# CRAT, vers le Scheme Conversationnel

Emmanuel Castro[1,2], Jean Sallantin[1], Stefano A. Cerri[1]

[1]LIRMM
[2]EURIWARE

LIRMM
151 rue Ada, Montpellier, France
34392 Cedex 5
{castro, js, cerri}@lirmm.fr

## Résumé

*Dans cet article, nous exposons les problèmes rencontrés pour rendre interopérable aussi bien des programmes que des agents. Nous montrons qu'il est nécessaire de considérer la construction des interfaces de communication entre logiciels au moment de leur exécution. Ensuite, nous posons un cadre d'étude des communications entre programmes et entre agents, appelé CRAT, basé sur l'évaluation d'expressions Scheme. Dans ce cadre, nous constatons la nécessité d'annoter sémantiquement les informations échangées par les programmes afin de pouvoir en assurer l'interopérabilité. Puis nous constatons que dans ce cadre, il est nécessaire de savoir gérer les ambiguïtés. Enfin nous proposons l'ébauche d'un langage pour utiliser CRAT : le Scheme Conversationnel.*

## Mots Clefs

Conversation, Agents, Messages, Interopérabilité, Scheme

## 1. Introduction

Depuis son apparition, le Web a profondément modifié la manière de faire de l'informatique. Il fait maintenant partie intégrante des systèmes d'exploitation des ordinateurs, comme en témoigne l'indissociabilité d'Internet Explorer et de Windows. Il n'a pas fini de rentrer au cœur des problèmes de développement des logiciels, et notamment des Agents. Dans cet article, nous étudions les problèmes que le Web pose dans la manière de développer des Agents, par exemple des agents de commerce électronique. Pour résoudre certains de ces problèmes, nous proposons de modifier la manière d'exécuter les programmes, ce que nous matérialiserons par le Scheme Conversationnel.

Avant d'aller plus loin, nous devons d'abord clarifier un peu le mot *Agent*. Wooldridge et Jennings en donnent une définition générale dans [1]. Pour eux, ce sont des machines capables d'autonomie, de capacités sociales (échange de messages), de réactivité et de pro-activité (prise d'initiative). À l'opposé, Newell en donne une définition plus conceptuelle que logicielle en disant que tout système (informatique), vu au niveau des Connaissances (*Knowledge Level*), est un Agent [2].

Les Agents qui nous intéressent dans cet article sont ceux évoluant sur le Web, c.-à-d. des Agents agissant toujours pour le compte d'un utilisateur [3]. Ils peuvent bien évidemment être des logiciels, mais nous pensons qu'il faut aussi y inclure les Personnes Humaines. Considérer une Personne comme un Agent permet d'intégrer explicitement les utilisateurs dans le système informatique à travers les messages qu'ils échangent, c.-à-d. mettre *l'utilisateur dans la boucle* [4].

Le but d'un Agent est de proposer des Services, c'est-à-dire des actions que d'autres agents lui demandent de faire. Là encore, l'utilisateur intervient. Pour fournir son Service, l'Agent a besoin d'informations dont un grand nombre ne sont accessibles qu'en interrogeant l'utilisateur client de ce Service. Il convient donc d'étudier comment programmer des agents traitant des informations dont l'accès dépend, directement ou indirectement, de la bonne volonté des utilisateurs.

Le Web apporte avec lui un autre problème : l'interopérabilité. Actuellement, sur le Web, n'importe qui peut publier un page-web que n'importe quel internaute pourra lire, sans problème d'interopérabilité, pourvu qu'il en ait obtenu l'URL. Encore, un site-web marchand peut faire évoluer son contenu et sa structure pour proposer de nouvelles offres commerciales sans risquer de problèmes d'interopérabilité avec ses futurs acheteurs. Cette construction complètement décentralisée du Web ne pose pas de problèmes d'interopérabilité quand les lecteurs des pages sont des humains. Mais quand on développe des agents automatisant l'accès à des pages-web, cette versatilité du Web est un obstacle. Elle l'est encore plus quand on veut mettre à disposition, non plus de simples pages-web, mais des agents. La mise à disposition d'agents de commerce électronique en est un bon exemple car la pression commerciale les poussera à évoluer sans cesse. Il était déjà difficile de permettre l'interopérabilité des agents au niveau des protocoles de communication, le Web rend encore plus difficile cette interopérabilité au niveau de la description sémantique des informations qui y sont échangées [3], surtout si ce qu'il faut décrire évolue sans cesse [4].

À partir du moment où des humains partagent à peu près la même langue et des connaissances suffisantes dans le domaine sur lequel portent des pages-web, le problème de l'interopérabilité ne se pose pas. Cependant pour des programmes il est réel. En effet, pour échanger des informations et interagir, ils ont besoin d'interfaces de communication. Elles sont traditionnellement définies une fois pour toutes lors de la conception du logiciel. Dans le contexte du Web, elles risquent d'être rapidement périmées. Elles ne pourront évoluer qu'avec la sortie d'une nouvelle version du logiciel. Les humains n'ont pas ce problème, ils savent adapter leurs interfaces de communication très rapidement, sans qu'il soit nécessaire de les « reprogrammer ». Il leur suffit d'apprendre de nouveaux mots, de nouvelles idées, ou de nouveaux gestes. Nous proposons donc de faire des programmes qui, comme le font les hommes, construisent leurs interfaces de communications à *run-time*.

Dans la suite de cet article, après avoir posé le scénario de l'exemple sur lequel nous travaillerons, nous étudierons plus en détails les problèmes de l'interopérabilité et les différentes stratégies pour l'atteindre. Puis nous proposerons le cadre CRAT permettant de voir les agents comme des fonctions, et les communications entre agents comme le passage de paramètres à ces fonctions. Ensuite nous montrerons le besoin d'avoir une représentation de la sémantique des informations échangées entre les agents, et les problèmes d'ambiguïté qu'elle amène. Enfin nous terminerons par l'esquisse d'un langage abstrayant les mécanismes que nous aurons mis en œuvre : le Scheme Conversationnel.

Mais d'abord, posons le scénario de commerce électronique impliquant des personnes et des agents logiciels qui nous servira d'exemple tout au long de l'article.

## 2. Exemple : Location d'Appartement sur le Web

Un *Étudiant* cherche à louer un appartement à Montpellier. Il possède un *Assistant Électronique* qui contient son profil utilisateur et qui va aussi lui servir d'intermédiaire avec le Web. Notre étudiant connaît aussi un agent *Moteur de Recherche* compétant dans le domaine immobilier. Ce dernier connaît, entre autre, un *Agent Immobilier* qui peut proposer des appartements à ses clients. La figure 1 montre les communications établies entre les différents agents. On supposera dans un premier temps, pour plus de simplicité, que les quatre agents (l'Étudiant, l'Assistant Électronique, le Moteur de Recherche et l'Agent Immobilier) partagent le même vocabulaire. Notons que le Moteur de Recherche aurait pu mettre en communication directe l'Agent Immobilier et l'Assistant Électronique, mais nous avons choisi de ne pas le faire dans cet article pour ne pas alourdir notre propos.

Voyons l'exemple, d'abord à travers les différentes étapes du scénario, puis par les dialogues entre les agents.

### 2.1. Scénario

L'Étudiant demande à son Assistant de l'aider à faire une tâche en utilisant son profil utilisateur. Ce dernier répond en lui demandant quelle tâche faire. L'Étudiant lui dit de contacter le Service de recherche d'appartement du Moteur de Recherche (appelons le Yahoo), alors l'Assistant le contacte et lui demande de trouver un appartement.

Le Moteur de Recherche demande à l'Assistant dans quelle ville il veut cet appartement. L'Assistant trouve dans le profil utilisateur de l'Étudiant qu'il veut son appartement à Montpellier, il l'indique donc au Moteur de Recherche. Nous ne nous étendrons pas sur la manière dont il a eu cette information, il a pu l'obtenir dans ses précédentes conversations avec l'Étudiant, ou encore en analysant son courrier.

Le Moteur de Recherche contacte l'Agent Immobilier de Montpellier et lui demande de trouver un appartement. Comme on pouvait s'y attendre, l'Agent Immobilier ne demande pas dans quelle ville devra être l'appartement, car toutes les requêtes qui lui sont faites concernent des biens immobiliers à Montpellier, il cherchera seulement à savoir le prix que le futur locataire voudra bien y mettre. Cette question est posée au Moteur de Recherche, qui ne sait pas. Ce dernier la pose à l'Assistant, qui ne sait pas non plus, enfin l'Assistant la pose à l'Étudiant, qui répond qu'il veut bien y mettre 260 € par mois. La réponse est retransmise jusqu'à l'Agent Immobilier. Ensuite, l'Étudiant donne, de sa propre initiative, une information qui n'avait pas été demandée. Il dit à tout hasard, à son assistant, qu'il veut bien être en sous location.

Ce qui n'était pas prévu, ni par le Moteur de Recherche, ni par l'Assistant, ni même par l'Étudiant, c'était que l'Agent Immobilier allait demander si le locataire de l'appartement allait être un étudiant ou pas, car il a un appartement pas cher à proposer, mais il est réservé aux seuls étudiants. L'Étudiant répond que oui, et la réponse suit le même chemin que pour le prix. Finalement, l'Agent Immobilier répond à l'agent Moteur de Recherche, qui transmet l'information jusqu'à l'Étudiant. – Fin du scénario –.

### 2.2. Dialogues

La figure 1 montre ce scénario du point de vue des conversations, en faisant voir le contenu des dialogues ayant lieu entre chacun des agents, exprimés en langue naturelle. Dans la section suivante nous verrons, en terme d'interopérabilité, les problèmes que pose l'échange d'informations avec des agents dont on ne connaît pas grand chose à priori, par exemple quand l'Agent Immobilier pose la question au sujet du statut du locataire, étudiant ou non.

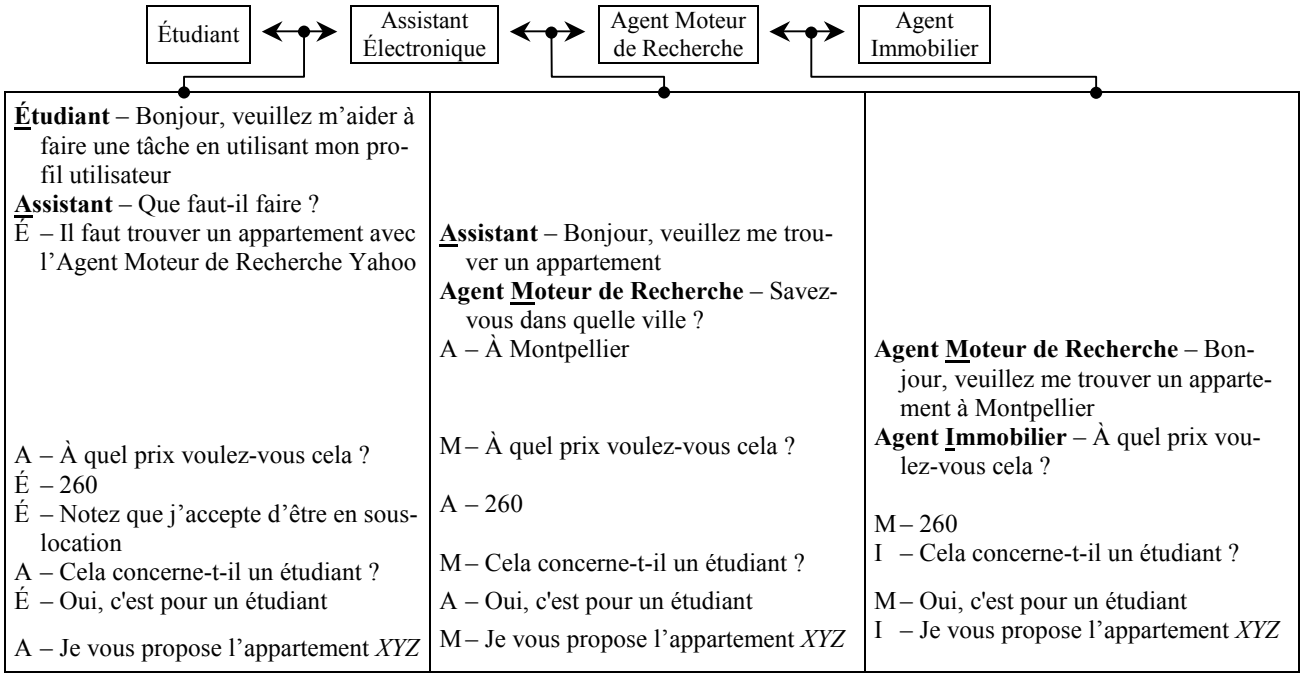


**Figure 1 – Dialogue entre les Agents**

## 3. Problème d'Interopérabilité

Dans cette section, nous montrons le rôle que jouent les Interfaces dans les problèmes d'Interopérabilité. Nous commencerons par expliquer ce qu'on entend par Interface, ensuite nous verrons les avantages et les inconvénients des deux manières de les implémenter : soit lors de la conception, soit lors de l'exécution. Enfin nous verrons l'intérêt qu'il y a à avoir des interfaces les plus génériques possibles.

### 3.1. Interfaces

Pour résoudre ensemble des problèmes, les Agents ont besoin de communiquer. Ils ont donc besoin d'avoir des interfaces de communication, et pour être interopérables, ils ont besoin de connaître les interfaces de communication de leurs interlocuteurs, c'est-à-dire l'ensemble de leurs mécanismes de communication, ce qui inclus un vocabulaire et une grammaire.

Si on prend l'exemple des langages de programmation, une interface définit le vocabulaire de communication d'un composant logiciel (objet Java, serveur Web, …). Elle est rendue publique par son concepteur, par exemple sous la forme d'un document destiné à des programmeurs et décrivant les fonctions disponibles pour communiquer avec ce composant, les informations dont elles ont besoin, ainsi que leur fonctionnement. Dans certains langages comme Smalltalk [5] ou Scheme [6], c'est le seul moyen de décrire une interface, mais dans d'autres, elles peuvent être décrite plus formellement (*interface* Java [7], *IDL* CORBA [8], fichier WSDL [9] du protocole de communication Web à base d'XML SOAP [10]).

Dans le monde des agents, où on échange explicitement des messages, les interfaces sont la spécification de ces messages, exprimés en XML ou au moyen d'objets. Le problème est alors reporté à celui des composants logiciels. En dehors de la programmation, mais toujours concernant l'aspect vocabulaire de la communication, il faut citer les Ontologies, qui cherchent à définir des concepts permettant de décrire, entre autres, des documents et des pages-web [11]. Pour en revenir à notre exemple de location d'appartement, l'interface de communication entre chaque paire d'agents est constituée par l'ensemble des messages (phrases) qu'ils utilisent pour communiquer (c.f. fig. 2).

Après voir exposé comment décrire les « mots » du vocabulaire de communication des programmes, regardons les problèmes venant de l'ordre d'utilisation de ces mots. Prenons l'exemple de l'écriture dans un fichier. Si on essaye d'écrire dans un fichier avec la fonction *write()* après avoir appelé la fonction *close()* en fermant l'accès, on obtient en général une erreur. Il nous faut donc une grammaire définissant l'ordre d'utilisation correct des mots. Le mécanisme de contrat du langage Eiffel permet de faire cela en indiquant les conditions d'appel des méthodes d'un objet [12]. Pour les Agents, les travaux sur les politiques de conversation (*Conversation Policy*) cherchent le moyen de définir la grammaire de la communication indépendamment de la structure interne des agents [13][14]. Mais dans bien d'autres cas, comme pour le vocabulaire, il faut s'en remettre à des documents informels en langue naturelle, non utilisables par des machines.

Après ce bref aperçu des formes que peuvent prendre les Interfaces, voyons quand et comment les implémenter.

## 3.2. Implémentation de l'Interface à la Conception

L'approche la plus traditionnelle pour assurer l'interopérabilité de deux composants logiciels, et donc de deux agents, est de définir une interface de communication, puis de l'implémenter complètement au moment de la conception des composants. Cette approche a l'immense avantage d'être simple à mettre en œuvre. Elle est basée sur une interface statique, ne changeant pas au cours de la durée de vie du programme. Elle assure que chaque fois qu'une information sera demandée, on saura comment la fournir. Enfin, l'interopérabilité peut y être en grande partie vérifiée au moment du design.

Le problème, quand on travaille avec des agents dédiés au Web, dans lequel on doit communiquer avec des Agents qu'on ne connaît pas, c'est justement que les interfaces sont définies une fois pour toutes sans possibilité de s'adaptater à ces Agents ou aux nouveaux besoins des utilisateurs. Ceux-ci sont inconnus des concepteurs lors du design. En cas de nouveaux besoins, les concepteurs de chaque composant doivent se mettre d'accord sur une nouvelle interface qui ne sera disponible que dans une nouvelle version de leurs logiciels respectifs. Il reste encore à vaincre l'inertie des utilisateurs de ces logiciels pour qu'ils en fassent la mise à jour, sachant que cette inertie augmente avec leur nombre. Les interfaces finissent alors par ne plus évoluer que très lentement et, afin de préserver leur interopérabilité, les nouveaux logiciels se trouvent contraints de respecter des interfaces périmées, mais déjà installées chez les utilisateurs. Ils finissent par souffrir du « syndrome du pré-installé ». Dans notre exemple, une telle démarche aurait pu aboutir l'impossibilité pour l'Agent Immobilier de demander si le futur locataire était étudiant. Cette question n'ayant pas été prévue au départ par les autres agents, ce besoin de l'Agent Immobilier n'aurait pas pu être pris en compte.

Un autre phénomène peut survenir quand on implémente les interfaces lors de la conception. Les concepteurs des différents composants peuvent ne pas s'entendre sur une nouvelle version de l'interface. Ils la font alors évoluer chacun de leur côté, rendant plus difficile l'interopérabilité, comme cela est déjà arrivé avec les différentes versions d'HTML ou le conflit entre Sun et Microsoft autour du langage Java, aboutissant au Java Standard d'un coté et à J++ et C# de l'autre.

## 3.3. Implémentation Conversationnelle de l'Interface à l'Exécution

Pour éviter les problèmes cités plus haut, nous proposons d'attendre l'exécution des programmes pour implémenter les interfaces, et plus précisément, d'attendre le moment où les programmes auront à communiquer. Les interfaces s'adapteront aux composants logiciels avec lesquels elles sont en communication et l'effort d'implémentation pourra se concentrer sur les parties des interfaces réellement utilisées. Elles seront alors adaptées au besoin réel des utilisateurs et à la manière dont on les utilise vraiment. Enfin, leur construction pourra profiter de toutes les informations disponibles au moment de la communication, y compris celles connues seulement par des utilisateurs humains.

Faire évoluer l'interface de communication d'un programme peut paraître une idée étrange, car le programme, une fois écrit, paraît statique, et car l'action d'implémenter paraît fortement liée au temps de la conception (au *design-time)*. Mais après tout, il est n'est pas saugrenu de faire des Interfaces Graphiques qui évoluent et s'adaptent aux utilisateurs [15].

Ayant constaté l'intérêt qu'il y a à implémenter les interfaces de communication lors de l'exécution d'un programme, il reste à résoudre le principal problème de cette approche : Comment implémenter à l'exécution ? C'est ce que nous nous éclaircirons dans la section 4, en proposant un cadre générique dans lequel pourrons venir s'inscrire différentes stratégies pour implémenter l'interface de communication.

## 3.4. Généricité et Interopérabilité

La prolifération des langages et des interfaces de communication nuit bien évidemment à l'interopérabilité. On peut souhaiter l'apparition de langages et d'interfaces génériques pour améliorer l'interopérabilité. Mais si la généricité ne provoque pas nécessairement de l'interopérabilité, elle l'encourage cependant [16].

Prenons le cas d'XML, il a permis à un nombre immense de logiciels de représenter des données arborescentes. Lorsque deux logiciels échangent des données en XML, leur analyse syntaxique devient triviale. Il n'y a qu'à utiliser la librairie, devenue standard, permettant de lire de l'XML [17]. Cela facilite l'interopérabilité des données. Mais d'un autre côté, XML sert de base à la définition d'un très grand nombre de langages décrivant « tout et n'importe quoi », et qui ne sont pas interopérable au niveau sémantique. On peut se demander alors si les bénéfices cités précédemment ne sont pas inutiles. XML a cependant permis de reporter le problème de l'interopérabilité à un niveau supérieur d'abstraction.

Nous cherchons à faire la même chose au niveau de la gestion des communications interactives entre programmes. Nous pensons que la généricité des communications, si elle ne résout rien par elle-même, permettra de reporter le problème de l'interopérabilité à un niveau plus sémantique, et qu'elle donnera une base pour développer des méthodes génériques d'aide à l'implémentation conversationnelle d'interface, dans le but d'obtenir une meilleure interopérabilité.

Nous allons maintenant exposer le cadre logiciel générique de communication entre agents qui nous permettra par la suite de développer des composants génériques d'implémentation d'interface.

## 4. Agents vus comme des Fonctions

Nous avons vu, dans la section 1, les contraintes que le Web apporte aux agents y travaillant : nécessité d'interopérabilité et nécessité d'évolutivité des interfaces. Pour gérer les problèmes dus à ces contraintes, nous proposons le cadre CRAT pour étudier les communications entre programmes et entre agents. Bien que ces communications puissent être très complexes, principalement à cause du fonctionnement asynchrone des agents et du besoin d'initiative mixte introduit par la présence d'utilisateurs [4][18], nous avons cherché le cadre de communication le plus simple possible basé sur la manière dont les programmes informatiques sont interprétés (exécutés). Nous nous sommes donc appuyés sur le modèle STROBE [19] et sur son évolution C+C [20], qui propose de représenter les agents au moyen d'interpréteurs et de leur boucle REPL (*read-eval-print-listen*). Après avoir présenté ce cadre, nous verrons comment le mettre en œuvre dans l'écriture de programmes en Scheme, permettant de transformer de simples fonctions en Agents ayant une gestion fine de leurs messages.

### 4.1. CRAT – Éléments Simples de Communication

Nous proposons de modéliser les échanges entre tous types de programmes avec 4 messages de base. Dans notre approche, quand deux agents logiciels communiquent, ou plus généralement quand deux morceaux de code communiquent, on cherche à les modéliser comme des Agents et à modéliser leur relation comme une relation Agent-Environnement–Agent-Script dans laquelle ils échangent des messages CRAT (CALL, REPLY, ASK et TELL). L'échange de ces messages peut être explicite comme dans une simple relation client-serveur, ou implicite comme dans une relation fonction-appelante–fonction-appelée. Mais bien que les communications que permet le cadre soient simples, la composition de plusieurs d'entre elles permettra de rendre compte de communications plus complexes.

#### 4.1.1. Organisation de la Communication

Dans le paragraphe précédent, nous avons évoqué deux agents, l'Agent Environnement et l'Agent Script. En fait, il s'agit d'abus de langages pour désigner, d'une part, l'agent ayant le rôle Environnement, et d'autre part, l'agent ayant le rôle Script, chaque agent pouvant avoir les deux rôles à la fois. Précédemment, nous avons parlé de fonctions alors que nous prétendons travailler avec des agents. C'est pour profiter du vocabulaire avec lequel on décrit la manipulation des fonctions, spécialement les notions d'Évaluation et d'Environnement, étudiées en détail en LISP [21][22]. Ces notions nous servent à décrire notre Agent Environnement et notre Agent Script. Notons encore que nous utilisons ici le mot Agent dans le sens d'entités logicielles échangeant des messages et ayant une boîte aux lettres pour les recevoir. Nous détaillons maintenant chacun des deux rôles qu'ils peuvent prendre.

L'Agent Environnement fournit des valeurs à partir de descriptions d'informations. Il tient son nom du parallèle avec les environnements des Évaluateurs LISP qui fournissent des valeurs à partir de nom de variable. L'Agent Script, lui, exécute une suite d'instructions – un script, d'où son nom. Un agent prend le rôle d'Agent Environnement quand il demande un Service à un autre agent. Ce dernier prend alors le rôle d'Agent Script et l'Agent Environnement s'engage à lui fournir les informations nécessaires à la réalisation du Service. C'est comparable à un passage de paramètres à une fonction dans un programme classique. Enfin pour finir la comparaison, si on regarde de plus près les mécanismes d'Évaluation, on constate que l'application d'une fonction commence aussi par créer un environnement contenant les paramètres à passer à la fonction (c.f. [21] §3.2.1), puis ensuite le corps de la fonction (le Script) est évalué dans cet environnement.

Quand un Agent Script exécute les fonctions lui permettant de réaliser le Service qu'on lui a demandé, il peut très bien être amené à demander un autre Service à un autre Agent. Dans ce cas, en plus du rôle d'Agent Script qu'il a envers son commanditaire, il prend un rôle d'Agent Environnement envers cet autre Agent. Illustrons cela avec l'exemple de la location d'appartement : l'Étudiant est Agent Environnement vis-à-vis de son Assistant Électronique car le premier fournit au second les informations qu'il demande (prix de l'appartement, statut du locataire). En retour, l'Assistant Électronique est Agent Script vis-à-vis de l'Étudiant (nous verrons son script dans §4.2.3), mais il est aussi Agent Environnement vis-à-vis de l'Agent Moteur de Recherche.

Voyons maintenant en détails les messages échangés entre l'Agent Environnement et l'Agent Script.

#### 4.1.2. Messages CRAT

Pour gérer la communication entre un Agent Environnement et un Agent Script, nous proposons 2 couples de messages : CALL – REPLY et ASK – TELL, qui sont gérés par les agents de la manière suivante.

- Ouverture et fermeture de la communication :
  - **CALL**(*service_description)* : l'Agent Environnement demande à l'Agent Script de lui fournir le Service décrit par *service_description*. Il s'engage à fournir les informations nécessaires à la réalisation du Service au moyen de messages TELL.
  - **REPLY**(*reply*) : l'Agent Script indique qu'il a fini son traitement, avec ou sans succès, et renvoi son résultat (*reply*). Cela ferme aussi la communication.

– Échange d'informations :
  - **ASK**(*data_request*) : l'Agent Script demande à l'Agent Environnement une information nécessaire à l'exécution de son script. L'information demandée est décrite par la Requête de Donnée *data_request*. L'Agent Script a le droit de suspendre son exécution jusqu'à obtention d'un message TELL satisfaisant à sa demande, mais ce n'est pas une obligation. Cela incite fortement l'Agent Environnement à répondre à ses demandes, s'il veut que le Service aboutisse.
  - **TELL**(*data_description*, *value*) : l'Agent Environnement envoie une information à l'Agent Script, soit motivée par un message ASK de l'Agent Script, soit de sa propre initiative. L'information est représentée par une Description de Donnée (*data_description*) et par sa valeur (*value*).

*Data_request* et *data_description* servent à annoter et à décrire les donnés que l'on recherche et celles que l'on propose. Nous les appelons des Annotations et ce terme sera utilisé quand il sera inutile de distinguer Requête et Description. Nous reviendrons dessus plus en détails au §5, nous reviendrons aussi sur la manière de calculer si un TELL (avec sa Description de Donnée) satisfait à la demande formulée par un ASK (avec sa Requête de Donnée) (c.f. §5).

Pour récapituler, nous avons 4 messages, que l'on peut aussi grouper par leurs émetteurs :
– Messages émis par l'Agent Environnement : CALL et TELL – Messages émis par l'Agent Script : ASK et REPLY

### 4.1.3. Structure des Agents

Dans ce paragraphe, nous étudions brièvement la façon dont les Agents doivent être structurés et à quel moment ils doivent être envoyés et attendus :

– L'Agent Environnement envoie un message CALL, ce qui ouvre un canal de communication sur lequel il écoute les messages ASK de l'Agent Script jusqu'à ce qu'il reçoive un message REPLY. Dans le même temps, il émet ses messages TELL, correspondant ou non à des messages ASK.
– L'Agent Script est en permanence en attente de messages CALL. Quand il en reçoit un, il se met aussi en attente de messages TELL correspondant aux CALL. Dans le même temps il envoie ses messages ASK. Quand il a fini son traitement, il envoie le résultat et ferme le canal de communication avec un message REPLY.

### 4.1.4. Illustration sur l'Exemple de Location d'Appartement (II)

La figure 2 montre le scénario exposé au §2, en terme de messages CRAT (CALL, REPLY, ASK, TELL). Vous pouvez les comparer aux dialogues de la figure 1. Voici le dictionnaire des symboles qui y sont utilisées :

– Descripteurs de Service (écrits en gras) :
  - **utilise-profil-utilisateur** : designe le service de l'Assistant Électronique qui propose de mettre à disposition le profil utilisateur pour faire une tâche.
  - **yahoo-trouve-appart** : désigne le service de l'Agent Moteur de Recherche (appelé Yahoo) qui propose de servir d'intermédiare avec des agents immobiliers.
  - **trouve-à-montpellier** : désigne le service de l'Agent Immobilier de Montpellier qui propose des appartement.

  Tous ces services renvoient la description d'un appartement, dans notre scénario c'est *appartement XYZ*.
– Descriptions et Requêtes de Données (écrits en italique) :
  Dans un soucis de simplicité, toutes les Annotations de notre exemple (les Requêtes et les Descriptions de Données) sont représentées dans le même langage. Elles sont encodés avec des identificateurs semblables à ceux des langages de programmation, ainsi un TELL satisfait un ASK s'ils contiennent tous les deux la même annotation (ex : *TELL(prix,*260*)* satisfait à la demande formulée par *ASK(prix)*).
  - *action :* décrit un Descripteur de Service. Le Service en question est n'importe quel service pour lequel l'utilisateur de l'Assistant Électronique (c.-à-d. l'Étudiant) veut que son profil utilisateur soit utilisé. Dans l'exemple, l'Étudiant choisit *yahoo-trouve-appart*, qui contient l'adresse de l'Agent Moteur de Recherche.
  - *prix* : décrit un nombre. Ce nombre doit être un prix en Euros.
  - *accepte-sous-location* : décrit la signification d'une valeur booléenne (oui/non). *Oui* indique l'acceptation d'une sous-location. *Non* indique le contraire.
  - *étudiant* : décrit la signitication d'un booléen. *Oui* indique qu'on parle d'un étudiant, *Non* indique le contraire.
  - *ville* : décrit un nom. Ce nom doit être le nom d'une ville.

On pourra remarquer que le fait d'indiquer quelque-chose qui n'a pas été demandé (*TELL(accepte-sous-location,* oui*)*) se fait avec le même type de messages que les réponses à des questions ASK. Pour le moment, nous n'étudirons pas comment traîter les messages non sollicités, car des problèmes se posent si un agent transmet des messages non-sollicités aux autres agents pour lequel il est Agent Environnement : On peut finir par être submergés par des messages non-sollicités inutiles.

À partir de tous ces messages et de quelques primitives permettant de les manipuler (c.f. §4.2.1), nous verrons au §4.2.3 comment construire nos 4 agents avec des programmes simples écrits en Scheme.

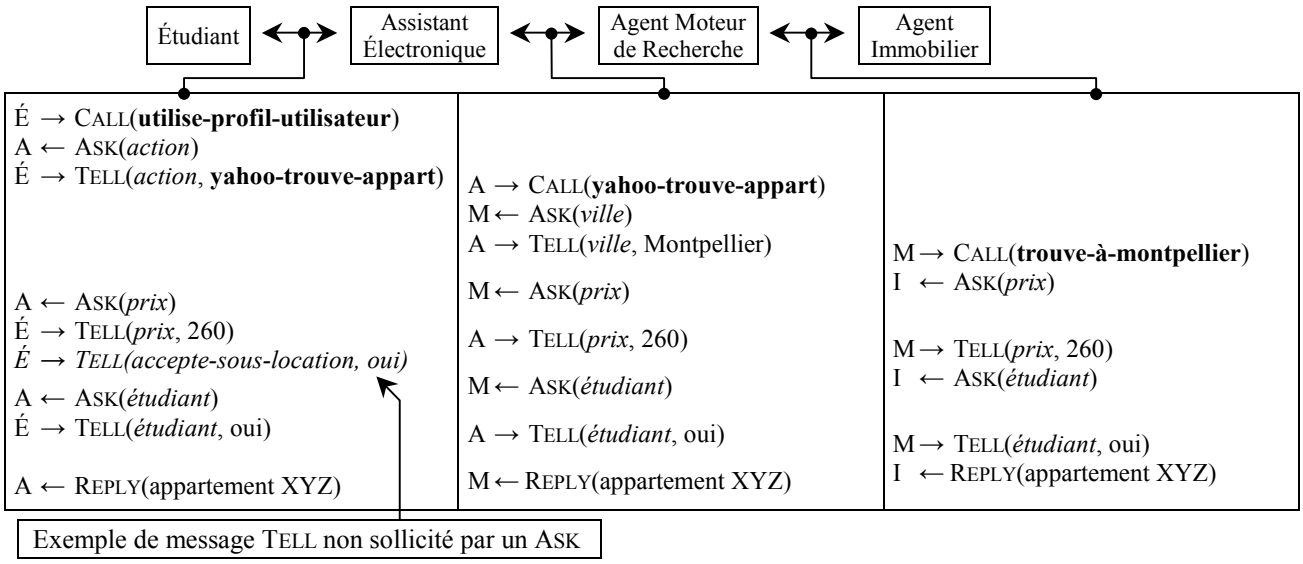


**Figure 2 – Messages CALL, REPLY, ASK, TELL entre les Agents**

### 4.1.5. Comparaison avec STROBE

Comme nous l'avons déjà dit plus haut, nous nous sommes inspirés de STROBE [19] et de C+C [20]. Le modèle STROBE propose 6 types d'actions de base, basées sur Scheme, pour représenter les communications entre agents :

- *assertion – acknowledge* : l'*assertion* sert à envoyer une nouvelle information en tentant de modifier l'environnement de l'interlocuteur, sans sollicitation particulière de cet interlocuteur. L'acquittement (*acknowledge*) sert à indiquer si l'action à réussi ou pas.
- *request – answer* : la requête (*request*) demande une information en tentant d'accéder à une variable de l'environnement de l'interlocuteur. La réponse (*answer*) envoie une information, comme le fait *assertion*, mais uniquement sur sollicitation.
- *order – executed* : l'ordre (*ordre*) demande l'application d'une fonction. L'action *exécuté* (*executed*) renvoi la valeur résultant de cette exécution.

Contrairement à STROBE qui considère l'environnement de chacun des deux agents en communication, nous ne considérons que l'environnement de l'Agent Environnement, et ce seulement en lecture. Cela nous permet de n'avoir que 4 types de messages. On se rapproche ainsi de la programmation fonctionnelle pure (sans affectation).

CALL et REPLY sont semblables à *order* et *executed*. Nous ne définissons pas d'action pour modifier les environnements, mais, comme nous l'avons vu plus haut, nous avons un message commun pour les envois d'informations sollicitées (*answer*) et non-sollicitées (*assertion*) : c'est TELL. Ne faisant pas d'affectation, nous n'avons pas besoin d'en recevoir l'acquittement, nous n'avons donc aucun message semblable à *acknowledge*. Il reste donc ASK qui est semblable à *request*. Enfin, à la différence des messages *assertion* et *request* qui travaillent explicitement avec des noms de variables, ASK et TELL travaillent avec des Annotations, qui sont une extension de la notion de variable (plus de détail dans le §5).

## 4.2. Agent vu comme une Fonction

Nous avons proposé un cadre de communication simple décrivant les relations entre un Agent Environnement et un Agent Script. Nous allons montrer comment le mettre en œuvre avec des fonctions Scheme, puis nous les utiliserons pour faire une version simple des agents de l'exemple de location d'appartement. L'idée de base sur laquelle repose une telle approche est que les fonctions communiquent naturellement au moyen du passage de paramètre. Dans un premier temps, nous remplaçons la communication utilisant les paramètres par l'utilisation explicite d'un ensemble de fonctions communication. Dans le §6 nous esquisserons comment abstraire ces fonctions pour en faire le Scheme Conversationnel.

Pour décrire et illustrer les primitives du Scheme Conversationnel, nous utilisons le langage Scheme. Une description concise et complète du langage est disponible dans [6]. Nous l'avons choisi pour son extrême simplicité, pour son typage dynamique [23], et car il est facile de le transcrire en d'autres langages de programmation à la mode, et en particulier Java, comme en témoigne l'intégration poussée faites entre les deux langages dans l'environnement de programmation en Scheme basé-sur-Java *Kawa* [24]. Ce que nous décrivons ici n'est donc pas nécessairement lié à Scheme, il est seulement le langage le plus adapté à ce qu'on veut décrire. Le principal obstacle de Scheme est, pour le néophyte, la notation parenthèsée des appels de fonctions. Par exemple, pour appeler une fonction `fonc` avec deux paramètres `a` et `b`, qu'on

écrirait classiquement `fonc(a,b)`, on écrit `(fonc a b)`. Le néophyte pressé se reportera à un tutoriel rapide, par exemple [25].

### 4.2.1. Primitives du noyau du Scheme Conversationnel

Nous allons montrer successivement comment décrire en Scheme, les Services, les Annotations, puis les Fonctions de Communication proprement dites.

#### Services

Les Services sont implémentés par des fonctions. Quand on lance un Agent, on lui fournit une liste de Descriptions de Services accompagnées de leurs fonctions respectives. Quand il reçoit un message CALL(*service_description*), il sélectionne la fonction associée au Service et l'exécute. Cette fonction est de la forme (*service-function* ***get-annotated-value call-with-annotated-value***), où *get-annotated-value* et *call-with-annotated-value* sont des fonctions passées en paramètre par l'Agent pour gérer les messages CRAT et qui peuvent changer suivant la manière dont il gère ses messages. Lorsqu'il exécute la *service-fonction*, l'Agent prend le rôle d'Agent Script. Enfin, quand l'exécution de la fonction se termine, l'Agent envoie un message REPLY(*result*) dans lequel *result* est la valeur de retour de cette fonction.

#### Annotations

Avant de décrire les fonctions *get-annotated-value* et *call-with-annotated-value*, nous devons éclaircir quelques points sur les Annotations. Une Valeur Annotée est une donnée dont l'usage pour lequel elle a été prévue est décrit au moyen de Descriptions de Données (*data_description* du message TELL). On y accède au moyen de Requêtes de Données (*data_request* du message ASK). La fonction (**make-annotated-value** *data_description value*) permet de construire une nouvelle Valeur Annotée à partir d'une Description et d'une valeur. Les Annotations peuvent être n'importe quel objet d'un langage de programmation. Dans l'exemple de la location d'appartement, une chaîne de caractère suffit.

#### Fonctions de Communication

Voici donc nos deux fonctions, dont le fonctionnement est résumé par la figure 3 :

- (***get-annotated-value*** *data_request*) : permet à un Agent Script de communiquer avec son Agent Environnement. Cette fonction permet d'accéder à une Valeur Annotée à partir de sa description (Requête de Donnée). Elle cherche si un message TELL(*data_description*, *x*) convenable n'a pas été reçu, c'est-à-dire tels que le *data_description* corresponde[1] à *data_request.* Si oui, renvoie *x*, sinon envoie un message ASK(*data_request*) à l'Agent Environnement, puis bloque l'exécution jusqu'à la réception d'un message TELL convenable.
- (***call-with-annotated-value*** *service_descrition annotated-value-list*) : permet de prendre le rôle d'Agent Environnement. Cette fonction permet à un Agent (appelons le l'Agent Appelant) d'utiliser le Service décrit par *service_description*. L'Agent fournissant ce Service (appelons le Agent Appelé) prend alors le rôle d'Agent Script. l'Agent Appelant, qui avait déjà un rôle d'Agent Script, prend le rôle d'Agent Environnement vis-à-vis de l'Agent Appelé. Le paramètre passé dans *annotated-value-list* donne les Valeurs Annotées que l'on met à la disposition de cet Agent Appelé en plus de celles auxquelles on peut déjà accéder avec *get-annotated-value*.
  Cette fonction commence par émettre un message CALL(*service_description*) à destination de l'Agent Appelé. Tant que le message REPLY(*result*) n'a pas été reçu, la fonction traite les messages ASK(*data_request*). Pour chaque message ASK, elle cherche une Valeur Annotée dont le *data_description* correspond[1] au *data_request,* soit dans la liste *annotated-value-list*, soit auprès de l'Agent Environnement de l'Agent Appelant (et récursivement auprès de ses

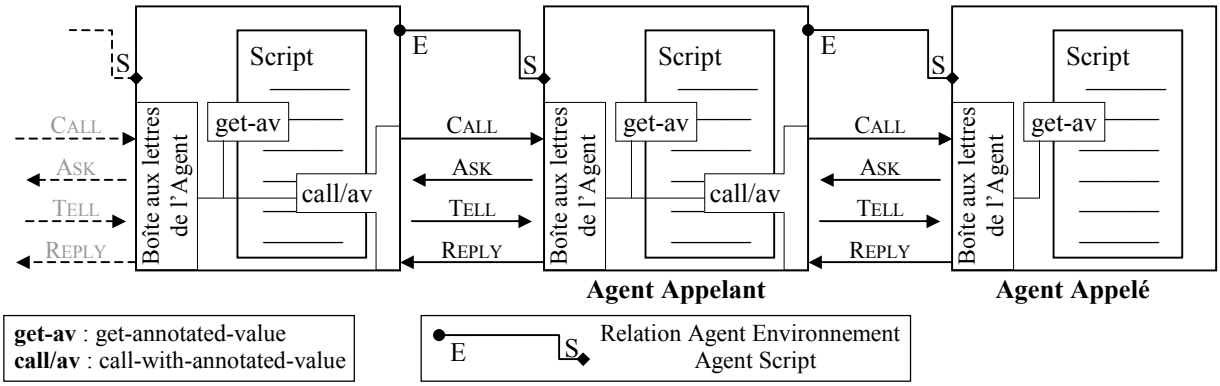


**Figure 3 – Agent Appelant – Agent Appelé**

[1] Dans notre exemple de la location d'appartement, la *correspondance* se fait avec une comparaison exacte de chaînes de caractères (en Scheme `string=?`)

« ancêtres »). Quand une valeur a été trouvée, la fonction renvoie un message TELL construit à partir de la Valeur Annotée sélectionnée. Il faut noter la grande liberté de choix qu'il y a dans la recherche de la Valeur Annotée.
Cette fonction peut aussi envoyer des messages TELL sans qu'ils aient été demandés (à l'initiative de l'Agent Appelant). La fonction se termine quand le message REPLY(*result*) est reçu. La valeur de retour est *result*.

### 4.2.2. Paramétrage des Agents

Le paramétrage de l'Agent avec ces deux fonctions permet à une même fonction d'implémentation de Service d'avoir des gestions très différentes des messages. C'est là que nous comptons mettre le code qui permettra d'adapter les communications de l'Agent avec ses interlocuteurs, c'est à dire le code qui nous permettra d'implémenter les interfaces de communication lors de l'exécution et de faciliter ainsi l'interopérabilité.

Nous avons testé avec succès plusieurs jeux de fonctions. Dans l'un d'eux, nous avons utilisé de vrais agents, avec des boîtes aux lettres, et de vrais messages, stockés dans des objets Scheme. Un autre jeu sur lequel il convient de s'attarder, et qie nous avons testé avec les mêmes fonctions de Services, est une version où tous les Agents fonctionnent dans le même processus. Dans cette version, *call-with-annotated-value* ne fait qu'appeler de manière traditionnelle la fonction implémentant le Service, en lui passant une fonction *get-annotated-value* construite avec *annotated-value-list*. Dans ce cas les messages ne sont que virtuels, ils sont attribués à certaines actions : CALL et REPLY pour l'appel et le retour de fonction ; ASK et TELL pour l'exécution de *get-annotated-value*. Un seul agent a le contrôle à un instant donné et on retrouve un modèle de programmation avec variables dynamiquement liées (c.f [22] §1.6.1), en effet, l'évaluation des Valeurs Annotées (*get-annotated-value*) se fait dans l'environnement d'exécution, matérialisé par l'Agent Environnement. Cela est semblable à l'évaluateur à variable dynamique sans forme sépciale décrit par Christian Queinnec dans [22] au §2.5.3, *get-annotated-value* et *call-with-annotated-value* permettant d'accéder et d'étendre l'environnement dynamique. Maintenant, pour eclaircir un peu les esprits, revenons à notre exemple de location d'appartement.

### 4.2.3. Illustration sur l'Exemple de Location d'Appartement (III)

Nous allons maintenant illustrer la fabrication d'Agents au moyen de fonction Scheme en montrant comment programmer les Agents de l'exemple de location d'appartement. La figure 4 montre les fonctions Scheme correspondant aux Services fournis par chaque Agent du scénario. On a utilisé des noms abrégés pour les primitives du Scheme Conversationnel, ***get-av*** pour *get-annotated-value*, ***call/av*** pour *call-with-annotated-value* et ***mk-av*** pour *make-annotated-value*. Les Annotations sont écrites en italique entre guillemets (elles sont encodées par des chaînes de caractères) :

- L'**Assistant Électronique** cherche d'abord un Service annoté *"action"*, puis l'appelle en lui indiquant que la ville concernée est `'Montpellier`.
- L'**Agent Moteur de Recherche** choisi l'Agent Immobilier dont il va utiliser les services pour fournir un appartement à l'Assistant Électronique. Les critères de choix sont la *"ville"* et éventuellement le *"pays"*. On peut noter que dans notre exemple, *"pays"* ne sera pas demandé car `"ville"` étant égale à `'Montpellier`, c'est l'Agent Immobilier de Montpellier qui est choisi (`trouve-à-montpellier`). On voit là que l'évaluation des valeurs annotées est une évaluation retardée.
- L'**Agent Immobilier** cherche à savoir le *"prix"* de l'appartement qu'on lui demande, si le prix est supérieur à 450 il propose l'appartement ABC, sinon si le *"prix"* proposé est supérieur à 250, il vérifie si c'est pour un *"étu-*

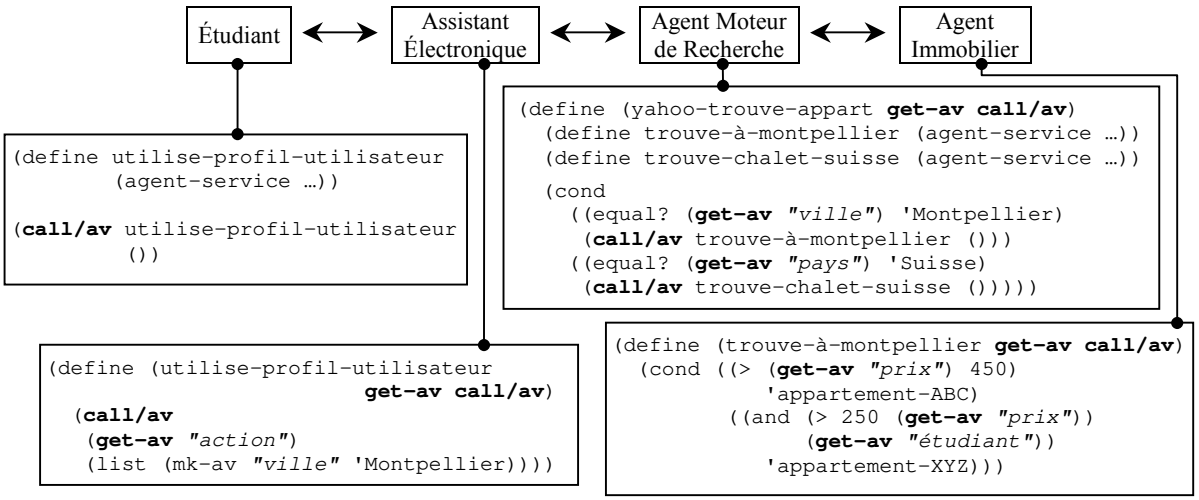


**Figure 4 – Code Scheme des Agents**

`diant"`, auquel cas il propose l'appartement XYZ.

– Enfin, le système est lancé quand l'**Étudiant** fait sa requête. Il invoque l'Assistant Électronique avec un *call/av* mais il ne fournit pas de liste de Valeurs Annotées. À la place, l'Étudiant, en tant qu'agent humain, donne ses informations par le biais de *call/av* et de *get-av* un peu spéciaux car ils lui servent d'interface utilisateur (éventuellement graphique, selon les goûts).

Pour connaître la liste des Valeurs Annotées nécessaires au fonctionnement de chacune des fonctions supportant les Agents, il suffit de regarder les communications entre ces Agents et de noter les Annotations contenues dans les messages échangés (c.f. fig. 2). On constate bien que la plupart des informations nécessaires à leur exécution ne sont pas citées dans le code source. L'Assistant Électronique ne « déclare » avoir besoin de que *action*, mais il a en fait eu besoin de demander : *action, prix* et *étudiant*. On constate donc, même dans ce petit exemple, que pour programmer un agent Assistant Électronique qui soit interopérable avec l'Agent Moteur de Recherche quand celui ci communique avec l'Agent Immobilier, il est nécessaire de construire la liste des paramètres – et donc l'interface de communication – à l'exécution, au cours de la conversation.

Dans la section suivante nous allons évoquer les problèmes intéressants qui surviennent lorsque les Annotations sont encodées dans des objets plus complexes que de simples chaînes de caractères.

## 5. Annotation et Ambiguïté

Pour aider à rendre interopérables des programmes aux cours de leur fonctionnement, nous avons besoin de descriptions sémantiques des données échangées par ces programmes. Au §4.1.5, nous présentons les Annotations comme une extension de la notion de variable. Nous ajoutons que c'est aussi une notion proche de la notion de type. Nous développons dans cette section ces deux idées un peu iconoclastes, suivies brièvement de comment faire des Annotations.

### Annotation *vs.* Variables

Pour comparer les Annotations aux variables, il faut d'abord donner une définition de ce qu'est une variable en programmation. Dans son livre fondateur sur l'interprétation des programmes informatiques, Abelson et Sussman introduisent les variables comme le moyen d'utiliser des noms pour faire référence à des objets informatiques ([21] au §1.1.2). C'est bien ce que nous avons fait dans notre exemple au §4.1.4 et §4.2.3 avec les Annotations. Elles y ont permis de faire référence à l'Agent Web (*web-trouve-appart*), au prix proposé (*prix*), *et cetera*. Voyons maintenant en quoi les Annotations sont aussi des Types.

### Annotation *vs.* Type

Les Types sont des informations utilisables par les machines, décrivant la structure des données et les opérations qu'il est possible de faire dessus. Ils permettent de s'abstraire de la manière dont sont rangés les octets représentant des objets informatiques. Dans les langages de programmation on peut observer deux tendances, non incompatibles, sur l'utilisation des types. La première, le typage dynamique, est la pratique qui consiste à associer le typage aux valeurs ; la seconde, le typage statique, consiste à associer le typage aux variables. Cela est à rapprocher du découpage fait dans les Annotations entre Descriptions de Données et Requêtes de Données.

### Annotation, Requête / Description

Dans le typage dynamique, à chaque donnée est associé son type de sorte que le type puissent être une donnée manipulée à l'intérieur d'un programme [26]. Cela peut être encodé par un couple (*Type*, *Valeur*), qui n'est pas sans rappeler une Valeur Annotée (*Description de Donnée*, *Valeur*). À la différence du Type, la Description de Donnée ne cherche pas à décrire directement la structure de la valeur ainsi que les opérations qu'il est possible de faire dessus, elle cherche à décrire pourquoi cette valeur est mise à disposition, et donc les opérations qu'il est souhaitable de faire dessus. C'est une information qu'il est souhaitable d'avoir pour aider à résoudre les problèmes d'interopérabilité lors de l'exécution.

Dans le typage statique les types sont associés aux variables. Ils ont deux utilisations, la première est la vérification des types des données transmises aux fonctions, c'est à peu près la seule utilisation qu'en font les machines à l'heure actuelle. La deuxième est la description des informations qu'on met dans les variables, et surtout dans les paramètres des fonctions. Cette utilisation du type est quasiment réservée au programmeur humain. Ce sont ces informations de typage, associées aux noms des variables, qui lui permettent de trouver, lorsqu'il écrit son programme, quelle donnée il va mettre dans quelle variable ou quelle donnée il va passer en paramètre à une fonction. C'est justement cette tâche que nous voulons automatiser pour permettre de rendre interopérables des programmes à *run-time*. La seule exception notable se rencontre dans les JavaBeans, où le typage des paramètres des fonctions accesseurs des propriétés des beans est utilisé pour choisir les interfaces de saisie des propriétés de ces beans.

### Contenu des Annotations

Au-delà des simples chaînes de caractères, il est possible d'utiliser des structures beaucoup plus riche pour encoder les Annotations. On peut envisager d'utiliser des listes d'identificateurs, des formules de logique propositionnelle, des formules de logiques des prédicats, et bien évidemment des ontologies. Quand on parle d'annotation et d'ontologie, on pense bien

évidemment à l'annotation des pages Web, qui sont des données comme les autres, et pour lesquelles des systèmes d'annotation sont développés [27][28]

### Ambiguïté

Avec des annotations très riches, il n'est plus possible de définir strictement la correspondance entre une Requête de Donnée et une Description de Donnée. On aura tendance à chercher la Valeur Annotée dont la Description et la plus proche de la Requête, mais se posera souvent le problème de choisir entre deux Valeurs Annotées équidistantes par rapport à la Requête. Par exemple, un Agent cherchant un véhicule pour transporter des personnes pourra se voir proposer d'abord un avion, puis un autobus. Il doit alors pouvoir ne pas s'arrêter obligatoirement à la première réponse valide, et doit pouvoir faire un choix entre deux réponses, quitte à demander l'aide d'un autre Agent, éventuellement humain. Là, il devient nécessaire d'avoir des stratégies de détection des ambiguïtés [29], et de résolution de ces ambiguïtés. La première de ces stratégies de résolutions d'ambiguïtés est l'interaction avec les utilisateurs intéressés à ce que les agents travaillent correctement. Cette souplesse permettra de faciliter l'interopérabilité en relâchant la nécessité d'un accord parfait entre les Requêtes et les Descriptions. La nécessité de gérer des informations ambiguës à l'intérieur des programmes en est le prix à payer.

## 6. Agent = Fonction + Évaluateur, vers le Scheme Conversationnel

Dans la section 4, nous avons vu que l'on pouvait faire des Agents satisfaisant l'équation Agent = Fonction + *get-av* + *call/av*, ce qui nous donne un langage avec des variables dynamiques (§4.2.2). L'utilisation généralisée de ces fonctions dans le code des Agents nous pousse tout naturellement à vouloir l'intégrer dans le langage, c'est à dire à créer un nouveau langage, dérivé du Scheme standard, dans lequel les appels à *get-av* et *call/av* serait automatiques, comme cela existe déjà avec variables dynamiques dans Common Lisp. L'idée est d'avoir des variables semblables aux paramètres des fonctions, mais allant chercher leur valeur dans l'environnement dynamique [30] au moyen des Annotations, les Valeurs Annotées devant être accessible depuis plusieurs Agents [31].

D'autre part, nous avons besoin de pouvoir contrôler le fonctionnement de l'environnement, c.-à-d. de contrôler une partie de l'évaluateur. Enfin on aura besoin de choisir l'environnement dans lequel évaluer une expression [32][33], c.-à-d. de choisir l'Agent avec lequel on communique lorsqu'on fait un calcul, notamment lors de l'utilisation de l'évaluation retardée dans les valeurs échangées dans les messages CALL, REPLY, ASK, TELL.

## 7. Conclusion et Perspectives

Dans cet article, nous avons exposé les problèmes rencontrés pour rendre interopérable aussi bien des programmes que des agents. Nous avons montré qu'il est nécessaire de considérer la construction des interfaces de communications entre logiciels au moment de leur l'exécution. Ensuite, nous avons posé un cadre d'étude des communications entre programmes et entre agents appelés CRAT, auquel nous avons associé un ensemble de fonctions qui nous ont permit de construire des Agents avec de simples fonctions Scheme. Cela a permit à des agents de manipuler des messages qui n'avaient pas été prévus lors la conception de ces agents (ex : *ASK(étudiant)*), une approche plus traditionnelle aurait pu empêcher ces messages de remonter jusqu'à un agent capable de les traiter. Dans ce cadre, nous avons constaté la nécessité d'annoter sémantiquement les informations échangées par les programmes pour pouvoir en assurer l'interopérabilité, et nous avons constaté que dans ce cadre, il est nécessaire de savoir gérer les ambiguïtés. Enfin nous avons proposé l'ébauche d'un langage pour utiliser CRAT : le Scheme Conversationnel.

Il reste encore beaucoup de travail à accomplir autour du Scheme Conversationnel, notamment une étude plus poussée des différentes variétés d'Annotations et une étude de l'adaptabilité des Agents que permet la paramétrisation des communications par les fonctions *get-av* et *call/av* avec des techniques de programmation par l'exemple[34].

## Bibliographie


[1] M. Wooldridge and N.R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115-152, 1995.

[2] A. Newell, The knowledge level, *Artificial Intelligence*, 18 (1982) 87-127.

[3] S. Ambroszkiewicz. Semantic Interoperability in Agentspace: proposal of agent interface to environment. *In Proc. of Workshop on Semantic Web: Models, Architectures and Management* at Fourth European Conference on Research and Advanced Technology for Digital Libraries, Lisbon, September 2000.

[4] Daniele Maraschi et Stefano Cerri, Relation entre les technologies de l'enseignement et les agents, *AFIA 2001, Atelier Méthodologies et Environnements pour les Systèmes Multi-Agents*, LEIBNIZ-IMAG, Grenoble, 25-29 juin 2001 (also available in English)

[5] William R. Cook. Interfaces and Specifications for the Smalltalk-80 Collection Classes, *OOPSLA 92 Proceedings* Association for Computer Machinery, pp. 1-15, New York, NY, 1992

[6] R. Kelsey, W. Clinger, J. Rees et al. Revised$^5$ Report on the Algorithmic Language Scheme, R. Kelsey, W. Clinger, J. Rees (eds.), Higher-Order and Symbolic Computation, Vol. 11, No. 1, September, 1998
[7] J. Gosling, B. Joy, G. Steele, G. Bracha, Chapter 9 in *The Java Language Specification Second Edition*, Addison Wesley, Jun 2000
[8] Object Management Group, Chapter 3 in *CORBA/IIOP 2.2 Specification*, February 1998
[9] Erik Christensen et al., *Web Services Description Language (WSDL) 1.1*, W3C Note, 15 March 2001
[10] Don Box et al., *Simple Object Access Protocol (SOAP) 1.1*, W3C Note, 08 May 2000
[11] J. Heflin and J. Hendler. Semantic Interoperability on the Web. *In Proceedings of Extreme Markup Languages 2000*. Graphic Communications Association, 2000. pp. 111-120
[12] Bertrand Meyer, Chapter 5 in *An Invitation to Eiffel: Introduction to the object-oriented language of choice*, http://www.eiffel.com/doc/manuals/language/intro/page.html
[13] Scott Moore. On conversation policies and the need for exceptions. *In Working Notes of the Workshop on Specifying and Implementing Conversation Policies*, pages 19--28, Seattle, Washington, May 1999.
[14] M. Greaves, H. Holmback, and J. M. Bradshaw. What is a conversation policy? In M. Greaves and J. M. Bradshaw editors, *Proceedings of the Autonomous Agents '99 Workshop on Specifying and Implementing Conversation Policies*, 1999
[15] Jean-David Ruvini, *Assistance à l'utilisation d'un environnement interactif : apprentissage des habitudes de l'utilisateur*, Thèse de Doctorat, Université Montpellier II, 6 octobre 2000
[16] Maxwell A.T. Jones, *Formal Generic Modelling*, Ph.D. Thesis, Nottingham Trent University, U.K., January 1998.
[17] David Megginson. *SAX 2.0 : The Simple API for XML*, http://www.megginson.com/SAX/index.html
[18] D. Maraschi et al, *A Conversational, Constructive view of Web knowledge: the Symbol Level*, Rap. de Recherche LIRMM.
[19] S. A. Cerri, "Shifting the focus from control to communication: the STReams OBjects Environments model of communicating agents" in *Collaboration between Human and Artificial Societies, Coordination and Agent-Based Distributed Computin*g, J.A. Padget Ed., Lect. Notes in Artificial Intelligence vol. 1624, pp. 71-101, Springer-Verlag, 1999.
[20] S. A. Cerri, J. Sallantin, E. Castro, and D. Maraschi. "Steps towards C+C: a Language for Interactions" in *Artificial Intelligence: Methodology, Systems, Application, AIMSA 2000*, S.A. Cerri and D. Dochev Eds., Lecture Notes in Artificial Intelligence vol 1904, pp. 33-46, Springer-Verlag, 2000.
[21] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs, second edition*. The MIT Press, Cambridge, Massachusetts, 1996.
[22] Christian Queinnec. *Les langages Lisp*. InterÉditions, Paris, France, 1994. (also available in English as Christian Queinnec. *Lisp in Small Pieces*. Cambridge University Press, 1996)
[23] S.A.Cerri, *Dynamic typing and lazy evaluation as necessary requirements for Web languages*, European Lisp User Group Meeting; Amsterdam (NL), June 6-8 1999,
*available at http://www.lirmm.fr/~maraschi/SMarino/Papers/DynamicTyping.pdf*
[24] Per Bothner, "Kawa: Compiling Scheme to Java". *40th Anniversary of Lisp Conference: Lisp in the Mainstream*. November 16-18, 1998, Berkeley, California,
*the Kawa system is available at http://www.gnu.org/software/kawa/*
[25] Bowdoin College – Computer Science Dept., *Writing and Running Scheme Programs: A Quick Tutorial,* http://www.bowdoin.edu/dept/cs/doc/langs/scheme.html
[26] Martín Abadi, et al. *Dynamic Typing in a Statically-Typed Language*. In Sixteenth Annual Symposium on Principles of Programming Languages, pages 213–227, January 1989.
[27] J. Heflin and J. Hendler. Dynamic Ontologies on the Web. In Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-2000). AAAI/MIT Press, Menlo Park, CA, 2000. pp. 443-449.
[28] Hillmann Diane, Using Dublin Core,Dublin Core Metadata Initiative working draft, 2000.
*available at http://dublincore.org/documents/usageguide/*
[29] Emmanuel Castro. Misunderstanding Detection using a Constraint Based Mediator. In ALCAA 2000, Biarritz, octobre 2000, *available at http://www.lirmm.fr/~castro/Paper/ALCAA2000*
[30] J. Lewis, M. Shields, E. Meijer, and J. Launchbury. Implicit parameters: Dynamic scoping with static types. In *27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL'00), Boston, MA, Jan. 19-21 2000
[31] Shinn-Der Lee and Daniel P Friedman. Quasi-static scoping: Sharing variable bindings across multiple lexical scopes. In POPL '93 – *Twentieth Annual ACM symposium on Principles of Programming Languages*, pages 479–492, Charleston (South Carolina, USA), January 1993. ACM Press
[32] Shinn-Der Lee and Daniel P. Friedman, *First-Class Extents*, TR-350, Computer Science Department, Indiana Univ., 1992.
[33] Christian Queinnec and David DeRoure. Sharing Code through First-class Environments. In proceedings of ICFP'96, ACM SIGPLAN International Conference on Functional Programming, May 1996
[34] H. Lieberman and al., *Your Wish is My Command:Programming by Example*. Morgan Kaufmann Publishers, Feb. 2001