On Generating Grounded Language in Cognitive Architecture

Shiwali Mohan

2011-04-11 Mon

For an intelligent agent that is situated in an environment and is collaborating with other agents (human/AI), the ability to communicate becomes extremely important. The representations that the agent uses for information processing and problem solving are very 'personal' to the agent, and it needs to be able to translate these representations to a communication protocol. If humans are involved, this communication protocol could be a natural language. In the first part of this project, I have looked at augmenting the Soar Cognitive Architecture with a Java natural language realization library - SimpleNLG. I have implemented a simple English grammar that allows a Soar agent to produce a structured output that is caught by a Java wrapper that produce English sentences using SimpleNLG Library.

Intelligent systems that explain their actions promote the user's understanding as they give the user more insight in the effects of their behavior on the environment. In order to provide individualized intelligent explanations, we need not only to evaluate a user's observable behavior, but we also need to make sense of the underlying beliefs, intentions and strategies. Throught the second part of my project I have looked at how semantic concepts and lexical concepts can be integrated to allow a proactive agent to communicate using language. I demonstrate the theory using a simple blocks-world example and I discuss how it can be generalized to more complex scenarions. I also look at how this theory can be extended to generating simple natural language explanation of an agent's actions.

1 Introduction

2 Motivation

3 Related Work

4 Background

Generating language consists of two logically distinguishable tasks, *Tactical Generation* involves making appropriate linguistic choices given the semantic input. However, once tactical decisions have been taken, building a syntactic representation, applying correct morphological operations and linearizing sentence as a string are comparatively mechanical tasks.

In this proposed work, Soar Cognitive Architecture has been used for Tactical Generation in a intelligent agent situated in a blocks-world domain and Java wrapper has been written to catch the structured output generated by a Soar agent to generated English sentences using SimpleNLG.

4.1 Soar

Soar is a symbolic cognitive architecture that uses a production system to encode its procedural knowledge. Soar has a working memory (WM), in which to store short-term knowledge describing its current state, and a production memory, in which to store long-term knowledge (encoded as condition => action rules).

Soar's WM is implemented as a directed, connected graph of working memory elements (WMEs). Each WME is a triple, consisting of an "identifier," "attribute," and "value." The identifier is a reference to an existing WME within working memory. The attribute is a string constant, providing a semantic descriptor of the augmentation provided by the WME to the associated identifier. The value can be a reference to an existing identifier or a constant (numeric or string). For any identifier, Soar implements an attribute-value pair set: thus no two WMEs can have the same identifier, attribute, and value.

The following sequence illustrates the Soar execution cycle:
Input => Proposal => Selection => Application => Output

During the input phase, environmental data is populated in a special input branch of working memory. Based upon the agent's current state, rules may match. During the proposal phase, these rules will nominate potential actions (termed "operators") that can be applied in the current state. Rules also fire to assert preferences as to the relative desirability of applying the proposed operators in the current state. During the selection phase, asserted preferences are compared, a decision is made, and a single operator is selected. In the application phase, more rules may fire as a result of operator selection. Finally, a special output section of working memory is sent to the environment in the output phase.

4.2 SimpleNLG

SimpleNLG is a Java library that provides interfaces offering direct control over the realisation process, that is, over the way phrases are built and combined, inflectional morphological operations, and linearisation. It defines a set of lexical and phrasal types, corresponding to the major grammatical categories, as well as ways of combining these and setting various feature values. In constructing a syntactic structure and linearising it as text with SimpleNLG, the following steps are undertaken:

1. Initialisation of the basic constituents required, with the appropriate lexical items

2. Using the operations provided in the API to set features of the constituents

3. Combining constituents into larger structures, again using the operations provided in the API which apply to the constituents in question

4. Passing the resulting structure to the lineariser, which traverses the constituent structure, applying the correct inflections and linear ordering depending on the features, before returning the realised string.

4.3 Integration - NLGSoar

The integration of SimpleNLG and Soar is relatively simple. The Soar agent writes structured "sentences" on a special language section of the output of its working memory which is caught by a Java wrapper. The Java wrapper uses the lexical information provided by Soar to create English sentences using the SimpleNLG API. The details of the grammar used is given in later section [section number]

5 Language Generation

5.1 Grammar

Following grammar rules are supported in the current implementation of NLGSoar

- S -> NP VP
- NP -> (DET)(ADJP)Noun
- ADJP -> (PRE-MOD)*Adjective(POST-MOD)*
- VP -> (ADVERB-PHRASE) Verb (COMPLEMENT)(INDIRECT-OBJECT)(PREPOSITIONAL-PHRASE)
- COMPLEMENT -> NP
- IDIRECT-OBJECT -> NP
- PREPOSITIONAL-PHRASE -> preposition NP
- ADVERB-PHRASE -> (PRE-MOD)*Adverb(POST-MOD)*

Apart from these, SimpleNLG also allows of adding multiple noun phrases to form subjects and complements. Sentences can be combined using co-ordinating or sub-ordinating conjunctions. Various features such as tense, form, negation can be specified of sentences and SimpleNLG generates appropriate English constructs.

5.2 Implementation Details

5.3 Examples

5.3.1 Specific Sentences

5.3.2 Scene Description

5.4 Limitations

6 Conceptual and Lexical Knowledge Integration

7 Explanation

7.1 Theory of Explanation

7.2 Blocks World Example

7.2.1 Limitations

8 Conclusions

9 Future Work