Interprétation de commandes en langage naturel pour les agents conversationnels à base d'ontologie

Laurent Mazuel, Nicolas Sabouret

Interprétation de commandes en langage naturel pour les agents conversationnels à base d'ontologie

L. Mazuel* N. Sabouret*
laurent.mazuel@lip6.fr nicolas.sabouret@lip6.fr

*Laboratoire d'Informatique de Paris 6 (LIP6)
104 av du Président Kennedy
75016 Paris – FRANCE

Résumé :
Dans cet article, nous nous intéressons à l'interprétation des commandes en langue naturelle d'un utilisateur à un agent artificiel. Nous proposons une architecture pour le traitement de ces commandes adaptable à différents types d'applications. Les algorithmes de traitement dépendent uniquement du code de l'agent et de l'ontologie de domaine de cet agent. Nous présentons ensuite une évaluation comparative de trois approches : l'approche descendante, reposant sur les contraintes syntaxique du langage de description de l'application, l'approche ascendante, reposant sur l'utilisation de connaissances sur l'ensemble des actions possibles de l'agent et finalement une proposition d'approche combinée.

Mots-clés : Communication humain-agent, introspection, ontologie de domaine, approche ascendante et descendante, evaluation comparative

Abstract:
This paper focuses on a generic architecture provided with a natural language (NL) algorithm for command interpretation that can be adapted to different agent's domains for human-agent communication. Our NL architecture only depends on the agent's code and its domain ontology. We consider two approaches for NL command interpretation : the top-down approach and the bottom-up approach. We propose to combine both approaches in a bottom-up based algorithm that makes use of agent's constraints. We propose a comparative evaluation of these three algorithms.

Keywords: Human-Agent communication, introspection, domain ontology, top-down and bottom-up approach, comparative evaluation

1 Introduction

1.1 Présentation du problème

La communauté des Systèmes Multi-Agent (SMA) s'intéressent particulièrement et depuis longtemps à la résolution de problèmes par des agents cognitifs autonomes [4]. A ce titre, elle étudie les protocoles et modèles d'interaction formels entre agents. Ainsi, il n'existait initialement que peu de travaux sur la communication humain-agent qui reste une problématique très ouverte.

D'autre part, la communauté Agents Conversationnels Animés s'intéresse particulièrement à l'interaction en langue naturel entre un utilisateur humain et un agent artificiel. Elle obtient de très bon résultats dans le domaine de l'interaction multi-modale[2] et des expressions d'émotions [9]. Cependant, au niveau compréhension des langues, ces approches s'appuient essentiellement sur des algorithmes de *pattern-matching* ad-hoc sans réelle analyse sémantique [1]. D'autre part, la communauté des systèmes de dialogues utilise les ontologies pour s'approcher d'une architecture plus générique [7]. L'idée importante sous-jacente à l'utilisation d'ontologies est de pouvoir généraliser les algorithmes de traitement sémantique afin de les faire dépendre uniquement de son formalisme. Ainsi, ces applications ne sont plus dépendantes que de l'ontologie et du résolveur de problème spécifique de la tache à accomplir. Par exemple, certains systèmes utilisent ainsi l'ontologie pour paramétrer un parseur générique [8]. Cependant, pour obtenir de bons résultats, ils imposent un formalisme de description d'ontologie très contraint, efficace pour un sous-ensemble d'application, mais loin des modèles ouvert clas-

siques. De plus, ce type d'ontologie décrit le modèle de l'application et non le domaine de travail de l'application. Nous pensons qu'il est possible d'extraire la sémantique des actions du code de l'application directement. L'ontologie ne serait plus alors une deuxième description de l'application, mais une description des liens sémantiques entre les différents concepts présents dans le domaine de l'application (ce qui constitue le rôle initial d'une ontologie).

1.2 Plan de l'article

Nous allons essayer de montrer qu'il est possible de définir un système d'interprétation de commandes en langue naturel (LN) basé uniquement sur une ontologie de domaine et des agents capables d'introspection. Nous proposons pour cela trois strategies, les approches descendante, ascendante et combiné.

Dans la seconde section, nous donnerons une description générale de notre modèle d'agent. La section 3 présente la chaîne de base de TAL et le gestionnaire de dialogue qui sont les parties de l'architecture commune aux trois algorithmes. Dans la section 4, nous introduisons les trois algorithmes génériques différents pour l'interprétation de commandes. La section 5 présente une évaluation préliminaire de ces algorithmes. La section 6 conclue l'article.

2 Notre modèle d'agent

2.1 Le modèle VDL

Nos agents sont programmés en utilisant le langage VDL (*View Design Language*)[1]. Le modèle VDL est basé sur la réécriture d'arbre XML : la description de l'agent est un arbre dont les noeuds représentent ses données ou ses actions. L'agent réécrit l'arbre à chaque étape de l'exécution suivant un certain nombre de mots-clés du langage présent dans les noeuds. Ce modèle permet aux agents d'accéder *à l'exécution* à la description de leurs actions afin de raisonner dessus pour de la planification, pour répondre à des questions d'états formelles [11], pour modéliser son comportement, *etc*.

Dans le modèle VDL, chaque agent est nanti d'une ontologie OWL. Cette ontologie doit contenir au minimum tout les concepts utilisés par l'agent (*i.e.* les concepts VDL)[2]. Il existe donc une fonction injective de l'ensemble des concepts VDL de l'agent vers l'ensemble des concepts définis dans l'ontologie.

2.2 Modèle d'actions

En VDL, les réactions[3] sont activées par des *évènements, i.e.* noeuds XML envoyés à l'agent en guise de commande. Ils sont la représentation formelle (*i.e.* en VDL) des commandes. Les réactions décrivent comment ces messages (envoyés par un utilisateur ou même un autre agent) doivent être traités. L'objectif du système décrit ici est de construire des évènements VDL à partir d'une commande utilisateur en LN.

En VDL, comme dans la plupart des modèles de représentation des actions, nous représentons une action par un tuple $r = \langle nom, P, E \rangle$ où nom est le nom de l'action, P l'ensemble des préconditions de l'action et E son ensemble d'effets. Nous pouvons définir quatre types de préconditions pour une réaction r de R, l'ensemble des réactions possible :

– $\mathcal{P}_e(r)$ est l'ensemble des préconditions d'évènement. Elles sont utilisées pour

[1] `http://www-poleia.lip6.fr/~sabouret/demos`

[2] Ces concepts sont présent dans le code XML de l'agent soit comme étiquette (*i.e. tag*), comme attributs ou contenu texte.

[3] Cet article traitant d'interaction utilisateur, nous limiterons nos actions aux *réactions* (par opposition au comportement proactif d'un agent).

aiguiller une certaine forme d'évènement vers une certaine classe de réactions, ou pour rejeter les évènements globalement mal formés.
– $\mathcal{P}_s(r)$ est l'ensemble des préconditions de structure. Elles sont utilisées pour vérifier la syntaxe précise d'un message et assurer que la réaction a toutes les informations nécessaires pour s'exécuter sans erreurs. Les préconditions de $\mathcal{P}_s(r)$ ne dépendent donc pas de l'état courant de l'agent, mais uniquement de la structure détaillée de l'évènement.
– $\mathcal{P}_c(r)$ est l'ensemble des préconditions de contexte. Ces préconditions ne dépendent que du contexte courant de l'agent.
– $\mathcal{P}_{cs}(r)$ est l'ensemble des préconditions contextuelles-structurelles, *i.e.* préconditions dépendantes de l'évènement *et* du contexte courant de l'agent.

Nous noterons $\mathcal{P}_e = \cup_{r\in\mathcal{R}}\mathcal{P}_e(r)$. Pour tout $e \in \mathcal{P}_e$, nous noterons $R_e(evt) = \{r \in \mathcal{R} | evt \in \mathcal{P}_e(r)\}$ l'ensemble des réactions dont l'exécution donnera lieu à des effets par l'évènement evt.

3 Architecture globale

Cette section présente les modules de LN communs aux trois algorithmes d'interprétation.

3.1 Outils de base de TAL

Dans notre projet (figure 1), le module lexical de base est celui du projet Open Source OpenNLP[4]. Ce module contient un étiqueteur, un lemmatiseur (simple lien vers le lemmatiseur WordNet contenu dans JWNL). Comme nous l'avions constaté dans l'évaluation de [10], l'utilisation d'un analyseur syntaxique n'est pas efficace pour les commandes en langue naturelle. En effet, les utilisateurs emploient plus souvent des mots clefs que des phrases

[4] `http://opennlp.sourceforge.net/`

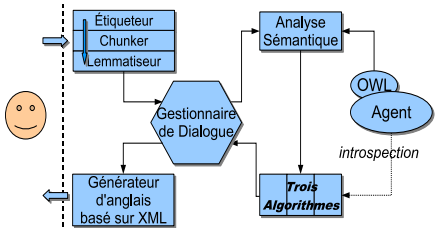

FIG. 1 – Architecture générale

bien structurées (*e.g.* "*drop object low*" ou "*take blue*"). Nous utilisons dans cet article une représentation du type "sac de mots" (après avoir enlevé les mots blancs repérés par leur étiquette)[5].

Notre objectif en analyse sémantique est d'utiliser les ontologies pour l'appariement des concepts utilisateurs avec ceux de l'agent VDL. Dans cet article, nous n'utilisons que la synonymie (*owl :sameAs*) de l'ontologie pour faire cet appariement. Cette analyse repose sur l'hypothèse de *connectivité sémantique* [12] enrichie par l'utilisation de la relation *owl :sameAs* : chaque concept qui apparaît dans une commande correcte est soit directement associé à un concept VDL, soit en relation *owl :sameAs* dans l'ontologie avec un concept VDL. Nous notons C l'ensemble de tous les concepts VDL liés (soit directement, soit via la relation *owl :sameAs*) à un terme apparaissant dans la commande utilisateur. C'est à partir de cet ensemble que nous construirons les évènements VDL au moyen des algorithmes décrits dans la section 4.

La dernière partie de notre chaîne est un générateur d'anglais qui transforme un noeud VDL en une phrase anglaise. Le résultat grammaticalement incorrect, mais suffisant pour que les utilisateurs comprennent les propositions du système dans

[5] L'objet de cet article n'étant pas l'interprétation syntaxique et sémantique des commandes, nous nous contenterons d'une version très simplifiée, afin de nous focaliser sur l'étude comparative. Nous n'évaluerons pas dans la section 5 les résultats uniquement liés à cette simplification.

notre évaluation.

3.2 Le gestionnaire de dialogue

Le gestionnaire de dialogue (GD) est responsable des dialogues avec l'utilisateur pour l'acceptation d'une commande, mais aussi pour la gestion des commandes incomplètes ou imprécises. Le paramètre d'entrée de notre GD est l'ensemble d'évènements créé par l'un des trois algorithmes que nous détaillerons dans la section 4 (approche descendante, ascendante, combinaison des deux). La première étape de notre GD est de découper cet ensemble, que nous noterons $\mathcal{G}$, en deux sous-ensembles disjoints :

- L'ensemble $\mathcal{E}$ des évènements possibles dans le contexte courant de l'agent.
- L'ensemble $\mathcal{F}$ des évènements impossibles. Ils ne correspondent pas forcement à une mauvaise modélisation par l'un des trois algorithmes, l'utilisateur pouvant parfaitement poser une commande impossible à réaliser par l'agent (impossible de manière générale ou de manière contextuelle).

$\mathcal{E}$ et $\mathcal{F}$ ne contiennent pas forcement *tous* les évènements possibles ou impossibles, ils représentent seulement un partitionnement des propositions faites par l'algorithme d'interprétation en cours. Le GD calcule ensuite le score d'appariement entre chacun des évènements de $\mathcal{E}$ et $\mathcal{F}$ et la commande utilisateur (la fonction de score est définie plus en détails dans [6]). A l'aide de ce score, nous retirons de $\mathcal{E}$ et de $\mathcal{F}$ les sous-ensembles d'évènements à score non-maximaux. En d'autres mots, $\mathcal{E}$ et $\mathcal{F}$ ne contiennent plus que les meilleurs évènements comparativement à la commande utilisateur parmi ceux générés par l'interprétation.

Pour l'exploitation de ces deux ensembles, nous définissons deux *seuils de compréhension* : p_{min} est la limite séparant les commandes *incomprises* des commandes *comprises sans certitude* et p_{max} est le score séparant les commandes *comprises sans certitude* des commandes *parfaitement comprise*s. Ces seuils correspondent, dans le principe, aux seuils "tell me" et "do it" de Patty Maes dans [5]. Le retour à l'utilisateur donné par le GD est dépendant de la position relative des deux scores $p_{\mathcal{E}}$ et $p_{\mathcal{F}}$ entre eux et par rapport aux deux seuils p_{min} et p_{max}. Nous différencions sept cas possibles [6]. Par exemple, si $p_{\mathcal{E}} \geq p_{max}$ la commande est considérée comme correctement comprise par le système quelque soit la valeur de $p_{\mathcal{F}}$. D'autre part, une commande sera considéré comme comprise mais impossible si $p_{\mathcal{E}} \leq p_{min}$ et $p_{max} \leq p_{\mathcal{F}}$, etc.

4 Algorithmes d'interprétation

Cette section présente les algorithmes construisant l'ensemble d'évènements $\mathcal{G}$ à partir de la commande et du code de l'agent.

4.1 L'approche descendante

L'approche descendante (*i.e. top-down*) consiste à *construire* une commande formelle (*i.e.* évènement en VDL) à partir de la commande en LN en considérant uniquement les contraintes structurelles imposées par le modèle formel du langage (*e.g.* [12, 13]). Elle est couramment utilisée en dialogue, en situation où il est difficile de prévoir à l'avance l'étendue des commandes ou de déterminer ce qu'il est possible d'effectuer à l'instant courant. Le principal défaut de cette approche est lié à la difficulté de définir des règles génériques de transformation dans le système. Ainsi, les systèmes actuels définissent un grand nombre de règles spécifiques de l'application, afin de réduire au maximum la création "d'évènements impossibles".

Dans notre approche, nous essayons de nous affranchir des règles d'interpretation

liées à l'application. C'est pourquoi nous n'utilisons que les préconditions de subsomption ($\mathcal{P}_e$) et l'état courant de l'agent pour construire l'évènement VDL à partir des concepts compris de l'utilisateur (C). Les préconditions de subsumption permettent de définir le squelette des évènements possibles pour un concept donné. Une analyse plus profonde de l'état interne de l'agent permet d'enrichir le squelette avec les concepts possibles. Plutôt que d'utiliser des règles strictes de grammaire, nous proposons de définir une méthode de construction d'évènements basée sur la syntaxe VDL et d'appliquer des heuristiques pour contraindre la construction selon la sémantique opérationnelle VDL.

Soit $E = \{e \in \mathcal{P}_e | tag(e) \in C\}$ et $\forall e \in E$, soit $C_e = C \backslash \{tag(e)\}$. Pour chaque $e \in E$, notre algorithme considère l'ensemble des feuilles L_e de e et cherche à l'intérieur du code les noeuds $t \in L_e$ qui contiennent au moins un concept $c' \in C_e$ dans leurs sous-éléments. $\forall e \in E$, nous notons Γ_e l'ensemble de ces noeuds et $\Gamma = \bigcup_{e \in E} \Gamma_e$. Puis, nous appliquons un algorithme de *fusion* qui permet de lier les différentes parties instanciées d'un même squelette de Γ (correspondant à différents concepts de la commande) en un seul évènement. Soit Γ^* l'ensemble des parties de l'ensemble Γ.

$$\mathcal{G} = \bigcup_{N \in max_{card} \Gamma^*} fusion(N)$$

Remarquons bien que l'approche descendante ne garantit pas que les évènements soient possibles ou non : elle construit simplement l'ensemble d'évènements qui s'apparie le mieux à la commande. Une conséquence de cette remarque est que $\mathcal{E} = \mathcal{E}_{max}$ et $\mathcal{F} = \mathcal{F}_{max}$.

Cependant, du fait que Γ peut être très grand et que le calcul de $fusion$ est NP-Difficile, nous réduisons Γ par l'utilisation *d'une heuristique de profondeur minimal*e : pour un couple donné $(e, c') \in E \times C_e$, nous ne gardons dans $\Gamma_e$que les noeuds dont la profondeur $depth(c_j)$ est minimale. L'heuristique est basée sur l'interprétation future des évènements (selon la sémantique opérationnelle VDL). Elle n'a aucun impact sur les évènements résultats : l'algorithme construira l'ensemble complet des meilleurs évènements.

4.2 L'approche ascendante

L'approche ascendante (*i.e.* bottom-up) classique utilise une liste *préétablie* de compétences et essaye de relier la commande en LN à une de ces compétences (*e.g.* [8]). Cette approche permet au développeur d'écrire des algorithmes génériques, au sens dépendant uniquement du formalisme de la liste de compétences. Cependant, ces listes sont définies de manière statique, le système n'a aucune conscience de ce qu'il est possible ou non de faire dans l'état courant. En pratique, ces listes doivent décrire toutes les situations de dialogues possibles (en tenant compte des erreurs possibles de l'utilisateur) ainsi que la traduction en requête formelle.

Pour éviter ce problème, nous proposons d'adopter une approche *ascendante constructive* basée sur l'analyse des préconditions. Notre approche utilise les informations contextuelles (obtenues de l'agent à l'exécution) pour déterminer quels évènements peuvent être acceptés par l'agent dans l'état courant. Ainsi, notre système construit la liste des évènements possibles, d'un point de vue agent, sans chercher à utiliser la commande de l'utilisateur.

L'approche ascendante utilise les préconditions d'évènement ($\mathcal{P}_e$) pour fournir les squelettes initiaux (comme l'approche descendante). Puisque notre objectif est de construire les évènements possibles (correspondant à la liste de compétences dans les approches classiques), nous retirons de la liste des squelettes $\mathcal{P}_e$ tout ceux qui sont liés à une action impossible dans l'état

courant (c'est-à-dire pour lesquels une précondition de contexte rend l'exécution de l'action impossible). Nous obtenons alors l'ensemble $\mathcal{P}_{e+c} =$

$$\left\{ e \in \mathcal{P}_e | \forall p \in \bigcup_{r \in R_e(e)} \mathcal{P}_c(r), \Psi(p,e) = \top \right\}$$

avec $\Psi(p,e)$ la fonction booléenne vérifiant si l'évènement e valide la precondition p.$\mathcal{P}_{e+c}$ est l'ensemble des squelettes qui seront forcement acceptés par l'agent dans l'état courant. C'est à partir de cet ensemble que nous allons construire l'ensemble d'événements $\mathcal{G}$.

L'idée de l'approche ascendante constructive est d'utiliser les préconditions structurelles ($\mathcal{P}_s$ et $\mathcal{P}_{cs}$) comme un ensemble de contraintes sur les évènements pour compléter les squelettes. Pour tout $e \in \mathcal{P}_{e+c}$, nous noterons $\tau(e,r) \in \Upsilon$ l'évènement construit à partir du squelette e et de la réaction $r \in R_e(e)$ (en n'analysant donc que $\mathcal{P}_s$ et $\mathcal{P}_{cs}$) par l'utilisation de notre algorithme de génération des cas de test. L'algorithme complet pour la méthode τ est trop long pour être présenté ici. Il repose fortement sur la sémantique opérationnelle du modèle VDL. Il est basé sur une interprétation récursive des termes VDL associé à un certain nombre de règle pour chaque mot-clé du langage.

L'ensemble $\mathcal{G}$ est ainsi calculé :

$$\begin{aligned} \mathcal{G} = \quad & \{\tau(e,r), \forall e \in \mathcal{P}_{e+c}, \forall r \in R_e(e) \\ & | \forall p \in \mathcal{P}_s(r) \cup \mathcal{P}_{cs}(r), \\ & \Psi(p, \tau(e,r)) = \top\} \end{aligned}$$

Remarquons que $\mathcal{G}$ est l'ensemble des évènements possibles : tous les évènements de $\mathcal{G}$ sont possibles pour l'agent et tous les évènements possibles de l'agent sont dans $\mathcal{G}$. Comme conséquence, nous aurons pour le GD que $\mathcal{E} = \mathcal{G}$ et $\mathcal{F} = \emptyset$.

4.3 L'algorithme combiné

L'approche ascendante constructive possède néanmoins une limitation : le système n'est pas capable de comprendre les commandes impossibles (au contraire de l'approche descendante). Contre ce problème, les approches à base de liste de compétences proposent d'utiliser des sous-compétences associées afin de traiter les différentes situations (action possible, impossible, paramètre incomplet, etc.). De manière similaire, nous voudrions que notre algorithme final possède cette capacité sans perdre l'aspect constructif.

Pour cela, nous avons combiné l'algorithme ascendant avec l'idée de l'algorithme descendant de gestion des commandes impossibles. Soit $\mathcal{G}_{bu}$ l'ensemble des évènements possible calculé par l'approche ascendante, notre objectif est de créer un ensemble $\mathcal{G}$ tel que $\mathcal{G}_{bu} \subseteq \mathcal{G}$, la partie supplémentaire représentant les évènements "*actuellement impossible*" (*i.e.* évènements qui ne peuvent pas être accepté par l'agent à l'état courant, mais qui le serait dans un état diffèrent).

Pour cela, nous utilisons le principe de relaxation de contrainte sur les préconditions de contexte (incluant les préconditions contextuelles structurelles). Nous ne pouvons pas relâcher les préconditions de structures, sous peine d'obtenir des évènements mal structuré :

$$\begin{aligned} \mathcal{G} = \quad & \{\tau(e,r), \forall e \in \mathcal{P}_e, \forall r \in R_e(e) \\ & | \forall p \in \mathcal{P}_s(r), \Psi(p, \tau(e,r)) = \top\} \end{aligned}$$

5 Évaluation préliminaire

5.1 Protocole

Notre expérience a été faite avec un agent simple appelé Jojo[6] inspiré du monde

[6] Vous pouvez essayer Jojo sur la page demo : `http://www-poleia.lip6.fr/~sabouret/demos`.

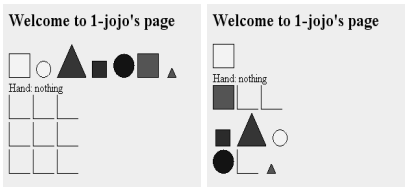

FIG. 2 – État initial et final du protocole

de cube de Winograd [14]. Cet agent possède deux actions : prendre un objet et poser un objet sur une "grille". Un objet est caractérisé par sa forme, sa couleur et sa taille. Une position est un couple de $\{upper, center, lower\} \times \{right, middle, left\}$.

Douze personnes ont fait l'expérience, quatre pour chaque algorithme. Aucune de ces personnes n'avaient utilisées le système auparavant. Ils n'avaient aucune information sur les capacités en TAL du système. L'objectif d'une personne était d'atteindre un état particulier de l'environnement de l'agent (figure 2), sans limitation de temps. Après l'expérience, un questionnaire permettait aux sujets de l'expérience de noter leurs impressions, commentaires libres et de donner une note sur certains critères.

5.2 Résultats principaux

La figure 3 montre la moyenne des temps mis pour atteindre l'objectif et la moyenne des notes donnée par les utilisateurs pour chaque algorithme. Il apparaît clairement que les utilisateurs préfèrent les approches du type ascendante (classique ou combinée). En effet, le retour à l'utilisateur et les propositions faites par l'agent sont les points les plus importants d'après l'évaluation des questionnaires. Ils réduisent d'environ 65% le temps requis pour l'accomplissement de la tache.

Une analyse plus profonde des traces des interactions lors de l'expérience confirme la nécessité pour l'utilisateur de savoir :

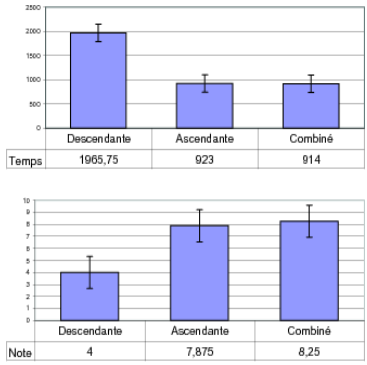

FIG. 3 – Moyennes des temps et des notes

1) ce qu'attend l'agent, ce qu'il ne peut pas faire. C'est ce qui explique les meilleurs résultats de l'approche ascendante sur l'approche descendante. Par exemple, lorsque l'utilisateur dit *"drop on the lower line*", l'approche ascendante propose la liste des cases vide en bas de la grille (sauf lorsqu'il n'en reste qu'une).

2) pourquoi il ne peut pas le faire. C'est ce qui permet, dans l'approche combinée, de corriger l'état de l'agent par rapport au souhait de l'utilisateur. Par exemple : la commande *"Take the red figure*", si la main est déjà pleine, est gérée par une réponse du type *"i can't because the hand is not empty"*. En réponse du retour contextualisé sur l'état de l'agent, les utilisateurs choisissent quasi systématiquement la proposition du système.

6 Conclusion & perspectives

Dans cet article, nous avons proposé un système d'interprétation de commande en LN uniquement paramétré que par le code de l'agent et l'ontologie du domaine. Bien que nous utilisions un langage agent spécifique, notre approche en est indépendante et peut être aisément adaptée à d'autres langages de description d'actions capable d'introspection.

La force de l'approche descendante est de pouvoir expliquer *pourquoi* une commande donnée ne peut être comprise, tandis que celle de l'approche ascendante est d'aider l'utilisateur en lui proposant les différentes actions possibles proche lorsque la commande est mal comprise ou partielle. Notre approche combinée propose les avantages des deux, alliée à notre gestionnaire de dialogue sans règle spécifique dépendante de l'application.

Dans la version qui est présentée ici, notre système ne fonctionne que sur une analyse sémantique minimale sur l'ontologie (synonymie). Par conséquent, il ne peut comprendre que les commandes formulées avec des concepts présents dans le code de l'agent. Notre objectif à terme est d'utiliser un calcul de similarité sémantique entre les concepts [3], afin de donner un score à l'approximation entre les concepts de l'agent et ceux de l'ontologie. Ceci devrait permettre au système de comprendre des commandes plus complexe et moins liées au code brut de l'agent.

Références

[1] S. Abrilian, S. Buisine, C. Rendu, and J.-C. Martin. Specifying Cooperation between Modalities in Lifelike Animated Agents. In *Working notes of the International Workshop on Lifelike Animated Agents : Tools, Functions, and Applications*, pages 3–8, 2002.

[2] T. W. Bickmore. Unspoken rules of spoken interaction. *Commun. ACM*, 47(4) :38–44, 2004.

[3] A. Budanitsky and G. Hirst. Evaluating wordnet-based measures of semantic distance. *Computational Linguistics*, 32(1) :13–47, March 2006.

[4] J. Ferber. *Les systèmes multi-agents : Vers une intelligence collective*. InterEditions, 1995.

[5] P. Maes. Agents that reduce workload and information overload. *Communications of the ACM*, 37(7) :30–40, 1994.

[6] L. Mazuel and N. Sabouret. Generic command interpretation algorithms for conversational agents. In *Proc. Intelligent Agent Technology (IAT'06)*, pages 146–153. IEEE Computer Society, 2006.

[7] D. Milward and M. Beveridge. Ontology-based dialogue systems. In *Proc. 3rd Workshop on Knowledge and reasoning in practical dialogue systems (IJCAI03)*, pages 9–18, August 2003.

[8] E.C. Paraiso, J.P. A. Barthès, and C. A. Tacla. A speech architecture for personal assistants in a knowledge management context. In *Proc. European Conference on AI (ECAI)*, pages 971–972, 2004.

[9] C. Pelachaud. Modelling gaze behaviour for conversational agents. In *Proc. Intelligent Virtual Agent (IVA'2003)*, pages 93–100, 2003.

[10] N. Sabouret and L. Mazuel. Commande en langage naturel d'agents VDL. In *Proc. 1st Workshop sur les Agents Conversationnels Animés (WACA)*, pages 53–62, 2005.

[11] N. Sabouret and J.P. Sansonnet. Automated Answers to Questions about a Running Process. In *Proc. CommonSense 2001*, pages 217–227, 2001.

[12] D. Sadek, Ph. Bretier, and E. Panaget. Artimis : Natural dialogue meets rational agency. In *IJCAI (2)*, pages 1030–1035, 1997.

[13] S. Shapiro. Sneps : a logic for natural language understanding and commonsense reasoning. *Natural language processing and knowledge representation : language for knowledge and knowledge for language*, pages 175–195, 2000.

[14] T. Winograd. *Understanding Natural Language*. New York Academic Press, 1972.