv n env)
       (define (foo list)
         (if (eq? n (car (car list)))
             (set-car! list ('n (getInter n) env))
             (foo (cdr list)))
         (foo other)))

      ;File 's operations
      (define (emptyFileMsgIn?)
       (null? fileMsgIn))
```

```
      (define (pushFileMsgIn kqmlmsg)
       (set! fileMsgIn (cons kqmlmsg fileMsgIn)))
      (define (popFileMsgIn)
       (let ((fileMsgIn-1 (reverse fileMsgIn)))
         (if (null? fileMsgIn-1)
             ()
             (begin (set! fileMsgIn (reverse (cdr (reverse fileMsgIn))))
                    (car fileMsgIn-1)))))

      (define (emptyFileMsgOut?)
       (null? fileMsgOut))
      (define (pushFileMsgOut kqmlmsg)
       (set! fileMsgOut (cons kqmlmsg fileMsgOut)))
      (define (popFileMsgOut)
       (let ((fileMsgOut-1 (reverse fileMsgOut)))
         (if (null? fileMsgOut-1)
             ()
             (begin (set! fileMsgOut (reverse (cdr (reverse fileMsgOut))))
                    (car fileMsgOut-1)))))

      ;Add an other
      (define (addOther n)
       (set! other (cons (list (send n 'getName ()) globalInter globalEnv) other)))
      (define (addOtherFrom n from)
       (set! other (cons (list n (getInter from) (getEnv from)))))

      ;Conversation's traitement
      (define (getConversation) conversation)
      (define (addConversation obj)
       (set! conversation (cons obj conversation)))

      ;REPL loop and message's traitement
      (define (listenObj)
       (begin (display "Object ")(display (getName)) (display " in LISTEN : ")
              (let ((kqmlmsg (popFileMsgIn)))
               (if (null? kqmlmsg)
                   (begin (display "No message....")(newline))
                   (begin (display "One message....")(newline)(readObj kqmlmsg))))))

      (define  (readObj kqmlmsg)
       (begin (display "Object ")(display (getName)) (display " in READ : ")
              (print-kqmlmsg kqmlmsg)(newline)
              (evalObj kqmlmsg)))

      (define (evalObj kqmlmsg)
       (begin (display "Object ")(display (getName)) (display " in EVAL : ")
              (display "...")(newline)
              ((getInter (send (sender kqmlmsg) 'getName ()))
              (parser kqmlmsg)
              (getEnv (send (sender kqmlmsg) 'getName ()))
              printObj)))

      (define (printObj exp)
       (if (kqmlmsg? exp)
           ;If the result is one message
           (begin (display "Object ")(display (getName)) (display " in PRINT 1 : ")
                  (print-kqmlmsg exp)(newline)
                  (sendmessage exp))
           (if (pair? exp)
              (if (pair? (car exp))
                  (if (eq? (car (car exp)) 'kqmlmsg)
                    ; If the result is a list of message
                    (mapper (lambda (x)
                              (begin (display "Object ")(display (getName)) (display
" in PRINT 2 : ")
                                     (print-kqmlmsg x)(newline)
                                     (mapper sendmessage exp)))
                            exp)
```

```
                    ; Else
                    (analize-answer exp))
                (analize-answer exp))
            (analize-answer exp))))

      (define (analize-answer exp)
       (begin (display "Object ")(display (getName)) (display " in PRINT 3 : ")
              (if (pair? (car exp))
                 (if (eq? (car (car exp)) 'no-such-performative)
                     (let ((sender (car (cdr (cdr exp))))
                          (receiver (car (cdr (cdr (cdr exp)))))
                          (performative (car (cdr (car exp)))))
                      (if (eq? performative 'broadcast)
                          (get-pair 'learn-broadcast-code-msg (getEnv (send sender
'getName ()))
                           (lambda (result1)
                            (let ((msg1 (make-kqmlmsg 'assertion receiver sender (car
(cdr result1)))))
                              (print-kqmlmsg msg1)(newline)(sendmessage msg1)
                              (get-pair 'learn-broadcast-msg (getEnv (send sender
'getName ()))
                           (lambda (result2)
                            (let ((msg2 (make-kqmlmsg 'assertion receiver sender (car
(cdr result2)))))
                              (print-kqmlmsg msg1)(newline)(sendmessage msg2)
                              (get-pair 'call-learn-broadcast (getEnv(send sender
'getName ()))
                           (lambda (result3)
                            (let ((msg3 (make-kqmlmsg 'assertion receiver sender (car
(cdr result3)))))
                              (print-kqmlmsg msg1)(newline)(sendmessage msg3)))
                           (lambda () 'notpossible))))
                           (lambda () 'notpossible))))
                           (lambda () (display "No method existing to teach you this
performative")))
                          (display "Unknown performative and no method to teach
it")))
                     (process-answer exp))
                 (process-answer exp))))

      (define (process-answer exp)
       (begin (display "Received and analize result : ") (display (car
exp))(newline)
              (if (emptyFileMsgOut?)
                 (display "No more messages...")
                 (sendmessage (popFileMsgOut)))))

      (define (parser kqmlmsg)
       (let ((content (content kqmlmsg)))
         (define (foo c)
           (if (pair? c)
              (cons (foo (car c)) (foo (cdr c)))
              (if (eq? c 'goodenv)
                  `',(getEnv (send (sender kqmlmsg) 'getName ()))
                  (if (eq? c 'goodeval)
                     `',(getInter (send (sender kqmlmsg) 'getName ()))
                     c))))
         `(kqmlmsg ,(performative kqmlmsg) ,(sender kqmlmsg) ,(receiver kqmlmsg)
,(foo content))))

      (define (dispatch message)
       (if (eq? message 'new1) new1
       (if (eq? message 'getName) getName
       (if (eq? message 'setName) setName
       (if (eq? message 'getGlobalInter) getGlobalInter
       (if (eq? message 'setGlobalInter) setGlobalInter
       (if (eq? message 'getGlobalEnv) getGlobalEnv
       (if (eq? message 'setGlobalEnv) setGlobalEnv
```

```
       (if (eq? message 'getInter) getInter
       (if (eq? message 'setInter) setInter
       (if (eq? message 'getEnv) getEnv
       (if (eq? message 'setEnv) setEnv
       (if (eq? message 'emptyFileMsgIn) emptyFileMsgIn
       (if (eq? message 'pushFileMsgIn) pushFileMsgIn
       (if (eq? message 'popFileMsgIn) popFileMsgIn
       (if (eq? message 'emptyFileMsgOut) emptyFileMsgOut
       (if (eq? message 'pushFileMsgOut) pushFileMsgOut
       (if (eq? message 'popFileMsgOut) popFileMsgOut
       (if (eq? message 'addOther) addOther
       (if (eq? message 'addOtherFrom) addOtherfrom
       (if (eq? message 'getConversation) getConversation
       (if (eq? message 'addConversation) addConversation
       (if (eq? message 'listenObj) listenObj
       (if (eq? message 'lifeLoop) lifeLoop
           (method-lookup super message)
       ))))))))))))))))))))))))

      (set! self dispatch))

    self))

;;; Object fonctions ;;;

(define (new-part class) (class))

(define (display-result x) (display "Le resultat de l'évaluation est : ")(display
x)(newline))

(define (method-lookup object selector)
  (if (compound? object)
      (object selector)
      (begin (display "Inapropriate object in method-lookup : ") (display object)
(newline))))

(define (send object msge args)
  (let ((method (method-lookup object msge)))
    (if (compound? method)
       (apply-procedure method args return)
       (if (null? method)
           (display "Message not understood")
           (begin (display "Inapropriate result of method-lookup : ")
                 (display object) (newline))))))

(define (object)
  (let ((super ())
       (self ()))
    (define (dispatch message)
      ())
    (set! self dispatch)
self))

;;; Converstation ;;;

(define (initialize-conversation obj1 obj2)
  (begin (send obj1 'addOther (list obj2))
        (send obj1 'addConversation (list obj2))
        (send obj2 'addOther (list obj1))
        (send obj2 'addConversation (list obj1))
        (void)))

(define (re-take-conversation obj1 obj2)
  (begin (send obj1 'addOther (list obj2 obj2))
        (send obj2 'addOther (list obj1 obj1))
        (void)))

(define (conversationLoop obj1 obj2)
```

```
  (let ((bool #t))
    (define (oneLoopMore)
      (begin (newline)
           (display "Do you like to continue conversation loop ? -- Y or N : ")
           (let ((answer (read)))
             (if (eq? answer 'N)
                (begin (set! bool ())(display "The end")(newline)())
                (if (eq? answer 'Y)
                    (while bool)
                    (oneLoopMore))))))
      (define (while test)
       (begin (send obj1 'listenObj ())
              (send obj2 'listenObj ())
              (oneLoopMore)))
      (while bool)))

;;; KQML like message ;;;

(define (sendmessage kqmlmsg)
  (send (receiver kqmlmsg) 'pushFileMsgIn (list kqmlmsg)))

(define (print-kqmlmsg kqmlmsg)
  (begin (display "KQMLMSG : ")
        (display (performative kqmlmsg))(display "  ")
        (display (send (sender kqmlmsg) 'getName ()))(display "  ")
        (display (send (receiver kqmlmsg) 'getName ()))(display "  ")
        (display (content kqmlmsg))))

(define kqmlmsg? ())
(set! kqmlmsg? (test-tag 'kqmlmsg))

(define (make-kqmlmsg performative sender receiver content)
  (list 'kqmlmsg performative sender receiver content))

(define (performative kqmlmsg) (car (cdr kqmlmsg)))
(define (sender kqmlmsg) (car (cdr (cdr kqmlmsg))))
(define (receiver kqmlmsg) (car (cdr (cdr (cdr kqmlmsg)))))
(define (content kqmlmsg) (car (cdr (cdr (cdr (cdr kqmlmsg))))))
```

xpmain.scm :

Ce fichier comprend le code correspondant à l'expérimentation elle-même ; c'est à dire l'instanciation des agents et le lancement de la conversation.

```
;;; xpmain.scm - Clement JONQUET - DEA Informatique LIRMM - Mai 2003 ;;;

(newline)

;;; Teacher's creation ;;;

(define teacher (simpleobj))
(send teacher 'new1 (list 'teacher))
(send teacher 'setGlobalInter (list evaluate-with-kqmlmsg-and-broadcast))

; Teacher's globalEnv definition... that give to the teacher the knowledge to learn
the broadcast performative
(define teacherenv
  '((learn-broadcast-code-msg
     (define learn-broadcast-code
       (let ((newproc
            (let ((oldproc (car (cdr (cdr (cdr (get-pair 'evaluate-kqmlmsg
                                                 goodenv
                                                 return
                                                 (lambda () 'notpossible))))))))
             (list 'compound
                   '(e r k)
                   (list (list (car (car oldproc))
                               (car (cdr (car oldproc)))
```

```
                              (list 'if
                                    '(eq? performative 'broadcast)
                                    '(k (mapper (lambda (x) (make-kqmlmsg (car content)
receiver x (car (cdr content))))
                                                (send receiver 'getConversation ())))
                                    (car (cdr (cdr (car oldproc)))))))
                    '())))))
        newproc)))
    (learn-broadcast-msg
     (define learn-broadcast (compound-to-reifier
                           (lambda (e r k)
                            (goodeval (car e) r k)))))
    (call-learn-broadcast
     (learn-broadcast (set! evaluate-kqmlmsg learn-broadcast-code)))))

(send teacher 'setGlobalEnv (list teacherenv))
(display "Teacher is created...")(newline)

;;; Student's creation ;;;

(define student (simpleobj))
(send student 'new1 (list 'student))
(send student 'setGlobalInter (list evaluate))
(display "Student is created...")(newline)

;;; Conversation ;;;

(initialize-conversation teacher student)
(display "Initialisation of the conversation...")(newline)(newline)

(define msg1 (make-kqmlmsg 'assertion teacher student '(define (square x) (* x
x))))
(define msg2 (make-kqmlmsg 'broadcast teacher student '(order (square 3))))

(sendmessage msg1)
(sendmessage msg2)

(conversationLoop teacher student)
(sendmessage msg2)
(conversationLoop teacher student)
```

load.scm :

Ce fichier est utilisé pour charger les différents modules présentés plus haut.

```
;;; load.scm - Clement JONQUET - DEA Informatique LIRMM - Mai 2003 ;;;

;;; Simple interpreter loading ;;;

(load "simple.scm")

;;; Lazy evaluation module loading ;;;

;(load "lazysimple.scm")

;;; global-env initialization ;;;

(initialize-global-env)

;;; File's loading in the global-env

(loadfile "simple.scm")
;(loadfile "lazysimple.scm")
(loadfile "evaluators.scm")
(loadfile "simpleobj.scm")

;;; Sharing global-env ;;;
```

```
(set-cdr! global-env (cons (cons 'global-env global-env) (cdr global-env)))
```

Lancement et déroulement de l'expérimentation

Etant donné que tout est interprété par notre méta-évaluateur, l'expérimentation est un peu longue (et oui la réflexivité coûte encore très cher !). Elle dure approximativement 5 minutes. Elle se lance de la manière suivante :

`(load "load.scm")` // charge le méta-évaluateur ainsi que les différents modules

`(loadfile "xpmain.scm")` // lance l'interprétation de xpmain.scm par notre méta-évaluateur

Pendant l'expérimentation, sont affichés le détail des messages que se transmettent le *teacher* et le *student*. L'utilisateur est invité à répondre Y ou N pour continuer la conversation. Cette conversation se déroule en deux étapes :

- La première où il faut répondre Y jusqu'a ce que "No messages..." s'affiche pour les deux agents. Elle consiste à l'apprentissage du performatif.
- La deuxième où il faut répondre une fois N pour relancer le message d'origine qui, cette fois ci, est traité.

Lorsque les deux agents affichent "No messages..." alors l'utilisateur peut répondre N est l'expérimentation se termine.

Annexe B. Fonction factorielle et de Fibonacci

Voici les définitions des fonctions factorielle itérative et Fibonacci mémoizée qui auraient puent être transmises par l'agent *teacher* à l'agent *student*, comme expliqué au §4.5.2 :

Fibonacci classique :

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))
```

Fibonacci mémoizée :

```
(define memo-fib
  (memoize (lambda (n)
             (cond ((= n 0) 0)
                   ((= n 1) 1)
                   (else (+ (memo-fib (- n 1))
                            (memo-fib (- n 2)))))))))

(define (memoize f)
  (let ((table (make-table)))
    (lambda (x)
      (let ((previously-computed-result (lookup x table)))
        (or previously-computed-result
            (let ((result (f x)))
              (insert! x result table)
              result))))))
```

Factorielle récursive :

```
(define (factorial n)
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

Factorielle itérative :

```
(define (factorial n)
  (define (fact-iter product counter max-count)
    (if (> counter max-count)
        product
        (fact-iter (* counter product)
                   (+ counter 1)
                   max-count)))
  (fact-iter 1 1 n))
```

Vous trouverez de plus amples informations sur ces procédures dans [ABE 96].

Annexe C. Soumission à JFSMA 2003

Le travail présenté dans ce mémoire a fait l'objet d'un article soumis à la conférence « Journées Francophones sur les Systèmes Multi-agents – JFSMA 2003 », Hammamet, Tunisie, 27-29 novembre 2003, présidé par Jean Pierre Briot (LIP6, Paris). Les réponses sont attendues pour le 5 juillet 2003. L'article est rédigé en français et fait une douzaine de pages. Il présente notre modèle ainsi que l'expérimentation mais ne détaille pas les extensions de notre travail. Le fichier est disponible en ligne sur http://www.lirmm.fr/~jonquet.

Pour plus de renseignements sur cette conférence : http://www.cck.rnu.tn/JFSMA/

Apprentissage issu de la communication pour des agents cognitifs

Clément Jonquet - Stefano A. Cerri

LIRMM - Université Montpellier 2
161, rue Ada
34392 Montpellier Cedex 5 - France
{cerri, jonquet}@lirmm.fr

RÉSUMÉ. *La communication entre agents cognitifs est un domaine de recherche en pleine effervescence. Nous proposons ici un modèle, basé sur le modèle STROBE, qui considère les agents comme des interpréteurs Scheme. Ces agents sont capables d'interpréter les messages d'une communication dans un environnement donné, avec un interpréteur donné, tous les deux dédiés à la conversation courante. Ces interpréteurs peuvent en outre évoluer dynamiquement au fur et à mesure des conversations et représentent la connaissance de ces agents au niveau méta. Nous proposons un mécanisme d'apprentissage à ce méta-niveau. Nous illustrons ce modèle théorique par une expérimentation de dialogue de type « professeur-élève », où un agent apprend un nouveau performatif à l'issue de la conversation. Les détails de l'implémentation ne sont pas fournis ici, mais sont disponibles.*

ABSTRACT. *Cognitive agent communication is a field of research in full development. We propose here a model, based on the model STROBE, which regards the agents as Scheme interpreters. These agents are able to interpret messages of a communication in both a given environment and an interpreter dedicated to the current conversation. These interpreters can moreover evolve dynamically progressively with the conversations and thus they represent meta level agent's knowledge. We propose a learning device for that level. We illustrate this theoretical model by a "teacher-student" dialogue experimentation, where an agent learns a new performative at the completion of the conversation. Details of the implementation are not provided here, but are available.*

MOTS-CLÉS : *communication agent, interprétation dynamique de message, réflexivité, reifying procedures, STROBE, langage de communication agent*
KEYWORDS: agent communication, message dynamic interpretation, reflexivity, reifying procedures, STROBE model, ACL

1. Introduction

La transmission du savoir est quelque chose d'essentiel pour toutes les sociétés humaines, c'est elle qui assure l'évolution et l'adaptation des ces sociétés à travers le temps. On ne peut imaginer où en serait l'homme s'il réapprenait à chaque génération à tailler des silex ou à contrôler le feu. Mais le problème ne se pose pas car les êtres humains possèdent une faculté d'apprentissage et d'adaptation qui n'est ni prévisible ni mesurable. Il n'en est pas de même pour les entités informatiques ! En effet, assurer la transmission du savoir, l'apprentissage et l'adaptabilité des sociétés d'agents est un véritable sujet qui promet encore de longues années de recherche. Nous tentons d'amener ici une petite pierre à cet énorme édifice en proposant un modèle d'apprentissage de connaissances basé sur la communication. Notre idée est de profiter de l'apprentissage comme effet secondaire de la communication. En effet, le but de l'éducation est de faire changer d'état son interlocuteur. **Ce changement se fait après l'évaluation des nouveaux éléments apportés par la communication**. Nous proposons dans cet article un modèle permettant de réaliser ce changement. Précisément, nous présentons un modèle, basé sur le modèle STROBE [CER 99], qui considère les agents comme des interpréteurs Scheme. Ces agents sont capables d'interpréter les messages d'une communication dans un environnement donné, avec un interpréteur donné, tous les deux dédiés à la conversation courante. En outre, nous montrerons comment ces évaluateurs, représentant les agents, peuvent modifier dynamiquement leur façon d'interpréter des messages. Ainsi, en communiquant, un agent apprend plus qu'une simple information, il modifie en fait sa façon de voir ces informations et devient capable d'en intégrer de nouvelles. Nous illustrerons ceci par une expérimentation de dialogue de type « professeur-élève ».

Ainsi, dans un premier temps, nous proposons un aperçu des problématiques associées aux notions de communication et d'apprentissage dans un SMA. Nous tenterons également de définir un scénario « idéal » pour l'évolution des sociétés d'agents et pour la communication dans l'avenir. En seconde et troisième partie, nous présenterons notre modèle qui consiste à considérer les agents comme des interpréteurs Scheme et qui leur fournit un ensemble de couples (environnement interpréteur) pour la représentation des autres. Ensuite, en quatrième partie, nous introduirons les outils qui nous permettent de faire évoluer dynamiquement l'interpréteur d'un agent, entre autres les mécanismes de réflexivité et de réification. L'expérimentation elle-même sera développée ainsi que le mécanisme des *reifying procedures*, en cinquième partie. Et enfin, nous essayerons de réaliser l'étude de cet exemple pour montrer ce qu'il apporte d'un point de vue théorique à une communication agent et quelles utilités ces genres de principes peuvent avoir dans un Système Multi-agents (SMA).

2. Communication et apprentissage dans les SMA

La simple mise en commun de plusieurs agents ne suffit pas à former un SMA, c'est le fait que ces agents communiquent qui le permet. C'est grâce à la communication que sont possibles la coopération et la coordination entre agent [FER 95]. Définir et modéliser la communication a toujours été difficile. Aujourd'hui, il existe de nombreux modèles et langages de communication mais peut-on dire qu'ils sont adaptés au monde agent ou à de nouvelles formes de communication comme celle que propose le Web ? Il ne s'agit pas de prendre les langages traditionnels de communication ou même les paradigmes actuels de programmation et de les adapter au Web et aux agents. **Il s'agit de développer de nouvelles architectures et de nouveaux langages conçus pour le Web et les agents.** En effet, les langages traditionnels sont ficelés et il est souvent très difficile de les faire évoluer. Pour être efficace, la communication doit être intrinsèque à un langage. Par exemple, en ce qui concerne l'apprentissage, il ne suffit pas simplement d'apprendre au niveau *donnée*, il faut aussi apprendre au niveau *contrôle* et au niveau *interprète* de ces programmes (données + contrôle). Un objet Java est capable de stocker des données mais il ne peut pas modifier sa propre structure pour en intégrer de nouvelles. Notre but est de fournir un modèle qui le puisse.

Si nous considérons le fait qu'une communication a des effets sur les interlocuteurs (acte perlocutoire), alors il nous faut obligatoirement considérer que les agents peuvent changer de but ou de point de vue au milieu de cette communication. Ils doivent donc **être autonomes et s'adapter pendant la communication** [CER 99]. Il faut aussi considérer que des agents peuvent interagir entre eux ou avec des humains suivant les mêmes principes [MCC 89] [CER 00]. Ce qui compte c'est **la représentation qu'un agent se fait de son interlocuteur**. Le problème de la sémantique des données que l'on échange se pose également. Actuellement, la notion d'ontologie est la réponse principale à cette question mais il en existe d'autres.

Historiquement, les SMA étaient construits avec un langage de communication intégré fonctionnant de manière ad-hoc. Aujourd'hui, la communauté SMA tend à fournir des *agent Communication Language* (ACL) applicables à un maximum d'interactions entre agents. En effet, fournir un ACL fort d'un point de vue sémantique donne un gros avantage pour la création et l'évolution d'un SMA [DIG 00]. Ces ACL sont basés sur la théorie des actes de langage[1]. Traditionnellement, les messages KQML ou FIPA-ACL fournissent un élément qui correspond à l'ontologie utilisée dans la communication. Cela permet de rendre les ACL indépendants de n'importe quel vocabulaire et donne à l'interlocuteur un moyen d'établir la correspondance concept / signification des éléments du contenu d'un message. Dorénavant on ne spécifie plus un ACL ad-hoc pour l'incorporer à un SMA, mais on construit une ontologie qui sera passée en paramètre des messages. Nous proposons dans ce papier une alternative à cet état de fait. Les ACL reçoivent souvent la

[1] Modèle issu de la philosophie du langage [AUS 70] et [SEA 71].

critique du manque de performatif. Notre expérimentation propose un exemple de solution à ce problème. Elle illustre une technique pour diffuser des nouveaux performatifs dans un SMA, par apprentissage direct d'agent à agent.

Face à ces langages, de nombreux modèles de communication ont été proposés. Une alternative sur la façon de considérer les agents est présentée dans [MAR 01]. Entre autres, STROBE [CER 99] s'intéresse à de nombreux principes importants pour une communication et se base sur les trois primitives Scheme : *STReam, OBject,* et *Environment*. Il met en avant des points comme la représentation de l'interlocuteur, la conservation de l'historique d'une conversation, l'apprentissage issu de la communication, etc. Nous reviendrons souvent aux propositions de STROBE car le modèle proposé ici s'en inspire.

Imaginons maintenant un scénario « idéal » de ce que pourrait être un SMA dans quelques années. Il considère toutes les entités du Web comme des agents d'une même société qui peuvent communiquer les uns avec les autres naturellement et se transmettre des connaissances. Cette société pouvant s'étendre sans aucune limite. Dans ce SMA, chaque agent est initialisé avec un minimum requis de connaissances (pour interagir) ainsi qu'avec une spécialité qui le caractérise et qu 'il peut transmettre aux autres. Cet agent possède un ensemble d'interpréteurs de messages qui représentent sa connaissance et son évolution dans le temps. Il apprend tout le reste au fur et à mesure de ses communications. Il apprend même à apprendre ! Pour chaque agent avec qui il communique, il a une représentation spécifique de celui-ci, ce qui lui permet de tenir compte de ce qu'il apprend tout en gardant son comportement et ses croyances d'origine intactes. D'un point de vue coopération / coordination, un agent peut demander à un autre d'interpréter pour lui tel ou tel programme et de lui renvoyer le résultat. Voir même, comme dans des architectures Grid [DER 01] où il est plus intéressant de déplacer le processus que les données, un agent peut transmettre à un autre un interpréteur qui lui permet d'effectuer sa tâche. Ces interpréteurs peuvent même être transmis avant une conversation, comme l'on transmet aujourd'hui une ontologie. Mais vu qu'aucun interpréteur n'évolue de la même manière, puisque deux conversations ne sont jamais les mêmes, cela donne à cette société d'agents une pluralité des connaissances incommensurable ! Son évolution devient totalement imprévisible et autonome. L'intégration des nouveaux agents se fait naturellement et petit à petit. Il n'est plus possible alors de prouver de manière théorique que tel ou tel agent sait accomplir une tâche ; le seul moyen est de regarder les solutions émergentes qui apparaissent lorsqu'un problème se pose.

Nous tentons dans cet article de proposer des idées pour rendre réalisable ce scénario encore utopiste aujourd'hui. Entre autres, nous allons voir comment un agent qui possède plusieurs interprètes de messages peut les modifier dynamiquement pour évoluer au fur et à mesure des conversations.

3. Les agents comme des interprètes Scheme

Le modèle que nous proposons a pour caractéristique principale le fait qu'il considère les agents comme des interpréteurs de messages. Cette idée est issue de STROBE [CER 99] qui s'inspire de la boucle classique d'évaluation et considère les agents comme des interpréteurs de type REPL (Read, Eval, Print, Listen). Lors d'une communication, chaque agent exécute une boucle REPL qui sont imbriquées les unes dans les autres (cf. Figure 2). Ce principe est important car il permet de considérer les agents comme des entités autonomes dont les interactions sont dirigées par une procédure d'évaluation des messages. Notre modèle utilise Scheme aussi bien pour le contenu des messages que pour leur représentation. De cette façon nous pouvons utiliser le même évaluateur pour évaluer le message et son contenu. Exemple d'expression Scheme représentée par des messages :

```
> (define x 2)        ⇔        > (assertion (define x 2))
: x                   ⇔        : (ack x)
> x                   ⇔        > (request x)
: 2                   ⇔        : (answer 2)
```

Ce point de vue est très intéressant car les agents bénéficient de tous les avantages liés à Scheme pour la représentation de connaissances grâce à la mémoire que constitue l'environnement. Par exemple, STROBE propose une structure d'environnement conservant l'historique, c'est à dire que les liaisons ne sont plus du type (var val) mais du type (var valn valn-1...val1). Cette structure devient accessible à des agents représentés par des interpréteurs.

Pour implémenter ce modèle nous avons écrit un méta-évaluateur Scheme reconnaissant un certain langage et nous y avons rajouté un module d'interprétation des messages. Scheme facilite l'abstraction à la fois et par les environnements [CER 99] et par les continuations [QUE 00]. Il explicite bien les trois niveaux : *donnée*, *contrôle*, *interprète*. L'apprentissage au niveau *données* consiste à affecter des valeurs à des variables déjà existantes ; Au niveau *contrôle*, définir de nouvelles variables ou fonction. Et enfin, celui qui nous intéresse, l'apprentissage au niveau *interprète* ou méta-niveau consiste à faire évoluer l'interprète Scheme. C'est pourquoi, en faisant évoluer son interprète, un agent apprend plus qu'une simple information ; Il change complètement sa façon de percevoir ces informations. C'est la différence entre apprendre une donnée et apprendre à traiter une classe de données.

4. Représentation des autres

Les performatifs de STROBE se traduisent par une expression Scheme et engendrent sur l'agent récepteur un certain comportement. En particulier, la mise à jour de son « modèle du partenaire ». En effet, pour ce qui est de la représentation de l'interlocuteur, STROBE met en avant le fait qu'il faut avoir un modèle du partenaire pour pouvoir reconstruire son état interne. Ce modèle propose que chaque dialogue soit interprété dans une paire d'environnement : le premier privé, appartenant à

l'agent, et le deuxième représentant le modèle du partenaire courant. C'est la notion d'environnements cognitifs. Notre travail exploite cette notion. En fait, il se greffe dessus car, comme le concept d'environnement cognitif fournit aux agents un environnement global (ou privé) et plusieurs environnements locaux de représentation de l'autre, notre proposition fournit aux agents non pas un évaluateur de message mais plusieurs évaluateurs dont un global (ou privé) et un pour chaque agent dont ils ont une représentation. Ainsi, l'évaluation des messages d'une conversation se fait avec un évaluateur donné dans un environnement donné. Notre travail se greffe sur ce concept d'environnement dans le sens où ces évaluateurs, pour être accessibles, doivent eux-mêmes être stockés dans ces environnements ! Donc nos agents possèdent les trois attributs suivants :

- **GlobalEnv** leur environnement global.
- **GlobalInter** leur interpréteur global.
- **Other** = {(name, interpreter, environment)} un ensemble de triplets correspondant aux représentations des autres.

Leur environnement global est privé et ne change pas. C'est cet environnement qui est dupliqué[2] lors de l'arrivée d'une nouvelle conversation et c'est son clone, stocké dans un élément de Other, qui est modifié au fur et à mesure de la conversation. Le principe est le même pour l'interpréteur global. La Figure 1 illustre ces représentations.

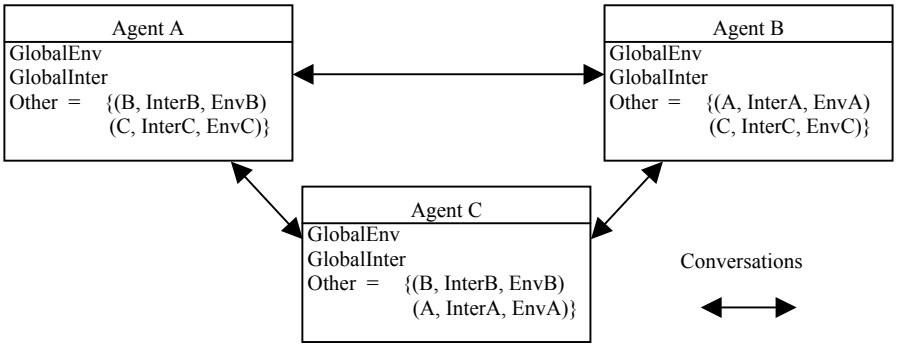


Figure 1 *Les trois attributs caractérisant un agent et ses représentations*

Nous avons maintenant posé les structures correspondant à nos agents et à leurs représentations. Pour assurer l'apprentissage à l'issue d'une communication, il faut maintenant expliciter les outils qui vont nous permettrent de modifier dynamiquement, au fur et à mesure des conversations, le(s) interprète(s) d'un agent.

[2] Pas forcément si on considère un agent qui voudrait reprendre une conversation dans le contexte d'une autre déjà existante ou ayant existée (avec son environnement et son interprète de message).

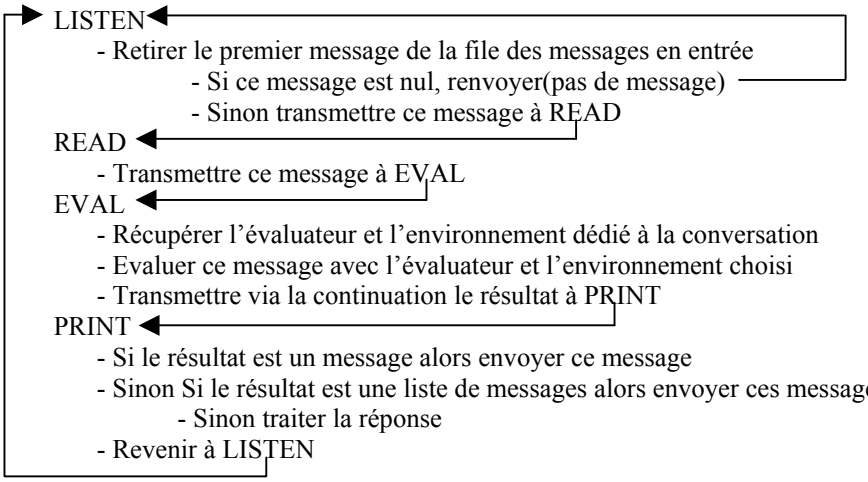


Figure 2 *La boucle REPL de nos agents*

5. Modification dynamique d'un interpréteur

5.1. Le méta-évaluateur Scheme

Le méta-évaluateur qui nous sert de support est issu du célèbre article de Jefferson et Friedman, *A Simple Reflective Interpreter* [FRI 92]. Cet évaluateur a l'avantage d'être facile à prendre en main et possède d'ores et déjà la propriété de réflexivité dont nous avons besoin. Il propose, en outre, le mécanisme des *reifying procedures*, qui comme nous le verrons plus loin permet à un programme utilisateur d'accéder à son contexte d'exécution[3] et éventuellement de le modifier. Cet évaluateur est de la forme `(evaluate expression environnement continuation)`. Il est défini principalement par deux procédures `evaluate` et `apply-procedure`[4], correspondant aux deux étapes de l'évaluation par environnement (cf. [ABE 96]).

5.2. Réflexivité et réification

En ce qui concerne la réflexivité, l'évaluateur proposé dans [FRI 92] est déjà implémenté de manière réflexive. C'est à dire que le code de l'évaluateur est écrit dans le sous-langage de Scheme que celui-ci reconnaît. Mais Jefferson et Friedman proposent également, dans leur article, une architecture dite à tour réflexive où à chaque niveau correspond un évaluateur, qui interprète le langage du dessus. Le niveau supérieur quant à lui interprète le code de l'utilisateur. Deux mécanismes

[3] Le contexte d'exécution d'une expression est constitué de l'environnement d'évaluation de cette expression et de la continuation qui correspond à la prochaine expression à évaluer avec le résultat.

[4] Correspondant aux fonctions `eval` et `apply` de [ABE 96] et `evaluate` et `invoke` de [QUE 94].

permettent de « se déplacer » dans la tour de façon à évaluer le code à n'importe quel niveau : la réflexivité permet de passer au niveau supérieur et la réification permet de redescendre. Ces deux points nous intéressent tout particulièrement.

Les *reifying procedures* fournissent le mécanisme nécessaire pour avoir accès au contexte d'exécution de l'interpréteur du niveau d'en dessous. L'idée principale est que les programmes des utilisateurs puissent avoir les mêmes accès que l'interpréteur lui-même. Grâce à cela, **les procédures implémentant l'évaluateur peuvent être accédées et modifiées par les programmes utilisateurs** de la même façon que l'environnement et les continuations. Cette propriété fait des *reifying procedures* l'outil idéal pour modifier dynamiquement nos interpréteurs. En effet, accéder au contexte d'exécution veut dire accéder à l'environnement dans lequel sont stockées les procédures définissant l'évaluateur et donc pouvoir les modifier. Nous avons vu que la réification était la méthode pour redescendre dans les niveaux d'évaluation. Pour appliquer une *reifying procedure* il faut donc avoir deux niveaux d'évaluation. Cela implique que toute l'expérimentation soit évaluée par notre méta-évaluateur pour réaliser le premier appel à `evaluate`, le deuxième étant effectué par l'agent lui-même lorsqu'il reçoit un message. Ainsi, la réflexivité de celui-ci devient obligatoire : notre méta-évaluateur doit pouvoir s'évaluer lui-même.

5.3. Protocole de communication

Les messages que nous considérons sont inspirés de la structure des messages KQML ou FIPA ACL[5] Ils seront notés par la suite kqmlmsg, ils sont de la forme : **(kqmlmsg performative sender receiver content)**. Pour que notre méta-évaluateur puisse interpréter les messages kqmlmsg, il faut lui ajouter un test qui les reconnaît dans la fonction `evaluate`, ainsi qu'une fonction qui les traite en évaluant leur contenu `evaluate-kqmlmsg`. Les performatifs reconnus sont : *assertion, order,request, ack, answer, executed* [CER 99] et, nous le verrons aussi, *broadcast*, qui fait l'objet de l'expérimentation. Lorsqu'un agent reçoit un message indexé par un performatif qu'il ne connaît pas il l'indique de manière particulière :

- Les assertions (*assertion*) ont pour but de modifier le comportement ou les représentations de l'interlocuteur. Leur réponse sont des messages de type accusé de réception (acknowledgement (*ack*)) signifiant un succès ou une erreur.

- Les requêtes (*request*) ont pour but de connaître une représentation de l'interlocuteur, comme par exemple la valeur d'une variable ou la fermeture d'une fonction. Leur réponse (*answer*) renvoie une valeur ou une erreur.

- Les ordres (*order*) demandent à l'interlocuteur d'exécuter une procédure. Le résultat est envoyé par celui-ci par un message de type exécuté (*executed*).

- Les messages diffusion (*broadcast*) consistent à envoyer en contenu un couple `(perform, content)` qui signifie que l'interlocuteur doit envoyer un message avec

[5] Notre protocole étant énormément simplifié, nos messages ne précisent que 4 paramètres. On aurait pu cependant en implémenter plus : language, ontology, in-reply-to, reply-with, etc.

comme performatif `perform` et comme contenu `content` à tous ses interlocuteurs en cours. Il n'y a pas de réponse définie pour les messages broadcast.

6. L'expérimentation : un dialogue « professeur-élève »

L'idée de notre travail est de profiter de l'apprentissage comme effet secondaire de la communication. En effet, le but de l'éducation est de faire changer d'état son interlocuteur. Ce changement se fait après l'évaluation des nouveaux éléments apportés par la communication. Pour cela nous utilisons les propriétés de réflexivité et de réification vues plus haut. Grâce à notre évaluateur réflexif, nous montrons qu'un agent peut modifier sa façon de voir les choses (c'est à dire d'évaluer des messages) en « ré-évaluant » son propre évaluateur à l'issue d'une conversation. L'expérimentation que nous présentons ici est un dialogue type « professeur-élève ». Un agent *teacher* demande à un autre agent *student* de diffuser (kqmlmsg de type broadcast) pour lui un message à tous ces correspondants mais le *student* ne connaît pas le performatif utilisé par le *teacher*. Par conséquent, le *teacher* transmet au *student* une série de messages (de type assertion et order) explicitant comment prendre en compte ce performatif. Il lui re-transmet alors son message d'origine qui obtient ce coup ci satisfaction. Le dialogue exact de l'expérimentation est décrit par la Figure 3

Nous avons développé pour les besoins de notre expérimentation des agents capables de communiquer, c'est à dire de s'échanger des messages les uns avec les autres et d'y produire des réponses significatives (ceci suivant le protocole définit plus haut)[6]. Ils ne font rien lorsqu'ils ne communiquent pas et leur autonomie se caractérise par le fait qu'ils apprennent seuls. Ils ont comme attributs, name, globalEnv, globalInter, other, ainsi que deux files de messages (une en sortie et une en entrée), et une structure stockant les conversations courantes. Leur comportement consiste uniquement à appliquer la boucle REPL (Figure 2).

Après toutes ces explications, nous pouvons maintenant décrire précisément comment est réalisé l'apprentissage d'un nouveau performatif. Nous allons détailler ici les messages que l'agent *teacher* envoie à l'agent *student* pour qu'il modifie son interprète. Le premier message est `learn-broadcast-code-msg`. Cette variable correspond dans l'environnement du *teacher* au code de la nouvelle fonction `evaluate-kqmlmsg` que le *student* doit générer pour, plus tard, l'affecter à cette fonction. Ce code est construit par le *student* en récupérant le corps de sa fonction `evaluate-kqmlmsg` et en lui ajoutant un `(if (eq? performative 'broadcast)` et le traitement associé.

[6] Nos agents sont en fait des objets programmés en Scheme d'après les techniques présentées par Normark dans *Simulation of Object-Oriented and Mechanisms in Scheme* [NOR 91]

TEACHER	STUDENT
Voici la définition de la procédure square : **(kqmlmsg 'assertion teacher student '(define (square x) (* x x)))**	Ok, je connais maintenant cette procédure : **(kqmlmsg 'ack student teacher '(*.*))**
Diffuse à tous tes correspondants : **(kqmlmsg 'broadcast teacher student '(order (square 3)))**	Désolé, je ne connais pas ce performatif : **(kqmlmsg 'answer student teacher `'(no-such-performative broadcast))**
Ok, voilà comment ajouter broadcast à la liste des performatifs que tu reconnais : Voilà le code que tu devras générer et ajouter à ta procédure `evaluate-kqmlmsg` : **(kqmlmsg 'assertion teacher student learn-broadcast-code-msg)** Ensuite voila la *reifying procedure* qui te permet de changer ce code : **(kqmlmsg 'assertion teacher student learn-broadcast-msg)** Exécute cette procédure : **(kqmlmsg 'order teacher student call-learn-broadcast)**	Ok, j'ai rajouté ce code dans une variable de mon environnement : **(kqmlmsg 'ack student teacher '(*.*))** Ok, je connais maintenant cette procédure : **(kqmlmsg 'ack student teacher '(*.*))** Ok, je viens de modifier mon évaluateur : **(kqmlmsg 'executed student teacher '(*.*))**
Diffuse à tous tes correspondants : **(kqmlmsg 'broadcast teacher student '(order (square 3)))**	Ok, je diffuse…

Figure 3 *Dialogue teacher/student pour l'enseignement de broadcast*

C'est une vison constructiviste de l'apprentissage[7]. Ainsi l'environnement du *teacher* possède la liaison suivante :

```
learn-broadcast-code-msg :
 `(define learn-broadcast-code
       (let ((newproc
          (let ((oldproc *récupération du code de evaluate-kqmlmsg*))
              *modification de oldproc pour lui ajouter
              le nouveau code (if (eq? performative…*)
          newproc))
```

[7] Il est important de remarquer que c'est le *student* qui reconstruit sa fonction à partir de celle qui existe déjà dans son environnement et non le *teacher* qui lui envoi le code de sa fonction `evaluate-kqmlmsg`. Cela permet au *student* de conserver d'autres modifications antérieures qu'il a pu avoir de sa fonction.

Le deuxième message envoyé par le *teacher* est le plus important des trois. C'est celui qui définit la *reifying procedure* `learn-broadcast` qui va modifier l'évaluateur du *student*. L'environnement de *teacher* possède aussi la liaison :

```
learn-broadcast-msg :
 `(define learn-broadcast (compound-to-reifier
                           (lambda (e r k) (evaluate (car e) r k))))
```

Lors de son évaluation `learn-broadcast` est transformée en appel d'une procédure composée `(compound)` avec comme argument : la liste de ses arguments, l'environnement dans lequel elle doit être évaluée et la continuation qu'il faut donner à cette évaluation. Le contexte d'exécution de `learn-broadcast` est alors accessible et donc modifiable ! Dans notre cas, `learn-broadcast` consiste à évaluer le `car` de la liste de ses arguments. Finalement, le *teacher* invite alors le *student* à appliquer cette fonction avec pour paramètre un appel à `set!` qui modifie `evaluate-kqmlmsg` par le code généré au préalable. La dernière liaison est donc :

```
call-learn-broadcast :
 '(learn-broadcast (set! evaluate-kqmlmsg learn-broadcast-code))
```

A l'issue du traitement du dernier message le *student* a modifié sa fonction `evaluate-kqmlmsg` et donc son interprète de message. Le code correspondant à cette fonction dans son environnement dédié à cette conversation est changé. Il est donc maintenant apte à traiter les messages indexés par le performatif broadcast.

7. Intérêts et extension de ces principes

Notre expérimentation montre comment prendre en compte un nouveau performatif, donc comment modifier la fonction d'interprétation des messages d'un agent. Mais les mêmes principes peuvent être utilisés pour modifier n'importe qu'elle partie de l'interpréteur d'un agent. Par exemple nous aurions pu faire un exemple qui rajoute `cond` ou `let*` au langage reconnu par notre méta-évaluateur. Voir même un agent qui enseigne à un autre comment transformer son évaluateur en évaluateur paresseux en changeant ses fonctions `evaluate` et `apply-procedure`. Grâce à ce protocole, nos agents possèdent en fait un ensemble d'interpréteurs qui représentent leurs connaissances. En effet, ils correspondent aux sous-langages que nos agents reconnaissent et donc à leurs facultés à effectuer une tâche. Comme vu dans le scénario « idéal », les agents peuvent effectuer des tâches pour d'autres ou même s'échanger leurs interpréteurs[8]. Leurs interpréteurs peuvent même être aussi transmis avant une conversation, plutôt que de transférer ontologie.

Imaginons une société d'agents ou un SMA qui suit ce genre de modèle alors n'importe quel agent peut apprendre quelque chose d'un autre. Si on construit un

[8] Ceci s'inspire de l'idée énoncée dans [ABE 96] : « *If we wish to discuss some aspect of a proposed modification to Lisp with another member of the Lisp community, we can supply an evaluator that embodies the change. The recipient can then experiment with the new evaluator and send back comments as further modifications.* »

agent avec une connaissance et une spécialité le caractérisant alors cette spécialité peut se diffuser au fur et à mesure de ses communications. Pour le Web, ce genre de principe est très intéressant. Considérons un agent qui joue le rôle de serveur pour une nouvelle application orientée Web et qui utilise une série de performatifs correspondant exactement à ce qu'il doit faire. S'il est construit avec le potentiel pour enseigner ces performatifs alors il s'intégrera très bien à une société d'agents en les enseignants au début des communications qu'il peut avoir. En outre, pour faire le lien avec XML, nous pouvons considérer une DTD ou un XML-Schema comme un « interpréteur » de donnée XML[9]. Le modèle présenté ici permet alors de faire évoluer dynamiquement ces DTD et donc les documents XML associés donnant au Web un dynamisme et une adaptabilité inégalable.

Le travail présenté ici s'inscrit dans la même lignée que le langage C+C [CER 00] qui propose de fournir aux agents un algorithme d'organisation temporel (*scheduling algorithm*) dynamique. Là où un objet répond à un appel de méthode sans réfléchir ou contester, un agent a un comportement qui dépend de son *scheduling algorithm* qui lui permet d'être autonome et de décider si et quand il consacre du temps aux autres. Là où l'objet est implémenté de manière fixe, l'agent possède un *scheduling algorithm* qui varie avec le temps. Nos interpréteurs réflexifs peuvent faire office de *scheduling algorithm.*

8. Conclusion

Nous avons essayé de montrer dans ce papier une méthode d'apprentissage pour les agents cognitifs issue de la communication. Cet apprentissage peut se faire par communication simple (niveau *donnée* et *contrôle*), ou par modification interne de l'agent (niveau *interprète*). Si les agents interprètent de façon dynamique les messages qu'ils reçoivent, ils deviennent adaptables et, sans aucune intervention extérieure, peuvent communiquer avec des entités qu'ils n'ont jamais rencontrées auparavant. En outre, comme leur évaluateur est modifié pour acquérir une connaissance, il pourrait l'être aussi pour apprendre à apprendre une connaissance. Dans ce cas le transfert du savoir deviendrait exponentiel au fur et à mesure des communications. Ce papier n'a pas simplement pour vocation de proposer un artéfact de plus de programmation à ajouter aux agents, mais l'idée est plutôt de montrer une technique, faite de manière simple et utilisable, d'évolution autonome des agents dans une société.

9. Annexe

Le modèle présenté ici à été sujet à une petite implémentation, non compète encore à ce jour, mais d'ores et déjà fonctionnelle. Elle est disponible en ligne sur http://www.lirmm.fr/~jonquet. Elle fut réalisée avec MIT Scheme 7.7.1, norme

[9] Puisque les documents XML sont des arbres et Scheme est idéal pour représenter les arbres. (cf. [KIS 02])

R5RS. Vous y trouverez entre autres les fichiers Scheme spécifiant le méta-évaluateur utilisé pour l'expérimentation, le module interprétant les kqmlmsg, le fichier implémentant les agents évoqués ici et le fichier correspondant à l'expérimentation.

10. Bibliographie

[AUS 70] Austin J.L., *Quand dire c'est faire*, Edition du seuil, Paris, 1970.

[ABE 96] Abelson H., Sussman G.J., Sussman J., *Structure and Intepretation of Computer Programs*, Second Edition, MIT Press, Cambridge, Massachusetts, 1996.

[CER 99] Cerri S.A., « Shifiting the Focus from Control to communication: The STReams OBject Environments (STROBE) model of communicating agents », In Padget, J.A. (ed.) Collaboration between Human and Artificial Societies, Coordination and agent-Based Distributed Computing, Berlin, Heidelberg, New York: Springer-Verlag, Lecture Notes in Artificial Intelligence, pp 71-101, 1999.

[CER 00] Cerri S.A., Sallantin J., Castro E., Maraschi D. « Steps towards C+C: a Language for Interactions », In Cerri, S. A., Dochev, D. (eds), *AIMSA2000: Artificial Intelligence: Methodology, Systems, Applications*, Berlin, Heidelberg, New York: Springer Verlag, Lecture Notes in Artificial Intelligence, pp 33-46, 2000.

[MAR 01] Maraschi D., Cerri S. A., « The relations between Technologies for Human Learning and agents », In procedings of the *AFIA 2001 Atelier: Methodologies et Environments pour les Systèmes Multi-agents,* Grenoble: Leibniz-Imag, pp. 61-73, 2001.

[DER 01] De Roure D., Jennings N., Shadbolt, N. « Research Agenda for the Semantic Grid: A Future e-Science Infrastructure » In Report commissioned for *EPSRC/DTI Core e-Science Programme*. University of Southampton, UK, 2001.

[DIG 00] Dignum F., Greaves M., « Issues in agent Communication : An introduction », Dignum F and Greaves M.(Eds.): *agent Communication, LNAI 1916*, pp1-16, Springer-Verlag Berlin Heidelberg, 2000.

[FER 95] Ferber J., *Les Systemes Multi-agents, vers une intelligence collective*, InterEditions, Paris, 1995.

[FRI 92] Friedman D. P., Jefferson S., « A Simple Reflective Interpreter », *IMSA'92, International Workshop on Reflection and Meta-Level Architecture*, Tokyo, 1992.

[KIS 02] Kiselyov Oleg, « XML, Xpath, XSLT implementations as SXML, SXPath, and SXSLT », *International Lisp Conference : ILC2002*, San Francisco, CA, Octobre 2002.

[MCC 89] McCarthy J., « Elephant 2000: A Programming Language Based on Speech Acts ». *Unpublished draft* Stanford University, www-formal.stanford.edu/jmc/elephant.pdf, 1989.

[NOR 91] Normark Kurt, « Simulation of Object-Oriented and Mechanisms in Scheme », *Institute of Electronic Systems*, Aalborg University, Denmark, 1991.

[QUE 94] Queinnec C., *Les langages LISP*, Interéditions, Paris, 1996.

[QUE 00] Queinnec C., « The Influence of Browsers on Evaluators or, Continuations to Program Web Servers », *ICFP'00*, Montréal, Canada, 2000.

[SEA 71] Searle J., *Les actes de langages, essai de philosophie du langage*, Herman Editeur, Paris 1971.

Annexe D. Soumission à ALCAA 2003

Ce travail a également fait l'objet d'une soumission à la conférence « Colloque agents Logiciels - Coopération Apprentissage - Activité humaine - ALCAA 2003 », Bayonne, France, 4-5 septembre 2003, présidé par Philippe. Aniorté (LIUPPA, Université Pau). Les réponses sont attendues sous peu. L'article est rédigé en anglais et fait une douzaine de pages. C'est la version traduite et améliorée de l'article soumis à JFSMA03. Le fichier est disponible en ligne sur http://www.lirmm.fr/~jonquet.

Pour plus de renseignements sur cette conférence : http://www.iutbayonne.univ-pau.fr/Rech/ALCAA2003/AppelaCom_Alcaa2003.htm

Cognitive agents Learning by Communicating

Clement Jonquet - Stefano A. Cerri

LIRMM - University Montpellier 2
161, rue Ada
34392 Montpellier Cedex 5 - France
{cerri, jonquet}@lirmm.fr

ABSTRACT. Cognitive agent communication is a research field in full development. We propose here an extension and an implementation of the STROBE model, which regards the agents as Scheme interpreters. These agents are able to interpret messages in a dedicated environment including an interpreter that learns from the current conversation. These interpreters evolve dynamically, progressively with the conversations, and thus represent evolving meta level agent's knowledge. We illustrate this theoretical model by a "teacher-student" dialogue experimentation, where an agent learns a new performative at the completion of the conversation. Details of the implementation are not provided here, but are available.

RÉSUMÉ. La communication entre agents cognitifs est un domaine de recherche en pleine effervescence. Nous proposons ici un modèle, basé sur le modèle STROBE, qui considère les agents comme des interpréteurs Scheme. Ces agents sont capables d'interpréter des messages dans un environnement donné incluant un interpréteur qui apprend de la conversation. Ces interpréteurs peuvent en outre évoluer dynamiquement au fur et à mesure des conversations et ils représentent la connaissance de ces agents au niveau méta. Nous illustrons ce modèle théorique par une expérimentation de dialogue de type « professeur-élève », où un agent apprend un nouveau performatif à l'issue de la conversation. Les détails de l'implémentation ne sont pas fournis ici, mais sont disponibles.

KEYWORDS: agent communication, message dynamic interpretation, reflexivity, reification, reifying procedures, STROBE model, ACL

MOTS-CLÉS : communication agent, interprétation dynamique de message, reification, réflexivité, reifying procedures, STROBE

1. Introduction

Knowledge communication is something very important for all human societies; it provides evolution and adaptation of these societies through time. One can't imagine where humanity would be if each generation had to learn again how to cut flints or control fire. Fortunately, this does not happen because human beings have both a learning and an important adaptation ability which is neither foreseeable nor measurable. It is not the same for data-processing entities! Indeed, to ensure knowledge communication, learning and adaptability of agent's societies is a research subject that still requires years of work. We try to identify, here, a small stone to this enormous building by proposing a model of knowledge learning, based on the communication. Our idea is to benefit from the learning side effect of communication. Indeed, the goal of education is to change the interlocutor's state. **This change is done after evaluating new elements brought by the communication**. We propose in this article a learning-by-being-told model that allows carrying out this change. Precisely, we present a model, based on the STROBE model [CER 99], which regards agents as Scheme interpreters. These agents are

able to interpret communication messages in a given environment, including an interpreter, dedicated to the current conversation. Moreover, we will show how these interpreters, which represent the agents, can dynamically adapt their way of interpreting messages. Thus, an agent learns, through a communication, more than simple Information, in fact it modifies its way of seeing this Information and acquires the ability to integrate new one. We will illustrate this by "teacher-student" dialogue experimentation.

The rest of the paper is organized as follows: section 2 proposes an outline of problems about communication and learning in a MAS (Multi-agent System), here we will also try to define an "ideal" scenario for the evolution of agents societies, and communication agents in the future; section 3 and 4 present our model which regards agents as Scheme interpreters and provides them a set of pairs (environment interpreter). Section 5 introduces the devices that enable us to make agent's interpreters evolve dynamically, specifically the mechanisms of reflexivity and reification. The experimentation itself will be developed as well as the mechanism of reifying procedures, in section 6. Finally, we will attempt to study this example to show what theoretical points are brought to an agent communication and which advantages these principles can have in a MAS.

2. Communication and learning in MAS

Simply grouping together several agents is not enough to form a MAS, it is the communication between these agents that makes it. Communication allows co-operation and coordination between agents [FER 95]. Defining and modelling communication have always been difficult. Nowadays, we can find many communication languages but could one say they are adapted to the agent world or to new forms of communication such as those proposed on the Web? We can't take traditional languages or even current programming paradigms and adapt them to the Web and agents. **We have to develop new architectures and new languages designed for the Web and the agents**. Indeed, traditional languages are frozen and it is often very difficult to make them move. To be effective, the communication must be intrinsic to a language. For example, for knowledge learning, the *data* level is not enough, the *control* level and the *interpreter* level of these programs are necessary. A Java object, for instance, is able to store data but it can hardly modify its own structure. Our goal is to provide a model that can do it and, at the same time, is as simple as possible. In our approach, expressive power of language constructs has the same priority as cognitive simplicity for building effective, new applications solving complex problems on the Web.

If we consider communication effects on the interlocutors, then we must consider that agents can change their goal or point of view during the communication. Therefore, **they must be autonomous and should adapt during communication** [CER 99]. We have also to consider the fact that agents can interact together or with humans following the same principles [MCC 89] [CER 00]. **What is important is the agent's representation of its interlocutors**. Semantics of exchanged data is also to take into account. For the moment, the concept of ontology is the main answer to this question but others exist.

Historically, MASs were built with an integrated communication language working in an ad-hoc way. Nowadays, the MAS community tends to use agent Communication Languages (ACLs) applicable to as many agents' interactions as possible. Indeed, providing a strong ACL with a strong semantic gives a large advantage for MAS creation and evolution [DIG 00]. These ACLs are based on the speech act theory[1]. Traditionally, KQML or FIPA-ACL messages provide an element that corresponds to the ontology used in the communication. This makes ACLs independent of any vocabulary and gives to the interlocutor the relation between the concept and the meaning of the message content elements. Nowadays, a specific or ad-hoc ACL is not incorporated in MASs but an ontology is built and given in messages parameters. We propose in this paper an alternative to this established practice. ACLs are often criticized about the number and the kind of performatives they provide. Our experimentation proposes an example of solution to this problem. It illustrates a technique to diffuse new performatives in MAS, by enabling agents to learn-by-being-told.

In reaction to all these languages, many models of communication have been proposed. An alternative to the way of considering agents is presented in [MAR 01]. Among other things, STROBE [CER 99] is interested in many significant principles for a communication being based on the three simple Scheme primitives: STReam, OBject, and Environment. It supports different points such as interlocutor representation, history conservation of

[1] This model comes from language philosophy ([AUS 70], [SEA 71]).

a conversation, learning-by-being-told etc. We will often refer to STROBE proposals because the model suggested here is inspired from it.

Now let us imagine an "ideal" scenario, such as those proposed in [IST 01], of what could be a SMA in a few years. It considers all the Web entities as agents of the same society that can naturally communicate together and transmit their knowledge. This society could extend without any limit. In this MAS, each agent is initialised with a necessary minimum of knowledge (to interact) and with a special knowledge characterizing it and transmittable to the others. These agents have a set of messages interpreters that represent their knowledge and its evolution in time. They progressively learn by communicating. They even learn how to learn and teach! For each agent with which they communicate, they have a specific representation of this agent, which enables to take account of what they learn while keeping their original behaviour and beliefs intact. From a co-operation/coordination point of view an agent can require another one to interpret on its behalf a program and return the result. As in Grid architectures [DER 01], an agent can also transmit to another the interpreter that allows it to realise its task. Moreover, these interpreters can be transmitted before a conversation, as nowadays ontologies are transmitted. Considering the fact that no interpreter evolves in the same way, because two conversations are never the same, this society of evolutionary agents would progressively acquire extraordinary knowledge richness! Its evolution becomes totally unforeseeable. New agent's integration is done naturally and gradually. It would not be possible to prove theoretically that an agent achieves a particular task; the only way would be to look at the emergent solutions which appear when a problem arises.

We try in this article to propose some ideas to progress towards this still utopian scenario. Among other things, we will see how an agent, which has several messages interpreters, can modify them dynamically to evolve progressively during the conversations.

3. Agents as Scheme interpreters

The proposed model considered agents as interpreters (for both messages and their content). This idea comes from STROBE that considers agents as REPL interpreters (Read, Eval, Print, Listen). While it communicates, each agent executes a REPL loop that is overlapping with the others ones (cf. Figure 2). This principle is very important because it regards agents as autonomous entities whose interactions are functionally controlled by a messages evaluation procedure. Our model uses Scheme as well for the messages contents and for their representation. In this way we can use the same interpreter to evaluate the message and its contents. Example of Scheme expression represented by messages:

```
> (define x 2)        ⇔          > (assertion (define x 2))
: x                   ⇔          : (ack x)
> x                   ⇔          > (request x)
: 2                   ⇔          : (answer 2)
```

This is a very interesting point of view because all the advantages related to Scheme for knowledge representation profit to agents, in particular the memory model provided by first class environments. For example, STROBE proposes an environment structure preserving history of values, i.e. bindings are not any more pairs like (var val) but are (var valn valn-1... val1). This structure becomes accessible to agents represented by interpreters.

To implement this model we wrote a Scheme meta-evaluator that recognizes a certain language and we added to it a messages interpretation module. Scheme is very useful because it is easy to conceive the interpretation process by abstracting both on the memory model (by abstracting on the environment [CER 99]) and on the control model (by abstracting on continuations [QUE 00]). It clarifies the three levels: *data*, *control* and *interpretation*. *Data* level learning consists in assigning values to already existing variables; *control* level learning consists in defining new variables or functions. And finally, the most interesting for us, *interpretation* level learning or, meta-level learning, consists in making evolve the Scheme interpreter itself. That's why, while making evolve its interpreter, an agent learns more than simple Information; he completely changes his way of perceiving and processing this Information. Here is the difference between learning a datum and learning how to process a class of data.

4. Representation of the others

STROBE performatives are translated into a Scheme expression and cause on the receiver agent a certain behaviour. In particular, updating its ”partner model”. In fact for the representation of each partner, STROBE proposes to have a partner model to be able to rebuild his internal state. This model proposes to interpret each dialogue in a pair of environments: the first, private, belongs to the agent and the second represents the current partner model. This is called Cognitive Environment [CER 96]. Our work exploits this concept. In fact, it is based on because, as the Cognitive Environment concept provides to the agents a global environment (or private) and several local environments representing the others, **our proposal provides to the agents not a message evaluator but several evaluators, including a global (or private) one and a specific one for each agent they have a representation of**. Thus, messages interpretation is done in both a given environment and an interpreter. Our work is based on this concept of Cognitive Environment because, to be accessible, these interpreters must be themselves stored in these environments. Thus our agents have the three following attributes:

- GlobalEnv their global environment.
- GlobalInter their global interpreter.
- Other = {(name, interpreter, environment)} a set of triplets corresponding to the representations of the others.

Their global environment is private and does not change. It is cloned[2] when a new conversation starts, and it is the clone, stored in an element of Other, which is progressively modified along the conversation. Figure 1 illustrates these representations.

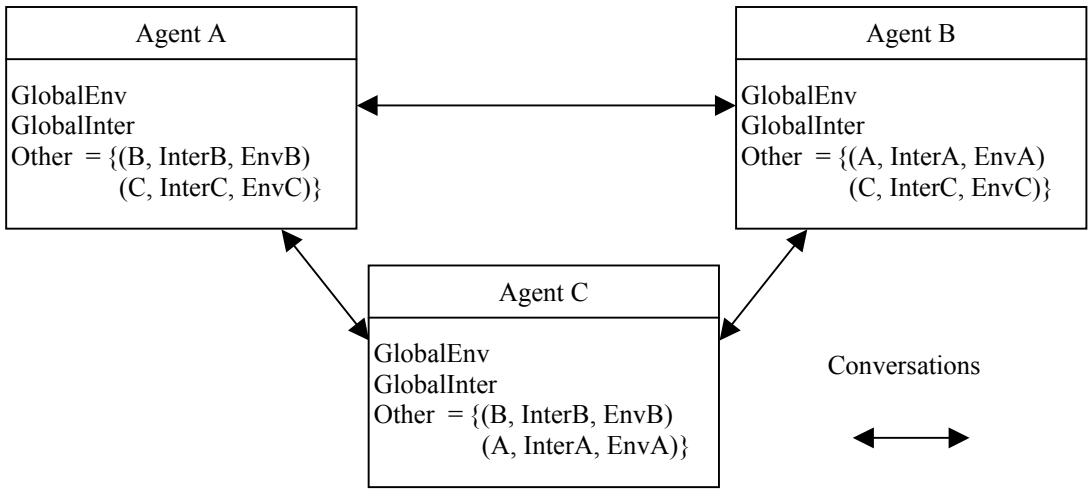


Figure 1. *The three attributes that define an agent and its representations*

Now we have both our agent's structure and their representations structure. In order to ensure learning by communication, it is now necessary to clarify devices that allow us to dynamically and progressively modify, by conversations, agent’s interpreter(s).

[2] Not necessarily if we consider an agent that would take back a conversation in another context, already existing (with this environment and this interpreter).

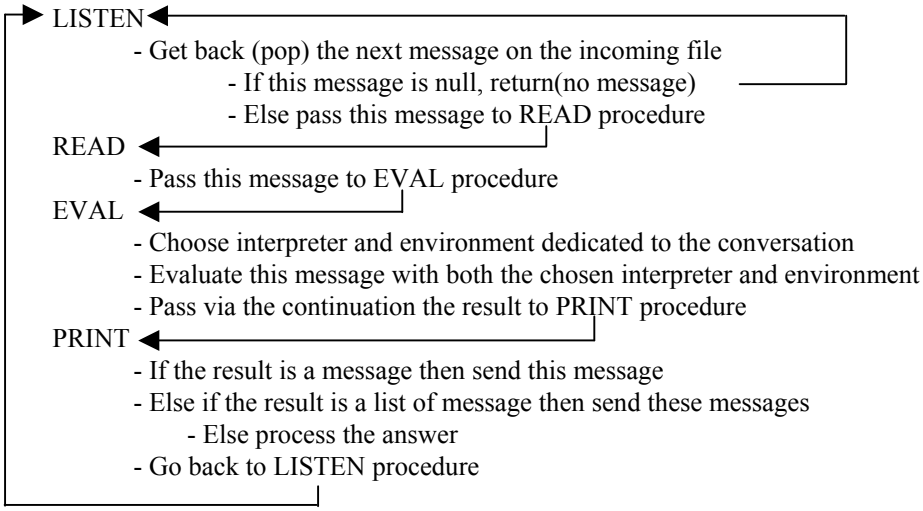


Figure 2. *The REPL loop of our agents*

5. Dynamic modification of an interpreter

5.1. The Scheme meta-evaluator

The meta-evaluator we used comes from the famous article written by Jefferson and Friedman, "A Simple Reflective Interpreter" [FRI 92]. This evaluator is user friendly (both easy to use and learn) and provides the reflexivity property we need. Moreover, it proposes the reifying procedure mechanism, which as we will further show, allows a user's program to access its execution context[3] and, if it is required, to modify it. This evaluator has the signature `(evaluate expression environment continuation)`. It is mainly defined by two procedures `evaluate` and `apply-procedure`[4], corresponding to the two steps of the environment evaluation.

5.2. Reflexivity and reification

About reflexivity, the evaluator proposed in [FRI 92] is already implemented in a reflexive way. It means that the evaluator code is written in the sub-language of Scheme that it processes. But Jefferson and Friedman also propose, in their article, an architecture called reflexive tower, a tower of interpreters. The interpreter at the bottom of the tower executes user input, and every other interpreter in the tower executes the interpreter immediately below it. Two mechanisms allow "moving" in the tower in order to evaluate the code at any level: reflexivity allows going up and reification allows going down. These two points are particularly interesting to us.

Reifying procedures provide the mechanism required to access to the execution context of the interpreter below. The main idea is that user's programs could have the same access rights as the interpreter itself. Owing to that, procedures implementing the evaluator can be accessed and modified by user's programs in the same way as the environment and the continuation. This property makes reifying procedures the ideal tool to dynamically modify our interpreters. Indeed, access to the execution context means access to the environment in which procedures defining the evaluator are stored and thus, means being able to modify them. Furthermore, as we saw it, reification is the method to go down in the evaluation levels. So, to apply a reifying procedure we need two levels of evaluation. Therefore, the whole experimentation must be evaluated by our meta-evaluator to carry out the first call to evaluate, the second being carried out by the agent itself when it receives a message. Thus, reflexivity of this one is obligatory: our meta-evaluator must be able to be evaluated itself.

[3] The execution context of an expression is defined by the environment of evaluation of this expression and the continuation to give to this evaluation.
[4] Corresponding to `eval` and `apply` of [ABE 96] and `evaluate` and `invoke` of [QUE 94].

5.3. Communication protocol

The messages we consider are inspired by the KQML or FIPA ACL[5] message structure. They will be identified from now on `kqmlmsg`, with the signature:

`(kqmlmsg performative sender receiver content)`.

In order our meta-evaluator interpret these `kqmlmsg` messages, it is necessary to add to it a test which recognizes them in the `evaluate` procedure, and a function which treats them by evaluating their contents, `evaluate-kqmlmsg`. Used performatives are: *assertion*, *request*, *order*, *ack*, *answer*, *executed* (cf. table 2 [CER 99]) and, as we will see, *broadcast*, which is the subject of the experimentation below. When an agent receives a message indexed by an unknown performative it answers by a message with `(no-such-performative performative)` content:

- *assertion* messages modify interlocutor behaviour or some of is representations. Their answers are acknowledgement (*ack*) message reporting a success or an error.
- *request* messages ask for one interlocutor representation, such as the value of a variable or the closure of a function. Their *answer* returns a value or an error.
- *order* messages require the interlocutor to apply a procedure. This interlocutor sends the result as the content of an *executed* message.
- *broadcast* messages consist in sending a message with a pair as content `(perform, content)` that means that the interlocutor must send a message with the performative `perform` and with the content `content` to all is current interlocutors. There is no answer defined for the broadcast messages.[6]

6. Experimentation: a "teacher-student" dialogue

The main idea of our work is to benefit from the learning side effect of communication. Indeed, the goal of education is to change the interlocutor's state. This change is done after evaluating new elements brought by the communication. To do that, we will use the reflexivity and reification properties already seen. With our reflective interpreter, we show that an agent can modify its way of seeing things (i.e. of evaluating messages) by "re-evaluating" its own evaluator at the end of a conversation. The experimentation we present here is a standard "teacher-student" dialogue.

An agent *teacher* asks to another agent *student* to broadcast a message to all its correspondents. However, *student* does not initially know the performative used by *teacher*. So, *teacher* transmits it a series of messages (*assertion* and *order*) clarifying *student* the way of processing this performative. Finally, *teacher* formulates again its request to *student* and obtains, this time, satisfaction. Figure 3 describes the exact dialogue realized in the experimentation.

For the experimentation we developed some agents able to communicate, i.e. exchanging messages together producing significant answers (following the defined protocol) [7]. They do not do anything when they do not communicate and their autonomy is defined by the fact that they can learn. They have the following attributes: name, globalEnv, globalInter, other, two files of messages (in/out), and a data structure storing the current conversation. Their behaviour consists in applying the REPL loop (Figure 2).

---

[5] Our protocol being extremely simplified, our messages specify only 4 parameters. However, we could have implemented other ones: language, ontology, in-reply-to, reply-with, etc.

[6] *broadcast* is a meta-performative, in the sense that it requires other agents to evaluate any performative.

[7] Our agents are in fact Scheme programmed object as we can see in the Normark article *Simulation of Object-Oriented and Mechanisms in Scheme* [NOR 91]

TEACHER	STUDENT
Here is the definition of square procedure: **(kqmlmsg 'assertion teacher student '(define (square x) (* x x)))**	Ok, I know now this procedure: **(kqmlmsg 'ack student teacher '(*.*))**
Broadcast to all your current correspondents: **(kqmlmsg 'broadcast teacher student '(order (square 3)))**	Sorry, I don't know this performative: **(kqmlmsg 'answer student teacher `'(no-such-performative broadcast))**
Ok, here is the method to add this performative to those you know: Here is the code you have to generate and add to your `evaluate-kqmlmsg` function: **(kqmlmsg 'assertion teacher student learn-broadcast-code-msg)** After, here is the reifying procedure which allows you to change this code: **(kqmlmsg 'assertion teacher student learn-broadcast-msg)** Run this procedure: **(kqmlmsg 'order teacher student call-learn-broadcast)**	Ok, I have added this code in a binding of my environment: **(kqmlmsg 'ack student teacher '(*.*))** Ok, I know now this procedure: **(kqmlmsg 'ack student teacher '(*.*))** Ok, I have just modified my evaluator: **(kqmlmsg 'executed student teacher '(*.*))**
Broadcast to all your current correspondents: **(kqmlmsg 'broadcast teacher student '(order (square 3)))**	Ok, I broadcast...

Figure 3. *Broadcast teaching "teacher-student" dialogue*

Based on these explanations, we can now describe precisely how the process of the new performative learning is carried out. We will detail *teacher* messages sent to *student* in order to modify its interpreter. The first message is `learn-broadcast-code-msg`. This variable corresponds in the `teacher` environment to the new `evaluate-kqmlmsg` function code that student must generate to, later, assign it to its function. This code is built by *student* by recovering its `evaluate-kqmlmsg` function body and by adding to it `(if (eq? performative ' broadcast)` and associated treatment. It is a constructivist learning point of view[8]. Thus, *teacher* environment has the following binding:

```
learn-broadcast-code-msg:
 `(define learn-broadcast-code
       (let ((newproc
          (let ((oldproc *evaluate-kqmlmsg code recorvering*))
               *oldproc is updated by adding new code (if (eq? performative…*)
          newproc))
```

The second message sent by *teacher* is the most significant. It defines the reifying procedure `learn-broadcast` that will modify *student* evaluator. *Teacher* environment has also the following binding:

```
learn-broadcast-msg :
 `(define learn-broadcast (compound-to-reifier
                                  (lambda (e r k) (evaluate (car e) r k))))
```

[8] It is important to notice that it is *student* who rebuild his function with the existing one in his environment and not *teacher who* send his `evaluate-kqmlmsg` function code. It allows *student* to keep some previous modification of his function.

During its evaluation, `learn-broadcast` is transformed into a call to a compound procedure with the following arguments: list of its arguments, the environment in which it must be evaluated and the continuation which it is necessary to give to this evaluation. Then, the execution context of `learn-broadcast` becomes accessible and modifiable! In our case, `learn-broadcast` evaluates `car` of its arguments list. Finally, *teacher* solicits *student* to apply its function with a call to `set!` (modifing `evaluate-kqmlmsg` by the code generated before). Figure 4 illustrates the steps of evaluation of this reifying procedure. Finally, the last binding is:

```
call-learn-broadcast :
 '(learn-broadcast (set! evaluate-kqmlmsg learn-broadcast-code))
```

After the last message process, *student* `evaluate-kqmlmsg` function is modified as well as its messages interpreter. The corresponding function code in its environment dedicated to this conversation is changed. Then *student* agent can process broadcast messages.

When the call to `learn-broadcast` combination is evaluate...

```
(evaluate
   (learn-broadcast (set! evaluate-kqmlmsg learn-broadcast-code))
   *good env*
   *good cont*)
```
[9]

...`learn-broadcast` is changed by is value in *student* environment...

```
(evaluate
   ((reifier (e r k) (evaluate (car e) r k) *good env*)
    (set! evaluate-kqmlmsg learn-broadcast-code))
   *good env*
   *good cont*)
```
[10]

...then the reifying procedure is transform in a `compound` procedure...

```
(evaluate
   ((compound (e r k) (evaluate (car e) r k) *good env*)
    (set! evaluate-kqmlmsg learn-broadcast-code))
   *good env*
   *good cont*)
```

...and `evaluate` call `apply-procedure`...

```
(apply-procedure
   (evaluate (car e) r k)
   ((set! evaluate-kqmlmsg learn-broadcast-code))
   *good cont*)
```

...which call to `evaluate` with `*good env*` and `*good cont*` accessible.

```
(evaluate
   (evaluate (car e) r k)
   ((e ((set! evaluate-kqmlmsg learn-broadcast-code)))
    (r *good env*) (k *good cont*)
    *good env*)
   *good cont*)
```

Figure 4. *Steps of the learn-broadcast reifying procedure evaluation*

[9] *good env* and *good cont* correspond to the environment and the interpreter dedicated to the conversation.

[10] A closure is stored in the environment as `(name type parameter body defenv)`. `type` can be `compound`, `reifier` or `primitive`, `defenv` is the definition environment.

7. Interests and extension

Our experimentation shows how to add a new performative to the ones already known by an agent, thus how to modify an agent messages interpretation function. However, the same principles can be used to modify any part of an agent's interpreter. For instance, we could have made an example which adds `cond` or `let*` to our language recognized by our meta-evaluator. Even more, an agent could teach to another how to make its evaluator lazy by changing some functions (`evaluate` and `apply-procedure`). With this protocol, our agents have a set of interpreters that represent their knowledge. Indeed, these interpreters correspond to their recognized sub-languages and thus to their faculties to carry out a task. As seen in the "ideal" scenario, agents can process programs for others or even exchange their interpreters[11]. Their interpreters can also be transmitted before a conversation. The "ontology" abstraction in ACLs is, therefore, extended by our model with an abstraction on the ACL itself. This may allow to experiment with ACLs equipped with different semantics [GUE 02] in order to choose, in a specific context the most adequate ACL.

Let us imagine an agent society or MAS that follows this model. Any agent can learn something from another. If an agent is built with a minimum of knowledge (to interact) and a speciality, then it can diffuse it progressively with its conversations. These principles are very interesting for the Web. Let us consider a new Web application server agent that uses a set of performatives corresponding exactly to its job. If it is built with the potential to teach these performatives then it will easily be integrated into an agents society by teaching these. Moreover, to extend these principles to XML, we can regard a DTD or an XML-Schema as a XML[12] data interpreter. Then the presented model allows these DTDs to evolve dynamically and idem for the associated documents XML; giving to the Web dynamism and adaptability.

The presented work follows the same line as the language C+C [CER 00] and the work reported in [GUE 99] that propose to provide to the agents a dynamic scheduling algorithm. When an object answers to a method call it does not contest or ask it why it must answer. However, an agent behaviour depends on its scheduling algorithm that enables it to be autonomous and to decide if and when it devotes time to the others agents. If an object implementation is fixed, an agent has a scheduling algorithm evolving during the time. Our reflective interpreters can embody dynamic scheduling algorithms.

Principles presented here could also be useful in others scenarios. For example, instead of the procedure square definition, let us imagine that *teacher* transfers to *student*, a procedure that implements an optimised algorithm to solve a problem. Consider, for instance, `(memo-fib n)`, which is the Fibonacci algorithm version that takes into account the memoization principle [ABE 96], transforming an exponential Fibonacci algorithm into a linear one (an example of dynamic programming). In this case, *student*, after broadcast learning, could choose some agents to perform some heavy computation using the Fibonacci algorithm as follow: It can asks all its current correspondents to process a Fibonacci number and after receiving all the answers, compares answer times to decide which of them are selected. It can do it because it has its own reference for this processing. This idea could be particularly interesting for communication protocols such as *contact net*, or for Grid computing where "effective" agents have to be selected to perform heavy computation.

8. Conclusion

We tried to show in this paper a learning method for cognitive agents based on communication. This learning process can be realised by simple communication (*data* or *control* level), or by agent internal modification (*interpreter* level). If agents interpret in a dynamic way their messages, they become adaptable and, without any external intervention, can communicate with entities that they have never met before. Moreover, as their evaluator is modified to acquire knowledge, it could be also modified to learn how to teach knowledge. Then knowledge transfer would progressively become possible and pertinent with communications. This paper does not simply propose another programming artefact to add to agents, but the idea is rather to show a technique, simple, usable and friendly, of autonomous evolution of agents in a society.

---

[11] It comes from the idea quoted from [ABE 96]: "*If we wish to discuss some aspect of a proposed modification to Lisp with another member of the Lisp community, we can supply an evaluator that embodies the change. The recipient can then experiment with the new evaluator and send back comments as further modifications"*

[12] Because XML documents are tree and Scheme is very handy to process on tree. (cf. [KIS 02])

9. References

[AUS 70] Austin J.L., *Quand dire c'est faire*, Edition du seuil, Paris, 1970.

[ABE 96] Abelson H., Sussman G.J., Sussman J., *Structure and Intepretation of Computer Programs*, Second Edition, MIT Press, Cambridge, Massachusetts, 1996.

[CER 96] Cerri S. A., "Cognitive Environments in the STROBE Model". Presented at EuroAIED: *the European Conference in Artificial Intelligence and Education*, Lisbon, Portugal (1996)

[CER 99] Cerri S.A., "Shifiting the Focus from Control to communication: The STReams OBject Environments (STROBE) model of communicating agents", In Padget, J.A. (ed.) *Collaboration between Human and Artificial Societies, Coordination and agent-Based Distributed Computing*, Berlin, Heidelberg, New York: Springer-Verlag, Lecture Notes in Artificial Intelligence, pp 71-101, 1999.

[CER 00] Cerri S.A., Sallantin J., Castro E., Maraschi D. "Steps towards C+C: a Language for Interactions", In Cerri, S. A., Dochev, D. (eds), *AIMSA2000: Artificial Intelligence: Methodology, Systems, Applications*, Berlin, Heidelberg, New York: Springer Verlag, Lecture Notes in Artificial Intelligence, pp 33-46, 2000.

[MAR 01] Maraschi D., Cerri S. A., "The relations between Technologies for Human Learning and agents", In procedings of the *AFIA 2001 Atelier: Methodologies et Environments pour les Systèmes Multi-agents,* Grenoble: Leibniz-Imag, pp. 61-73, 2001.

[DER 01] De Roure D., Jennings N., Shadbolt, N. "Research Agenda for the Semantic Grid: A Future e-Science Infrastructure" In Report commissioned for *EPSRC/DTI Core e-Science Programme*. University of Southampton, UK, 2001.

[DIG 00] Dignum F., Greaves M., "Issues in agent Communication: An introduction", Dignum F and Greaves M.(Eds.): *agent Communication, LNAI 1916*, pp1-16, Springer-Verlag Berlin Heidelberg, 2000.

[FER 95] Ferber J., *Les Systemes Multi-agents, vers une intelligence collective*, InterEditions, Paris, 1995.

[FRI 92] Friedman D.P., Jefferson S., "A Simple Reflective Interpreter", *IMSA'92, International Workshop on Reflection and Meta-Level Architecture*, Tokyo, 1992.

[GUE 02] Guerin, F. "Specifying agent Communication Languages", *Ph.D. Thesis*, Dept. of Electrical and Electronic Engineering, Imperial College, 2002.

[GUE 99] Guessoum, Z. ,Briot, J.-P. "From Active Objects to Autonomous agents". IEEE Concurrency, vol. 7-3, pp. 68-76, 1999

[IST 01] ISTAG, "Scenarios for Ambient Intelligence in 2010", Final Report Compiled by K. Ducatel, M. Bogdanowicz, F. Scapolo, J. Leijten & J-C. Burgelman, IPTS-Seville, February 2001.

[KIS 02] Kiselyov O., "XML, Xpath, XSLT implementations as SXML, SXPath, and SXSLT", *International Lisp Conference: ILC2002*, San Francisco, CA, Octobre 2002.

[MCC 89] McCarthy J., "Elephant 2000: A Programming Language Based on Speech Acts". *Unpublished draft* Stanford University, www-formal.stanford.edu/jmc/elephant.pdf, 1989.

[NOR 91] Normark K., "Simulation of Object-Oriented and Mechanisms in Scheme", *Institute of Electronic Systems*, Aalborg University, Denmark, 1991.

[QUE 94] Queinnec C., *Les langages LISP*, Interéditions, Paris, 1996.

[QUE 00] Queinnec C., "The Influence of Browsers on Evaluators or, Continuations to Program Web Servers", *ICFP'00*, Montréal, Canada, 2000.

[SEA 71] Searle J., *Les actes de langages, essai de philosophie du langage*, Herman Editeur, Paris 1971.

10. Annexe

The presented model was the subject of a small implementation, witch is still not completed today, but already functional. It is available on line on http://www.lirmm.fr/~jonquet. It was develop with MIT Scheme 7.7.1, standard R5RS. You will find below the Scheme files specifying the meta-evaluator used for the experimentation, the `kqmlmsg` message interpreter module, the file implementing our agents, and finally the file corresponding to the experimentation.

11. Acknowledgements

This work was performed to fulfil in part the requirements for a DEA Informatique (M.Sc. in Computer Science) by one of the authors (CJ). The support of the EU project LEGE-WG (Learning Grid Excellence Working Group) is gratefully acknowledged.

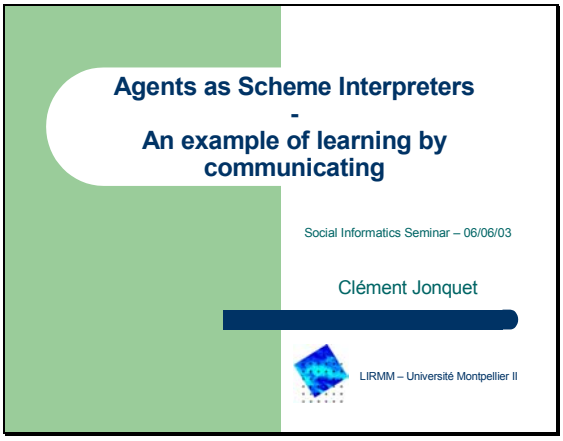
Annexe E. Soumission à RFIA 2004

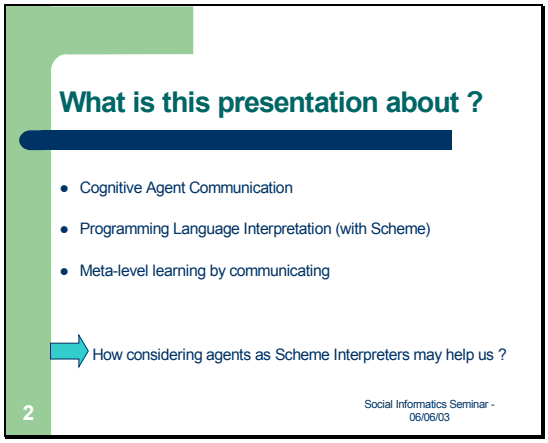
Ce travail a également fait l'objet d'une soumission à la conférence « 14ème Congrès Francophone AFRIF-AFIA de Reconnaissance des Formes et Intelligence Artificielle – RFIA 2004 », Toulouse, France, 28-30 janvier 2004, présidé par Jean Charlet (STIM/AP-HP, Paris) et Michel Dhome (LASMEA, Clermont-Ferrand). Les réponses sont attendues pour le 30 septembre 2003. L'article est rédigé en anglais et fait une dizaine de pages. Il présente rapidement notre modèle et l'expérimentation, et détaille par ailleurs l'exemple d'application au e-commerce (spécification dynamique). Le fichier est disponible en ligne sur http://www.lirmm.fr/~jonquet mais étant donné qu'il n'était pas entiérement fini lors de la rédaction de ce mémoire il n'est pas joint ici.

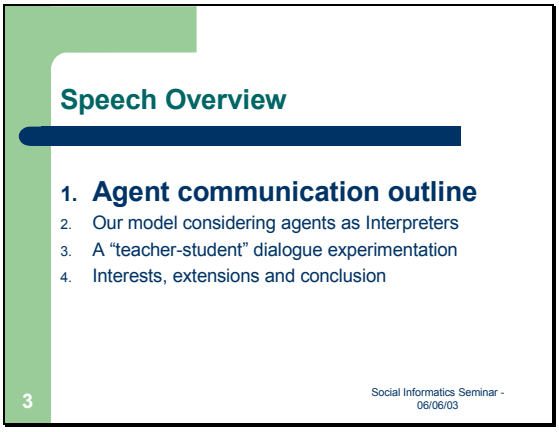
Pour plus de renseignements sur cette conférence : http://www.laas.fr/rfia2004/

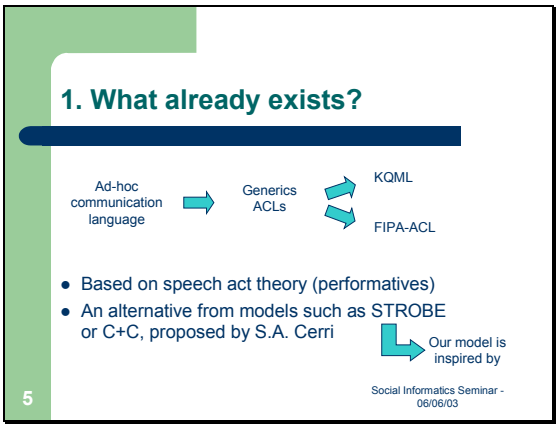
Annexe F. Séminaire Social Informatics

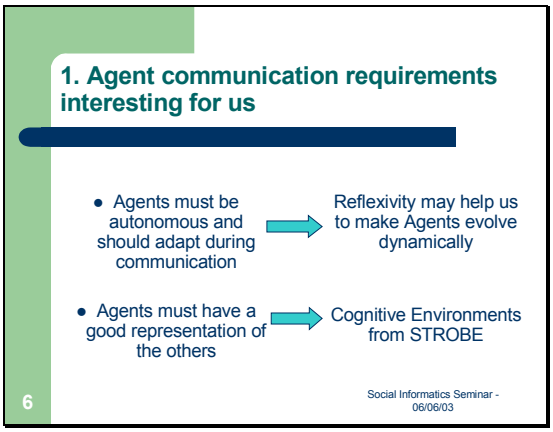
Le travail présenté dans ce mémoire a également fait l'objet d'un séminaire au LIRMM le 06/06/03 organisé par l'équipe E-dialogue. Cette présentation s'est déroulée en anglais et a durée approximativement une heure. Le fichier correspondant aux transparents de ce séminaire est disponible en ligne sur http://www.lirmm.fr/~jonquet.











Speech Overview

1. Agent communication outline
2. **Our model considering Agents as Interpreters**
3. A "teacher-student" dialogue experimentation
4. Interests, extensions and conclusion

2. Agents As Scheme Interpreters

- Agents as REPL (Read-Eval-Print-Listen) interpreters
- During communication Agents execute a REPL loop
- Evaluate both the message and its content

⇒ Agents = Autonomous entity controlled by a procedure (`evaluate`)

⇒ How to make evolve these interpreters dynamically ?

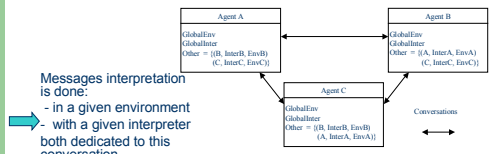
2. What about Scheme ?

- Memory model provided by first class environments
- Control model provided by continuations
- 3 knowledge-levels modification:

Data	Control	Interpreter
⇩	⇩	⇩
(set! a 3)	(define square …)	Add a special form

2. Representation of the others

- STROBE: Cognitive Environments
- In these environments we add an interpreter



Messages interpretation is done:
- in a given environment
- with a given interpreter

both dedicated to this conversation

2. Our agents' behaviour

```
LISTEN
    - Get back (pop) the next message on the incoming file
        - If this message is null, return (no message)
        - Else pass this message to READ procedure
READ
    - Pass this message to EVAL procedure
EVAL
    - Choose interpreter and environment dedicated to the conversation
    - Evaluate this message with both the chosen interpreter and environment
    - Pass via the continuation the result to PRINT procedure
PRINT
    - If the result is a message then send this message
    - Else if the result is a list of message then send these messages
        - Else process the answer
    - Go back to LISTEN procedure
```


2. What devices are we going to use ?

- Our reflective meta-evaluator (inspired by the one of Jefferson and Friedman)
- Reflexivity and reification and…
- Reifying procedures mechanism, that allows user programs to access their evaluation context

⇒ Ideal tool to dynamically modify our Agents' interpreters

2. Communication protocol

- Message format:

  (kqmlmsg performative sender receiver content)

- Performatives:
  - ***assertion*** ex: `(define square …)`? answers: ***ack***
  - ***request*** ex: `a`? answers: ***answer***
  - ***order*** ex: `(square a)`? answers: ***executed***
  - ***broadcast*** ex: `(order (square 3))`? no answers

⇒ Message evaluation procedure `evaluate-kqmlmsg`

Speech Overview

1. Agent communication outline
2. Our model considering Agents as Interpreters
3. **A "teacher-student" dialogue experimentation**
4. Interests, extensions and conclusion

3. "Teacher-Student" experimentation dialogue

- Idea: benefit of the learning side effect of communication
- *Student* learns the ***broadcast*** performative by being told by *teacher*: its meta-level learning
- We develop agents able to communicate

3. Dialogue details and demonstration

TEACHER	STUDENT
Here is the definition of square procedure: `(kqmlmsg 'assertion teacher student '(define (square x) (* x x)))`	Ok, I know now this procedure: `(kqmlmsg 'ack student teacher '(*.*))`
Broadcast to all your current correspondents: `(kqmlmsg 'broadcast teacher student '(order (square 3)))`	Sorry, I don't know this performative: `(kqmlmsg 'answer student teacher '(no-such-performative broadcast))`

Ok, here is the method to add this performative to those you know:
Here is the code you have to generate and add to your `evaluate-kqmlmsg` function:
`(kqmlmsg 'assertion teacher student learn-broadcast-code-msg)`
After, here is the reifying procedure which allows you to change this code:
`(kqmlmsg 'assertion teacher student learn-broadcast-msg)`
Run this procedure:
`(kqmlmsg 'order teacher student call-learn-broadcast)`
	Ok, I have added this code in a binding of my environment:
	`(kqmlmsg 'ack student teacher '(*.*))`
	Ok, I know now this procedure:
	`(kqmlmsg 'ack student teacher '(*.*))`
	Ok, I have just modified my evaluator:
	`(kqmlmsg 'executed student teacher '(*.*))`

Broadcast to all your current correspondents: `(kqmlmsg 'broadcast teacher student '(order (square 3)))`	Ok, I broadcast...

3. Teacher's messages details

1. The `evaluate-kqmlmsg` procedure that student must generate
2. Reifying procedure definition
3. Reifying procedure application

`evaluate-kqmlmsg` ***student*** procedure is modified: It can perform the ***broadcast*** performative

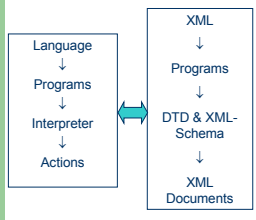
Speech Overview

1. Agent communication outline
2. Our model considering Agents as Interpreters
3. A "teacher-student" dialogue experimentation
4. **Interests, extensions and conclusion**

4. Interests of this model

- Other modification of Agent interpreters are possible
- An example of Web server autonomy
- Replacing ontologies ?

4. XML Rapprochement



- We can consider DTD or XML-Schema as XML interpreters
- DTD & XML-Schema may evolve dynamically with our model
- Web dynamism, plurality and adaptability

4. Extension: Non Deterministic Evaluation

- We can consider our Agents as non deterministic interpreters (Abelson et Sussman)
- Agents can process the `amb`, `require`, and `try-again` specials forms
- Interesting for constraint programming

Construct constraints programs by communicating

4. Non Deterministic Evaluation: An example

Logic puzzles:

"Baker, Cooper, Fletcher, Miller, and Smith live on different floors of an apartment house that contains only five floors.

```
(define (multiple-dwelling)
  (let ((baker (amb 1 2 3 4 5))
        (cooper (amb 1 2 3 4 5))
        (fletcher (amb 1 2 3 4 5))
        (miller (amb 1 2 3 4 5))
        (smith (amb 1 2 3 4 5)))
   (require (distinct? (list baker cooper
  fletcher miller smith)))
```

Baker does not live on the top floor.

```
(require (not (= baker 5)))
```

Cooper does not live on the bottom floor.

```
(require (not (= cooper 1)))
```

Fletcher does not live on either the top or the bottom floor.

```
(require (not (= fletcher 5)))
(require (not (= fletcher 1)))
```

Miller lives on a higher floor than does Cooper.

```
(require (> miller cooper))
```

Smith does not live on a floor adjacent to Fletcher's.

```
(require
  (not (= (abs (- smith fletcher)) 1)))
```

Fletcher does not live on a floor adjacent to Cooper's.

```
(require
  (not (= (abs (- fletcher cooper)) 1)))
```

Where does everyone live?"

```
(list (list 'baker baker)
      (list 'cooper cooper)
      (list 'fletcher fletcher)
      (list 'miller miller)
      (list 'smith smith))))
```

4. Non deterministic evaluation: modelling e-commerce dialogues

CLIENT	SNCF

I want a ticket from Montpellier to Paris
`(require (eq? depart montpellier))`
`(require (eq? dest paris))`
	Definition of a new `find-ticket` procedure :
	`(define (find-ticket)`
	`  (let`
	`  ((depart (amb 'ens-ville*)) (dest (amb 'ens-ville*))`
	`  (prix (amb 'ens-prix*)) (date (amb 'ens-data*)`
	`  (require (not (eq? depart dest)))`
	`  (require (eq? depart montpellier))`
	`  (require (eq? dest paris))`
	`  (list (list 'depart depart)`
	`  (list 'destination dest)`
	`  (list 'prix prix)`
	`  (list 'date date))))`
	When ?

Tomorrow before 10AM
`(require (< date 'demain10H*))`
Please, give me a proposition for a ticket!
`(find-ticket)`
	`find-ticket` procedure modification adding a new constraint. Then procedure execution:
	Ok, train 35, departure tomorrow 9.30AM, from Montpellier to Paris, 150€
	`((depart montpellier) (destination paris) (prix 150) (date 'dem9H30*))`

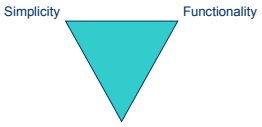
Is it possible to pay less than 100€ ?
`(require (< prix 100))`
`(find-ticket)`
	idem
	Ok, train 31, departure tomorrow 8.40AM, from Montpellier to Paris, 95€
	`((depart montpellier) (destination paris) (prix 95) (date 'dem8H41*))`

Another proposition please ? `(try-again)`	`find-ticket` procedure execution: *Ok, train 32, departure tomorrow 9.15AM, from Montpellier to Paris, 98€* `((depart montpellier) (destination paris) (prix 98) (date 'dem9H15*))`
Ok, I accept this one…	

4. Extension: GRID Computing

- *Teacher* teaches an optimised algorithm procedure such as `fact-iter` or `memo-fib`
- *Student* learns the ***broadcast*** performative
- Then, *student* may play a GRID Computing server role:
  - Ask its correspondents to perform a little computation
  - Depending on the answer time, choose the best in order to perform a heavy computation

Conclusion



Meta level agents learning by communicating

Extraordinary adaptability and autonomy for MAS

Important potential applications (Grid, e-commerce, Web..)