Lisaac agent

Petit modèle agent pour langage à prototype

Benoît Sonntag* — **Pierre-Alexandre Voye****

* *INRIA-Lorraine / LORIA, 615 Rue du Jardin Botanique, BP 101, 54602 VILLERS-LES-NANCY Cedex, FRANCE*
bsonntag@loria.fr
** *La roche, 44360 Vigneux de Bretagne*
pierreavoye@oceanetpro.net

RÉSUMÉ. *Nous proposons dans cet article les bases d'un modèle simple de langage orienté agents réactifs pour le langage Lisaac, langage objet à prototype compilé. Nous présentons les axiomes de bases à implémenter dans le compilateur afin d'intégrer des mécanismes agents, en s'adaptant aux spécificités d'un langage objet à prototype. Nous proposons l'intégration de mécanismes comme la gestion de messages multicast, la définition du comportement réactif de l'agent, un mécanisme de structuration de SMA, et un mécanisme d'extraction structuré de données.*

ABSTRACT. *In this article, we suggest, the basis of a simple agent oriented language for the compiled and prototype-based language Lisaac. We show the basis axioma that we have to implement in the compiler in order to implement agent mecanism and adapt the model to a prototype-based and compiled language. We suggest a mecanism such as multicast messages, a reactive behaviour of the agent, a MAS structuration mecanism, and a structured data extracting system.*

MOTS-CLÉS : *Programmation Orienté Agent, modèle agent, langage à prototype, Lisaac*

KEYWORDS: *Agent Oriented Programming, agent model, prototype-based language, Lisaac*

1. Introduction

Bon nombre de logiciels sont modélisables en termes de petites unités logicielles collaborant ensemble afin de résoudre les problèmes qu'on lui soumet. Les paradigmes de programmation objets ont considérablement amélioré la tâche du concepteur logiciel en lui apportant un mode de découpage plus proche d'un mode de pensé humain. On peut d'ailleurs remarquer que la montée en niveau des langages de développement logiciel est concomittente à une proximité toujours plus accrue des modes de pensés et de représentations humains. Le paradigme objet trouve néanmoins ses limites dans la réalisation de logiciels excédents plusieurs dizaines de millions de lignes en imposant des architectures très complexes, peu souples et peu fiables (Livshitz, 2004). Cette constatation nous a amené à imaginer un modèle entièrement dédié à simplifier la tâche du développeur et surtout à lui permettre d'exprimer au mieux son intuition.

Nous pouvons synthétiser plusieurs caractéristiques de la représentation humaine :

• un être humain conceptualise le monde en terme d'objets animés ou non d'intentionnalités ;

• les objets ont tendance à se comporter en fonction de stimuli extérieur ;

• ces objets se transmettent des informations entre eux, et ces informations peuvent donner lieu à une modification d'un comportement ;

• une cause implique un effet et vice-versa ;

• les informations manipulées dans les programmes représentent des perceptions humaines, et possèdent donc une structure ontologique

Notre modèle a pour objectif d'intégrer ces concepts tout en restant dans le cadre d'un langage objet à prototype classique. Si la modélisation d'une fourmillière nécessite un langage orienté agent, des calculs matriciels se satisfont parfaitement d'un langage orienté objet classique. Les deux approches sont complémentaires. Par conséquent, l'une des contraintes que nous avons imposées à notre modèle est de concevoir une extension élégante dans son uniformité au langage Lisaac. Ainsi, notre modèle respecte les grandes lignes de ce langage et en particulier son caractère typé.

Les extensions ainsi prévues sont principalement orientées vers la conception de SMA réactifs , l'approche par but étant particulièrement problématique à compiler d'une part et d'autre part s'intègre difficilement à un langage objet à prototype classique. Néanmoins, le langage Lisaac étant un langage objet à prototype classique, on peut facilement doter un agent d'une représentation du monde. de plus le système de message présenté permet d'aller plus loin que la définition d'agent réactif purs, bien que l'on ai pas cherché à fournir des outils permettant de réaliser des systèmes multi-agents cognitifs.

Nous identifions plusieurs mécanismes permettant d'atteindre ce but :

1) Chaque agent doit pouvoir se comporter et réagir en fonction de son environnement direct. On défini donc des clauses environnementales déclenchant un comportement spécifique à la satisfaction de ses clauses.

2) Il est nécessaire, pour aider le concepteur d'un SMA, de récupérer facilement des données sur un ensemble d'objets ou d'agents. Nous proposons un système d'extraction de données, inspiré de SQL et adapté au modèle objet. Ici, un identifiant ou une référence d'un objet n'est plus nécessaire à son accès.

3) Les agents doivent pouvoir s'envoyer des messages en multicast en conditionnant la réception de ceux-ci à des critères définissables.

4) Le concepteur de SMA doit pouvoir posséder un outil lui permettant de représenter plusieurs niveaux de systèmes multi-agent, un SMA pouvant être un agent d'un autre SMA.

1.0.1. *Comparaison de notre approche avec d'autres contributions*

De nombreux langages ont été créés pour implémenter aisément des systèmes multi-agents. Ces langages tentent la plupart du temps de permettre de définir des SMA dotés d'agents cognitifs en offrant des fonctionnalités plus ou moins poussées en fonction de l'objectif initial.

Citons

1) AgentSpeak (Weerasooriya *et al.*, 1995) est un langage dans lesquels les agents sont autonomes et dotés d'états mentaux comme des croyances, objectifs, plans et intentions. AgentSpeak implémente un système de *trigger* permettant de créer des SMA constitués d'agents réactifs. Ce langage est exclusivement conçu pour créer des SMA. AgentSpeak n'est pas, à la base, un langage objet à prototype et ne permet pas d'utiliser les propriétés objets classiques.

2) Viva (G., 1996) est un langage combinant la programmation orienté aspect et des concepts issu de PROLOG. De même que AgentSpeak, il permet de définir croyances, objectifs, intentions et tâches. Il permet de définir des prédicats à la manière de SQL ainsi que le comportement de l'agent en fonctions des états mentaux et des évènements. VIVA est un langage orienté agent, suivant les précepts défini par Shoham3 (**?**) Cette approche très intéressante s'intègre difficilement avec un langage objet à prototype, constituant en elle même un paradigme fouillé de programmation et surtout trop spécifique. Viva n'est pas un langage envisageable dans un contexte industriel.

3) DIMA (Guessoum *et al.*, 1997) est l'extension de la plate-forme Actalk constituée afin de pouvoir programmer des SMA cognitifs dans lesquels chaque agent est doté d'un module de raisonnement. La réponse du module est calculé en fonction des ses perceptions, connaissances (grâce à une base de règles) ou de messages. Actalk étant basé sur SmallTalk (un langage objet à classe non typé), notre approche, moins ambitieuse et à base de prototype, reste orthogonal à ces travaux.

2. Le langage Lisaac - un langage objet à prototype

Le choix du langage objet pour l'application de notre modèle agent se justifie essentiellement sur deux points cruciaux. Tout d'abord, les langages à classe reflètent

d'une manière moins intuitive la vision que l'on peut donner à un agent. Le paradigme *classe/instance* des langages à classe compilés implique que la description statique d'un classe n'est pas directement présent et vivant dans l'univers de notre application. Une instance est alors nécessaire pour qu'un premier représentant d'une classe soit présent et vivant en mémoire. Une classe ne permet pas de visualiser celle-ci directement comme un agent, mais plutôt comme un schéma de construction d'un agent. En effet, une classe est un moule qui, dans les langages objets à classes compilés, ne sont vivant qu'après instanciation. Ce qui donne lieu à des différence entre le modèle et l'instanciation. Une classe B héritant de A, est à l'instanciation un objet unique intégrant A et B. A *contrario*, la description d'un prototype constitue directement un objet présent en mémoire, utilisable tel quel ou clonable si nécessaire. Ainsi, un prototype est directement en adéquation à l'image d'un premier "individu agent", comme Adam et Ève vivant dans notre SMA. Un objet B héritant de A donne lieu à deux objets A et B indépendants. Le second point est d'ordre plus pragmatique. Si nous voulons démocratiser la programmation agent, de nombreux programmeurs sont sensible à un point essentiel : les performances à l'exécution. Le langage Lisaac est actuellement l'unique langage à prototype possèdant un compilateur ayant de bonnes performances à l'exécution (Sonntag, 2005).

Dans le cadre de cet article, nous ne pouvons pas réaliser une présentation détaillé du langage Lisaac. Le manuel de référence (Sonntag *et al.*, 2003) est disponible sur le site du projet Isaac (`http ://IsaacOS.loria.fr`).

Pour donner une vue globale de Lisaac, notons simplement qu'il est syntaxiquement et sémantiquement proche du langage objet à prototypes Self (Ungar *et al.*, 1987), ou relativement proche du langage SmallTalk (Goldberg *et al.*, 1983), (Goldberg, 1984) (ce dernier étant néanmoins un langage à classe). En revanche, contrairement à Self, Lisaac se distingue par un système de type proche du langage Eiffel (Meyer, 1992), avec entre autre la généricité (type paramétrique).

D'un point de vue pratique, le langage Lisaac se manipule d'une façon assez similaire à un langage comme Eiffel, voire Java, avec les particularités de l'objet prototype. C'est un langage qui, au delà de ses particularités est très rapidement maîtrisable pour un développeur habitué à un langage comme Java.

3. Caractéristiques de notre petit modèle agent

3.1. *Régles et déclaration d'un agent*

Un ensemble de règles doivent être respectées pour définir un agent :

Règle n˚1 : Un prototype représentant un agent se distingue par l'affectation de la mention AGENT dans le slot `category` de la section HEADER du Lisaac (voir fig. 1 ligne (1)).

Règle n˚2 : Un agent doit obligatoirement hériter du prototype AGENT directement ou indirectement via l'arbre d'héritage (voir fig. 1 ligne (2)). Notons que le prototype

```
section HEADER
  + name := FOURMI;
  - category := AGENT;   // (1)
section INHERIT
  * parent_fourmi:AGENT; // (2)
section PUBLIC
  - is_in:AGENT := FOURMILIERE; // (3)
```

Figure 1. *Exemple de déclaration de l'agent* FOURMI

AGENT n'est pas de catégorie agent. Le prototype AGENT implante un ensemble de fonctionnalités génériques et nécessaires aux agents. Ce point fait l'objet de la section 3.6. De plus, cela nous permet de respecter les règles de typage du langage Lisaac lors de la manipulation des agents.

Règle n°3 : Un prototype de catégorie agent est obligatoirement un prototype feuille et ne peut donc pas être parent d'un autre prototype. Cela implique l'absence d'un prototype de catégorie agent dans l'arbre d'héritage d'un agent. Cette règle permet une cohérence et une simplification du modèle d'exécution parallèle des agents que nous développons en section 3.3.

Règle n°4 : La déclaration du slot `is_in` de type AGENT est obligatoire dans la description du prototype AGENT (voir fig. 1 ligne (3)). Ce slot est utilisé au niveau de la hierarchisation récursive des SMA et dans la communication inter-agent. Nous autorisons toutefois son absence pour un seul agent dans le système, la section 3.2 développe la présence de ce slot.

Rappelons la significations des symboles '+','-','*' précédent les déclarations de variables.

- Le symbole '+' signifie que le slot peut être cloné.
- '-' implique que nous partageons la valeur de ce slot avec tous les clones de cet objet.
- '*' signifie que le slot contient directement une représentation de de l'objet (il ne vaut pas '`null`', il contient déjà un objet du type du slot).

3.2. *Définition récursive d'un agent*

Une des bases de la définition d'un système multi-agentsrepose sur la récursivité de la notion : un SMA peut être lui-même un agent d'un autre SMA. Dans notre exemple, chaque organe d'une fourmi est un agent appartenant au SMA définissant l'entité d'une fourmi. De même, chaque fourmi peut être considérée comme un agent du SMA représentant la fourmillière. Dans le même ordre d'idée, plusieurs

fourmillières peuvent définir un SMA englobant l'univers d'application de notre simulation.

Le slot `is_in` imposé par la règle 4 permet d'architecturer les niveaux entre eux. Dans l'exemple de la figure 1, l'agent FOURMI appartient à l'agent ou méta-agent, ou encore au SMA FOURMILLIERE. Cette appartenance étant réalisée par un slot classique, elle reste dynamique.

Un SMA (ou méta-agent) peut alors diriger et interagir en donnant des directives à ces agents de deux manières :

- via un message classique entre agents (voir section 3.4) ;
- via un message en *multicast* en utilisant comme sélection la valeur du slot `is_in`.

L'agent représentant le SMA de l'univers d'application est l'unique agent n'ayant pas de slot `is_in`. Dans notre exemple, l'ensemble des agents FOURMILIERE appartiennent au SMA englobé dans l'agent WORLD. Celui-ci étant l'univers d'application, il ne possède pas de slot `is_in` décrivant son appartenance.

3.3. *Un agent : un prototype toujours actif*

À la différence d'un objet qui n'est actif que lors d'un envoi de message, un agent doit pouvoir être toujours actif et avoir son propre comportement indépendemment des autres.

Ainsi, chaque agent a son propre contexte d'exécution. Le clonage d'un agent a pour conséquence d'obtenir un nouveau contexte d'exécution pour le fonctionnement de celui-ci. La définition de ce clonage particulier est décrite dans le prototype AGENT ; parent imposé par la régle 2 de la déclaration d'un agent. Aussi, pour éviter tout problème d'incohérence des données communes entre deux agents clonés, nous réalisons une duplication récursive de ces données (*deepclone*). Si le programmeur est désireux de partager des données à plusieurs agents, cela est toujours rendu possibles par la construction d'un agent contenant ces données. Cette encapsulation permet de gérer de manière fiable la cohérence de ces données lors d'accès multiples à un instant T.

3.4. *Communication inter-agent*

La communication est réalisée par un envoi de message classique de type `agent.slot`.

Si le slot contient du code (méthode ou procédure), deux cas sont possible :

1) Si l'appelant réclame une réponse, par exemple `a := agent.slot`, le flot d'exécution de l'appelant est stoppé jusqu'à obtention de cette valeur de retour.

2) Si l'appelant ne réclame pas de réponse, le message est envoyé, stocké dans une FIFO de l'agent appelé, et l'appelant continu son flot d'exécution. Le message sera traité ultérieurement selon le choix et le comportement de l'agent visé.

Au niveau du prototype AGENT, nous avons deux slots particuliers permettant de gérer et de maîtriser la FIFO interne des messages asynchrones :

• `has_new_message` de type BOOLEAN permet de savoir si un message est en attente dans la FIFO.

• `pop_message` qui a pour objectif de déclencher l'exécution d'un message en attente dans la FIFO.

Si le slot appelé contient une donnée, nous renvoyons directement celle-ci sans passer par la FIFO et sans couper le flot d'exécution de l'agent appelé et l'agent appelant.

Les objets passés entre agents (arguments ou valeurs de retour) sont traités selon leurs types de la manière suivante :

• Si l'objet est de type `Expanded`, objet de petite taille, comme un entier, un booléen, un caractère..., l'objet est directement passé d'un agent à l'autre.

• Si l'objet est d'un type plus complexe (chaîne de caractère, structure, ...), nous réalisons automatiquement un clonage récursif (*deepclone*) pour éviter les incohérences possible causé par le partage de cet objet durant une exécution parallèle. Si des objets complexes sont à faire passer on les stockent dans un agent virtuel.

Une gestion plus complexe d'envoi de message de type *mail* est alors rendu possible par l'implantation d'une librairie basée sur ces primitives de bases.

3.5. *Comportement*

Un agent, intrinsèquement actif, doit décrire son comportement réactif. Celui-ci exprime comment l'agent se comporte à chaque instant et plus exactement quel comportement adopte t-il en fonction de son environnement.

Chaque comportement est lié à des conditions de son environnement ou à un état interne. C'est pour cela que nous décrivons le déclenchement d'un comportement à l'aide de clauses satisfaites lors de l'application de celui-ci. Chaque clause implémente le concept de cause à effet : La satisfaction d'une clause, autrement dit d'une cause directement liée à l'agent donne lieu à un effet (le comportement proprement dit).

Les clauses se posent sous forme d'une expression booléenne comportant un ensemble de termes et de connecteurs logiques.

Dans un langage objet classique, un message ayant un identifiant est à la base du déclenchement d'une action. Ici, nous remplaçons l'identifiant de message par la clause en question. De ce fait, le déclenchement de l'action (effet) est directement lié à la satisfaction de sa clause.

Exemple :

```
section PUBLIC
  - (is_hungry) <- to_eat_action;
```

Si la clause (`is_hungry`) est vrai, l'agent réalise l'action de manger.

Tant que la clause est satisfaite, l'action se répète. Il peut être nécessaire de réaliser une action préalable au comportement répétitif (`Preface`) et une action de finalisation de ce comportement (`Postface`). Exemple :

```
section PUBLIC
  - (is_hungry) <-
  Preface { to_make_food; }
  ( to_eat_action; )
  Postface { to_clear_table; };
```

Ici, si la clause (`is_hungry`) est vrai, l'agent commence par faire à manger (`to_make_food`), puis fait l'action de manger (`to_eat_action`) tant que la clause est vrai. Avant de démarrer un autre comportement, l'agent débarrassera la table (`to_clear_table`).

Comme l'action répétée peut être interrompue à tout moment, le code présent dans la partie `Postface` permet de rendre la cohérence des données causée par la rupture brutal de l'action.

Il est possible que plusieurs clauses soit satisfaites à un même instant. Le choix du comportement prioritaire sera réalisé par une "force" de priorité définie après une clause.

Exemple :

```
section PUBLIC
  - (is_hungry) Priority 5 <- to_eat_action;

  - (has_enemy) Priority 10 <- to_beat;
```

Si l'agent a un ennemi, il fait l'action de se battre, même s'il a faim.

Notons que la priorité par défaut est de 0. En cas d'égalité des priorités, l'ordre de déclaration des comportements rentre en jeu.

3.6. *Le prototype* AGENT

Nous avons décrit dans les sections précédente un certain nombre de fonctionnalités présentes dans ce prototype. Ici, nous les rappelons rapidement en ajoutant certaines notions qui nous paraissent importantes.

```
section HEADER
  + name := AGENT;
  - category := MICRO;

section PUBLIC
  - clone:SELF <- /* code system */
  // Nouvelle défi nition du slot 'clone' (deep-clone)
  // Avec création d'un nouveau contexte d'exécution

  - has_new_message:BOOLEAN <- /* code system */
  // Vrai, si un message est en attente dans la FIFO

  - pop_message <- /* code system */
  // Retire et exécute un message de la FIFO

  - (has_new_message) Priority 0 <- pop_message;
  // Comportement par défaut: Exécution au plus tôt d'un message en attente

  - time:UINTEGER <- /* code system */
  // Renvoi l'âge de l'agent en milliseconde

  - timein:UINTEGER;
  // Variable contenant un temps, nécessaire au calcul d'une durée.

  - timeout limit_time:UINTEGER :BOOLEAN <-
  // Calcul la durée écoulé entre 'time' et 'timein'.
  // Renvoi Vrai, si cette durée est inférieur ou égal à la
  // valeur 'limit_time' ou si 'timein' est égal à 0.
  (timein = 0) || {(time - timein) <= limit_time};
```

3.7. *Exemples de comportements attendus*

Si nous voulons qu'un comportement soit interrompu pour répondre à un message, nous pouvons utiliser la forme suivante :

```
  - (state = 0) <- ... // current_behaviour
```

Par défaut, la priorité d'un comportement est 0, le comportement par défaut défini dans le prototype AGENT sera prioritaire (car déclaré de priorité 0, mais avant notre comportement). Notre comportement reprendra automatiquement son exécution après le traitement du message (au passage de `has_new_message` = FALSE).

Si nous ne désirons pas interrompre un comportement par un message, nous pouvons utiliser la forme suivante :

```
- (state = 0) Priority 1 <- ... // current_behaviour
```

La priorité est supérieur au traitement d'un message.

Si nous voulons qu'un comportement de priorité supérieur à 0 soit néanmoins interrompu pour répondre à un message, nous pouvons utiliser la forme suivante :

```
- ((state = 0) & (! has_new_message)) Priority 1 <-
( ... /* Current behaviour */ )
Postface { pop_message; };
```

Un système multi-agents doit être réactif, et se conceptualise en temps réel. Le programmeur doit avoir la possibilité de conditionner l'exécution d'un comportement avec une limite de temps. Pour limiter l'exécution d'un comportement A à 30 millisecondes pour ensuite passer à un comportement B, nous pouvons utiliser la forme suivante :

```
- ((state = 0) & (timeout 30)) <-
Preface { timein := time; }
( ... /* Current behaviour A */ )
Postface { state  := 1; timein := 0; };

- (state = 1) <- ... // Code of behaviour B
```

Si nous voulons changer de comportement de manière synchrone à un nouvelle état d'un autre agent :

```
- ((state = 0) & (other_agent.state != 1)) <-
( ... /* Current behaviour */ )
Postface { state := 1; };

- (state = 1) <- ... // New behaviour
```

3.7.1. *Extraction de données*

Les systèmes multi agents supposent une communication accrue, en particulier les agents complexes et cognitifs. Une modélisation complexe nécessite une bonne connaissance de l'environnement extérieur, en particulier des connaissances complexes sur les propriétés d'agent et objets extérieur. Dans les langages classiques, le programmeur est obligé de parcourir inlassablement des collections pour y trouver les données qu'il cherche. Dans un langage agent, c'est la communication qui pallie ce manque.

Nous avons cherché à définir un mécanisme utilisable par les agents et par les objets. C'est pourquoi le modèle *Lisaac agent* intègre un mécanisme d'extraction des données inspiré entre autre de SQL, il s'agit d'envoyer des requêtes permettant de

filtrer des données à partir d'une ou plusieurs collections d'objets, ainsi que de déterminer une liste d'objets/agents répondant à certains critères.

Dans le cadre de cet article court, nous ne pouvons pas développer ces aspects de notre modèle. Le principe étant très proche des mécanismes décrits dans (Seriai, 2003), nous avons préféré mettre l'accent sur précédents points de notre modèle.

4. Conclusion

Le modèle *Lisaac Agent* se veut une synthèse pragmatique entre un langage à objet à prototype classique et un langage permettant d' implémenter des systèmes multi-agentsréactifs. Nous proposons une extension d'un langage de haut niveau qui se veut simple, intuitive tout en restant puissant et polyvalent.

Concevoir des agent cognitifs avec notre approche est beaucoup plus difficile, cette extension se positionne clairement dans le domaine d'agent réactifs. Nous le percevons comme le prolongement naturel de l'approche objet. Néanmoins, le programmeur pourra développer des systèmes multi-agents capables de manifester des comportements cognitifs grâce à l'émergence de comportement intelligents au sein de SMA réactifs. Cela dit, la possibilité d'envoyer des messages en multicast permettront au programmeur de concevoir des systèmes multi-agents faisant montre de fonctionnalités s'approchant de systèmes multi-agents cognitifs.

Une première implentation naïve de notre modèle est fonctionnelle. Si elle ne permet pas de réaliser des mesures d'efficacité sérieuses en terme de performance d'exécution, elle a le mérite de valider la faisabilité et de tester la pertinence de notre approche décrite dans cet article.

5. Bibliographie

G. W., « VIVA knowledge-based agent programming », april, 1996.

Goldberg A., *Smalltalk-80, The Interactive Programming Environment*, Addisson-Wesley, Reading, Massachusetts, 1984.

Goldberg A., Robson D., *Smalltalk-80, the Language and its Implementation*, Addison-Wesley, Reading, Massachusetts, 1983.

Guessoum Z., Briot J.-P., Dojat M., « Des objets concurrents aux agents autonomes », *Cinquièmes Journées Francophones sur l'Intelligence Artificielle Distribuée et les Systèmes Multi-Agents (JFIADSMA'97)*, vol. 33, number 5, Hermès Science Publications, p. 93-107, april, 1997.

Livshitz V., « `http ://java.sun.com/developer/technicalArticles/Interviews/livschitz_qa.html` », , The Next Move in Programming : A Conversation with Sun's Victoria Livschitz, February, 2004.

Meyer B., *Eiffel, The Language*, Prentice Hall, Englewood Cliffs, 1992. ISBN 0-13-247925-7.

Seriai A., « QUEROM : An Object-Oriented Model to rewriting query using views », *International Conference on Enterprise Information Systems*, 2003.

Sonntag B., « Le projet Isaac : une alternative objet de haut niveau pour la programmation système », *4ième Conférence Française sur les systèmes d'Exploitation, (CFSE'4)*, ACM Press, Mars, 2005.

Sonntag B., Colnet D., Boutet J., Lisaac - Programmer's Reference Manual, Technical report, LORIA, 2003. LORIA.

Ungar D. M., Smith R. B., « Self : The Power of Simplicity », *2nd Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*, ACM Press, p. 227-241, 1987.

Weerasooriya, Rao, Ramamohanarao, « Design of a concurrent agent-oriented language », *Intelligent Agents : Theories, Architectures, and Languages , no. 890 in Lecture Notes in Artificial Intelligence*, Springer-Verlag, p. 386-402, april, 1995.