Generic natural language command interpretation in ontology-based dialogue systems

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

Abstract
This paper presents a general architecture towards a more generic approach to conversational agents. Our architecture contains generic (in sense of application independent) natural language (NL) modules that are based on ontologies for command interpretation. We focus on the presentation of the event generator and dialogue manager modules which rely on a bottom-up approach for matching the user’s command with the set of currently possible actions.

1 Introduction

Recent works on Embodied Conversational Agents (ECA) [7] and more generally conversational systems [2] showed that natural language (NL) interaction is one crucial step in the course toward a more natural human-agent interaction. However, the chosen approaches in ECA mostly rely on ad-hoc pattern matching without semantic analysis [1]. The dialogue system community, on the other hand, proposes to use ontologies to improve genericity [8, 11]. 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 [8, 12] 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. [11]) 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).

This paper focuses on command interpretation for intelligent agents. We propose 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 [9]) deals with misunderstood or indefinite commands.

Our paper is organised as follows. In the second section, we give a general overview of our agent model. The third section presents the Natural Language Processing algorithm we use. We first present the parser. We then detail our algorithm for command interpretation. We also present our dialogue manager that deals with clarification. Section 4 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 use a specific language that allows us 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. 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

1 http://www-poleia.lip6.fr/~sabouret/demos
planning, question answering, behaviour recognition, etc [13]. In this paper, agent’s introspection is used to explain why a given command is not possible. Moreover, this architecture allows XML-based NL generation directly from the agent’s code.

In the VDL model, every agent is provided with a domain ontology written in OWL\(^2\). This ontology must contain at least all the concepts used by the agent (i.e. the VDL concepts), either as XML tag, 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 \textit{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

In VDL, as in most action representation models, actions are defined as a tuple \(< N, P, E >\) where \(N\) is the name of the action, \(P\) is the set of preconditions and \(E\) the set of effects. Moreover, interaction with the user and other agents is achieved by \textit{external events}. External events are the formal representation of commands, sent to the agent at runtime.

For a given action, we can outline four kinds of preconditions in VDL for an action \(r \in \mathcal{R}\), the set of actions of the agent:

- \(\mathcal{P}_e(r)\) is the set of event preconditions. It defines the set of triggering events using term \textit{subsumption}.
- \(\mathcal{P}_s(r)\) 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 \(\mathcal{P}_s\) 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, \textit{i.e.} preconditions that only depend on the agent’s internal state.
- \(\mathcal{P}_{cs}(r)\) is the set of contextual structure preconditions, \textit{i.e.} preconditions that depend both on events (selected by \(\mathcal{P}_e\)) and on the agent’s internal state.

We note \(\mathcal{P}_e = \bigcup_{r \in \mathcal{R}} \mathcal{P}_e(r)\). 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 actions that are linked with \(e\).

### 3 NL processing for commands

This section presents the algorithms used for building VDL events from natural language commands. Our main objective is to improve genericity. We define a generic algorithm for command interpretation, based on the ontology and the operational semantic of our agent’s programming language. We first give a general overview of the NLP toolchain and the global system’s architecture. We afterwards present the algorithm and the dialogue manager.

#### 3.1 NLP toolchain and global architecture

In our project, the lexical module is based on the default OpenNLP\(^3\) tokenizer, tagger and chunker (see Figure 1). Additionally we make use of a homemade lemmatizer\(^4\). As anticipated in [10], 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. “\textit{drop object low}” or “\textit{take blue}”). Thus, we represent the content of sentences as bags of words, after removed stop-words identified by their tags\(^5\).

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 with the \(\text{owl:sameAs}\) transitive and reflexive relation for concept synonymy in OWL.

Our semantic analysis method relies on the hypothesis of \textit{semantic connectivity} [14], enriched with synonymy: every concept that appears within a relevant command is either directly associated with a VDL concept or is in a \text{sameAs} relation with an agent’s concept.

If we note \(S\) the bag of words that represent the user sentence, we can build the set \(C\) of known concepts that appear within the user’s command:

\[
C = \{v \in C_{VDL} | \exists s \in S. \text{map}_{VDL}(v) = s \text{ or } \text{map}_{VDL}(v) \text{ owl:sameAs } s\}
\]

The last part of our chain is an XML-based English NL generator that transforms any VDL node into an English sentence, based on [4].

\(^2\)\url{http://www.w3.org/TR/owl-guide/}

\(^3\)\url{http://opennlp.sourceforge.net/}

\(^4\)We preferred OpenNLP to the widely-used TreeTagger for compatibility reasons with our Java online version.

\(^5\)This leads us to lose important information. [10] give a representation of user’s sentences as a set of constraints that deals with this issue. We didn’t implement this within our system yet.
3.2 General overview

Our approach is based on Allen’s bottom-up approach [3]. 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. [12, 8]). However, the competences list has to forecast all possible dialogues (even problem cases) and their translation into formal commands (possibly with parameters). 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. [5]) and showed interesting results for testbeds generation.

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. Similarly, using constraints relaxation on context preconditions and contextual structure preconditions ($P_c$ and $P_{ca}$), it builds the lists of “currently impossible” events, i.e. events that are not acceptable by the agent in its current state but that would be accepted in a different state.

The next section presents the algorithm which computes the possible and the impossible events set and select relevant events. Section 3.4 presents the Dialogue Manager (DM) that deals with the sets of possible and impossible events to generate better feedback to user.

3.3 Events generation & selection

The event generation algorithm is responsible for building a set of potential event. It is the core of our NL command processing system. It allows the system to use the agent’s actions description (extracted from the agent’s code itself) so as to build the set of events that can be carried out by the agent. This avoids the use of a priori defined static competences lists.

Our algorithm builds both the set of possible events $E$ or the set of “currently impossible” events $F$. We use event preconditions ($P_e$) to provide the initial skeleton of the event. Since $P_e$ filters external events using subsumption, all events build by addition of sub-elements to a skeleton $e \in P_e$ will be accepted and reciprocally, all accepted events must be build from a skeleton. We note $\Upsilon$ the (infinite) set of all possible VDL nodes. To compute $E$, we remove from $P_e$ events that cannot be processed currently:

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

with $eval: \Upsilon^2 \rightarrow \{\top, \bot\}$ the precondition evaluation function: $\forall p \in P(r), eval(p, e) = \top$ iff the precondition $p$ is valid with respect to the event $e$ and the current agent’s state. $P_{e++}$ is the set of event skeletons that are accepted by the agent with respect to constraint preconditions ($P_c$). Note that $P_{e++} \subseteq P_e$.

Now we use structure and contextual structure preconditions ($P_s$ and $P_{ca}$) as a set of constraints on the events to refine event skeletons into actual events. For all $e \in P_e$, we note $refine(e, r) \in \Upsilon$ the event obtained from the skeleton $e$ and the set of preconditions $P_s(r) \cup P_{ca}(r)$ of the action $r \in R_e(e)$ using our test-bed generation based algorithm. The complete algorithm for $refine$ 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.

This leads to the set of syntactically correct events:

$$E = \left\{ refine(e, r), \forall e \in P_{e++}, \forall r \in R_e(e) \right\}$$

$$F = \left\{ refine(e, r), \forall e \in P_e, \forall r \in R_e(e) \right\}$$

$E$ is the set of possible events: all events in $E$ will be accepted by the agent and all accepted events belong to $E$. Conversely, $F$ is the set of “currently impossible” events, i.e. events that are not acceptable by the agent in its current state but that would be accepted in a different state.

Once possible and currently impossible events have been generated, the selection algorithm tries to select the most appropriate one with respect to the user’s command. The general idea is to compute the probability of every event in $E \cup F$ and to determine the maximum probability event in $E$ and $F$.

For every node $n \in \Upsilon$ and for any 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 probability $p(e)$ of an event $e \in E \cup F$ is:

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

We build the subsets $E'$ and $F'$ of maximum-probability events. For $X \in \{E, F\}$, we note $X' = \{e \in X | p(e) = \text{max}(\{p(x), x \in X\})\}$.

We can build the set $E$ of possible events and the set $F$ of “currently impossible” events:

$$X = \left\{ \begin{array}{ll} \emptyset & \text{if max}(\{p(x), x \in X\}) = 0 \\ X' & \text{otherwise} \end{array} \right.$$

Moreover, $\forall e \in F$, we note $np(e)$ the set of invalid preconditions that make this event impossible to process:

$$np(e) = \{ p \in P_c(e) \cup P_s(e) \cup P_{ca}(e) | eval(p) = \bot \}$$
3.4 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.

We make use of two thresholds in $[0.0, 1.0]$ (as $p_E$ and $p_F$). $p_{\text{min}}$ 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 [9]. 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 $|E| = 1$) or informs the user about an ambiguity (when $|E| > 1$).

2. If $p_E < p_{\text{max}}$ and $p_F < p_{\text{max}}$, the system is unsure about the user’s command. It asks for a confirmation or a reformulation, depending on the position of $p_E$ and $p_F$.

3. 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 $\text{np}(e), e \in F$.

4. 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 Conclusion & perspectives

In this paper, we proposed a general NLP architecture for command interpretation based on the idea that generic algorithms can be parameterized by the agent’s code and a domain ontology. Our system relies on a constructive bottom-up approach based upon the action preconditions. Even if we use the VDL language for programming our agents, the approach is language-independent and can be easily adapted to others introspection-capable models.

We conducted a preliminary evaluation of our system that shows that the feedback provided by the DM allows the user to align on the agent’s ontology. Our system tells the user why a given command cannot be performed and shows the system’s expectations. Users have the feeling that the system is “more clever” with our constructive bottom-up DM than with classical approaches. This evaluation also shows that the limitation of our system resides in the minimal semantic analysis on the ontology (synonymy). To overcome this issue, we propose to use advanced semantic distance measures (as given by [6] for instance) for associating the human command concepts with the agent’s concepts in the ontology.

References


