Les systèmes multi-agents basés sur des réseaux de Petri colorés : technique de pointe ou technologie dépassée ?

François Barthélemy[1]

[1] Institut d'informatique des Facultés Notre-Dame de la Paix (FUNDP), Namur (Belgique)
fbarthel@info.fundp.ac.be

Résumé. En 1997, Moldt et Wienberg ont écrit un article sur l'utilisation des réseaux de Petri colorés pour représenter un système multi-agents. L'idée, très bonne à l'époque, a aujourd'hui un peu vieilli. Dans l'article qui suit, une analyse critique de l'article « *Multi-agents systems based on coloured Petri nets* » [1] est réalisée. Les auteurs s'étaient basés sur les techniques des réseaux de Petri colorés et sur la programmation orientée-agent de Shoham. Un exposé de ces techniques est donné. Les idées originales des auteurs sont ensuite exposées pour laisser finalement place à une critique actuelle de ces idées.

1 Introduction

Dans ce document, il sera question d'analyser une architecture multi agents basée sur des réseaux de Petri colorés. Afin de bien pouvoir aborder cette question, une brève introduction aux systèmes multi agents ainsi qu'aux réseaux de Petri colorés sera exposée dans un premier temps. Ensuite, l'idée proposée par Moldt et Wienberg en 1997 sera exposée. Dans la partie suivante, une critique ainsi que quelques idées seront développées.

2 Introduction aux concepts

2.1 Réseaux de Petri colorés

Les réseaux de Petri colorés sont, comme tous les réseaux de Petri, composés d'un ensemble de modules qui contiennent chacun un réseau de places, de transitions et d'arcs. La spécificité des réseaux colorés est d'associer une couleur de marque différente à chaque processus.

Selon [2], ils sont souvent employés comme un langage orienté graphique pour le design, la spécification, la simulation et la vérification de systèmes. Les réseaux de Petri colorés sont, par exemple, utilisés dans les domaines des protocoles de communication, les systèmes distribués ou encore les systèmes de production automatisés.

2.2 Réseaux de Petri colorés orientés objets

Dans tout système orienté objet, un objet invoque une méthode d'un autre objet en lui envoyant un message. Il y a moyen de combiner cette vue orientée objet avec celle des réseaux de Petri colorés.

Tout objet (ou instance d'une classe) peut être représenté par un réseau de Petri (figure 1) ; on appellera cette représentation un *objet-réseau.* Dans celle-ci, les variables sont remplacées par des places, les méthodes par des transitions. Les différents objets peuvent se transmettre des messages en utilisant des boîtes de sortie et des boîtes d'entrée (*out_pool* et *in_pool*), ces boîtes sont représentées par des places. Elles constituent une sorte d'interface entre les objets et le gestionnaire des messages. Ce dernier est l'entité qui se charge de transmettre les messages d'un objet vers un autre. On verra que dans l'architecture proposée, grâce à l'utilisation de réseaux colorés, ce rôle est assez simple.

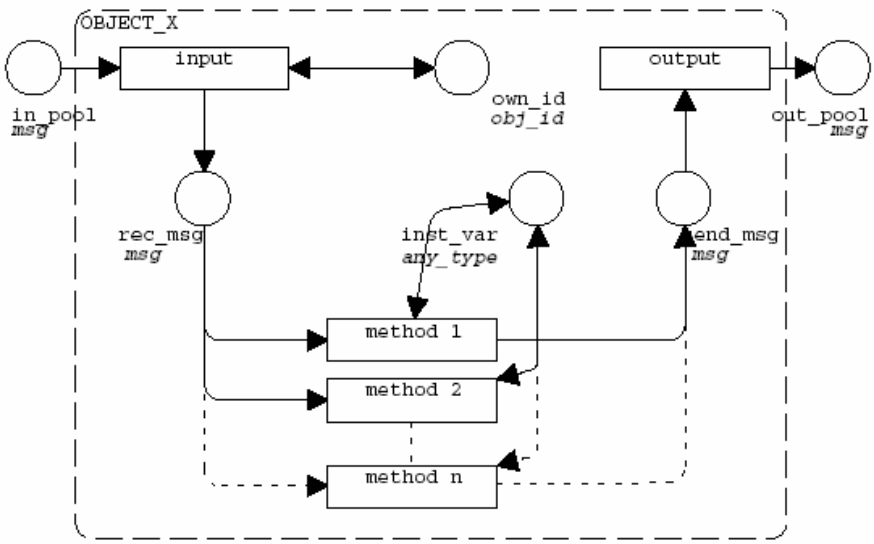

fig. 1 : objet-réseau venant de [3]

Dans ce modèle, chaque message est représenté par une marque comportant certaines informations. Lorsqu'un objet reçoit un message (présence d'une marque dans *in_pool*), la transition *input* sera tirée si le message lui est bien adressé. Il faut en effet la présence de marques de couleurs identiques dans *own_id* et dans *in_pool* pour que la transition *input* puisse être tirée. Une marque sera alors placée dans *rec_msg* où le nom de la méthode à effectuer sera lu de la marque. La méthode en question sera effectuée en utilisant éventuellement les variables locales. Le résultat aura la forme d'un message et sera placé dans *end_msg*. De là, il sera envoyé dans *out_pool* où son acheminement sera pris en charge par le gestionnaire des messages. Le rôle du gestionnaire de messages dans ce système est assez simple. En effet, vu qu'un contrôle

est fait à l'entrée des objets pour voir si le message leur est adressé, il suffit de fusionner (d'un point de vue schématique) *in_pool* et *out_pool*.

Les classes peuvent également être représentées par ce système (figure 2). En effet, une classe définit le type de ses instances et contient quelques méthodes propres comme la destruction et la création d'objets (d'où leur appellation d'usines à objets). Une classe peut envoyer des messages et peut dès lors également être vu comme un objet.

Il faudrait donc un mécanisme interprétable en réseau de Petri pour représenter les classes, leur fonctionnement et la gestion des objets de ces classes. Tout cela est réalisable en enrobant les *objets-réseau* qui ont le même état et le même comportement dans une *classe-réseau*. Afin d'arriver à ce résultat, on garde la même structure que les *objets-réseau* et on étend le type de chaque place avec l'identifiant de l'objet.

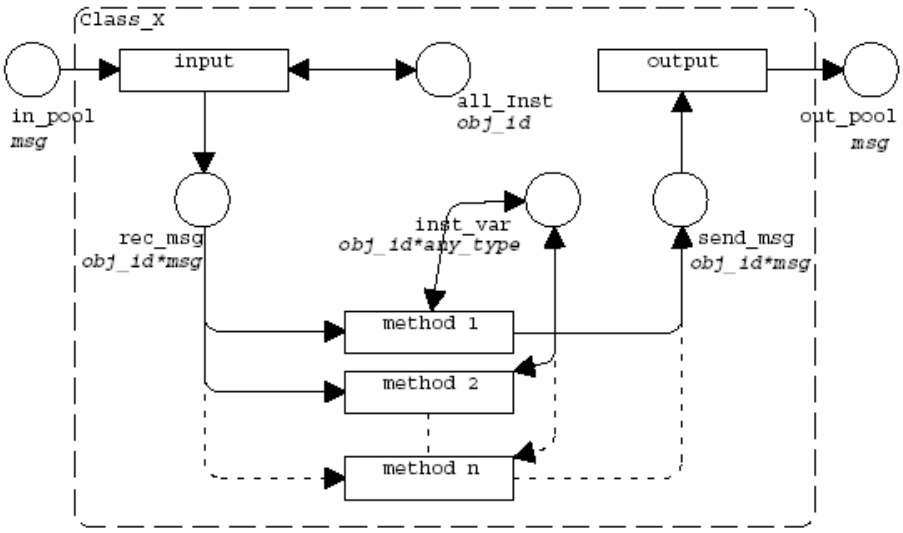

fig. 2 : classe-réseau venant de [3]

Le fonctionnement de cette *classe-réseau* est identique au fonctionnement d'un *objet-réseau* au point près que les méthodes propres aux classes ont été rajoutées.

2.3 Programmation orientée agent de Shoham

Shoham a proposé dans [4] une méthode de programmation d'agents. Il se base sur la programmation logique pour représenter l'état courant de l'agent. Cet état courant de l'agent sera appelé *état mental* par Shoham. L'utilisation d'un langage logique permet de représenter l'état mental de l'agent sous forme de *faits*, de *croyances* et d'*engagements* envers d'autres agents. Dans cette représentation, une action est représentée par un fait qui devient vrai une fois que l'action est effectuée.

Le comportement de l'agent est commandé par des règles d'engagement (*commitment rules*) qui déterminent les actions à effectuer par l'agent suivant les messages reçus et l'état mental de l'agent. Chaque règle d'engagement contient une condition sur les messages, une condition sur l'état mental et une action. Pour savoir si une règle d'engagement est déclenchée ou non, l'agent compare la condition sur les messages avec les messages reçus, il compare la condition sur l'état mental et ses croyances du moment. Si la règle est activée, alors l'agent est engagé à réaliser l'action. Ces actions peuvent être de plusieurs sortes : communicatives (envoyer des messages), privées ou internes (exécuter des sous-routines). Ainsi, comme dit dans [5], une règle pourrait être paraphrasée de la façon suivante :

SI je reçois un message d'un agent qui me demande d'effectuer une action à un temps donné et que je crois que :
- cet agent est un ami ;
- je peux effectuer cette action ;
- au moment donné, je n'ai rien de prévu,

ALORS engagement à réaliser l'action au temps donné.

Shoham insiste également sur le fait que les messages sont porteurs d'informations. Le fonctionnement général d'un agent peut être vu de la façon suivante :

1. lire tous les messages courants, mettre à jour les croyances et donc les engagements (si nécessaire)
2. exécuter tous les engagements du cycle courant dont les conditions sont satisfaites
3. goto (1).

3 Idée originale des auteurs

L'idée originale des auteurs de l'article analysé est de combiner des éléments des théories suivantes :

- l'approche orientée-objet qui est à la base des agents et des systèmes multi-agents
- l'utilisation des réseaux de Petri comme une méthode de description formelle et pratique pour spécifier, implémenter et simuler des systèmes d'agents
- l'utilisation de la programmation logique pour rejoindre le langage de programmation orienté-agent de Shoham.

En résumé, leur idée est de transposer la vue de Shoham dans un environnement distribué en utilisant les réseaux de Petri colorés. Ainsi, le fait que les actions soient représentées par des prédicats qui devenaient vrais avec Shoham est traduit en Petri par une transition qui peut être tirée si une marque concernant le prédicat en question existe. Shoham fait une distinction entre le fait de s'engager à faire

une action et l'exécution de l'action en elle-même ; cette distinction est traduite en Petri en représentant ces deux tâches par deux transitions distinctes qui peuvent se déclencher indépendamment et même de manière concurrentielle.

Pour ce qui est des systèmes multi-agents, Moldt et Wienberg spécifient que le système et les agents qui le composent doivent avoir les propriétés suivantes :

- indépendance : un agent peut accomplir des actions, soit lui-même soit par délégation. L'exécution de ces actions doit être traitée de manière concurrentielle ; un agent totalement accaparé par une action ou attendant le résultat d'une action effectuée par un autre serait totalement inutile.
- communication : les agents sont capables de communiquer entre eux via des messages. Ces messages peuvent être considérés comme des *speech act* comme définis par [6].
- intelligence : les agents sont capables d'utiliser une représentation symbolique des données et ils sont capables de déduire de nouvelles connaissances.

Ainsi, l'utilisation des réseaux de Petri semble être un bon choix de formalisme. En effet, il permet, entre autres, d'avoir une spécification statique et dynamique du système. De tels systèmes peuvent également être évalués via l'utilisation d'outils qui permettent d'exécuter un réseau de Petri. En outre, les réseaux de Petri offrent une représentation graphique qui reste formelle.

Il y a deux façons de regarder un tel système : la vue globale de toute la société d'agents et la vue d'un seul agent. Bien que ces vues soient interconnectées et que les propriétés des agents influencent les propriétés du système, on ne peut pas parler d'héritage des propriétés. En effet, par exemple, des agents peuvent très bien être exempts de blocages sans que l'on aie pour autant une société d'agents qui évite les blocages. Nous allons maintenant exposer brièvement ces deux vues.

3.1 La société d'agents

La société d'agents consiste en un ensemble d'objets qui ont chacun leur propre identité. Une classe d'agents et les instances qui lui sont reliées peuvent être représentées par un seul réseau de Petri (qui doit être associé à une classe-réseau). Ce réseau de Petri commun peut être décomposé en un ensemble d'objets-réseau (un par agent existant).

D'un point de vue général, on peut constater que les différentes classes-réseau possibles diffèrent entre elles seulement du point de vue des méthodes particulières de leurs agents. Elles possèdent toutes des méthodes pour créer et détruire des agents (gestion des instances). On peut remarquer que chaque agent créé par une même classe se voit initialement doté de la même base de connaissances que tous ses frères. Dès lors, toutes les classes d'agents ont la même structure que celle exposée dans la figure 3.

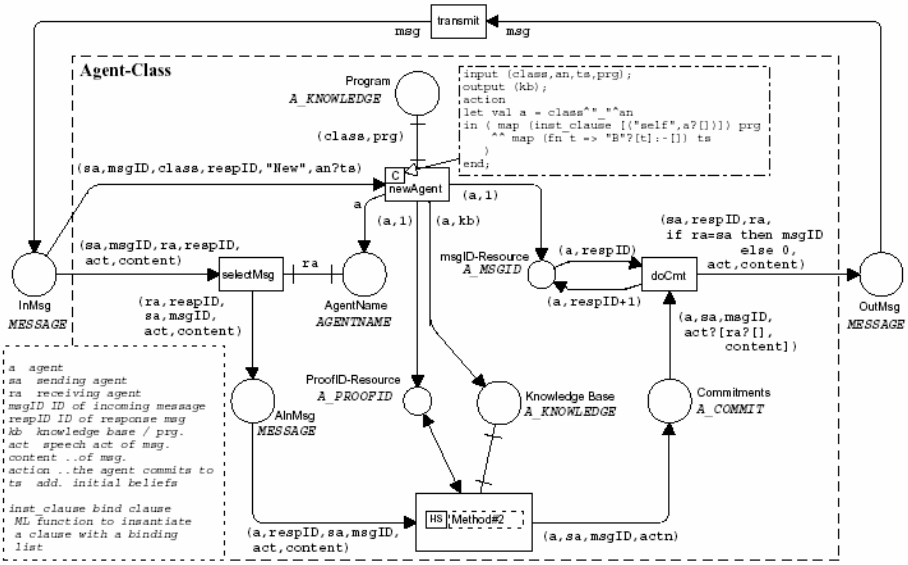

fig. 3 : une société d'agents représentée comme une classe-réseau venant de [1]

3.2 Un seul agent

Quand on analyse les agents, chacun d'entre eux doit être considéré comme l'instance d'une classe. Comme montré dans la figure 4, chaque agent peut recevoir et envoyer des messages. Si on regarde leur fonctionnement, on peut voir que les messages sont traités par une méthode centrale qui transforme ces messages en prédicats qui pourront ensuite être exécutés par un *prouveur* de théorèmes. A cet effet, celui-ci accède à la base de connaissances de l'agent et en déduit les actions à réaliser. Ces actions sont alors exécutées lors du tirage de la transition suivante. Ces actions peuvent aussi bien être des actions à effectuer par l'agent lui-même que des messages à envoyer à d'autres agents.

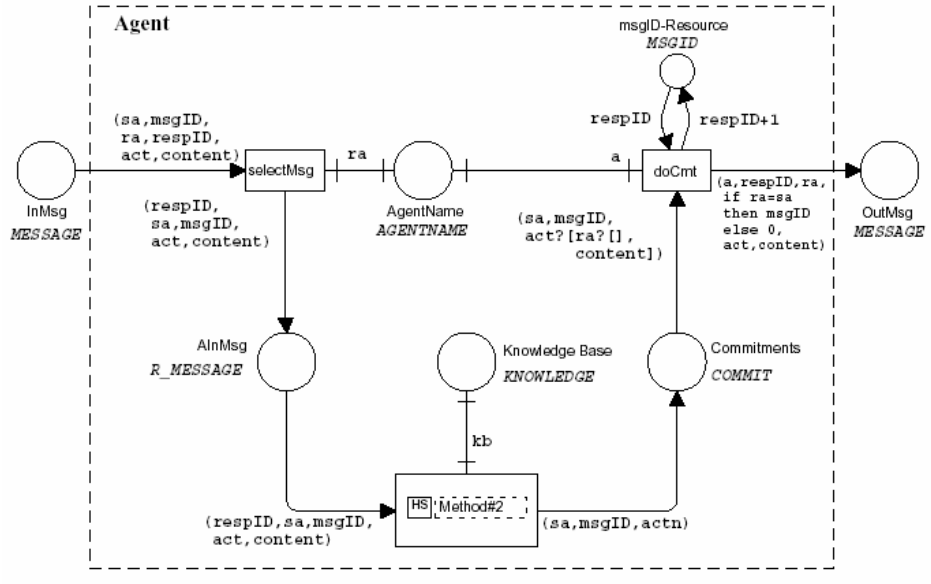

fig. 4 : un agent représenté comme un objet-réseau venant de [1]

Intéressons nous un instant au traitement des messages. On peut tout d'abord signaler que chaque message en entrée contient une action à effectuer ou une information; cependant, l'agent qui reçoit le message peut réaliser une toute autre action qui lui serait dictée par ses règles d'engagement. Chaque message va être décomposé en ses différentes parties par la transition *makeqry* :

1. *sa* est le nom de l'envoyeur.
2. *msgID* est l'identifiant du message (unique pour *sa*).
3. *ra* est le nom de l'agent auquel le message est adressé.
4. *respID* est le numéro du message dont le message courant est la réponse.
5. *type* est le type du message (*request* ou *inform*).
6. *content* est un prédicat qui contient toutes les données du message.

Grâce à toutes ces informations, l'agent va pouvoir décider quelle action effectuer. Il va faire ce choix en analysant ses règles d'engagement qui peuvent être vues comme étant de la forme suivante :

(COMMIT *msgcond mntlcond* (agent action)*)

Dans cette formule, *msgcond* peut être considéré comme un pattern qui doit correspondre au message en entrée. *mntlcond* peut être vu comme une condition sur l'état mental de l'agent, c'est-à-dire une combinaison logique de croyances et d'engagements. Si les deux conditions sont respectées, l'agent va rédiger une requête par <*agent action*> existant dans la règle. Il va engager les *agents* à effectuer les *actions*. Ces requêtes seront transformées en messages par la transition *makeCmt*. Toutes ces opérations sont montrées dans la figure 5.

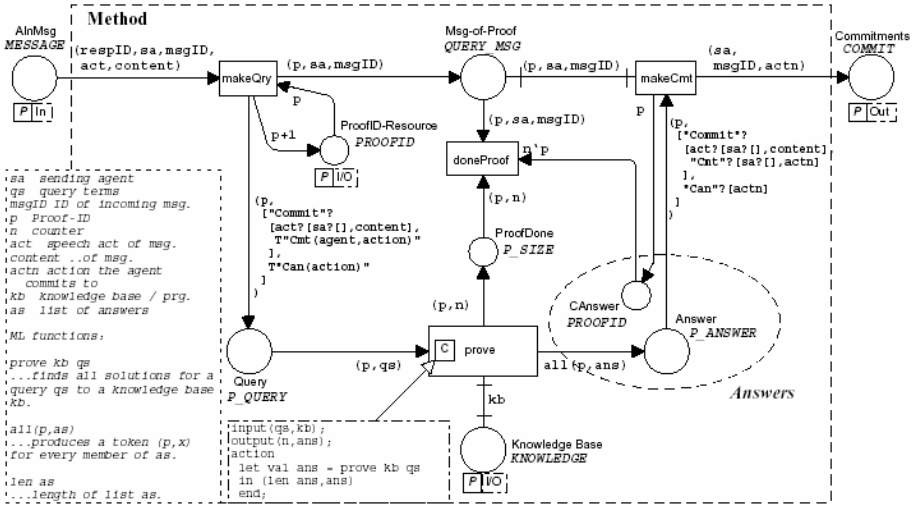

fig. 5 : raffinement de la transition du traitement des messages venant de [1]

On peut finalement rajouter que le système défini par Moldt et Wienberg utilise un *prouveur* de théorèmes semblable à celui de Prolog. Il sera évidemment exécuté de manière concurrente (afin de ne pas bloquer le système) et sera géré de manière incrémentale. Cette dernière propriété signifie que, plusieurs solutions pouvant être générées suite à l'analyse d'un même prédicats, ces solutions sont exprimées sous forme de marques dès leur production et la sortie du système peut être "augmentée" au fur et à mesure. Le *prouveur* de théorèmes sera ici développé comme un sous-réseau contenu dans l'architecture de chaque agent.

4 Critiques

L'utilisation des réseaux de Petri comme outil de représentation est parfois ambigu. En effet, cette représentation ne permet que l'utilisation de places, de transitions et de marques. Moldt et Wienberg ont considéré que les messages qui peuvent être transmis entre agents peuvent être de deux types (information et requête), cependant, d'autres types de messages sont maintenant souvent utilisés dans des architectures multi-agents : ordre, interdiction, question,… ([7]) Autant l'idée des auteurs pouvait se défendre en 1997, autant elle semble un peu obsolète aujourd'hui. Une représentation via des schémas UML sera tout aussi intuitive mais permettra de donner plus de détails sur l'architecture interne.

L'architecture multi agents présentée par les auteurs est une architecture assez élémentaire. Pour des agents plus sophistiqués, la représentation via réseau de Petri serait fort peu intuitive. En effet, beaucoup d'agents utilisent des entrées venant de senseurs ou de capteurs dans le monde réel. Ces entrées peuvent difficilement être transmises sous forme de messages et encore plus difficilement transposées en

prédicats logiques. D'autre part, si on se base sur la définition d'un agent développée par Franklin et Graesser dans [8], on peut voir qu'un agent doit avoir les propriétés suivantes : autonomie, réactivité, persistance et pro-activité (agir sur l'environnement autrement qu'en réponse à des stimuli). Certaines de ces caractéristiques et leurs incidences dans la structure des agents ne pourraient pas être intuitivement traduites par des réseaux de Petri. Ainsi, le fait induit par la réactivité que l'environnement envoie des stimuli et le fait que l'agent doit observer son environnement (pro-activité) ne peuvent pas être simplement représentés par des réseaux de Petri.

Le fait d'avoir choisi d'utiliser la programmation logique pour représenter l'état mental des agents et donc aussi pour effectuer les choix des actions à effectuer est très intéressant d'un point de vue purement scientifique. Cette approche permet en effet d'avoir des agents qui peuvent apprendre et évoluer au cours de leur durée de vie. Il suffit en effet de rajouter de nouvelles règles ou d'adapter d'anciennes pour faire évoluer l'agent. De plus, cette approche permet d'approcher le vieux rêve des informaticiens de résoudre un problème en donnant simplement sa spécification. Cependant, il faut également remarquer que certaines données du monde réel et certaines actions sont difficilement traduisibles en prédicats. Cet inconvénient insiste une fois de plus quant au fait que l'architecture développée par les auteurs ne s'adapte pas aux environnements dynamiques (par exemple, le monde réel).

5 Conclusion

L'utilisation des réseaux de Petri pour représenter des systèmes multi agents est une idée intéressante dans le sens où la symbolique assez simple des réseaux de Petri permet une compréhension rapide de l'architecture globale. Cependant, la représentation en réseau de Petri peut paraître aujourd'hui obsolète sous certains aspects. Des représentations plus puissantes et tout aussi intuitives existent de nos jours. Dès lors, l'utilisation de réseaux de Petri colorés comme base pour le développement et la représentation de systèmes multi agents peut encore être envisagée pour des systèmes simples dont les agents travaillent dans un environnement statique (non réel) ; pour des systèmes plus complexes, l'utilisation des réseaux de Petri est à éviter.

6 Bibliographie

[1] Moldt, M., Wienberg, F.: Multi-Agent-Systems based on Coloured Petri Nets. Publié dans "Proceedings of the 18th International Conference on Application and Theory of Petri Nets", Toulouse (1997)

[2] ??: Very Brief Introduction to CP-nets, http://www.daimi.au.dk/CPnets/intro/verybrief.html (dernière modification le 28/12/2003) (consulté le 10/05/2004)

[3] Maier, D., Moldt, M.: Objektorientierte Konzepte – Dargestellet mit gefärbten Petrinetzen. Fachbereichsbericht FBI-HH-M-261/96, Universität Hamburg, Fachberei Informatik (1996)

[4] Shoham, Y.: Agent-Oriented Programming, AI 60, 51-92 (1993)

[5] Wooldridge, M., Jennings, N. R.: "Agent Theories, Architectures, and Languages: a Survey," in Wooldridge and Jennings Eds., Intelligent Agents, Berlin: Springer-Verlag, 1-22 (1995)

[6] Searle, J. R.: “Speech Acts”, Cambridge University Press (1969)

[7] Vlassis, N.: “A concise introduction to Multi-Agents systems and distributed AI”, Intelligent Autonomous Systems, Informatics Institute, University of Amsterdam, chapitre 6 (2003)

[8] Franklin, S., Graesser, A.: “Is it an Agent, or just a Program? : A Taxonomy for Autonomous Agents”, Proceedings of the Third International Workshop on Agent Theories, Architectures and Languages (1996)