Generic command interpretation algorithms for conversational agents

Laurent Mazuel and Nicolas Sabouret
Laboratoire d’informatique de Paris 6 - LIP6, France
{laurent.mazuel, nicolas.sabouret}@lip6.fr

Abstract

This paper focuses on the human-machine communication within the framework of intelligent agents. We propose a generic architecture provided with a natural language (NL) algorithm for command interpretation that can be adapted to different agent’s domains. Our NL architecture only depends on the agent’s code and its domain ontology. We consider two classical approaches for NL command interpretation: the top-down approach, which relies on the agent’s model syntactical constraints, and the bottom-up approach which relies on the set of the agent’s possible actions. 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.

1 Introduction

1.1 Presentation of the problem

The Multi-Agent System (MAS) community has long been focused on distributed problem solving through autonomous cognitive agents [11]. One of the core issues in MAS research is the study of agent interaction models and protocols [1] at the formal level. However, few works has been done on human-agent communication. Nevertheless, recent work on the use of MAS for ambient intelligence [12] and semantic web services [17] in the context of mixed human-agent communities raised new interests for this problematic.

Two communities close to the agent world took a close look to this issue. The embodied conversational agents (ECA) [8] community on the one hand focused on multimodal interaction, social behavior [5] and emotion expression [18]. However, the chosen approaches mostly rely on ad-hoc pattern matching without semantic analysis [2]. The dialogue system community, on the other hand, proposes to use ontologies to improve genericity [10, 15]. The main idea behind the use of ontologies is to specify generic algorithm that only depends on the ontology formalism. Thus, applications only depend on the ontology and the specific application problem-solver. Systems like [10, 16] use the ontology to parameterize a generic parser. However, in such systems, the ontology formalism itself is ad-hoc. It strongly depends on the application type and does not allow generic knowledge representation. Moreover, these ontologies describe the application model as well as the application actions. Our claim is that it should be possible to extract the meaning of actions from the code itself. The ontology then is no longer an application descriptor. It only provides the complementary semantic information on relations between the application concepts (which is the initial role of ontologies). Moreover, systems that use generic knowledge representation (e.g. [15]) rely on application-dependant parsers. However, the parser use 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 enlighten).

1.2 Plan of the paper

This paper focuses on the command interpretation for intelligent agents. We try 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 (like [13]) deals with misunderstood or indefinite commands.

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

Our paper is organised as follows. In the second section, we give a general overview of our agent model. Sec-
Figure 1. VDL Embodied Conversational Agents based on the LEA model

tion 3 presents the NLP tool chain and the dialogue manager, which are the part of the NLP architecture that are in common with the three algorithms. In section 4, we introduce three different generic algorithms for NL command interpretation: a top-down one, a bottom-up one and a combination of both. Section 5 presents a preliminary evaluation of these algorithms. Section 6 concludes the paper.

2 Overview of our model

Our aim is to be able to programme 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 behaviour and their activity. To this purpose, we rely upon a specific language that allows to access at runtime to the description of the agent’s internal state and actions.

2.1 The VDL model

Our agents are programmed using the View Design Language (VDL) language\(^1\). 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 to the description of their actions and to reason about it for planning, formal question answering [20], behaviour recognition [21], etc. The VDL agent model can be used for web services composition [9], Embodied Conversational Agents [19], social behaviour simulation...

In the VDL model, every agent is provided with a domain ontology written in OWL [25]. This ontology must contains at least all the concepts used by the agent (i.e. the VDL concepts), either as XML tags (except for VDL keywords), attributes names and values or CDATA contents. We note \(C_{VDL}\) the set of the VDL concepts and \(C_{OWL}\) the set of the OWL concepts, individuals and properties. We define a injective map \(\text{map}_{VDL}\) defined on the set \(C_{VDL}\) and taking values in the set \(C_{OWL}\), to match VDL concepts on the ontology.

2.2 Actions in VDL

We can exhibit two kinds of behaviour for an autonomous agent provided with interaction capabilities [11]:

- The reactive behaviour is used when the agent performs operations in answer to a command (a typical example is a start/stop operation).
- The proactive behaviour is the capability for the agent to run independently from any command.

In this paper, as we focus on the human-machine interaction, we will only work on reactions. In VDL, reactions are triggered by external events, i.e. XML nodes sent to the agent at runtime for command. They are the formal representation of commands. External events correspond to the content of “request” ACL messages [1] whereas reactions describe how such messages (send 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.

Message processing in MAS protocols can be decomposed into two stages. In the first stage, a parser checks the message’s syntax (eventually, the message could be rejected). It ensures that the reaction will be able to process the event and it switches the event to the correct reaction. In the second stage, the reaction processes the event itself according to the agent’s internal state and reaction’s definition (i.e. behaviour). It must extract relevant information (i.e. parameters expected by the reaction) from the event and then perform 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 characterise four kinds of preconditions for a reaction \(r\) in \(R\), the set of agent reactions :

- \(P_e(r)\) 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 set. 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.

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

3 Global architecture

This section presents the common NL modules 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. Additionally we make use of a homemade lemmatizer\(^3\). As anticipated in [14], the use of a grammar based syntactic parser is not relevant for NL commands. In fact, users often command the system with keywords rather than well-structured sentences (e.g. “drop object low” or “take blue”). Thus, we represent the content of sentences as bags of words, after removed stop-words identified by their tags\(^4\).

The semantic analysis is the core of our model. Our aim is to use ontologies for concepts matching. In the current stage, we simply use synonymy for linking the user’s command with VDL concepts. For $(c_1, c_2) \in C_{OWL}^2$, we define the distance between $c_1$ and $c_2$ w.r.t. synonymy as:

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

with owl:sameAs being the transitive and reflexive relation for concept synonymy in OWL.

\(^2\)http://opennlp.sourceforge.net/

\(^3\)We did not select the OpenNLP lemmatizer for lightweight reasons. Similarly, we preferred OpenNLP to the TreeTagger (http://www.ims.uni-stuttgart.de/projekte/corplex/TreeTagger) for compatibility reasons with our Java online version.

\(^4\)This representation clearly leads us to lose important information. We discuss this point in section 5.

Our semantic analysis method relies on the hypothesis of semantic connectivity [22]: every concept that appears within a relevant command must be defined in the ontology. This hypothesis is enriched with our $dist_{syn}$ operator:

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

More precisely, if we note $S$ the bag of word that represent the user sentence, we can build the set $C$ of known concepts as follows:

$$C = \{ v \in C_{VDL} | \exists s \in S.dist_{syn}(\text{map}_{VDL}(v), s) = 0 \}$$

$C$ contains the set of all VDL concepts that appear within the user’s command. Similarly, we build the set $U$ of not-understood concepts:

$$U = \{ s \in S | \forall v \in C_{VDL}.dist_{syn}(\text{map}_{VDL}(v), s) = 1 \}$$

$U$ contains the set of command words that do not appear within the agent’s description. Note that the construction of $C$ and $U$ is only a preliminary step for the algorithm described in Section 4.

The last part of our chain is an English NL generator that transforms any VDL node into an English sentence, by appending the translation of concepts obtained by a depth-first search of the node. Generally, this recursive algorithm prints the node’s tag, its attributes as “attribute is value”, its content (if any) and then all its sub-elements. For instance:

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

will become “take position is out shape is square”. Moreover, we use specific rules for VDL keywords.

The resulting outputs are awful from a syntactical point of view, but it is sufficient for users to understand the system’s proposal or explanations. However, it is possible to improve it significantly using XML-based NL generator like [4] that do not require any template.

Section 4 shows how the set $C$ of known concepts is used to build VDL events using three possible NL algorithms.

![Figure 2. General architecture](image-url)
3.2 User’s feedback: the dialogue manager

The dialogue manager (DM) is responsible for both command acknowledgement 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 algorithm described in details in section 4 (the top-down, bottom-up or combined algorithms). The three algorithms have the same output, to be compliant with the DM input.

Our DM takes a set $G$ of external events proposed by semantic analysis algorithms (presented in section 4). It first partitions this set into two subsets:

- The set $E$ of possible events in the current agent’s context;
- The set $F$ of impossible events, i.e. events that correspond to the user’s NL command but that are not processable at least at the current time.

To this purpose, let $P(r) = P_e(r) \cup P_s(r) \cup P_{cs}(r)$. It is the set of preconditions that are used to compute whether a given reaction will be able to process a given event. We note $\Upsilon$ the (infinite) set of all possible VDL nodes. Let $eval : \Upsilon \rightarrow \{\top, \bot\}$ the precondition evaluation function:

$$\forall p \in P(r), eval(p, e) = \top \iff \text{the precondition } p \text{ is valid with respect to the event } e \text{ and the current agent’s state}.$$ 

The sets $E$ and $F$ are computed as follows:

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

$$F = G \setminus E$$

Note that $E$ and $F$ do not need to contain all possible or impossible events. They only contain the propositions made by our NL algorithms.

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

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

The DM computes the matching degree of events in $E$ and $F$ with respect to the user’s command. For every node $n \in \Upsilon$ and for any recognised concept $c \in C$, we note $contains(n, c) = \exists x \in sub(n) | c \in \{tag(x), attributes(x), content(x)\}$, with $sub(n)$ 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 $e \in E$ or $e \in F$ is:

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

We build $E_{\text{max}}$ and $F_{\text{max}}$ the best matching events. Let $p_e = \max\{\{p(e), e \in E\}\}$ and $p_F = \max\{\{p(e), e \in F\}\}$. Then

$$E_{\text{max}} = \begin{cases} 0 & \text{if } p_e = 0 \\ \{ e \in E | p(e) = p_e \} & \text{otherwise} \end{cases}$$

and same for $F_{\text{max}}$. $p_e$ and $p_F$ are the matching degree of elements in $E_{\text{max}}$ and $F_{\text{max}}$. In other words, the DM only takes into consideration possible and impossible events that best match the user’s command. This has no impact on feedback built by the DM.

We define two thresholds for our algorithms. $p_{\text{min}}$ is the minimum value for an event to be considered as possibly understood command and $p_{\text{max}}$ 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 [13]. She proposed empirically to use a margin for accepting events: $p_{\text{min}} = 0.3$ and $p_{\text{max}} = 0.8$.

The answer given by the DM depends on the position of $p_e$ and $p_F$ with respect to $p_{\text{min}}$ and $p_{\text{max}}$:

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

   **U:** Take something red
   
   **S:** I can either take object square red, take object triangle red.

2. If $p_{\text{min}} < p_e < p_{\text{max}}$ and $p_F < p_e$, the best understood event is not possible ($p_F > p_e$) but something close was understood which is still possible ($p_F > p_{\text{min}}$). The DM asks the user for a reformulation. It displays both failed preconditions of impossible events ($\bigcup_{e \in F_{\text{max}}} np(e)$) and the list of possible events $E_{\text{max}}$.

   **U:** Put it on 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

3. If $p_{\text{min}} < p_e < p_{\text{max}}$ and $p_F < p_e$, the impossible events can be ignored, but still the system is not sure about the user’s command ($p_F < p_{\text{max}}$). It asks for a confirmation by displaying events $e \in E_{\text{max}}$.

   **U:** Take the blue or red triangle form (while there is no blue triangle)
4. If \( p_E \leq p_{\text{min}} \) and \( p_{\text{max}} \leq p_F \), the system correctly understood an impossible command. It tells the user that this command is not possible by giving the list of failed preconditions \( np(e), e \in F_{\text{max}} \).

U: Take the blue red figure (with something already in hand)

S: I can’t because: the content of hand is not empty.

5. If \( p_E \leq p_{\text{min}} \) and \( p_{\text{min}} < p_F < p_{\text{max}} \), the system might have understood something but this command cannot be performed. The DM asks the user for confirmation.

6. If \( p_E \leq p_{\text{min}} \) and \( p_F \leq p_{\text{min}} \), the system didn’t understand the command and tells it to the user.

4 NL command processing algorithms

This section presents the algorithms that build the set of VDL events \( G \) from natural language commands. This set is passed to the dialogue manager. As stated by Allen [3], two methods can be considered for processing NL commands: the top-down and the bottom-up approach. We first propose implementations of these two methods in a generic manner, i.e. as algorithms that only depend on the ontology and the operational semantic of our agent’s programming language. We finally give 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. [22, 24]). 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 relies in the difficulty to keep 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 agent due to \( P_e \), \( P_c \) or \( P_s \). The introspection capability of the VDL model can be used to compute explanations on why a given event could not be processed\(^5\). However, production of these explanations requires additional and ad-hoc structure preconditions in \( P_s \) that are not necessary for the programme itself. Shapiro [24] proposes to use “impossible events” to compute such ad-hoc preconditions for inclusion within the agent’s knowledge base.

Our implementation of this approach only uses subsumption preconditions (\( P_s \)) and the agent’s internal state for building the VDL event from know 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 [23]), 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 \( E = \{ e \in P_e | \text{tag}(e) \in C \} \) and \( \forall e \in E \), let \( C_e = C \setminus \{ \text{tag}(e) \} \). For every \( e \in E \), our algorithm considers the set of leaves \( L_e \) of \( e \) and searches within 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 \( \Gamma_e \) the set of these nodes and \( \Gamma = \bigcup_{e \in E} \Gamma_e \). We then apply a fusion algorithm that merges several possible event skeletons in \( \Gamma \) (corresponding to different concepts in the user’s command) into one single event. It builds the maximum common subsumed node.

For every set of nodes \( N \), let \( \text{fusion}(N) = \max_{\subseteq} \{ x \in T | \forall n \in N, x \leq n \} \). The idea of the top-down algorithm for computing \( G \) is to build the fusions of the largest possible subsets of \( \Gamma \), i.e. the events that best match the user’s command. Let \( \Gamma^* \) be the power set of \( \Gamma \).

\[
G = \bigcup_{N \in \max_{\subseteq} \Gamma^*} \text{fusion}(N)
\]

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, \( E = E_{\text{max}} \) and \( F = F_{\text{max}} \).

However, because the set \( \Gamma \) is possibly very large, and because computing \( \text{fusion} \) is NP-hard, we reduce \( \Gamma \) using a minimal depth heuristic: for a given couple \((e, e') \in E \times C_e \), we only keep in \( \Gamma_e \) the node with the minimal \( \text{depth}(c_j) \). This heuristic is based on the forthcoming interpretation of events (according to VDL’s operational semantics). It has no impact on the resulting events: the algorithm still builds the complete set of best matching events.

4.2 The bottom-up approach

The classical bottom-up approach makes use of a early defined list of competences and tries to match the natural language command onto one of the possible formal command (e.g. [16, 10]). This allows programmers to write “generic” NL algorithms that only depend on this competences list. However, since the competences list is defined statically, the system has no knowledge on what is possible

\(^5\)For instance: “Take the red figure” 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.
or not in the current state. Concretely, the competences list has to forecast all possible dialogues (even problem cases) and their translation into formal commands (possibly with parameters). Moreover, this huge amount of information is mostly redundant with the agent’s rules defined for the problem solving itself.

To avoid this issue, we propose to adopt a constructive bottom-up approach based on preconditions analysis. 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 issue has been widely studied for software validation (e.g. [6]) and showed interesting results for testbeds generation. This can be adapted to event generation if we consider that preconditions are used to test messages. Thereby our system builds the list of possible events from the agent’s point of view, without concern about whether any of those matches the user’s command.

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

$$
P_{e+} = \left\{ e \in P_e | \forall p \in \bigcup_{r \in R_e(e)} P_c(r), \text{eval}(p, e) = T \right\}
$$

$P_{e+}$ is the set of event skeletons that are accepted by the agent with respect to context preconditions ($P_c$). Note that $P_{e+} \subseteq P_e$.

The idea of the bottom-up approach is to use structure and contextual structure preconditions ($P_s$ and $P_{cs}$) as a set of constraints on the events to refine event skeletons into actual events. For all $e \in P_e$, we note $\text{refine}(e, r) \in \Upsilon$ the event obtained from the skeleton $e$ and the set of preconditions $P_s(r) \cup P_{cs}(r)$ of the reaction $r \in R_e(e)$ using our test-bed generation based algorithm. The complete algorithm for $\text{refine}$ is too long to be presented here. It strongly relies on the VDL model’s operational semantics. It is based on a recursive interpretation of VDL terms with different rules for each VDL keyword.

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

$$\mathcal{G} = \{ \text{refine}(e, r), \forall e \in P_{e+}, \forall r \in R_e(e) \mid \forall p \in P_s(r) \cup P_{cs}(r), \text{eval}(p) = T \}$$

Note that $\mathcal{G}$ is the set of possible events\(^6\): all events in $\mathcal{G}$ are accepted by the agent and all accepted events belong to $\mathcal{G}$.

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). However, competence-list-based approaches often separate a competence (associated to actions) into two or more sub-competences, so as to deal with the different situations (the action is possible, it is not, it is only partially possible for a given reason...). Similarly, 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.

To this purpose, we propose to combine the bottom-up approach with the top-down idea that it is possible to build events 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 $\mathcal{G}$ with the set $\mathcal{G}_{bu}$ of “currently impossible” events:

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.

To this purpose, we use constraints relaxation on context preconditions and contextual structure preconditions ($P_c$ and $P_{cs}$):

$$\mathcal{G}_{bu} = \{ \text{refine}(e, r), \forall e \in P_e, \forall r \in R_e(e) \mid \forall p \in P_s(r), \text{eval}(p) = T \}$$

The set of events passed to the dialogue manager is:

$$\mathcal{G} = \mathcal{G}_{bu} \cup \mathcal{G}_{bu}$$

5 Preliminary evaluation

In this section, we evaluate our three algorithms for building events from natural language commands. 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.

5.1 Protocol

Our experiment was conducted using a simple agent called Jojo\(^7\) inspired from Winograd’s block’s world [26].

\(^6\)The same bottom-up generation algorithm is used by our forward-chaining planner in VDL agents.

\(^7\)You can try Jojo on our demo page: http://www-poleia.lip6.fr/~sabouret/demos. Examples in the dialogue manager’s algorithm come from this experiment’s corpus.
This agent has two possible actions: to take an object or to drop it into a given position in a “grid”. An object is characterised by its shape ($\text{shape} \in \{\text{square}, \text{triangle}, \text{circle}\}$), its colour ($\text{color} \in \{\text{red}, \text{green}, \text{blue}, \text{white}\}$) and its size ($\text{size} \in \{\text{tiny}, \text{small}, \text{medium}, \text{big}\}$). A position is a couple in $\{\text{upper}, \text{center}, \text{lower}\} \times \{\text{right}, \text{middle}, \text{left}\}$.

We had twelve subjects for this experiment, four on each algorithm. None of them had ever used the system before. They were given no information on the system’s NLP capabilities. The aim was to reach a given particular state (see Figure 3), without time limitation. Subjects were afterwards submitted a questionnaire on their appreciations on the system’s NLP capabilities.

Our expectations were that 1) the top-down approach should better understand the user’s command since it builds the best-matching possible formal representation; 2) it could provide explanations on impossible commands; 3) the bottom-up approach should lead to easier user interaction since it proposes commands to the user and 4) the combined algorithm takes the advantage of both.

5.2 Overall results

Figure 4 shows the average time for task achievement and the average score given 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-down one. Indeed, the feedback to the user and the agent propositions are seen as very important from the user’s point of view. Objectively, it reduces (by more than 65%) the required time for the same task.

A deeper analysis of filled questionnaires confirms the relevance of a clear 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 in the correct cell. On the contrary, explanations provided by the system from top-down generated events are often confusing. 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 make any analysis on 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 recognised 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 is that it keeps the top-down capability to provide explanations about what is not possible. For instance: “Take the red figure” 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 leads to a correct explanation. Using such feed-backs on the agent’s context, users often switched to possible commands. Also note that the difference between the bottom-up and the combined algorithms should increase when in less intuitive contexts than the block’s world: users will not be able to guess correct formulations without any explanation on impossible actions. However, we have not tested this thoroughly yet.

Moreover, the evaluation of the system outlines the lack of semantic interpretation of 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.

6 Conclusion & perspectives

In this paper, we have proposed a generic NL system for command interpretation 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. It can be used for human-agent communication in semantic web service composition [9]. However, even if we use the VDL language for programming agents, our approach is not de-
dependent of the language and can be easily adapted to others introspection-capable programming languages.

The strength of top-down algorithms is to allow the system to tell the user why a given command cannot be performed, whereas the bottom-up approach can propose guidelines to the user about the system’s expectations, when a command is not correctly understood. Our combination algorithm relies on a generic dialogue architecture and there are no specific rules for the interaction problem, like clarification, imprecise command or impossible command.

Currently, our system works with a minimal semantic analysis on the ontology (synonymy). As a consequence, it can only understand commands that are formulated with concepts defined within the agent’s code. Our aim is to make use of real semantic distance measures for associating the human command concepts with the agent’s concepts in the ontology, as proposed by [7] for instance. This should allow the system to understand commands that require a semantic interpretation of the context.

Another limitation of our system (that does not appear within our evaluation because the agent’s domain does not admit it) is that our “bag of words” syntactic analysis, presented in section 3.1, leads us to lose important information. For example, the command “from Boston to London” will result into the same bag as “from London to Boston”. [14] give a representation of user’s sentences as a set of constraints that deals with this issue. We shall include such “soft” syntactic parsers in our system.

References