Generic command interpretation algorithms for conversational agents

Laurent MAZUEL and Nicolas SABOURET
Laboratoire d'Informatique de Paris6 (LIP6),
104 av du Président Kennedy, 75016 Paris, France
{laurent.mazuel, nicolas.sabouret}@lip6.fr

AbstractThis paper focuses on human-machine communication with intelligent agents, it proposes a generic architecture with an algorithm for natural language (NL) command interpretation which makes it easy to define different applications using the description and domains of the different agents, since all that is required is their respective codes and domain ontologies. There are two classical approaches for NL command interpretation: the top-down approach, which relies on the syntactical constraints of the agent's model, and the bottom-up approach which relies on the set of the agent's possible actions. The present work combines the two in a new bottom-up based algorithm that makes use of agent's constraints. the three algorithms are then compared, and results show that the combined approach gives best results.

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

1. Introduction

Fore a long time, the Multi-Agent System (MAS) community has focused on distributed problem solving through autonomous cognitive agents [1]. One of the core issues in MAS research is the study of agent interaction models and protocols [2] at the formal level. Although little work had been done on human-agent communication, recent work on the use of MAS for ambient intelligence [3] and semantic web services [4] in the context of mixed human-agent communities has rekindled interest in this question.

Two communities working closely with the agent world have chosen to examine this issue. The embodied conversational agents (ECA) [5] community has focused on multi-modal interaction, social behavior [6] and expression of emotions [7]. However, their approaches rely mainly on ad-hoc pattern matching without semantic analysis [8]. The dialogue system community, on the other hand, proposes to use ontologies to improve genericity [9,10]. The main idea behind the use of ontologies is to specify generic algorithms that only depend on the ontology formalism. Thus, applications only depend on the ontology and the specific application problem-solver. Systems like [9,11] use their ontology to parameterize a generic parser. However, in such systems, the ontology formalism itself is ad-hoc,

depends very much on the application type and does not allow generic knowledge representation (*e.g.* concept/relation ontologies). Moreover, these ontologies describe the application model as well as the application actions. In other words, the application is written in a specific non-semantic language which will be executed and in a semantic form in the ontologies. Other systems use generic knowledge representation (*e.g.* [10]) and rely on application-dependent parsers. However, the parser uses the structure of the ontology to understand over-specified or under-specified commands like "switch the light on" (the system will propose the different possible locations to be lit).

We suggest that it should be possible to extract the meaning of actions from the code itself (without having to describe both a semantic form and an executable form for any given action). The ontology is therefore no longer an application descriptor but simply provides the complementary semantic information on relations between the application concepts (which is the initial role of ontologies).

This paper focuses on NL command interpretation for intelligent agents. The aim is to show that it is possible to define a generic NL system based upon a domain ontology and agents capable of introspection. The system extracts the set of possible actions from the agent's code and matches these actions with the user's command using the ontology as a glue. In addition, a score-based dialogue manager (such as [12]) deals with misunderstood or indefinite commands.

Whereas the architecture relies on a specific agent's model that allows code introspection, the Natural Language Processing (NLP) modules (for interpretation of terms, feedback, dialogue and clarification) can be generalized, in order to depend only on domain ontology and the description of the agent. In other words, NLP algorithms remain the same for all the agents in the domain, and it is their ontologies that make the link between the algorithms, the agent's code, the NL commands and the user's questions.

The paper is organized as follows. The second section provides a general overview of our agent model. Section 3 presents the NLP tool chain and the Dialogue Manager, which are the parts of the NLP architecture that the three algorithms have in common. Section 4 introduces three different generic algorithms for NL command interpretation: a top-down one, a bottom-up one and a combination of the two. Section 5 illustrates the three algorithms. Section 6 offers a preliminary evaluation of the three algorithms, section 7 presents related work and section 8 suggests future lines of research.

2. Overview

Our aim is to be able to program cognitive agents that can be controlled by natural language commands, and that are capable of reasoning about their own actions, so as to answer questions about their behavior and their current action. To do so, we rely upon a specific language in order to access *at runtime* the description of the agent's internal state and actions.

Figure 1. VDL Embodied Conversational Agents based on the LEA model. The system asks "Could you get me the red torch?"

2.1. The VDL model

Our agents are programmed using the *View Design Language* (VDL)[1]. The VDL model is based on XML tree rewriting: the agent's description is an XML tree whose nodes represent either data or actions. The agent rewrites the tree at every execution step according to these specific elements. This model allows agents to access *at runtime* the description of their actions and to reason about it for planning, formal question answering [13], behavior recognition [14] for example. The VDL agent model can also be used for web services composition [15], Embodied Conversational Agents [16], social behavior simulation, etc.

In the VDL model, every agent is provided with a domain ontology written in OWL [17]. This ontology must contain all the concepts used by the agent (*i.e.* the VDL concepts), either as XML tags (except for VDL keywords), attribute names and values or CDATA contents. We note C_{VDL} the set of VDL concepts and C_{OWL} the set of OWL concepts. We define an *injective* map map_{VDL} defined on the set C_{VDL} and taking values from the set C_{OWL} in order to match VDL concepts on the ontology.

2.2. Actions in VDL

It is possible to identify two kinds of behavior for an autonomous agent provided with interaction capabilities [1]:

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

- Reactive behavior corresponds to actions which are performed *only* when the agent receives a command (a typical example is a start/stop operation).
- Proactive behavior is the ability of the agent to run independently of any command.

Since this paper focuses on human-machine interaction, only work on *reactions* will be considered. In VDL, reactions are triggered by *external events*, *i.e.* XML nodes sent to the agent at runtime for command. These nodes are the formal representation of commands. External events correspond to the content of "request" ACL messages [2] whereas reactions describe how such messages (sent by other agents or by the user) must be processed. The aim of the NLP system presented in this paper is to build VDL events from a user's command. No previous work has been done previously on NL interaction with a VDL agent and the only way to communicate directly with a VDL agent was to use its specific XML-based formalism.

Message processing in MAS protocols can be broken down into two stages. In the first stage, a parser checks the syntax of the message (the message could be rejected). It ensures that the reaction will be able to process the event and switches it to the correct reaction. In the second stage, the reaction processes the event itself according to the agent's internal state and the definition of the reaction (*i.e.* behavior). It must extract the relevant information (*i.e.* parameters expected by the reaction) from the event and then carry out modifications. However, these modifications will be performed only if the current agent's *context* (internal state) allows it.

In VDL, as in most action representation models, actions are defined as a tuple $< N, P, E >$ where N is the action name, P is the set of preconditions of the action and E its effects. The parser and context verification must be implemented within the agent, using *preconditions*. Based on the previous definitions, we characterize four kinds of preconditions for a reaction r in $\mathcal{R}$, the set of agent reactions:

- $\mathcal{P}_e(r)$ is the set of event preconditions. They are used to ensure that a given action is triggered by a given class of events. Their interpretation relies on *subsumption* for checking the structure of the received event.
- $\mathcal{P}_s(r)$ is the set of structure preconditions. It is used to check the message's syntax and to ensure that the action will be able to process the event. Preconditions in $\mathcal{P}_s(r)$ do not depend on the agent's internal state, but only on the received event.
- $\mathcal{P}_c(r)$ is the set of context preconditions. Such preconditions only depend on the agent's internal state. For example, a (simulated) robot cannot move when it runs out of energy.
- $\mathcal{P}_{cs}(r)$ is the set of contextual structure preconditions, *i.e.* preconditions that depend both on events (selected by $\mathcal{P}_e$) and on the agent's internal state. For example, a robot cannot catch an object when this object is out of reach.

We note $\mathcal{P}_e = \cup_{r \in \mathcal{R}} \mathcal{P}_e(r)$. For all $e \in \mathcal{P}_e$, we note $R_e(e) = \{r \in \mathcal{R} | e \in \mathcal{P}_e(r)\}$ the set of reactions that process the event e.

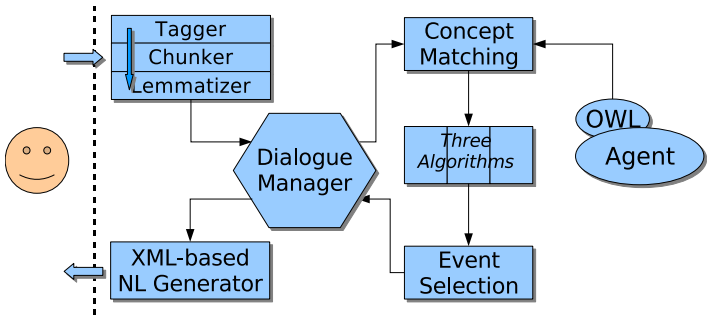

Figure 2. General architecture

3. Global architecture

This section presents the NL modules which are common to the three algorithms and that are used within our NL command interpretation architecture.

3.1. NLP toolchain

In our project (see figure 2), the lexical module is based on the default OpenNLP[2] tokenizer, tagger and chunker. The WordNet lemmatizer provided by OpenNLP is also used and allows the system to detect concepts represented by more than one word (*e.g.* "*dark red*", *"extra large*").[3] As anticipated in [18], the use of a grammar-based syntactic parser is not relevant for NL commands, since users often choose keywords to command the system rather than well-structured sentences (*e.g.* "*drop object low*" or "*take blue*"). For this reason, recent work proposes to compute a logical form of the command [18,19], using a system of logical rules rather than a grammar-based parser. This kind of approach is more robust in real applications, since the user's sentences are often grammatically defective. We have also chosen a logical representation of the command, based on an analysis of the relation's ontology, in order to detect the functional relations in the user command [20]. This logical form is mostly useful when interpreting complex relations (*e.g. "the biggest triangle*") or symmetrical commands (*e.g.* "*from London to Boston*" and "*from Boston to London*" which have the same bag of words). However, this type of representation has no impact on the three algorithms evaluated in this paper. Moreover, the technical study of this model is not the objective of this paper. Hence, in order to read the algorithms more easily, we will suppose that the user's sentence is a "bag of words" (*i.e.* a set of words).

Before getting to the semantic interpretation, the first important point is to convert the bag of words S into a set C of equivalent concepts in the ontology that can be understood by the agent, precisely because classical dialogue systems rely on the hypothesis of *semantic connectivity* [21]:

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

[3] Currently, we do not use WordNet for semantic interpretation. We give a rapid overview of our future project with WordNet in section 8.

> "the semantic content of the utterance corresponds to a path in a semantic network describing the application domain knowledge."

In other words, all the words used in a command must have an equivalent concept in the ontology. To respect this hypothesis, we take into account a simple semantic distance measure on the ontology. For $(c_1, c_2) \in C^2_{OWL}$, we define the distance between c_1 and c_2 w.r.t. synonymy as:

$$dist_{ONT}(c_1, c_2) = \begin{cases} 0 \text{ if } c_1 = c_2 \text{ or } c_1 \text{ owl:sameAs } c_2 \\ 1 \text{ else} \end{cases}$$

where *owl:sameAs* is the transitive and reflexive relation for concept synonymy in OWL. This formula simply uses synonymy to link the user's command to VDL concepts. In this way, the $dist_{ONT}$ operator enriches the Sadek hypothesis, which can be rewritten as:

> Every concept that appears within a relevant command is either directly associated to a VDL concept or is in an OWL *sameAs* relation with an agent's concept.

We have defined in [22] a new semantic distance measure for an ontology based on a generalization of the distance measure of Jiang & Conrath [23]. Our measure takes into account all possible relations in the ontology and not only the subsumption *same-as* relation. Once again, this semantic measure has no impact on the three algorithms we want to evaluate. However, the evaluation of the system highlights the lack of semantic interpretation of the commands, which make the system unable to understand complex command like *"take the smallest triangle"* or "*drop it in place of the red form*". This result was expected since we only use the *owl:sameAs* relation for "semantic" relations. Since semantic distances measures are beyond the scope of this paper, work currently in progress on them is not presented here.

Now we can build the set C of *known concepts*, the bag of concepts that represent the user's sentence, as follows:

$$C = \{v \in C_{VDL} | \exists s \in S'. dist_{ONT}(map_{VDL}(v), s) = 0\}$$

where map_{VDL} is the injective map between the VDL concepts and the ontology (see section 2.1) and C contains the set of all VDL concepts that appear in the user's command. Note that the construction of C is only a preliminary step for the algorithm described in Section 4, which shows how the set C of known concepts is used to build VDL events using the three possible NL algorithms.

The last part of our chain is an English NL generator that transforms any VDL node (*i.e.* a VDL precondition or a VDL formal command) into an English sentence.[4] Since we want our system to depend only on the agent VDL code and the ontology, this English generator is very important. It is managed by the Dialogue Manager presented in the following section. The English generator algorithm appends the translation of concepts obtained by a depth-first search of

[4] Unlike the classical template approach in which each dialogue state of the system must be rewritten for each new application.

the node. Generally, this recursive algorithm prints the node tag, its attributes as "attribute is value", its content (if any) and then all its sub-elements. For instance, the formal command:

```
<take position="out"> <shape>square</shape> </take>
```

will become "*take position is out shape is square*".

Moreover, when we have to translate a VDL precondition into English (for example to explain why a problem has occurred), we must pay attention to the VDL keywords inside. Each VDL keyword is associated to a specific translation rule (but is application-independent). For example, the precondition:

```
<equals><event-get/><size>small</size></equals>
```

with the VDL keyword *equals* (all sub-nodes must be equal) and the VDL keyword *event-get* (representing a content in an event) will be translated as: *"the content of the event must be equal to size is small"*.

From a syntactical point of view the resulting output is very poor, but in our experiments (section 6) it was sufficiently clear for the users to understand the system's proposal or explanations. However, it is possible to improve it significantly by using an XML-based NL generator like [24].

3.2. User's feedback: the Dialogue Manager

The Dialogue Manager (DM) is responsible for both command acknowledgment and management of misunderstood or imprecise commands. The DM will produce different answers depending on the different contextual situations. The input of the DM is computed by one of the event creation algorithms described in detail in section 4 (top-down, bottom-up or combined). The three algorithms have the same output, in order to be compatible with the DM input. This output is a set $\mathcal{G}$ of external events. The DM first partitions this set into two subsets:

- The set $\mathcal{E}$ of possible events in the current agent's context;
- The set $\mathcal{F}$ of impossible events, *i.e.* events that correspond to the user's NL command but that cannot be processed (at least at the current time).

Let $\mathcal{P}(r) = \mathcal{P}_c(r) \cup \mathcal{P}_s(r) \cup \mathcal{P}_{cs}(r)$., which is the set of preconditions that are used to compute whether a given reaction will be able to process a given event. We note Υ the (infinite) set of all possible VDL nodes. Let $eval : \Upsilon^2 \rightarrow \{\top, \bot\}$ be the precondition evaluation function: $\forall p \in \mathcal{P}(r), eval(p, evt) = \top$ iff the precondition p is valid with respect to the event evt and the current agent's state.

The sets $\mathcal{E}$ and $\mathcal{F}$ are computed as follows:

$$\mathcal{E} = \{evt \in \mathcal{G} | \exists r \in R_e(evt), \forall p \in \mathcal{P}(r), eval(p, evt) = \top\}$$

$$\mathcal{F} = \mathcal{G} \backslash \mathcal{E}$$

Note that $\mathcal{E}$ and $\mathcal{F}$ do not need to contain *all* possible or impossible events. They only contain the proposals made by our NL algorithms.

In addition, for all $evt \in \mathcal{F}$, we note $np(evt) \in \bigcup_{r \in R_e(evt)} \mathcal{P}(r)$ the set of preconditions which prevents reactions from processing this event. Since $\mathcal{F}$ is a set of impossible events, $np(evt) \neq \emptyset$. The set $np(evt)$ is computed as follows:

$$np(evt) = \bigcup_{r \in R_e(evt)} \{p \in \mathcal{P}(r) | eval(p, evt) = \bot\}$$

The DM computes the matching degree of events in $\mathcal{E}$ and $\mathcal{F}$ with respect to the user's command. For every node $n \in \Upsilon$ and for any recognized concept $c \in C$, we note $contains(n, c) = \exists x \in sub(n) | c \in \{tag(x), attributes(x), content(x)\}$, where $sub(n)$ is the set of all direct and indirect sub-elements of $n \in \Upsilon$. In other words, $contains(n, c)$ is true *iff* c appears anywhere within node n. The matching degree of $e \in \mathcal{E}$ or $e \in \mathcal{F}$ is:

$$p(e) = \frac{card(\{c \in C | contains(e, c)\})}{card(C)}$$

We build $\mathcal{E}_{max}$ and $\mathcal{F}_{max}$ the best matching events. Let $p_{\mathcal{E}} = max(\{p(e), e \in \mathcal{E}\})$ and $p_{\mathcal{F}} = max(\{p(e), e \in \mathcal{F}\})$. Then

$$\mathcal{E}_{max} = \begin{cases} \emptyset & if\ p_{\mathcal{E}} = 0 \\ \{e \in \mathcal{E} | p(e) = p_{\mathcal{E}}\} & otherwise \end{cases}$$

and the same for $\mathcal{F}_{max}$. $p_{\mathcal{E}}$ and $p_{\mathcal{F}}$ are the matching degree of elements in $\mathcal{E}_{max}$ and $\mathcal{F}_{max}$. In other words, the DM only takes into consideration possible and impossible events that best match the user's command.

We define two thresholds for our algorithms. p_{min} is the minimum value for an event to be considered as a possibly understood command and p_{max} is the further limit beyond which the event is considered as a correct representation of the user's command. They correspond respectively to the "tell me" and "do it" thresholds for Patty Maes in [12]. She proposed empirically to use a margin for accepting events: $p_{min} = 0.3$ and $p_{max} = 0.8$.

The answer given by the DM depends on the position of $p_{\mathcal{E}}$ and $p_{\mathcal{F}}$ with respect to p_{min} and p_{max}:

1. If $p_{\mathcal{E}} \geq p_{max}$, the command is considered as correctly understood by the system. The DM either sends the event to the agent (when $|\mathcal{E}_{max}| = 1$) or informs the user about an ambiguity (when $|\mathcal{E}_{max}| > 1$). For instance[5]:

   U: Take something red.
   S: I can either "take object shape is square color is red" or "take object shape is triangle color is red".

2. If $p_{min} < p_{\mathcal{E}} < p_{max}$ and $p_{\mathcal{E}} < p_{\mathcal{F}}$, the best understood event is not possible ($p_{\mathcal{F}} > p_{\mathcal{E}}$) but something close was understood which is still possible ($p_{\mathcal{E}} > p_{min}$). The DM asks the user for a reformulation. It displays both the failed preconditions of impossible events ($\cup_{evt \in \mathcal{F}_{max}} np(evt)$) and the list of possible events $\mathcal{E}_{max}$.

   U: Put it in the upper left cell *(with upper left cell already occupied).*
   S: I can't because: the content of upper left cell is not empty. Therefore, I can either: - drop object in the upper middle, in the upper right, in the center left or in the lower left.

[5] All examples are taken from the evaluation in section 6.

3. If $p_{min} < p_{\mathcal{E}} < p_{max}$ and $p_{\mathcal{F}} \leq p_{\mathcal{E}}$, the impossible events can be ignored, but still the system is not sure about the user's command ($p_{\mathcal{E}} < p_{max}$). It asks for a confirmation by displaying events $e \in \mathcal{E}_{max}$.

   U: Take the blue or red triangle shape *(but there is no blue triangle).*
   S: Do you mean "take object shape is triangle color is red"?

4. If $p_{\mathcal{E}} \leq p_{min}$ and $p_{max} \leq p_{\mathcal{F}}$, the system correctly understood an impossible command. It tells the user that this command is not possible by giving the list of failed preconditions $np(evt), evt \in \mathcal{F}_{max}$.

   U: Take the blue object *(with something already in hand).*
   S: I can't because: the content of hand is not empty.

5. If $p_{\mathcal{E}} \leq p_{min}$ and $p_{min} < p_{\mathcal{F}} < p_{max}$, the system might have understood something but this command cannot be performed. The DM asks the user for confirmation. The result will be the same as for (3) except that the command is impossible. After a possible confirmation from the user, the agent will assert that the event is not possible.

   U: Take the blue or red triangle shape *(but there is no blue triangle and with something already in hand).*
   S: Do you mean "take object shape is triangle color is red"?
   U: Yes.
   S: I can't because : the content of hand is not empty.

6. If $p_{\mathcal{E}} \leq p_{min}$ and $p_{\mathcal{F}} \leq p_{min}$, the system didn't understand the command and tells the user.

4. NL command interpretation algorithms

This section presents the algorithms that build the set of VDL events $\mathcal{G}$ from natural language commands. This set is passed to the Dialogue Manager. As observed by Allen [25], two methods can be considered for processing NL commands: top-down and bottom-up. We first propose generic implementations of these two methods, *i.e.* as algorithms that only depend on the ontology and the operational semantics of our agent's programming language, and then suggest a combined algorithm that integrates the top-down idea of building impossible events within a bottom-up approach.

4.1. The top-down approach

The top-down approach consists in *building* the formal command from the NL command, considering only the structural constraints imposed by the formal model (*e.g.* [21,19]). It is mostly used when the system cannot ensure that the command would be possible at the current time. The main drawback of the top-down approach lies in the difficulty of keeping generic rules in the system. Without application-specific rules on the syntactic structure of events, the system can produce "impossible events", *i.e.* events that will not be accepted by the system because of $\mathcal{P}_e$, $\mathcal{P}_c$ or $\mathcal{P}_s$. For instance, Shapiro [19] proposes to use "impossible events" to compute such *ad-hoc* preconditions for inclusion within the agent's

knowledge base. In most other systems, the impossible events are not considered and the only answer sent to the user is something like "I don't understand your command". On the contrary, our agent model VDL makes use of its introspection ability to compute explanations as to why a given event could not be processed[6]. These explanations are produced by analyzing of the failed preconditions for a given event (the $np(e)$ function defined earlier in section 3.2). This allows us to define a top-down implementation with generic rules without having to take impossible events into account.

Consequently, our implementation of the top-down approach only uses subsumption preconditions ($\mathcal{P}_e$) and the agent's internal state to build the VDL event from known concepts (C). Subsumption preconditions allow us to define the skeleton of the possible events that correspond to a given concept. A deeper analysis of the agent's internal XML code allows the system to enrich this event. Rather than using strict grammar rules (like in [26]), we propose to define an event construction method based upon the VDL syntax and to apply heuristics to constrain the construction with regard to the VDL operational semantics.

Let $E = \{e \in \mathcal{P}_e | tag(e) \in C\}$ and $\forall e \in E$, let $C_e = C \backslash \{tag(e)\}$. For every $e \in E$, our algorithm considers the set of leaves L_e of e and searches in the agent's code for nodes whose tag $t \in L_e$ and that contain at least one concept $c' \in C_e$ in their sub-elements. $\forall e \in E$, we note Γ_e the set of these nodes and $\Gamma = \bigcup_{e \in E} \Gamma_e$. We then apply a *merge* algorithm that merges several possible event skeletons in Γ (corresponding to different concepts in the user's command) into one single event. The algorithm builds the maximum common subsumed node.

For every set of nodes N, let $merge(N) = max_{\preceq} \{x \in \Upsilon | \forall n \in N, x \preceq n\}$ where Υ is the (infinite) set of all possible VDL nodes. The idea of the top-down algorithm for computing $\mathcal{G}$ is to merge the largest possible number of subsets of Γ, *i.e.* the events that best match the user's command. Let Γ^* be the power set of Γ then:

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

Note that the top-down algorithm does not consider whether events are possible or not: it simply builds the set of best matching events and passes it to the DM. As a consequence, for the DM, $\mathcal{E} = \mathcal{E}_{max}$ and $\mathcal{F} = \mathcal{F}_{max}$.

However, because the set Γ is possibly very large, and because computing *merge* is NP-hard, we reduce Γ using a *minimal depth heuristic*: for a given couple $(e, c') \in E \times C_e$, we only keep in Γ_e the node with the minimal $depth(c_j)$. This heuristic is based on the forthcoming interpretation of events (according to VDL's operational semantics). It is a minimal heuristic to decrease recall, improve precision and only suppresses events that have been built incorrectly in terms of VDL operational semantics. It is possible that incorrect events remain, but the remaining set still has the *complete* set of best matching events and correct events.

[6]For instance: *"Take the red* object" when the action is not possible (due to a non-empty hand) leads the system to tell the user that the hand was not empty.

4.2. The bottom-up approach

The classical bottom-up approach makes use of a *previously defined* list of competences and tries to match the natural language command to one of the possible formal commands (*e.g.* [11,9]). This allows programmers to write "generic" NL algorithms that only depend on this list of competences. However, since this list is defined statically, the system has no knowledge about what is or is not possible in the current state. Concretely, the list of competences has to contain all possible dialogues (even problem cases) and their translation into formal commands (possibly with parameters). To make matters worse, there is often a big overlap between this huge amount of information and the agent's rules that have been defined for the problem solving itself.

To avoid this problem, recent work has considered a *dynamic bottom-up approach* (*e.g.* [27]). In these systems, only a minimal set of competences is defined in advance. If an NL command does not exactly match a competence, the system tries to generate a new one with the correct specification using parts of the other competences. We propose to go further and to adopt a *constructive bottom-up approach* based on the analysis of preconditions. In this way, we do not have to describe an initial competences list, since the primary actions described in the agent are enough to build a competence *dynamically*. Our approach uses contextual information (obtained from the agent's code at runtime) to determine which events can be processed by the agent in the current state. This problem has been widely studied for software validation (*e.g.* [28]) and has given interesting results for testbed generation. It can be adapted to event generation and NL command processing. Preconditions can be seen as tests that are applied to messages, which means that our system builds the list of possible events from the agent's point of view, without worrying whether or not any of them matches the user's command.

The bottom-up algorithm uses event preconditions ($\mathcal{P}_e$) to provide the initial event skeletons. Since our aim is to compute a set of possible events, we first remove from $\mathcal{P}_e$ those events that cannot be processed by the agent due to the current agent's context:

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

$\mathcal{P}_{e+c}$ is the set of event skeletons that are accepted by the agent with respect to context preconditions ($\mathcal{P}_c$). Note that $\mathcal{P}_{e+c} \subseteq \mathcal{P}_e$.

The idea of the bottom-up approach is to use structure and contextual structure preconditions ($\mathcal{P}_s$ and $\mathcal{P}_{cs}$) as a set of constraints on the events to refine event skeletons into actual events. For all $e \in \mathcal{P}_e$, we note $refine(e, r) \in \Upsilon$ the event obtained from the skeleton e and the set of preconditions $\mathcal{P}_s(r) \cup \mathcal{P}_{cs}(r)$ of the reaction $r \in R_e(e)$ using our testbed generation-based algorithm. The complete algorithm for $refine$, which is too long to be presented here, relies heavily on the VDL model's operational semantics. It is based on a recursive interpretation of VDL terms with different rules for each VDL keyword (see figure 3). In this example, *event, ctx-str, exist* and *get* are VDL keywords. The <*event*><*take*/></*event*> node indicates that the root of the XML command must be <*take*/>. The precon-

Part of an agent's code:

```
<a1>
  <b>7</b>
  <c>8</c>
</a1>
<a2>
  <b>9</b>
  <c>10</c>
</a2>
```

Two preconditions:

```
<event><take/></event>
<ctx-str>
  <exist>
    <get><event-get><take/></event-get></get>
    <b/>
  </exist>
</ctx-str >
```

Two events generated by these preconditions:

```
<take><a1/></take>
<take><a2/></take>
```

Figure 3. An example of event generation

dition can be translated into English as "the node *take* in the event (*i.e.* <*event-get*><*take/*><*/event-get*>) must contain a node 1) with a child *b* and 2) that exists in the VDL code". Since there are two nodes in the code which validate this precondition, the event-generation algorithm can build two events (respectively for *a1* and *a2*).

The set of events $\mathcal{G}$ is then computed by:

$$\mathcal{G} = \{refine(e,r), \forall e \in \mathcal{P}_{e+c}, \forall r \in R_e(e) \\ |\forall p \in \mathcal{P}_s(r) \cup \mathcal{P}_{cs}(r), eval(p, refine(e,r)) = \top\}$$

Note that $\mathcal{G}$ is the set of possible events[7]: all events in $\mathcal{G}$ are accepted by the agent and all accepted events belong to $\mathcal{G}$. Thus, $\mathcal{E} = \mathcal{G}$ and $\mathcal{F} = \emptyset$.

4.3. A combined algorithm

One limitation of the constructive bottom-up approach is that our system will not be able to understand user commands that correspond to impossible events (whereas the top-down algorithm can build such impossible events). The classical competence-list based approach also encounters this problem. This is generally

[7] The same bottom-up generation algorithm is used by our forward-chaining planner in VDL agents.

solved by defining a set of special competences which are triggered by the incorrect message and return a particular template-based answer for each incorrect state. Unfortunately, this reinforces the disadvantages of the competence-list approach. On the contrary, we want to keep the idea of event generation, but with the ability to generate impossible events to match a possible incorrect command from the user. Therefore, we would like our algorithms to be able to tell the user that a given command is correct but not possible in the current state.

In order to do this, we propose to combine the bottom-up approach with the top-down idea that it is possible to build events that are incompatible with the agent's current context. Let $\mathcal{G}_{bu}$ be the set of possible events as computed by the bottom-up approach. The idea of our combined approach is to enrich $\mathcal{G}_{bu}$ with the set of "*currently impossible*" events such that $\mathcal{G}_{bu} \subseteq \mathcal{G}$:

> *Currently impossible events* are events that are not acceptable by the agent in its current state but that would be accepted in a different state.

This notion of *currently impossible events* is the key aspect of our combined algorithm. It relies on the idea that some preconditions ($\mathcal{P}_c$ and $\mathcal{P}_{cs}$) are heavily dependent on the agent's current context. For instance, let us consider a robot that can perform a given action iff *it has enough energy*. This context precondition (having enough energy) can be true at some point during runtime and false at another. Similarly, if the robot has to keep close to an object so as to be able to pick it up, the context-structure precondition *I am next to object X* (X being a parameter of the event/command) depends, for a given X, on the current state. For the same object X, it can be true at one point and false at another. For example, let us take the event "take object A" for an agent that is far from A. This event is *currently impossible*. In a different state, the context-structure precondition would be evaluated as true and the event would be possible.

More formally, *currently impossible events* are events associated with $\mathcal{P}_c$ or $\mathcal{P}_{cs}$ that fail though they would succeed in a different state. In order to build such *currently impossible events*, we simply use constraint relaxation on context preconditions and contextual structure preconditions ($\mathcal{P}_c$ and $\mathcal{P}_{cs}$) when generating the set of events $\mathcal{G}$. To relax the $\mathcal{P}_c$precondition, we do not use the $\mathcal{P}_{e+c}$ defined in section 4.2 but $\mathcal{P}_e$ directly. This allows us to build a set of events from skeletons of actions that are contextually currently impossible, but that would be possible in another state. The $\mathcal{P}_{cs}$ precondition describes a parameter of the agent's context that can be used in an event (*e.g.* in the command *"buy me a ticket for the Pink Floyd show"*, "*Pink Floyd*" is dependent on the agent's context). Using the $\mathcal{P}_{cs}$ preconditions only when building the candidate events but not when testing them increases the number of potential events.

Finally, the formula to compute the set $\mathcal{G}$ is very close to that of the bottom-up approach, but with fewer constraints. The set of events $\mathcal{G}$ is computed by:

$$\mathcal{G} = \{refine(e,r), \forall e \in \mathcal{P}_e, \forall r \in R_e(e) \\ |\forall p \in \mathcal{P}_s(r), eval(p, refine(e,r)) = \top\}$$

Note that our algorithm cannot guarantee that this constraint relaxation choice will not introduce too many false events which would mean that the system proposes too many impossible events. However, we suggest that it will improve

the bottom-up algorithm result, as the evaluation presented in the next section will show.

5. A complete example

This section presents a complete example of natural language processing using our architecture. It is illustrated using the combined algorithm approach, since it is the best algorithm in the evaluation, and also allows us to present briefly the results of the event generation algorithm.

5.1. The "jojo" agent

The example here is that of the agent used for the evaluation in section 6. The agent is a simple agent called Jojo[8] inspired by Winograd's block's world [29]. This agent has two possible actions: to take an object or to drop it into a given position in a three-by-three "grid". An object is characterized by its shape ($shape \in \{square, triangle, circle\}$), its color ($color \in \{red, green, blue, white\}$) and its size ($size \in \{tiny, small, medium, big\}$). A position is a couple in $\{upper, center, lower\} \times \{right, middle, left\}$. For example, a formal event using this VDL formalism is:

```
<take><object>
  <shape>square</shape>
  <color>red</color>
  <size>medium</size>
</object></take>
```

5.2. Building of the set of recognized concepts

Let the agent "Jojo" be in the state of figure 4. The agent has a white circle in its hand. There are two objects in the grid, a blue circle in the lower left cell and a red square in the center. We consider the following command:

"hi jojo, release the circle on the lower line!"

The first step is to tag, chunk and lemmatize the command. The result of this first step is the model:

[NP hi:PRP] [ADVP jojo:RB] [?? ,:,] [VP release:VB] [NP the:DT circle:NN] [PP on:IN] [NP the:DT lower:JJR line:NN]

Next, the system must compute the set C of recognized concepts, by searching for words in the ontology which can be linked to the set of all VDL concepts. The result is the set:

C = {[release VDL:drop], [circle VDL:circle], [lower VDL:lower] [line VDL:line]}

[8] You can try Jojo on our demo page: `http://www-poleia.lip6.fr/~sabouret/demos`. The examples in the Dialogue Manager's algorithm come from the experimental corpus.

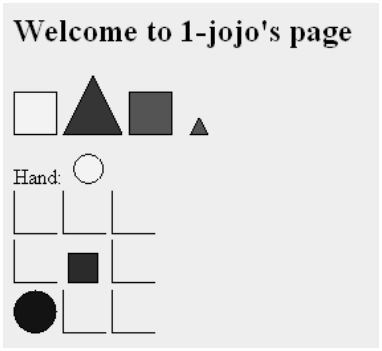

Figure 4. An initial state of our agent "Jojo".

In Jojo's ontology, the verb "release" is declared as equivalent to the verb "drop" using the *sameAs* relation. Therefore the semantic distance defined in section 3.1 returns 1 to $dist_{ONT}(drop, release)$. The final "bag of words" that we will consider is the set:

$$C = \{drop, circle, lower, line\}$$

5.3. Event generation

Seven events are generated for the "take an object" action, one for each object present in the agent's world. An event "take an object" is like:

```
<take><object>
  <shape>square</shape>
  <size>medium</size>
  <color>white</color>
</object></take>
```

Ten events are generated for the "drop to a cell" action, one for each cell in the grid (9 events) and one corresponding to outside of the grid. An event "drop to a cell" is like:

```
<drop><position>
  <line name=''upper''/>
  <row name=''left''/>
</position></drop>
```

Finally, the set $\mathcal{G}$ of generated events contains 17 events, which are not all possible at the current state.

5.4. The Dialogue Manager's response

Firstly, the Dialogue Manager has to separate the set $\mathcal{G}$ into the set $\mathcal{E}$ of possible and the set $\mathcal{F}$ of impossible events. In the current state, the possible events are only drop events, corresponding to a cell in the grid or outside of the grid. Hence, the set $\mathcal{E}$ contains 8 events:

1. Seven events for the seven free cells in the grid, one "drop to a cell" action for each cell.
2. One event for outside of the grid.

Therefore, the set $\mathcal{F}$ contains 9 events:

1. Two events for the two cells in the grid which already have an object.
2. Seven events for the seven objects which the agent is unable to take, because it already has an object in its hand.

The next step of the Dialogue Manager is to compute the $p_{\mathcal{E}}$ and $p_{\mathcal{F}}$ relevance score, which means computing the maximum relevance sets $\mathcal{E}_{max}$ and $\mathcal{F}_{max}$. These scores correspond to the maximum relevance score between the sets of concepts C and $\mathcal{E}$ (resp. $\mathcal{F}$). The two events of $\mathcal{E}$ which maximize the score, the two events in $\mathcal{E}_{max}$, are:

```
<drop><position>              <drop><position>
  <line name=''lower''/>        <line name=''lower''/>
  <row name=''center''/>        <row name=''right''/>
</position></drop>            </position></drop>
```

because the only two free cells in the lower line are the center cell and the right cell. This means that the score $p_{\mathcal{E}}$ is:

$$p_{\mathcal{E}} = \frac{|\{drop, lower, line\}|}{|\{drop, circle, lower, line\}|} = 3/4$$

Similarly, there is only one event which maximizes the score in the set $\mathcal{F}$ to build the set $\mathcal{F}_{max}$:

```
<drop><position>
  <line name=''lower''/>
  <row name=''left''/>
</position></drop>
```

because the lower left cell is already taken by an object. Thus, we have $p_{\mathcal{F}} = 3/4$.

Now we have the parameters $p_{\mathcal{E}} = 3/4$, $p_{\mathcal{F}} = 3/4$, $|\mathcal{E}_{max}| = 2$ and $|\mathcal{F}_{max}| = 1$. The thresholds used by the Dialogue Manager are $p_{min} = 0.3$ and $p_{max} = 0.8$. This corresponds to policy number 3: $p_{min} < p_{\mathcal{E}} < p_{max}$ and $p_{\mathcal{F}} \leq p_{\mathcal{E}}$. This strategy ignores the set of impossible events $\mathcal{F}$ and asks the user for clarification and confirmation:

```
Your command is imprecise, please give information.
 I can either:
 - drop position line name is lower row is center
 - drop position line name is lower row is right
```

Note that the algorithm has skipped the last event for the lower line because it was dynamically marked to be impossible.

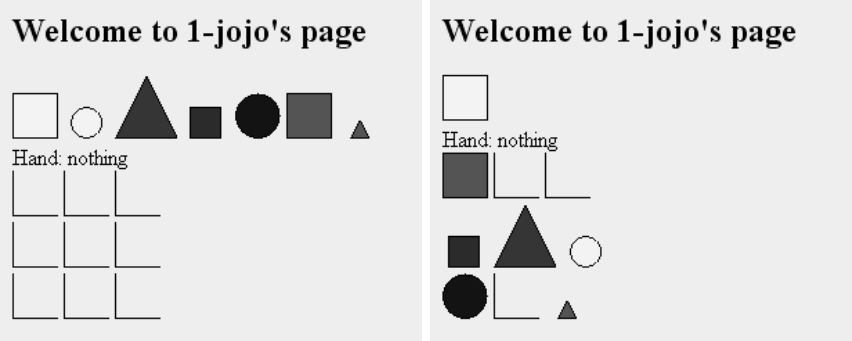

Figure 5. Initial state and goal of the experiment

6. Preliminary evaluation

The top-down algorithm computes the best matching event from the user command, without considering if the resulting events are possible or not. The bottom-up algorithm computes the set of all possible events in the current agent's context. The combined approach computes the set of possible or "*currently impossible*" events, using a bottom-up algorithm. The DM filters the result so as to separate possible and impossible events and to determine which events best match the user's command.

In order to evaluate the algorithms, we chose a single task and made three identical tests (one for each algorithm) using the same protocol. The first section presents the protocol in detail, the second the overall results.

6.1. Protocol

Our experiment was conducted using the simple agent called Jojo presented in section 5.1. We had twelve subjects for this experiment, four for each algorithm. None of them had ever used the system before. They were given no information about the system's NLP capabilities. The aim was to reach a given particular state (see Figure 5), with no time limitation. Subjects were afterwards asked to fill out a questionnaire on what they thought about the system's NLP capabilities.

We expected that:

1. the top-down approach would understand the user's command better since it builds the best-matching possible formal representation and
2. it would provide explanations for impossible commands;
3. the bottom-up approach would lead to easier user interaction since it proposes commands to the user;
4. the combined algorithm offers the advantages of the first two.

6.2. Overall results

Figure 6 shows the average time necessary to complete the task and the average score awarded by users to the system, for all three algorithms. It clearly shows that users prefer the bottom-up approaches (classical or combined) to the top-

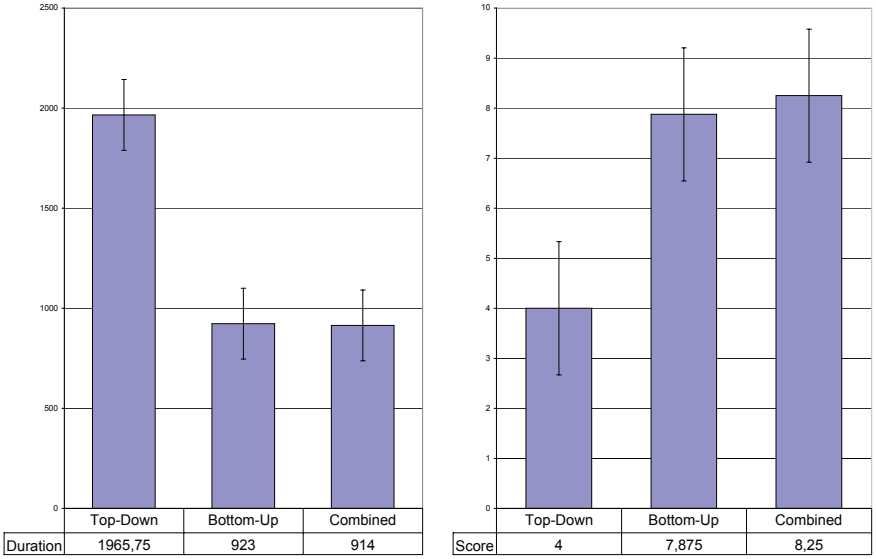

Figure 6. User evaluation and task duration

down one, since the feedback to the user and the agent's proposals are seen as very important from the user's point of view and reduce by more than 65% the time required for the same task.

A deeper analysis of the questionnaires confirms the usefulness of good feedback about what the agent expects from the user. For instance, when the user asks: "*Drop on the lower line*", the bottom-up approaches propose all empty lower cells. From a mixed-initiative planning perspective, the user knows exactly what the system expects. Moreover, if only one empty cell exists, the system drops directly into the correct cell. On the contrary, explanations provided by the system from top-down generated events are often confusing, as for instance: "*I can't drop in the upper left cell because it is not a cell*". The reason for these failures is that the top-down algorithm does not analyze the structure of the events it tries to generate. In the previous example, "*Drop in the upper left cell*" leads to a failure because "*cell*" is recognized as a concept whereas it should not be added to the event's structure.

What makes the combined approach significantly better than the bottom-up one (from the user's point of view) is that it retains the top-down capability to provide explanations about what is not possible. For instance: *"Take the red object*" when the action is not possible (due to a non-empty hand) leads the system to tell the user that the hand was not empty. Similarly, "*Drop upper left*" when the hand is empty or the cell already occupied also leads to a good explanation. Using such feedback on the agent's context, users often switched to possible commands. In addition, the difference between the bottom-up algorithm and the combined one should increase when in less intuitive contexts than the block's world: users will not be able to guess correct formulations without any explanation about impossible actions. However, we have not tested this thoroughly yet.

7. Related work

To our knowledge, there is no work on NL interaction which proposes an evaluation of the bottom-up and top-down approaches, thus, making it impossible to compare the importance of our results with similar experiments. However, we will try to present the difference that exists between our ontology-based generic NL architecture and the other NL interaction systems. This will be done by looking at the two major weaknesses presented in the introduction: the separation of the knowledge representation from the NL interpretation algorithms and the specific problem-solver language adapted for semantic interpretation and introspection.

7.1. Knowledge representation

Currently, everyone agrees that it is necessary to separate the knowledge representation from the NL algorithms [9,10,11,30], as this improves the capacity to instantiate an NL-core algorithm to a new system with a new domain. However, the borderline between the two and the choice of the content formalism for the knowledge representation are very fuzzy. We suggest that it is possible to use a generic ontology formalism which represents only the domain relation and the concepts. The semantic interpretation must be based on a semantic similarity measure in order to make interpretation possible. We have defined a new complex measure based on recent advances in this domain for our system in [22]. Similarly, Flycht-Eriksson [31] uses a generic ontology formalism with different kinds of relations. However, the NL algorithms are domain-specific, so this system does not use a similarity measure. Milward's system [10], which uses a separate and generic ontology, makes inferences to understand overspecified or underspecified commands. However, the ontology is a hierarchy of concepts and is in fact an application of the very simple Rada measure [32] which is not efficient for complex semantic interpretation [33]. Another example, SmartKom [30], aims at defining a multimodal embodied conversational agent with speech recognition[9]. They use a separate ontology [34] for disambiguation and they have defined a contextual score, called OntoScore [35]. The ontology is defined in OIL. However, the contextual score is only used to disambiguate the set of proposals made by the speech recognizer and not for the semantic interpretation. The semantic interpretation is handcrafted in a specific language called M3L which takes into account multimodal communication (speech and gestures).

Many systems do not use a generic ontology formalism but a specific one. This causes many problems when shifting to a new domain, because the specific formalism is very hard to master and difficult to use. For example, SnepS [36] is a dialogue system architecture to command a robot in the real world based on a top-down approach[10]. This system defines its own knowledge representation called SnepS-Log based on semantic networks and logical operators compatible with NL semantics [19]. A command is modeled in a logical form using a set of Sneps-Log rules from the application. The approach is very similar to that of Artimis [21] in which the logical model is an extended version of the BDI logic.

[9] `http://www.smartkom.org`

[10] `http://www.cse.buffalo.edu/sneps/`

They also use predefined rules to build the representation of the user's command. Compared to our generic formalism approach, the logic formalism is specific to their systems and very difficult to use. For example, SnepS-Log defines a set of new operators with a very specific semantics. Similarly, BDI programming requires some experience.

Another example is the TRIPS system [37] that uses an ontology to describe the parameters of their generic parser [9]. There are two ontologies (LF and KR), one for the syntactic structure and the other for the domain information. The LF ontology is built to be "as general as possible, [a] relatively flat structure and linguistically motivated" from an extended version of the FRAMENET ontology [38] while the KR ontology is "domain-specific concept, [with] roles organized for efficient reasoning". In practise, the LF ontology defines too much application-specific information which is also defined in the problem solver. Moreover, the linguistic equivalences are handcrafted and this is a very costly process. As for, the KR ontology, it is a set of patterns used to trigger the correct action: a pattern algorithm is not a semantic interpretation and requires many rules to obtain efficient results. Another example is the work of Paraiso & Barthes. They have defined an architecture to implement conversational agents [11]. They use a domain ontology and WordNet for disambiguation with direct synonymy recognition (using the synset definition). However, the domain ontology is a very strict formalism which describes the action and the keywords needed to trigger it. Moreover, the way they use WordNet does not take into account recent work on semantic similarity and complex requests cannot be understood. Last but not least, the CEDERIC system [27] also separates its ontology. It has a semantic similarity function for semantic interpretation, but unlike our system, the ontology is defined by using a specific formalism, the KM language. This language describes the actions, the domain knowledge, the discourse model and the lexicon all in the same model, which means that the ontology is more application specific than a domain-application ontology. For example, a new application working on the same knowledge model requires rewriting many parts of this ontology. Moreover, each entry in the ontology is linked to a handcrafted weight which is used for the semantic similarity function. This similarity function is ad-hoc for the interpretation algorithm and cannot be extended to another formalism.

7.2. Problem-solver language and natural language interpretation

Another major aspect in these systems is the problem-solving specification. Everyone agrees (as in the matter of the role of ontologies), that to improve the genericity of a system, a specific language must be used to describe the problem solver. There are two kinds of approaches. The classical approach describes the problem solver first in the real execution code and then, once again, in a specific semantic language which represents the functionalities [9,19]. The more recent approaches try to define a language that is *at one and the same time* the code to be executed and the semantics of its code [30,27,11]. This reduces drastically the time needed to write a new architecture and it is this second approach that we have chosen. We use the VDL language which is both an action description language [39] and a semantic language [40], (*i.e.* a language capable of introspec-

tion). Similarly, the CEDERIC system [27] defines an action description model with preconditions and effects (as does our agent model VDL). They have defined a few set of plans and, in the case of complex commands, the system is able to modify or combine many plans to construct a new one. This is, to our knowledge, the only system which proposes a dynamic bottom-up approach. However, as we pointed out in the previous section, the KM language describes, all in the same model, the action, the domain knowledge, the discourse model and the lexicon.

Most of the current systems make use of the first approach. One example is COLLAGEN [41], a new language especially defined to program user assistants. This language allows the developer to describe all the discourse plans (a *recipe*) between the agent and a user. Thus, it is possible to define many different agents that will share the same plan recognition algorithm. However, this language defines the interaction plan and not the problem-solver code, which has to be written separately. Similarly, the KR ontology of TRIPS [9] defines the interaction pattern with the user and branches into a hardcode problem-solver.

8. Conclusion & future lines of research

This paper has proposed a generic NL command interpretation system within the framework of the intelligent agent technology. Our algorithms can be parameterized by the agent's code and the domain ontology. The system is integrated in the VDL multi-agent platform and can be used for human-agent communication in semantic web services composition [15]. However, even if we use the VDL language for programming agents, our approach is not dependent on the language and can easily be adapted to other programming languages, provided they are capable of introspection (*i.e.* one can generate the set of possible and impossible events).

Our current system uses the "combined" algorithm with a logical syntactic representation of the user's command [20] (instead of the "bag of words" C) and a semantic similarity measure [22] based on [23] (instead of the simple *owl:sameAs* relation). We are currently evaluating the impact of these extensions on natural language interpretation.

To match the user concepts to the ontology, the Sadek hypothesis is not efficient, since the ontologies do not define all the synonyms and definitions for each word. To improve this, one solution is to use work on disambiguation with WordNet [33] to *anchor* all the terms in the command to the ontology, *i.e.* to match a *concept* from the ontology for each user concept. In addition, the semantic similarity we want to use allows us to find the closest (semantically speaking) application concept to the anchoring of a user concept. Our measure relies on the assumption that an ontology contains many other relations than *is-a*, *part-of* or *inverse-of*. This information can be used to determine the relatedness degree of two concepts, in order to build the best set of application concepts representing the command.

References

[1] Ferber, J.: Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)

[2] : The FIPA ACL Message Structure Specifications. http://www.fipa.org/specs/fipa00061/ (2002)

[3] Gárate, A., Herrasti, N., López, A.: GENIO: an ambient intelligence application in home automation and entertainment environment. In: Proc. of the 2005 joint conference on Smart objects and ambient intelligence (sOc-EUSAI '05), ACM Press (2005) 241–245

[4] Paurobally, S., Tamma, V., Wooldridge, M.: Cooperation and Agreement between Semantic Web Services. In: W3C Workshop on Frameworks for Semantics in Web Services. (2005)

[5] Cassel, J., Sullivan, J., Prevost, S., Churchill, E.: Embodied Conversational Agents. MIT Press (2000)

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

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

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

[9] Dzikovska, M.O., Allen, J.F., Swift, M.D.: Integrating linguistic and domain knowledge for spoken dialogue systems in multiple domains. In: Proc. of IJCAI-03 Workshop on Knowledge and Reasoning in Practical Dialogue Systems. (August 2003)

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

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

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

[13] Sabouret, N., Sansonnet, J.: Automated Answers to Questions about a Running Process. In: Proc. CommonSense 2001. (2001) 217–227

[14] Sabouret, N., Sansonnet, J.: Learning Collective Behavior from Local Interactions. In Dunin-Keplicz, B., Nawarecki, E., eds.: From Theory to Practice in Multi-Agent Systems, Proc. CEEMAS 2001. Volume 2296 of LNAI-LNCS. Springer-Verlag (2002) 273–282

[15] Charif-Djebbar, Y., Sabouret, N.: An Agent Interaction Protocol for Ambient Intelligence. In: Proc. of the 2nd International Conference on Intelligent Environments (IE'06). (2006) 275–284

[16] Sabouret, N., Golsenne, M., Martin, J.: VDL+LEA: complémentarité entre interaction multi-modale et interaction multi-agents. In: Proc. 1st Workshop sur les Agents Conversationnels Animés (WACA). (2005) 13–22

[17] Smith, M.K., Welty, C., McGuinness, D.L.: Owl web ontology language guide. http://www.w3.org/TR/owl-guide/ (February 2004)

[18] Milward, D.: Distributing representation for robust interpretation of dialogue utterances. In: ACL. (2000) 133–141

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

[20] Mazuel, L.: Utilisation des ontologies pour la modélisation logique d'une commande en langue naturel. In: Rencontre des Étudiants Chercheurs en Informatique pour le Traitement Automatique des Langues (RECITAL 2007). (6 2007)

[21] Sadek, D., Bretier, P., Panaget, E.: Artimis: Natural dialogue meets rational agency. In: IJCAI (2). (1997) 1030–1035

[22] Mazuel, L., Sabouret, N.: Degré de relation sémantique dans une ontologie pour la commande en langue naturelle. In: Plate-Forme AFIA, Ingénierie des Connaissances 2007 (IC 2007). (7 2007)

[23] Jiang, J., Conrath, D.: Semantic similarity based on corpus statistics and lexical taxonomy. In: Proc. on International Conference on Research in Computational Linguistics, Taiwan (1997) 19–33
[24] Bateman, J.A.: Enabling technology for multilingual natural language generation: the kpml development environment. Nat. Lang. Eng. **3**(1) (1997) 15–55
[25] Allen, J., Miller, B.W., Ringger, E.K., Sikorski, T.: A robust system for natural spoken dialogue. In: ACL. (1996) 62–70
[26] Seneff, S.: Tina: A natural language system for spoken language applications. Computational Linguistics **18**(1) (1992) 61–86
[27] Eliasson, K.: Case-Based Techniques Used for Dialogue Understanding and Planning in a Human-Robot Dialogue System. In: Proc. of IJCAI07. (2007) 1600–1605
[28] Botella, B., Taillibert, P., Gotlieb, A.: Utilisation des contraintes pour la génération automatique de cas de test structurels. In: Test de logiciel. Volume 21. RSTI-TSI (2002) 1163–1187
[29] Winograd, T.: Understanding Natural Language. New York Academic Press (1972)
[30] Wahlster, W.: SmartKom: Symmetric Multimodality in an Adaptive and Reusable Dialogue Shell. Proc. of the Human Computer Interaction Status Conference (2003) 47–62
[31] Flycht-Eriksson, A.: Design of Ontologies for Dialogue Interaction and Information Extraction. In: Proc. Workshop on Knowledge and reasoning in practical dialogue systems (IJCAI'03). (2003)
[32] Rada, R., Mili, H., Bicknell, E., Blettner, M.: Development and Application of a Metric on Semantic Nets. IEEE Transactions on Systems, Man, and Cybernetics **19**(1) (1989) 17–30
[33] Budanitsky, A., Hirst, G.: Evaluating wordnet-based measures of semantic distance. Computational Linguistics **32**(1) (March 2006) 13–47
[34] Gurevych, I., Porzel, R., Slinko, E., Pfleger, N., Alexandersson, J., Merten, S.: Less is more: using a single knowledge representation in dialogue systems. In: Proceedings of the HLT-NAACL 2003 workshop on Text meaning, Morristown, NJ, USA, Association for Computational Linguistics (2003) 14–21
[35] Porzel, R., Gurevych, I., Muller, C.: Ontology-based contextual coherence scoring. In: Proc. of the Fourth SIGdial Workshop on Discourse and Dialogue, Sapporo, Japan (July 2003)
[36] Shapiro, S., Rapaport, W.: The SNePS Family. Computers & Mathematics with Applications **23**(2-5) (1992) 243–275
[37] Ferguson, G., Allen, J.: Trips: An integrated intelligent problem-solving assistant. In: Proc. AAAI/IAAI. (1998) 567–572
[38] Johnson, C., Fillmore, C.: The FrameNet tagset for frame-semantic and syntactic coding of predicate-argument structure. In: Proc. ANLP-NAACL. (2000) 56–62
[39] Sabouret, N.: Active Semantic Web Services: A programming model for agents in the semantic web. In: Proc. EUMAS. (2003)
[40] Sabouret, N.: A model of requests about actions for active components in the semantic web. In: Proc. STAIRS 2002. (2002) 11–20
[41] Rich, C., Sidner, C.L., Lesh, N.: Collagen: Applying collaborative discourse theory to human-computer interaction. AI Magazine **22**(4) (2001) 15–26