Using XML for Representing Domain Dependent Knowledge in Dialogos

G. Boella[1], R. Damiano[1], M. Danieli[2], and L. Lesmo[1]

[1] Dipartimento di Informatica – Università di Torino
Cso Svizzera 185
{guido,rossana,lesmo}@di.unito.it
[2] Loquendo
Via Nole, 55 - Torino
Morena.Danieli@loquendo.com

Abstract. In this paper we describe the extension of the EasyDial developer interface of Dialogos, the spoken dialog system of Loquendo. EasyDial has been integrated with an XML-based representation of data structures and procedural knowledge which are dependent on the application domain. A set of XSLT programs translates this knowledge into textual data and C language procedures which are integrated in Dialogos after the processing by EasyDial. The adoption of an XML-based representation for declaring the domain dependent knowledge speeds up the application development process significantly.

1 Introduction

The increasing demand for spoken dialog systems in different domains, ranging from providing information about trains or flights to personal services for accessing e-mail by telephone and to answering machines, is making more and more apparent the necessity for a rapid development of new applications, not necessarily involving the experts who implemented the dialog system shell.

In order to fulfill this requirement, the dialog system shell must be encapsulated in a further level of abstraction which hides the details concerning data structures and centralizes the knowledge shared by the modules of the system.

In this paper, we describe an approach to domain knowledge representation in Dialogos, the LOQUENDO spoken dialog system [1], and EasyDial, the developer toolkit for Dialogos [4], which aims at facilitating the development of spoken language applications in the telecommunication domain.

This approach is based on an XML based representation of the knowledge to be acquired by the system in order to provide its service; in particular, this knowledge concerns the structure of the sentences to be generated. Based on the XML representation, an XSLT program produces both the data in the internal representation format of the system and a relevant portion of the C language procedures to be used by the Dialogos engine in a given application.

XML has been chosen to define languages for representing the system knowledge since:

1. Its rapid diffusion as a standard ensures that it will be easier to find developers able to use it.
2. The precise definition of the syntax of documents via Document Type Definitions prevents errors during editing, while XML editors facilitate the editing by suggesting options and by signalling errors.
3. Its portability ensures the availability of programs for processing it on all platforms, both for producing XML documents and for operating on it (e.g., XSLT translators).
4. The automatic translation of XML documents in programming language procedures restricts the possibility of errors by the developers. In fact, the procedures are represented at a higher level of detail, via languages which are defined by means of XML.
5. The abstraction from implementation details and the rigorous DTD declarations make XML documents easily adaptable to new versions of the dialog system.

In the following, we will describe examples of use of XML in four cases: the representation of system parameters, the generation of dialogic turns, the generation of natural language phrases that describe the value of parameters, and the partially automated generation of procedures for managing parameters.

2   The Parameter Definition

The *parameters* of Dialogos are knowledge structures which contain data which must be acquired by the system to provide the service.

For example, in order to build a query to a database of flights, at least the cities of departure and arrival and some information about the date of flight are required. So DEPARTURE_CITY, ARRIVAL_CITY, and DEPARTURE_DATE, are parameters in Dialogos' terms, and must be acquired by the system in order to provide the caller with an answer. Parameters may have atomic values (for example "Rome") or complex values obtained from the meaningful composition of a set of atomic values (for example, "the eleventh of November").

The XML definition of parameters allows to gather some information about items which are often spread across different files. In particular, the XML definition specifies the structure of each parameter (e.g., which parameters it subsumes or which parameters it is composed of), its type, and the information about the initiative of the system towards it: whether it must be requested or confirmed. Finally, the information about the compatibility of the possible atomic sub-parameters is specified as well (see next Section).

The parameter definitions not only allow to centralize the information about the parameters themselves, but they constitute the source for the generation (by an appropriate XSLT program) of the data structures in C language which are used by Dialogos, and which were previously to be hand-written. In particular, the generation process exploits the knowledge about the type of parameters, and the subsumption information (i.e., the same C structure is shared by more than one parameter).

3   The Generation of Sentences

As described in [3], the generation of sentences in Dialogos is organised in two levels: *Dialog Moves* (DMs, like *Request* for asking information, *Verify* for checking the system understanding, etc.) and *Dialog Acts* (DAs).

DAs contain the knowledge about the form of the sentences which must be generated to perform a certain Dialog Move. They depend on the set of parameters (e.g., which are needed and which have already been acquired), and show a large variability. For example, a *Verify* DM concerning the departure city of a flight differs from one concerning the departure time:

1 Vuole partire da Torino ?
(Do you want to leave from Turin ?)

2 Vuole partire alle quattro del pomeriggio ?
(Do you want to leave at four o' clock ?)

Moreover, the same parameter (say DEPARTURE_TIME) can be expressed in different ways according to what the caller said:

3 Vuole partire fra le tre e le cinque ?
(Do you want to leave between three and five o' clock ?)

Finally, the form of the same DA may vary according to the context; e.g., within the context of a misunderstanding [2]:

4 Mi scusi, non ho capito. Vuole partire fra le tre e le cinque ?
(Sorry, I did not understand. Do you want to leave between three and five ?)

When the developing shell of Dialogos, EasyDial, was not supported by the XML module, such knowledge took the form of a long list of templates including all possible combinations of parameters and contextual flags. For example, the fourth DA listed above was represented as:

PARAMETER: TIME_INTERVAL
CONTEXTUAL_FLAG: NOT_UNDERSTOOD
PATTERN: Mi scusi, non ho capito. Vuole $\sim$?

where the first attribute denotes the parameter to be verified, the second one identifies the current state of the context (this DA should be used in case the previous caller's turn was not understood properly by the system) and the third one the pattern to be generated (provided that the gap '$\sim$' be filled with the value of the parameter, see next Section).

The verbs '*partire*' ('leave') and '*arrivare*' ('arrive') (which are missing from the template) are generated together with the value of the parameter, depending on the parameter to which the template applies (departure or arrival) and on the presence of a contextual flag (e.g., in case of the destination city of a return flight, the verb takes the form of '*ritornare*', 'to go back').

The XML based representation of Dialog Acts captures the above generalities and provides a more compact representation. An XSLT program is used to generate automatically all possible combinations of parameters and contextual flags. The output of the XSLT program is in the human-readable format which currently constitutes the input of EasyDial, so that the development system needs not to be modified and the generated Dialog Act can be checked and refined by hand.

The DAs which generate sentences 1–3 above (with many others) can be captured by a single XML construct together; it takes the following (somewhat simplified) form:

```
<DA type="VERIFY" NOT_UND="YES">
  <PARAMETER name="TIME"/>
  <PATTERN>Vuole ~ ?</PATTERN>
</DA>
```

Since the parameter TIME can take different forms according to the availability of its component parameters, different DAs will be generated, among which the one above, where only the TIME_INTERVAL was involved.

Since not all the combinations of component parameters make sense, (e.g., HOUR, the hour of departure, is not compatible with TIME_INTERV, "*Vuole partire alle cinque, fra le tre e le sei?", "*Do you want to leave at five, between three and six?"), the illegal combinations of component parameters are contained in the XML parameter definition described in the previous section, which can be accessed by the XSLT program when generating the alternatives.

4   The Generation of Parameters

As discussed in the previous section, the generation of DAs does not cope with the problem of generating the values of the parameters.

This task is accomplished by a set of procedures, which in the previous version of the system, had to be written by hand in C language.

In order to automatize the task, we defined by means of XML a simple programming language: a DTD has been built for representing at a higher level of abstraction the instructions necessary for generating the value of parameters. In this way, one XML document contains the rules for generating the value for each parameter (and for the component ones).

As a short example consider the representation corresponding to a C switch construct for transforming a number in the name of the corresponding weekday.

```
<ITEM>
  <PAR NAME="WEEK_DAY"/>
  <CASE><NUMCONST VALUE="1"/><STRING TEXT="'lunedi'"/>
  </CASE>
  <CASE><NUMCONST VALUE="2"/><STRING TEXT="'martedi'"/>
  </CASE>
  ...
</ITEM>
```

The automatic translation via XSLT of this construct accounts for the tasks of checking if WEEK_DAY is already bound, retrieving its value and assigning the output string to the appropriate variable (see Figure 2 for an example XSLT rule).

Note that in the hand written functions all these tasks, together with the management of possible failures in case of missing values, had to be complied with for each parameter.

Since the XML syntax may seem to some people clumsier than standard programming languages, it must be noted that the developer has at his disposal an XML editor. In the research project, we are using *Xeena* (developed by AlphaWorks)[1] which checks the correspondence with the DTD and suggests the developer the syntax of the generation rules (see in Figure 1 a snapshot of the *Xeena* editor).

5   Functions for the Parameters

Besides the functions for generating natural language descriptions of the value of the parameters, the application developer currently has to write a number of functions for checking the consistency of parameters and for transforming their values or for retrieving them from the database or from the application. Some typical examples are the consistency control that Dialogos applies for validating the recognition results of time expressions, and the ones applied for checking that the return date is after the departure date acquired in a previous dialog turn. An automatic generation of these procedures is more problematic than the description of the natural language form of parameters, since they are more application-dependent: currently, they must be written by a developer who has a deep knowledge of the structure of the database and of the underlying application.

Instead of developing an XML-based language for describing this knowledge at a higher level, we have chosen a semi-automatic approach. Starting from the knowledge about the parameters (see Section 2), a skeleton of each function is generated in an automatic way. This approach is inspired by the *JavaBeans* methodology for generating the 'interface' of java classes.

In fact, a number of functions are structured in a similar way and are associated to the parameters depending on their structure.

For example, the prototypical function for checking the internal consistency of a parameter composed of simpler ones (like the parameter TIME described above, which is composed of HOUR, TIME_INTERV, PART_OF_DAY) is structured in the following way:

1. declare local variables;
2. check the existence and retrieve the value of component parameters;
3. perform a domaind dependent consistency check on the possible combinations of component parameters;
4. return the result.

While the task 3 is complex and is different for each parameter, the procedures for executing the remaining ones can be generated automatically by knowing the structure of the parameter and its type, which (as we saw in Section 2) are stored in an XML document.

Even if the conceptual part of the work of the application developer must be made case by case, the existence of the structure of the functions to be written is of great help. Moreover, the automated generation of functions restricts the possibility of errors or missing configurations of parameters which should be detected by means of runtime debugging.

---

[1] The *Xeena* editor of IBM Alphaworks can be found at http://www.alphaworks.ibm.com/tech/xeena.
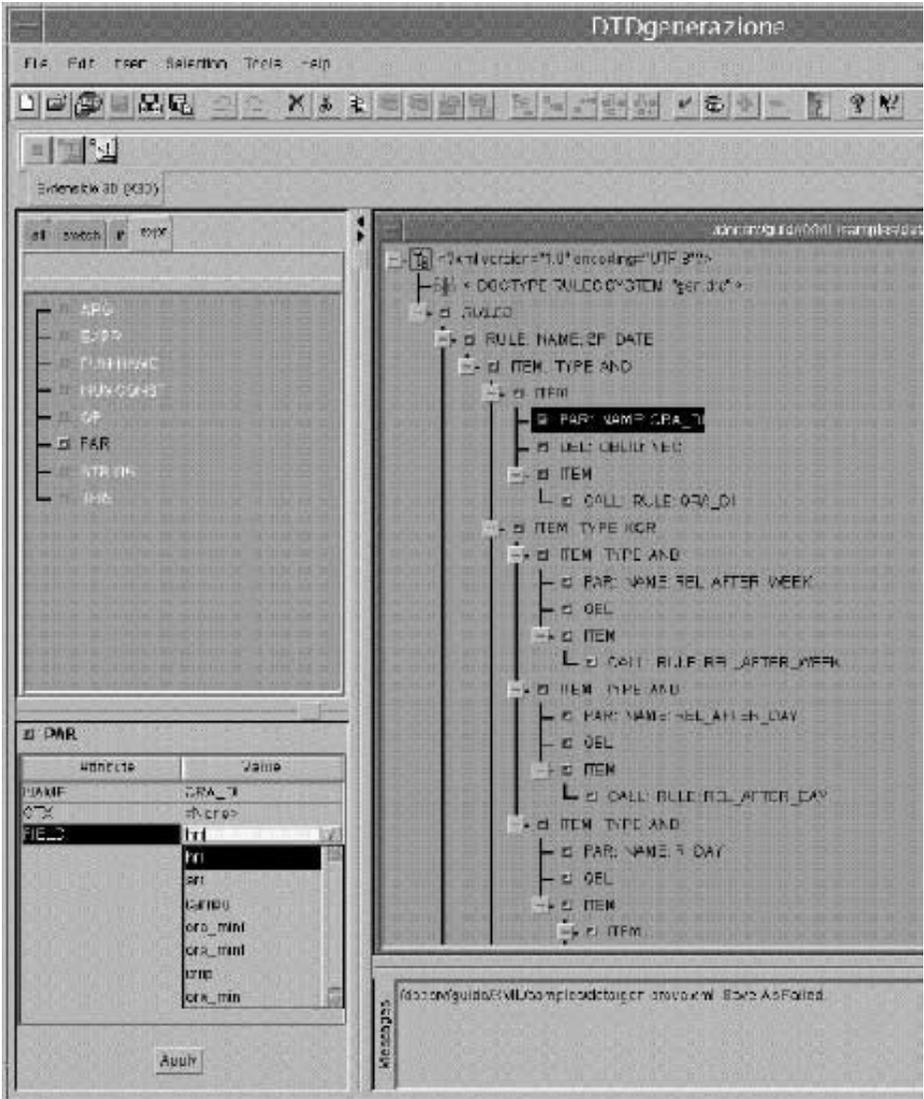
Fig. 1. The Xeena XML-editor displaying generation rules.

6   Conclusion

In this paper, we have described the work on the *EasyDial* developer interface for Dialogos. We have defined via XML some languages which are used as a representation formalism for storing both knowledge about data structures and procedural knowledge. This knowledge is then directly processed by EasyDial and integrated in Dialogos without any further intervention of the application developer.

```
<xsl:template match="PAR">
if (exist_param_gps(<xsl:apply-templates select="@NAME"/>))
<xsl:if test="following-sibling::CASE">
char string<xsl:number level="multiple"/>[50]="";
  string<xsl:number level="multiple"/> =
    get_value_param_gps(<xsl:apply-templates select="@NAME"/>);
  switch(string <xsl:number level="multiple"/>) {
  <!-- switch on the value of the local variable -->
</xsl:if>
<xsl:apply-templates select="following-sibling::CASE/STRING"/>
<!-- ... -->}
<xsl:if test="following-sibling::ITEM[position()=last()]">
if(strgen == ""){
 <xsl:apply-templates select="following-sibling::ITEM"/> } </xsl:if>
<xsl:if test="not(following-sibling::ITEM[position()=last()])">
 if(strgen == ""){local-flag=false;} </xsl:if>
else  { <xsl:apply-templates select="following-sibling::OBL"/> };
</xsl:template>
```

Fig. 2. An example of XSLT rule for translating generation rules. Text in italics corresponds to actual C code. Some pieces of them are generated or not according to XSLT conditionals. For instance, the input XML specifications can include constructs such as 'CASE'; in correspondence to it, a C '*switch*' will be generated. The 'xsl:apply-templates' tag takes care of calling recursively the XSLT rules on the rest of the XML document.

In this task, XSLT has proven to be a flexible and complete instrument for translating an XML document into data structures and C procedures, even if it was originally developed for dealing with document styles.

Acknowledgements

References

1. D. Albesano, P. Baggia, M. Danieli, R. Gemello, E. Gerbino e C. Rullent, "A Robust System for Human-Machine Dialogue in Telephony-Based Applications", in *International Journal of Speech Technology*, 2, 1997, pp. 101-111
2. M. Danieli, "On the use of expectations for detecting and repairing human-machine miscommunication", in *Proceedings of the AAAI Workshop on Detecting, Repairing and Preventing Human-Machine Miscommunication*, Portland (Oregon), 1996, pp. 87-93
3. M. Danieli, E. Gerbino e L. Moisa, "Dialogue strategies for improving the usability of telephone human-machine communications", in *Proceedings of the ACL-97 Workshop: Interactive Spoken Dialogue Systems*, Madrid , 1997, pp 114-120
4. L.Moisa, C.Pinton, C. Popovici, "EasyDial: A Tool for Task-Oriented Dialogue Systems over the Telephone", Proceedings of the IEEE 9th International Workshop on "Database and Expert Systems Applications", Wien, 1998, pp. 176-181