Design of a Generation Component for a Spoken Dialogue System

Graham Wilcock
University of Helsinki
00014 Helsinki, Finland
graham.wilcock@helsinki.fi

Kristiina Jokinen
University of Art and Design Helsinki
00560 Helsinki, Finland
kristiina.jokinen@uiah.fi

Abstract

The paper describes the design of a generation component for a spoken dialogue system. After discussing three existing models of generation, a model based on new information focus is adopted. The design uses a pipeline architecture implemented in XML and Java. XSL templates are used to give a flexible form of template-based generation.

1 Introduction

The Interact project (Jokinen, 2001) is a collaboration between Finnish universities and IT companies which aims to develop a generic dialogue model that will overcome the rigidity of practical dialogue systems and enable users to interact with applications in a natural way. The point of departure is to experiment with tools and methods that allow the system to approach various problematic situations more flexibly.

The dialogue system, depicted in Figure 1, is built on the Jaspis agent-based development architecture (Turunen and Hakulinen, 2000). On the most general level it contains managers (Input Manager, Dialogue Manager and Presentation Manager) which handle coordination between the system components and functional modules. Within each manager there are several agents which take care of the various tasks typical for the functional domain. The modules and agents communicate via a shared knowledge base (Information Storage) where all the information about the system state is kept.

The Dialogue Manager consists of several dialogue agents dealing with dialogue situations such as recognition of dialogue acts and task goals. The agents function in parallel, but the architecture also permits us to experiment with various competing agents for the same subtask: in this case a special evaluation agent chooses the one that best fits in the particular situation (Jokinen et al., 2001). The decision about the system's next act is determined by the dialogue controller agent that collects the information produced by the other agents, and constructs an Agenda, a list of concepts that is transformed into a natural language expression by the generation agents. The interface between the dialogue manager and the generation component is described in more detail by Jokinen and Wilcock (2001).

In this paper we describe the design and implementation of an experimental generation component. The design is relatively orthodox, having a pipeline architecture and using template-based generation methods. The implementation uses Java and XML.

In Section 2 we describe three existing models of generation, and discuss whether they are appropriate for a spoken dialogue system. In Section 3 we describe the new information-based model of generation which we adopt. In Section 4 we describe the design of the generation component and the XML-based implementation.

2 Three Models of Generation

In this section we describe three models of generation, which originate from work in machine translation, text generation, and telephone dialogue systems respectively.

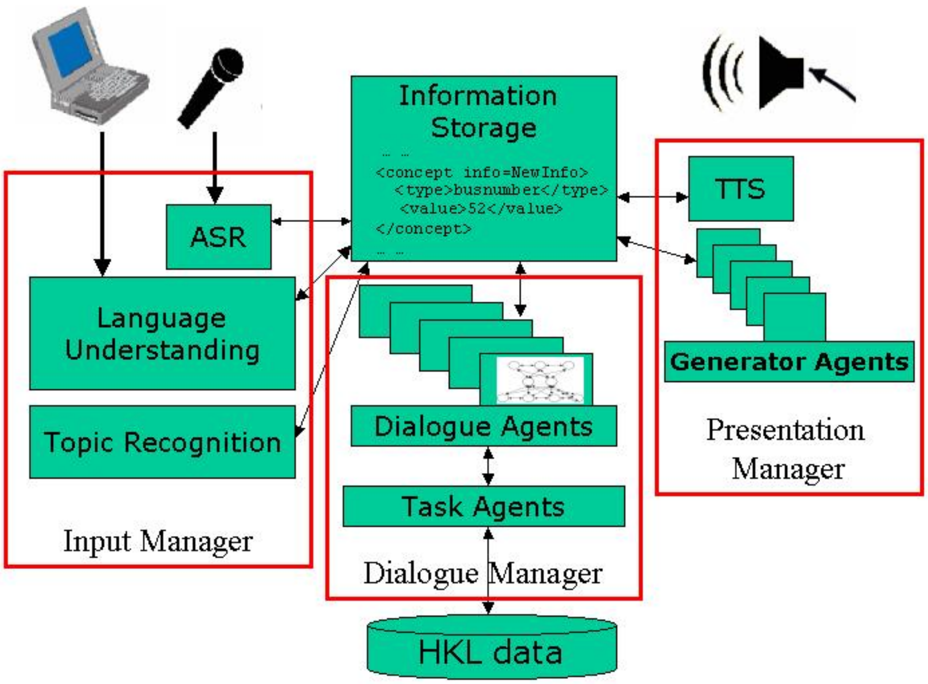

Figure 1: Interact Dialogue System Architecture

2.1 A Machine Translation Model

Machine translation systems often concentrate on analysis and transfer problems, so the generation stage is rather simple. Other systems use bidirectional grammars with a bag generation algorithm. Generation starts from a bag of items, which may be semantic terms in a logical form (Phillips, 1993), or lexical items (Carroll et al., 1999). In both cases, an essential property of the generation algorithm is that the order of information in the bag is not significant. It has no effect on surface realization order.

In machine translation, treating the source information as unordered is attractive, as the surface order in the target language should not be tied to the surface order in the source language. The usual approach in bag generation has therefore been to generate all possible grammatical sentences (all sentences licensed by the target language grammar) which match the source information (i.e. use all and only the items in the source bag). This produces a combinatorial explosion of different sentences which are grammatically legal. The emphasis of research efforts in bag generation has therefore been on finding ways to increase the efficiency of the algorithms (Carroll et al., 1999).

In this model, the generation problem has been seen either as purely syntactic (find all syntactically permitted sentences which use the given lexical items), or as a relationship between semantics and syntax (find all syntactically permitted sentences which realize the given semantic terms). Usually, information structure is ignored. This model does not attempt to select the best realization from the multiple possible alternatives, if they are all syntactically equally correct.

The form of bag generation described by Phillips (1993) for use in machine translation has also been applied to dialogue response generation. The algorithm is suitable for incremental generation, and an implementation of an incremental version of the algorithm is described by Wilcock (1998).

In this incremental model of generation, the initial bag of terms is incomplete, and more semantic terms are added after generation has started. The generator starts generating from the incomplete bag, and must find a way to include the additional semantic terms when they are supplied. In this model, although the order of terms inside the bag is not significant, the order in which terms are added influences the utterance very strongly.

Wilcock (1998) suggests *utterance rules* to control the behaviour of the generator. From an incomplete bag and a partial utterance, the generator attempts to continue the utterance as further semantic terms are added. When the generator cannot find a way to continue, a repair is performed. Like the underlying bag model, however, this incremental model does not take information structure explicitly into account as a significant factor in generation.

2.2 A Text Generation Model

In the model of generation adopted in text generation systems (Reiter and Dale, 2000), information structure is recognised as a major factor. This model usually has a pipeline architecture, in which some stages explicitly deal with text planning and referring expressions, ensuring that topic shifts and old and new information are properly handled.

We adopt a version of this pipeline architecture in our generation component, suitably adapted for a spoken dialogue system. Content determination is handled by the dialogue manager, and text planning is rather simple. In a dialogue system, as Santamarta (1999) says, "the length of the generated text is rather short, namely a turn. A turn can consist of one word up to a couple of sentences."

On the other hand, spoken dialogue systems impose additional requirements on generation, in particular for handling prosody (Santamarta, 1999; Theune, 2000), which do not arise in text generation systems.

2.3 A Fixed Dialogue Model

The third model of generation for dialogue systems is based on using a dialogue grammar or a dialogue description language, such as VoiceXML (VoiceXML Forum, 2000).

This model, however, suffers from the disadvantage that it tends to increase the rigidity of the system, by enforcing a form-filling approach which makes the user fit in with the system's demands. This is the exact opposite of the aim of the Interact project, namely to increase the adaptivity and flexibility of the system to meet the user's needs. Adaptivity is one of the issues addressed by Jokinen and Wilcock (2001).

3 A Spoken Dialogue System

In this section we describe a generation model which is designed for highly interactive spoken dialogue systems. We illustrate it with simple examples in English, equivalent to examples from the Interact project domain.

3.1 Generation from NewInfo

The model of generation which we adopt is described by Jokinen et al. (1998). Response planning starts from the new information focus, called *NewInfo.* The generator decides how to present NewInfo to the user: whether to presented it by itself or whether to wrap it in appropriate linking information.

(1) User: *Which bus goes to Malmi?*
System: *Number 74.*

(2) User: *How do I get to Malmi?*
System: *By bus - number 74.*

In example (1) NewInfo is the information about the bus number, while in (2) NewInfo concerns the means of transportation. In both cases, NewInfo is presented to the user by itself, without linking to the Topic.

(3) *When will the next bus leave for Malmi?*
(a) *2.20pm.*
(b) *It will leave at 2.20pm.*
(c) *The next bus to Malmi leaves at 2.20pm.*

Whether NewInfo should be wrapped or not depends on the changing dialogue context. When the context permits a fluent exchange of contributions, wrapping is avoided and the response is based on NewInfo only, as in example (3a). When the context requires more clarity and explicitness, NewInfo is wrapped by Topic information as in (3b) in order to avoid misunderstanding. When the communication channel is working well, wrapping can be reduced, but when there are uncertainties about what was actually said, wrapping must be increased as in example (3c) to provide implicit confirmation. These examples are discussed in more detail by Jokinen and Wilcock (2001).

3.2 The Agenda

The dialogue manager creates an *Agenda*, a set of concepts which it marks with Topic and NewInfo tags. All information in the system is held in XML format. Here is the Agenda for response (2) *By bus - number 74*:

```
<agenda>
  <concept info="NewInfo">
    <type>means-of-transportation</type>
    <value>bus</value>
  </concept>
  <concept info="Topic">
    <type>destination</type>
    <value>malmi</value>
  </concept>
  <concept info="NewInfo">
    <type>bus</type>
    <value>exists</value>
  </concept>
  <concept info="NewInfo">
    <type>busnumber</type>
    <value>74</value>
  </concept>
</agenda>
```

The generator can freely use the tagged pieces of information in order to realise the system's intention as a surface string, but it is not forced to include in the response all the concepts that the dialogue manager has designated as relevant in the agenda. Thus the dialogue manager and the generator communicate via the specifically marked conceptual items in the shared knowledge-base, but they both make their own decisions on the basis of their own reasoning and task management. The dialogue manager need not know about particular rules of surface realisation while the generator need not know how to decide the information status of the concepts in the current dialogue situation.

If the real-time requirements of the system allow sufficient time, the generator can decide on the optimum way to wrap the new information, but if there is extreme urgency to produce a response, the generator can simply give the new information without wrapping it. If this leads to misunderstanding, the system can attempt to repair this in subsequent exchanges. In this sense, the model offers an *any-time* algorithm.

4 An XML-based Implementation

In this section we describe the implementation of the generation component. The implemented system, based on Java, XML and XSL transformations, can generate responses like the examples in section 3.1 Further details of the implementation are discussed by Wilcock (2001).

4.1 A Pipeline Architecture

The generator starts from an agenda specified in XML by the dialogue manager, as shown in section 3.2, and produces a response which is also specified in XML, to be passed to the speech synthesizer as shown in section 4.3. We are therefore generating XML from XML. The simplest way to do this is to apply a set of XML transformations specified in XSL (XML Stylesheet Language). We do this using the Xalan XSL Processor (Apache XML Project, 2001).

With Xalan it is easy to set up a sequence of transformations, in which the output of one transformation becomes the input to the next transformation. This kind of "piping" is a rather natural way to implement the pipeline architecture which is so frequently used in natural language generation systems as described by Reiter and Dale (2000).

4.2 Template-based Generation

Following the NewInfo-based model of generation, the aggregation stage selects those concepts marked as NewInfo as the basis for generation, and also decides whether NewInfo will be the only output, or whether it will be preceded by the Topic linking concepts. In order to implement detailed syntactic ordering, we use *aggregation templates* as a form of sentence plan specification.

The role of template-based generation has been debated in the NLG community (Becker and Busemann, 1999). We adopt a form of template-based generation using aggregation templates, which include linguistic features and slots which can be filled in flexible ways. Our aggregation templates are similar in some ways to the syntactic templates described by Theune (2000).

Appropriate templates are selected by the aggregation stage, based on the concept types in the agenda and their status as Topic or NewInfo. The aggregation templates are implemented by means of XSL named templates. The use of XSL templates is a rather natural way to implement NLG templates.

In the aggregation stage, variable slots in the aggregation templates are filled simply by copying concepts from the agenda into the slots, pending further processing by the lexicalization and referring expression stages.

In the lexicalization and referring expression stages of the pipeline, the concepts in the aggregation templates are replaced by lexical items and referring expressions using XML-to-XML transformations. The transformations can be implemented by Java programs using the DOM model, or by XSL stylesheets, as discussed in section 4.5.

4.3 Speech Markup

The final stages of response generation perform syntactic and morphological realization and produce an XML output for the speech synthesizer in Java Speech Markup Language (Sun Microsystems Inc, 1997). The response *By bus - number 74* in example (2) is marked up in JSML as follows.

```
<jsml>
  <sent> By
    <emp> bus </emp>
  </sent>
  <break size="large"/>
  <sent> Number
    <sayas class="number"> 74 </sayas>
  </sent>
</jsml>
```

Here `<sent>` marks sentence boundaries, `<emp>` shows that the word *bus* should be spoken with emphasis, `<break>` shows that a pause is required before the second part of the response, and `<sayas>` tells the speech synthesizer that 74 should be pronounced "seventy-four".

4.4 XSL or Java?

The ease of setting up a pipeline architecture with XSL raises the question of whether XSL transformations are suitable for general use in NLG systems, or whether we need general purpose programming languages or other NLG tools. The problem with using general purpose programming languages is of course that this requires the participation of skilled programmers. One of the major advantages of XSL is that it is *not* a general-purpose programming language, but falls perhaps into the category of scripting languages which can be used successfully by non-programmers.

In any case, XSL can be combined with general purpose programming languages by embedding *extension functions* in the XSL templates. These functions can be written in Java (Apache XML Project, 2001). This means that even where general purpose programming languages are required for specific purposes, such as complex morphology, a pipeline of XSL transformations can still be used as a general framework. This is discussed further by Wilcock (2001).

4.5 TrAX, DOM or Translets?

We have experimented with three different implementations of the design described here. In one implementation, the XML-to-XML transformations required at each stage of the pipeline are specified by means of XSL templates in XSL stylesheets which are processed by the Xalan processor. As the same unchanged stylesheets are parsed again, every time a dialogue response is generated, this implementation is too slow for use in a real-time dialogue system.

In a second implementation, the XML-to-XML transformations at each stage of the pipeline are performed by Java programs which manipulate the XML structures using the DOM document model. This implementation using compiled Java code is therefore fast, and can be used in a real-time dialogue system. However, the programs require detailed navigation around the nodes in the document tree structures using the DOM API, and of course the use of Java requires the appropriate object-oriented programming skills, so one of the main advantages of using XML and XSL is lost.

The third implementation uses Translets. Translets, developed by Sun Microsystems in the XSLTC compiler, are currently being integrated into the Xalan processor (Apache XML Project, 2001). Translets are compiled from XSL stylesheets into Java classes (bytecode). The translets, after they have been developed and tested, only need to be compiled once and can then be executed repeatedly, giving the same speed as other compiled Java classes.

This appears to combine the advantages of both approaches. The XML transformations are written in the form of XSL stylesheets, avoiding the need for skilled Java programming and low-level node navigation, but the resulting translets offer high speed because they are already compiled, avoiding the need for repeated stylesheet parsing.

5 Conclusion

We have described the design of a generation component for a spoken dialogue system. The design combines an orthodox NLG pipeline architecture and template-based generation with a model of response planning from new information focus and an XML-based implementation. Further details and discussions are available in the other papers cited.

References

Apache XML Project. 2001. Xalan-Java (latest version). http://xml.apache.org/xalan-j/.

Tilman Becker and Stephan Busemann, editors. 1999. *May I Speak Freely? Between Templates and Free Choice in Natural Language Generation.* Proceedings of the KI-99 Workshop. DFKI Saarbrücken.

John Carroll, Anne Copestake, Dan Flickinger, and Victor Poznanski. 1999. An efficient chart generator for (semi-)lexicalist grammars. In *7th European Workshop on Natural Language Generation.*

Kristiina Jokinen and Graham Wilcock. 2001. Confidence-based adaptivity in response generation for a spoken dialogue system. In *Proceedings of the 2nd SIGdial Workshop on Discourse and Dialogue*, pages 80–89, Aalborg, Denmark.

Kristiina Jokinen, Hideki Tanaka, and Akio Yokoo. 1998. Planning dialogue contributions with new information. In *Proceedings of the Ninth International Workshop on Natural Language Generation*, pages 158–167, Niagara-on-the-Lake, Ontario.

Kristiina Jokinen, Topi Hurtig, Kevin Hynnä, Kari Kanto, Mauri Kaipainen, and Antti Kerminen. 2001. Self-organizing dialogue management. In *Proceedings of 2nd Workshop on NLP and Neural Networks*, NLPRS-2001, Tokyo.

Kristiina Jokinen. 2001. The Interact project. *Elsnews*, 10.2:10.

John Phillips. 1993. Generation of text from logical formulae. *Machine Translation*, 8:209–235.

Ehud Reiter and Robert Dale. 2000. *Building Natural Language Generation Systems.* Cambridge University Press.

Lena Santamarta. 1999. Output generation in a spoken dialogue system. In Becker and Busemann (1999), pages 58–63.

Sun Microsystems Inc. 1997. Java Speech Markup Language Specification, version 0.5.

Mariët Theune. 2000. *From Data to Speech: Language Generation in Context.* Ph.D. thesis, Eindhoven University of Technology.

Markku Turunen and Jaakko Hakulinen. 2000. Jaspis - a framework for multilingual adaptive speech applications. In *Proceedings of 6th International Conference on Spoken Language Processing*, Beijing.

VoiceXML Forum. 2000. Voice eXtensible Markup Language VoiceXML, version 1.00. http://www.voicexml.org/spec.html.

Graham Wilcock. 1998. Approaches to surface realization with HPSG. In *Proceedings of the Ninth International Workshop on Natural Language Generation*, pages 218–227, Niagara-on-the-Lake, Ontario.

Graham Wilcock. 2001. Pipelines, templates and transformations: XML for natural language generation. In *Proceedings of 1st NLP and XML Workshop*, NLPRS-2001, Tokyo.