CAML – A Universal Configuration Language for Dialogue Systems

Gergely Kovásznai[1,2], Constantine Kotropoulos[1], and Ioannis Pitas[1]

[1] Dept. of Informatics,
Aristotle Univ. of Thessaloniki,
Thessaloniki, Greece
{kovasz,costas,pitas}@zeus.csd.auth.gr

[2] On leave from the Institute of Mathematics and Informatics,
Univ. of Debrecen,
Debrecen, Hungary
kovasz@math.klte.hu

Abstract. In this paper, a novel architecture of a universal dialogue system and its configuration language, so-called Conversational Agent Markup Language (CAML), is proposed. The dialogue system embodies a CLIPS engine in order to enable CAML to formulate procedural and heuristic knowledge. CAML supports frames, functions, and categories that enable it: (a) to process wildcards, to control the inner state through variables, and to formulate procedural knowledge in contrast to Phoenix/CAT Dialog Manager; (b) to support nested macros, to control the inner state through variables, to assign priorities and weights to states, and to interface with external databases in contrast to Dialog Management Tool Language (DMTL); (c) to implement context-free grammars, to extract semantic content from user input through frames, to allow numeric variables, and to interface with external databases as opposed to Artificial Intelligence Markup Language (AIML). The proposed system is extensible in the sense that it can be embedded in any conversational system that receives and emits XML content. Such a dialogue system can be incorporated in multimodal interfaces, such as talking head applications, conversational web interfaces, conversational database interfaces, and conversational programming interfaces.

1 Introduction

Nowadays, human-machine interaction is changing dramatically toward spoken dialogue. Many dialogue systems (DSs) or conversational agents have been developed for web applications, database interfaces or even chat bots [2]. State-of-the-art DSs vary in their architecture, aims, and configuration. All of them include three basic modules, namely a dialogue manager, a language parser, and a language generator. Henceforth, the aforementioned modules will be referred to as the DS core.

The features of any state-of-the-art DS core depend highly on the application they have been designed for. Usually, the application is well-defined, but limited

V. Mařík et al. (Eds.): DEXA 2003, LNCS 2736, pp. 896–906, 2003.
© Springer-Verlag Berlin Heidelberg 2003

to a restricted domain, e.g. train/airplane reservation systems. Furthermore, the core is highly influenced by the additional modules embedded in the same DS (e.g., speech recognizer, speech synthesizer) in terms of the structure of input (output) data received (emitted). Accordingly, any state-of-the-art DS core cannot be used as a universal one, that can be embedded in any DS. For example, the core of a chat bot cannot be transformed to the core for a conversational database interface. The aim of this paper is to propose an architecture for a DS core which is arbitrarily *extensible*, i.e., it can be extended with any external resource and external module in order to be embedded in any DS. Another aim of the paper is to design a novel configuration language for the core, which is *universal* in the sense that it can be attached to any spoken language or any possible topic offering syntactic and semantical independence. Other strong points of the configuration language are the low-level procedural knowledge formulation (for experienced users) and its facilities to perform tasks closely related to dialogue management (for naive users) that make it easily used. We call the proposed configuration language Conversational Agent Markup Language (CAML). It is an XML-compliant language. A DS core configured by CAML is called a CAML core.

2 Overview of a Dialogue System Core

A general architecture of a DS core is shown in Figure 1. Its three modules perform language parsing, dialogue management, and language generation. The configuration language of the core must manage all the aforementioned tasks and interrelate them.

Several techniques have been applied to these tasks. Since CAML is designed to be a universal language, it supports the most flexible, commonly used techniques. Although the use of fixed techniques may limit the utility of CAML, its

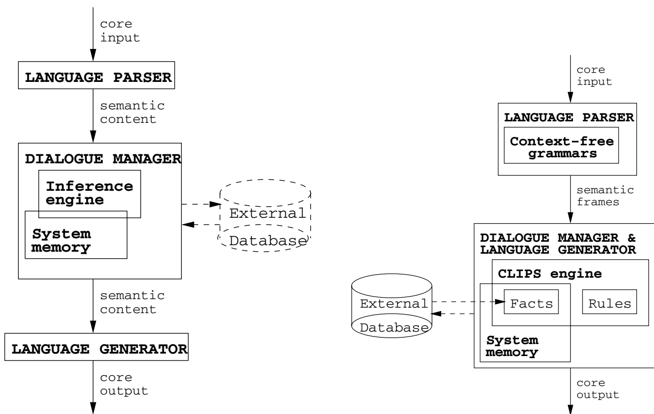


Fig. 1. Architecture of a dialogue system core and the CAML core (on the right).

extensibility makes it possible to replace these techniques with external ones, as explained in Section 4.

2.1 Language Parsing

Language parsing extracts semantic information from the core input. Language parsers employ several techniques. CAML supports *robust parsing* [1]. To configure a robust language parser, a grammar must be specified in order to analyze the core input. In terms of the grammar used, several types of language parsers exist. CAML uses context-free (CF) grammars [3]. Since CF grammars are not able to formulate complex data and relations that are common in real life, CAML provides also facilities to combine CF grammars with procedural data in order to alleviate the just described deficiency of CF grammars. In a CF grammar, a set of non-terminal symbols and a set of CF rewriting rules are specified. If the user input can be derived by a CF grammar, one or more parse-trees can be constructed for it. Language parsers vary even in the representation of semantic information extracted from a parse-tree. CAML uses the *semantic frames* [1]. CAML provides a facility for users to define the frames used in the extraction of semantic information.

A CAML core analyzes the input in the following way: (1) it checks whether the input can be derived from a non-terminal symbol in the grammar; (2) if it can be derived, it constructs one of the possible parse-trees based on the priorities of non-terminal symbols that are employed to define the tree having the highest priority; (3) it extracts the semantic information from the parse-tree into frames.

2.2 Dialogue Management and Language Generation

Dialogue management determines the next state of the DS core according to its current state and the currently extracted semantic information. It is also responsible for generating the semantic content used during language generation in order to provide a spoken output. The CAML core contains only one module to perform the tasks related to these two phases, i.e., (1) to load the frames extracted during language parsing; (2) to trace the inner state of the core (to place the data into the system memory and to retrieve them if needed); (3) to infer the next state of the core (i.e., the content of the system memory) from its current state and the frames loaded; and (4) to generate output. Some dialogue managers provide an interface for external databases, i.e., to perform queries in the databases and to load the results into the system memory.

The CAML core contains a *CLIPS engine*. CLIPS is a tool for building expert systems [9]. As a part of the CAML core system memory, CLIPS facts are specified during the dialogue management and they are modified by CLIPS rules defined by the user.

The system memory is a set of variables. A variable can get either a literal or a frame value. A frame value is implemented as a CLIPS fact. CAML provides the users facilities to formulate the procedural knowledge inherent in CLIPS language and heuristic knowledge by defining CLIPS rules. CAML provides the

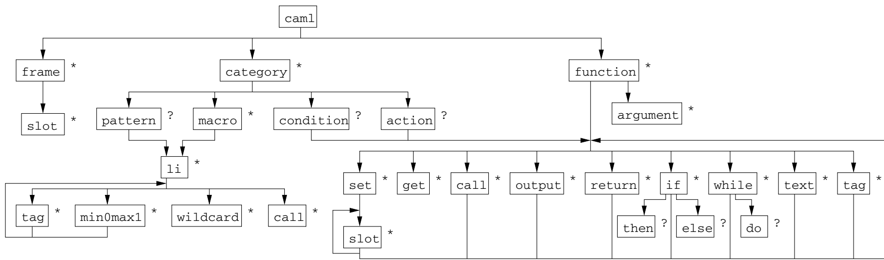


Fig. 2. CAML DTD

following facilities in order to perform the aforementioned tasks: (1) it loads the extracted frames into variables; (2) it declares variables, it sets and gets their values; (3) it formulates the procedural and heuristic data in order to specify CLIPS rules and to execute them; (4) it formulates the procedural data in order to specify core output. CAML provides facilities to query in external databases and to load the results into variables.

3 Specification of CAML

In this section, the syntactic elements of CAML are introduced. Since CAML is an XML-compliant language, its structure can be specified by a Document Type Definition (DTD) as can be seen in Figure 2. The use of syntactic elements is demonstrated by examples which belong to a simple DS providing information about museums.

3.1 Top Level Syntactic Elements

Three syntactic elements of CAML are accentuated, namely the frames, the functions, and the categories. They are located on the highest syntactic level, while the rest can be located only within another syntactic element. A frame used during language parsing can be defined by the use of a `<frame>` tag containing zero or more `<slot>` tags. As part of procedural knowledge, functions are defined as parameterizable units of procedural data. They are implemented by `<function>` tags.

```
<frame name="query">
  <slot name="museum"/>
  <slot name="monument"/>
</frame>
```

```
<function name="display-museum">
  <output>
    The following information was found about the museum
    <argument number="1"/> located in <argument number="2"/>.
    <argument number="3"/>
  </output>
</function>
```

Fig. 3. Frames and functions.

The most important syntactic element of CAML is the category. A category can be defined by a `<category>` tag. Categories contain all parse-specific and heuristic data on the points where the execution of procedural data starts. Each category contains some rewriting rules, a condition on its validity, and an action (i.e., procedural data) to be executed if the category is valid. Actually, two types of categories can be defined by using the same syntax. If the given category specifies at least one rewriting rule, it formulates parse-specific and strictly procedural data. Otherwise, it formulates heuristic data, a CLIPS rule. The execution of a CAML content attaches to the categories inside, and comprises the following tasks: (1) to read core input; (2) to find the category having the highest priority to which the input fits; (3) to construct the parse-tree having the highest priority for the input and the category; (4) to extract the frames from the parse-tree; (5) to execute the actions of the categories within the parse-tree; (6) to instruct the CLIPS engine to start the rule execution.

The tags `<pattern>` and `<macro>` contain so-called patterns, i.e., the right-hand-side of rewriting rules. Within a pattern a non-terminal symbol can be specified by the use of a: (1) `<call>` referring to a `<category>` or a `<macro>`; (2) `<opt>` specifying an optional pattern; (3) `<wildcard>` which can be matched with any text. The use of `<wildcard>` tags is a very important facility to define partially specified patterns. The text matched with a wildcard can be stored in a variable. Accordingly, we may check in a `<condition>` tag, whether its value is equal to the value of another variable or can be found in an external database. The tags `<condition>` and `<action>` contain procedural data. Each `<condition>` tag must return a boolean value TRUE or FALSE.

By the use of the parameter `priority`, a priority number can be assigned to a category, which is used to define the parse-tree having the highest priority or to schedule the execution of CLIPS rules. The parameter `extracted` determines whether the category takes part in the extraction of frames [6]. A set of categories, a constructed parse-tree, and an extracted frame are shown as examples in Figure 4.

3.2 Procedural Knowledge

Procedural data can be specified within either a `<function>`, or a `<condition>`, or an `<action>` tag. Procedural data is a set of CLIPS function calls, e.g., `(str-cat "Hi" " Joe")` or `(= (+ 2 2) (- 7 3))`. CAML provides some tags to ease the situation of naive users, that is, to exempt them from using CLIPS language, e.g., `<output>` for sending data to core output, `<if>` for checking a conditional statement, `<while>` for making a loop in the execution, `<call>` for calling an arbitrary CLIPS function or command, `<return>` for returning a value from a CLIPS function, `<text>` for specifying a CLIPS string literal.

Special tags are needed to control variables. The tags `<set>` and `<get>` are used to read and write a value in an object referred by a qualified name, i.e., a name consisting of segments separated by dots. The first segment is the name of a variable, the rest is a sequence of names of slots. E.g., `person.name.surname` refers to the slot "surname" of the slot "name" of the variable "person". The

```
<category name="tell museum" priority="10">
  <pattern>
    <li><call>speak</call>
      about <call>museum</call></li>
  </pattern>
  <macro name="speak">
    <li>speak</li>
    <li>tell me</li>
  </macro>
</category>

<category name="museum" extracted="museum">
  <pattern>
    <li>
      <opt>museum called</opt>
      <call>museum name</call>
      <opt>located</opt> in
      <call>museum location</call>
    </li>
  </pattern>
</category>

<category name="museum name"
        extracted="name" priority="9">
  <pattern>
    <li><wildcard name="museum_name"/></li>
  </pattern>
  <condition>...</condition>
</category>

<category name="museum location"
        extracted="city" priority="9">
  <pattern>
    <li><wildcard name="museum_city"/></li>
  </pattern>
  <condition>...</condition>
</category>
```

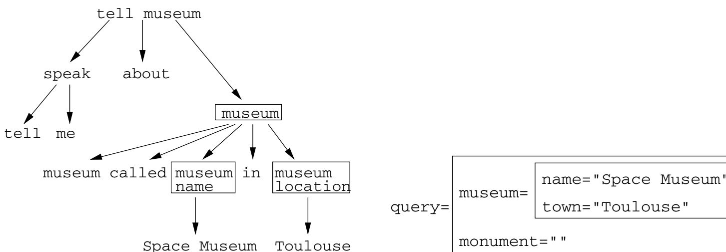


Fig. 4. A parse-tree and an extracted frame for a set of categories related to the input "tell me about museum called Space Museum in Toulouse".

referred object can get either a literal or a frame value. A frame value can be formulated by the use of `<slot>` tags. In Figure 5, the use of `<set>` and `<get>` is shown. Below each `<get>`, the value returned by the tag is written.

In order to read the content of a frame extracted during language parsing, a `<get>` with the parameter `type="frame"` can be used. In this case, the first segment of the qualified name is interpreted as the name of the given frame. An example is shown in Figure 5.

```
<set name="x">
  <slot name="name">Louvre Museum</slot>
  <slot name="location">
    <slot name="country">France</slot>
  </slot>
</set>
  .
  .
  .
<set name="x.location.city">Paris</set>
```

```
<get name="x.location.country"/>
➥ France

<get name="x.location"/>
➥ <slot name="country">France</slot>
  <slot name="city">Paris</slot>

<get name="query.museum" type="frame"/>
➥ <slot name="name">Space Museum</slot>
  <slot name="city">Toulouse</slot>
```

Fig. 5. Controlling variables.

```
<category name="museum_rule">
  <condition>
    <return>
      <call name="neq">
        <get name="query.museum" type="frame"/>
        <text></text>
      </call>
    </return>
  </condition>
  <action>
    <set name="museum_query" type="sql"
      database="jdbc:mysql://zeus.csd.auth.gr/museumdb?user=visitor&password=04eg35">
        select name,city,text from museums where
          name='<get name="query.museum.name" type="frame"/>' and
          city='<get name="query.museum.city" type="frame"/>'
    </set>
    <while>
        <call name="neq">
          <get name="museum_query"/>
          <text></text>
        </call>
      <do/>
        <call name="display-museum">
          <get name="museum_query.name"/>
          <get name="museum_query.city"/>
          <get name="museum_query.text"/>
        </call>
        <get name="museum_query" type="sql"/>
    </while>
  </action>
</category>
```

Fig. 6. Interfacing an external database and formulating complex procedural data in a category specifying a CLIPS rule.

In order to interface an external database, the parameter `type="sql"` can be used in a `<set>` or a `<get>`. In this case, the content of a `<set>` is interpreted as an SQL command. The database is accessed through JDBC interface, so it is addressed by a JDBC URL. The first result of the SQL query is loaded into the object referred by the qualified name. The next result can be loaded by the use of a `<get>` with `type="sql"`. An example can be found in Figure 6.

4 Extensibility of the CAML Core

One of the aims of the CAML core is its extension by external resources if needed. The DS core reads a text input and emits a text output, traditionally in a spoken language. The CAML core uses the same input channel and the same output channel in order to be extensible. In order to communicate with external resources through these channels, the CAML core reads *structured text-typed input* and emits *structured text-typed output*. A quite easy way to structure text data is the use of an *XML*-compliant language, e.g., if an external language parser is used to extract semantic information from a user input, this information can be easily incorporated in the input of the CAML core by the use of XML tags. Similarly, semantic content can be embedded in the output of the CAML core in order to make it accessible for an external language generator. The most important reason of making a DS core able to read structured input and to emit

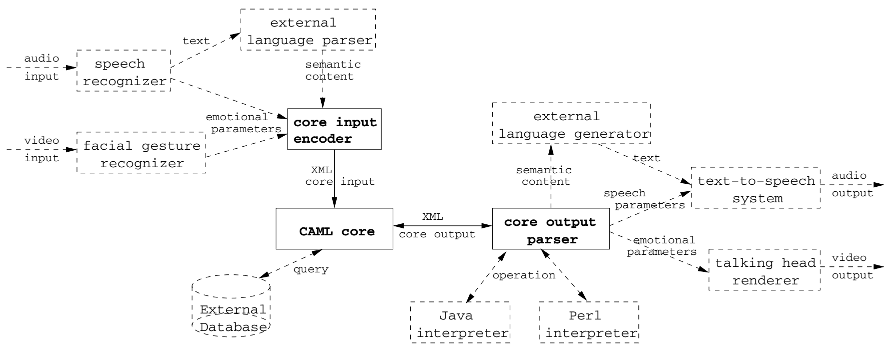


Fig. 7. Architecture of a possible dialogue system including the CAML core.

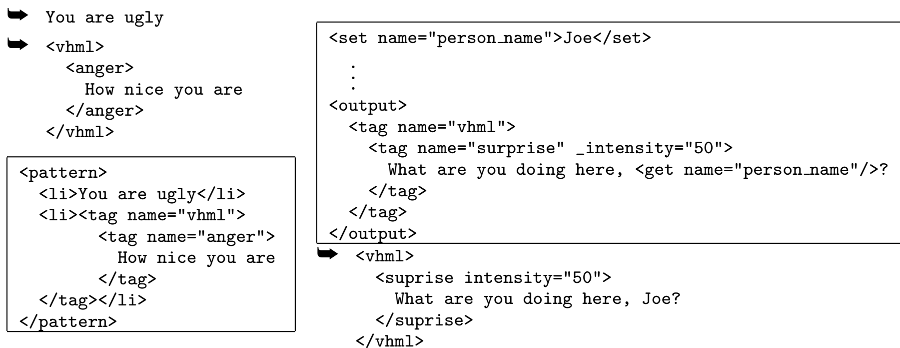
structured output is the *multi-modal nature of human communication*. Humans communicate by expressing their thoughts in several ways simultaneously, i.e., not only by the words uttered, but also by prosody of speech, facial gestures, hand movements, etc. Three possible applications are:

1. *Talking head applications*: an animated human head which the user can conduct a dialogue with. The core output could be in Virtual Human Markup Language (VHML) [5] in order to incorporate information about emotions, gestures, movements, etc.
2. *Web pages with conversational interfaces*: a web page which provides a natural language interface to help web navigation. Core output could be in Hypertext Markup Language (HTML) in order to generate whole web pages or web page segments, and to execute scripts.
3. *Conversational programming interfaces*: a DS which can "translate" user requests into procedural data in a programming language and execute them. The source code in a given programming language could be encapsulated by an XML tag in core output, e.g., `<cplusplus>...</cplusplus>` for C++, `<java>...</java>` for Java, etc.

In order to embed the CAML core in a DS, two modules are necessary. The first module encodes the core input in XML (core input encoder) and the second module parses the core output (core output parser). In Figure 7, a possible architecture of such a DS is shown.

In order to support XML core input and core output, CAML allows embedding XML tags in patterns of categories and within `<output>` tags. Since not every XML tag is included in the CAML DTD, its reserved characters must be encoded into XML standard entities (e.g., "<" into "`<`", "`"`" into "`"e;`"). CAML provides a tag called `<tag>` to exempt naive users from encoding them manually. In Figure 8, the use of `<tag>` is demonstrated by two examples with inputs that should be matched and the output that is generated.

Bi-directional communication is needed through the output channel of a CAML core in some cases. A typical example is the case of conversational programming interfaces, because there we need to load the results of operations



```
➥ You are ugly
➥ <vhml>
     <anger>
       How nice you are
     </anger>
   </vhml>

<pattern>
  <li>You are ugly</li>
  <li><tag name="vhml">
        <tag name="anger">
          How nice you are
        </tag>
  </tag></li>
</pattern>

<set name="person_name">Joe</set>
  ⋮
<output>
  <tag name="vhml">
    <tag name="surprise" _intensity="50">
      What are you doing here, <get name="person_name"/>?
    </tag>
  </tag>
</output>
➥ <vhml>
     <suprise intensity="50">
       What are you doing here, Joe?
     </suprise>
   </vhml>
```

Fig. 8. Structured core input and core output.

implemented in any programming language back into the core. To implement this, the core output parser must be able to specify return values for all XML tags embedded in core output and then to load them back through the output channel of the CAML core, as return values of the given `<output>` tags. The specification of these return values is a technical problem which has to be solved by the core output parser, hence it does not belong to the CAML and the CAML cores.

5 Comparison with Other Configuration Languages

In this section, the abilities of CAML are compared to the abilities of three state-of-the-art configuration languages for a DS core according to six aspects, presented in Table 1. CAML has been inspired by these languages. Any of them is used in a specific area. CAML can be conceived as a superset of these languages, with respect to universality and abilities.

Table 1. Abilities of other configuration languages.

	Language parsing	Controlling inner state	Procedural knowledge	Heuristic knowledge	Language generation	Interfacing databases
Phoenix/CAT	good	good	no	poor	poor	yes
DMTL	poor	poor	no	medium	poor	no
AIML	poor	medium	medium	no	medium	no

The *Phoenix Semantic Parser* [6] was developed by the Center of Spoken Language Research and used in *CAT Dialog Manager*, which was incorporated in CU Communicator [7] for flight, car, and hotel rental agents. The configuration language of the Phoenix Semantic Parser provides facilities to specify a set of context-free grammars and to extract the semantic information from the core input into frames. Only fully-specified core inputs can be formulated. The CAT Dialog Manager provides facilities to load the frames extracted by the Phoenix

Semantic Parser and to perform dialogue management tasks on them. These tasks can be specified in CAT's proprietary configuration language, which assigns so-called text templates to the slots of the frames. The inner state of the system is controlled through frames and global variables. The level of determination to display text templates is not specified by the CAT documentation, but it could be non-deterministic. System-defined weights are assigned to parse-trees, but user is not able to specify weights. Only three types of text templates (core outputs) can be assigned to the slots: prompts, confirmations, and sql queries.

The *Dialogue Management Tool Language (DMTL)* is the configuration language for the *Dialog Management Tool (DMT)* [8]. This language was tested embedding VHML content in talking head applications. Context-free grammars can be specified, but there is no facility to generate parse-trees higher than three levels. Only fully-specified core inputs can be formulated. There is no facility to extract semantic information from inputs. The states of the system can be defined and linked to each other, but no data can be placed into the system memory by the user. Validity conditions can be attached to the states, however it is not real procedural knowledge. Weights can be assigned to core outputs, but not to states or core inputs. Two types of core outputs can be specified, namely responses and signals.

The *Artificial Intelligence Markup Language (AIML)* was developed by A. L. I. C. E. AI Foundation as the configuration language of *Alicebot systems*, which are used in chat bots. AIML parsing is not based on context-free grammars, but rather on simple, linear pattern matching. Partially specified core inputs can be formulated. There is no facility to extract semantic content from the core input. Variables can be used, but only text values can be assigned to them. Procedural data can be formulated mostly to read, write or check the value of a variable. There are only a few facilities to operate with such a value. These is no possibility to operate with numeric values. Patterns are matched with the core input in alphabetic order, there are no facilities to set their priority. The core output is generated by the execution of procedural data. Only text output can be emitted.

Conclusions

We have designed a novel universal configuration language and an extensible architecture for a dialogue system core. We have outlined several state-of-the-art areas, where such a core can be used. The proposed language has been compared to other state-of-the-art configuration languages and its features are found to comprise a superset of those offered by the other configuration languages. After the design phase, we have developed the first version of the proposed CAML core.

Acknowledgement. This work has been supported by the European Union funded Research Training Network "Multi-modal Human-Computer Interaction" (MUHCI).

References

1. R. Suereth, *Developing Natural Language Interfaces.* N.Y., McGraw-Hill, 1997.
2. N. Ole Bernsen, H. Dybkjær, and L. Dybkjær, *Designing Interactive Speech Systems.* London, Springer-Verlag, 1998.
3. X. Huang, A. Acero, and H.-W. Hon, *Spoken Language Processing.* Upper Saddle River, N.J., Prentice Hall PTR, 2001.
4. A. L. Gorin, A. Abella, T. Alonso, G. Riccardi, and J. H. Wright, "Automated Natural Spoken Dialog", *IEEE Computer*, vol. 35, no. 4, pp. 51–56, April 2002.
5. A. Marriott, "VHML – Virtual Human Markup Language", in *Proc. OzCHI 2001 Workshop*, 2001.
6. W. Ward, "Understanding Spontaneous Speech: the Phoenix System", in *Proc. ICASSP 91*, 1991, pp. 365–367.
7. B. Pellom, and W. Ward, S. Pradhan, "The CU Communicator: An Architecture for Dialogue Systems", in *Proc. ICSLP*, Beijing China, November 2000.
8. C. Gustavsson, L. Strindlund, and E. Wiknertz, "Dialogue Management Tool", in *Proc. OzCHI 2001 Workshop*, 2001.
9. CLIPS, *http://www.ghg.net/clips/CLIPS.html.*