Multilingual authoring using feedback texts

Richard Power and **Donia Scott**
ITRI, University of Brighton
Lewes Road, Brighton BN2 4AT, UK
FirstName.LastName@itri.bton.ac.uk

Abstract

There are obvious reasons for trying to automate the production of multilingual documentation, especially for routine subject-matter in restricted domains (e.g. technical instructions). Two approaches have been adopted: Machine Translation (MT) of a source text, and Multilingual Natural Language Generation (M-NLG) from a knowledge base. For MT, information extraction is a major difficulty, since the meaning must be derived by analysis of the source text; M-NLG avoids this difficulty but seems at first sight to require an expensive phase of knowledge engineering in order to encode the meaning. We introduce here a new technique which employs M-NLG during the phase of knowledge editing. A 'feedback text', generated from a possibly incomplete knowledge base, describes in natural language the knowledge encoded so far, and the options for extending it. This method allows anyone speaking one of the supported languages to produce texts in all of them, requiring from the author only expertise in the subject-matter, not expertise in knowledge engineering.

1 Introduction

The production of multilingual documentation has an obvious practical importance. Companies seeking global markets for their products must provide instructions or other reference materials in a variety of languages. Large political organizations like the European Union are under pressure to provide multilingual versions of official documents, especially when communicating with the public. This need is met mostly by human translation: an author produces a source document which is passed to a number of other people for translation into other languages.

Human translation has several well-known disadvantages. It is not only costly but time-consuming, often delaying the release of the product in some markets; also the quality is uneven and hard to control (Hartley and Paris, 1997). For all these reasons, the production of multilingual documentation is an obvious candidate for automation, at least for some classes of document. Nobody expects that automation will be applied in the foreseeable future for literary texts ranging over wide domains (e.g. novels). However, there is a mass of non-literary material in restricted domains for which automation is already a realistic aim: instructions for using equipment are a good example.

The most direct attempt to automize multilingual document production is to replace the human translator by a machine. The source is still a natural language document written by a human author; a program takes this source as input, and produces an equivalent text in another language as output. Machine translation has proved useful as a way of conveying roughly the information expressed by the source, but the output texts are typically poor and over-literal. The basic problem lies in the analysis phase: the program cannot extract from the source all the information that it needs in order to produce a good output text. This may happen either because the source is itself poor (e.g. ambiguous or incomplete), or because the source uses constructions and concepts that lie outside the program's range. Such problems can be alleviated to some extent by constraining the source document, e.g. through use of a 'Controlled Language' such as AECMA (1995).

An alternative approach to translation is that of generating the multilingual documents from a *non-linguistic* source. In the case of automatic Multilingual Natural Language Generation (M-

NLG), the source will be a knowledge base expressed in a formal language. By eliminating the analysis phase of MT, M-NLG can yield high-quality output texts, free from the 'literal' quality that so often arises from structural imitation of an input text. Unfortunately, this benefit is gained at the cost of a huge increase in the difficulty of obtaining the source. No longer can the domain expert author the document directly by writing a text in natural language. Defining the source becomes a task akin to building an expert system, requiring collaboration between a domain expert (who understands the subject-matter of the document) and a knowledge engineer (who understands the knowledge representation formalism). Owing to this cost, M-NLG has been applied mainly in contexts where the knowledge base is already available, having been created for another purpose (Iordanskaja et al., 1992; Goldberg et al., 1994); for discussion see Reiter and Mellish (1993).

Is there any way in which a domain expert might author a knowledge base without going through this time-consuming and costly collaboration with a knowledge engineer? Assuming that some kind of mediation is needed between domain expert and knowledge formalism, the only alternative is to provide easier tools for editing knowledge bases. Some knowledge management projects have experimented with graphical presentations which allow editing by direct manipulation, so that there is no need to learn the syntax of a programming language – see for example Skuce and Lethbridge (1995). This approach has also been adopted in two M-NLG systems: GIST (Power and Cavallotto, 1996), which generates social security forms in English, Italian and German; and DRAFTER (Paris et al., 1995), which generates instructions for software applications in English and French. These projects were the first attempts to produce *symbolic authoring* systems – that is, systems allowing a domain expert with no training in knowledge engineering to author a knowledge base (or *symbolic source*) from which texts in many languages can be generated.

Although helpful, graphical tools for managing knowledge bases remain at best a compromise solution. Diagrams may be easier to understand than logical formalisms, but they still lack the flexibility and familiarity of natural language text, as empirical studies on editing diagrammatic representations have shown (Kim, 1990; Petre, 1995); for discussion see Power et al. (1998). This observation has led us to explore a new possibility, at first sight paradoxical: that of a symbolic authoring system in which the current knowledge base is presented through a natural language text *generated by the system.* This kills two birds with one stone: the source is still a knowledge base, not a text, so no problem of analysis arises; but this source is presented to the author in natural language, through what we will call a *feedback text.* As we shall see, the feedback text has some special features which allow the author to edit the knowledge base as well as viewing its contents. We have called this editing method 'WYSIWYM', or 'What You See Is What You Meant': a natural language text ('what you see') presents a knowledge base that the author has built by purely semantic decisions ('what you meant').

A basic WYSIWYM system has three components:

- A module for building and maintaining knowledge bases. This includes a 'T-Box' (or 'terminology'), which defines the concepts and relations from which assertions in the knowledge base (or 'A-Box') will be formed.
- Natural language generators for the languages supported by the system. As well as producing output texts from complete knowledge bases, these generators will produce feedback texts from knowledge bases in any state of completion.
- A user interface which presents output or feedback texts to the author. The feedback texts will include mouse-sensitive 'anchors' allowing the author to make semantic decisions, e.g. by selecting options from pop-up menus.

The WYSIWYM system allows a domain expert speaking any *one* of the supported languages to produce good output texts in *all* of them. A more detailed description of the architecture is given in Scott et al. (1998).

2 Example of a WYSIWYM system

The first application of WYSIWYM was DRAFTER-II, a system which generates instructions for using word processors and diary managers. At present three languages are supported: English, French and Italian. As an example, we will follow a session in which the author encodes instructions for scheduling an appointment with the OpenWindows Calendar Manager. The desired content is shown by the following output text, which the system will generate when the knowledge base is complete:

> **To schedule the appointment:**
> Before starting, open the Appointment Editor window by choosing the Appointment option from the Edit menu.
> Then proceed as follows:
>
> 1 Choose the start time of the appointment.
> 2 Enter the description of the appointment in the What field.
> 3 Click on the Insert button.

In outline, the knowledge base underlying this text is as follows. The whole instruction is represented by a `procedure` instance with two attributes: a `goal` (scheduling the appointment) and a `method`. The `method` instance also has two attributes: a `precondition` (expressed by the sentence beginning 'Before starting') and a sequence of `steps` (presented by the enumerated list). Preconditions and steps are procedures in their turn, so they may have methods as well as goals. Eventually we arrive at sub-procedures for which no method is specified: it is assumed that the reader of the manual will be able to click on the Insert button without being told how.

Since in DRAFTER-II every output text is based on a procedure, a newly initialised knowledge base is seeded with a single `procedure` instance for which the goal and method are undefined. In Prolog notation, we can represent such a knowledge base by the following assertions:

```
procedure(proc1).
goal(proc1, A).
method(proc1, B).
```

Here `proc1` is an identifier for the `procedure` instance; the assertion `procedure(proc1)` means that this is an instance of type `procedure`; and the assertion `goal(proc1, A)` means that `proc1` has a goal attribute for which the value is currently undefined (hence the variable `A`).

When a new knowledge base is created, DRAFTER-II presents it to the author by generating a feedback text in the currently selected language. Assuming that this language is English, the instruction to the generator will be

```
generate(proc1, english, feedback)
```

and the feedback text displayed to the author will be

> Achieve **this goal** by applying *this method.*

This text has several special features.

- Undefined attributes are shown through *anchors* in bold face or italics. (The system actually uses a colour code: red instead of bold face, and green instead of italics.)
- A red anchor (bold face) indicates that the attribute is obligatory: its value must be specified. A green anchor (italics) indicates that the attribute is optional.
- All anchors are mouse-sensitive. By clicking on an anchor, the author obtains a pop-up menu listing the permissible values of the attribute; by selecting one of these options, the author updates the knowledge base.

Although the anchors may be tackled in any order, we will assume that the author proceeds from left to right. Clicking on **this goal** yields the pop-up menu

choose
click
close
create
.....
save
schedule
start

(to save space, this figure omits some options), from which the author selects 'schedule'. Each option in the menu is associated with an 'updater', a Prolog term (not shown to the author) that specifies how the knowledge base should be updated if the option is selected. In this case the updater is

```
insert(proc1, goal, schedule)
```

meaning that an instance of type `schedule` should become the value of the `goal` attribute on `proc1`. Running the updater yields an extended knowledge base, including a new instance `sched1` with an undefined attribute `actee`. (Assertions describing attribute values are indented to make the knowledge base easier to read.)

```
procedure(proc1).
goal(proc1, sched1).
  schedule(sched1).
  actee(sched1, C).
method(proc1, B).
```

From the updated knowledge base, the generator produces a new feedback text.

> Schedule **this event** by applying *this method*.

Note that this text has been completely regenerated. It was not produced from the previous text merely by replacing the anchor **this goal** by a longer string.

Continuing to specify the goal, the author now clicks on **this event**.

appointment
meeting
.....

This time the intended selection is 'appointment', but let us assume that by mistake the author drags the mouse too far and selects 'meeting'. The feedback text

> Schedule the meeting by applying *this method*.

immediately shows that an error has been made, but how can it be corrected? This problem is solved in WYSIWYM by allowing the author to select any span of the feedback text that represents an attribute with a specified value, and to cut it, so that the attribute becomes undefined, while its previous value is held in a buffer. Even large spans, representing complex attribute values, can be treated in this way, so that complex chunks of knowledge can be copied across from one knowledge base to another. When the author selects the phrase 'the meeting', the system displays a pop-up menu with two options:

Cut
Copy

By selecting 'Cut', the author activates the updater

```
cut(sched1, actee)
```

which updates the knowledge base by removing the instance `meet1`, currently the value of the `actee` attribute on `sched1`, and holding it in a buffer. With this attribute now undefined, the feedback text reverts to

> Schedule **this event** by applying *this method*.

whereupon the author can once again expand **this event**. This time, however, the pop-up menu that opens on this anchor will include an extra option: that of pasting back the material that has just been cut. Of course this option is only provided if the instance currently held in the buffer is a suitable value for the attribute represented by the anchor.

Paste
appointment
meeting
.....

The 'Paste' option here will be associated with the updater

```
paste(sched1, actee)
```

which would assign the instance currently in the buffer, in this case `meet1`, as the value of the `actee` attribute on `sched1`. Fortunately the author avoids reinstating this error, and selects 'appointment', yielding the following reassuring feedback text:

> Schedule the appointment by applying *this method*.

Note incidentally that this text presents a knowledge base that is *potentially complete*, since all obligatory attributes have been specified. This can be immediately seen from the absence of any red (bold) anchors.

Intending to add a method, the author now clicks on *this method*. In this case, the pop-up menu shows only one option:

method

Running the associated updater yields the following knowledge base:

```
procedure(proc1).
goal(proc1, sched1).
  schedule(sched1).
  actee(sched1, appt1).
    appointment(appt1).
method(proc1, method1).
  method(method1).
  precondition(method1, D).
  steps(method1, steps1).
    steps(steps1).
    first(steps1, proc2).
      procedure(proc2).
      goal(proc2, F).
      method(proc2, G).
    rest(steps1, E).
meeting(meet1).
```

A considerable expansion has taken place here because the system has been configured to automatically instantiate obligatory attributes that have only one permissible type of value. (In other words, it never presents red anchors with pop-up menus having only one option.) Since the `steps` attribute on `method1` is obligatory, and must have a value of type `steps`, the instance `steps1` is immediately created. In its turn, this instance has the attributes `first` and `rest` (it is a list), where `first` is obligatory and must be filled by a procedure. A second procedure instance `proc2` is therefore created, with its own goal and method. To incorporate all this new material, the feedback text is recast in a new pattern, the main goal being expressed by an infinitive construction instead of an imperative:

> To schedule the appointment:
> First, achieve *this precondition.*
> Then follow these steps.
>
> 1 Perform **this action** by applying *this method.*
> 2 *More steps.*

Note that at any stage the author can switch to one of the other supported languages, e.g. French. This will result in a new call to the generator

```
generate(proc1, french, feedback)
```

and hence in a new feedback text expressing the procedure `proc1`.

> Insertion du rendez-vous:
> Avant de commencer, accomplir *cette tâche.*
> Exécuter les actions suivantes.
>
> 1 Exécuter **cette action** en appliquant *cette méthode.*
> 2 *Autres sous-actions.*

Clicking for example on **cette action** will now yield the usual options for instanciating a goal attribute, but expressed in French. The associated updaters are identical to those for the corresponding menu in English.

choix
cliquer
fermer
.....
enregistrement
insertion
lancement

The basic mechanism should now be clear, so let us advance to a later stage in which the scheduling procedure has been fully encoded.

> To schedule the appointment:
> First, open the Appointment Editor window.
> Then follow these steps.
>
> 1 Choose the start time of the appointment by applying *this method.*
> 2 Enter the description of the appointment in the What field by applying *this method.*
> 3 Click on the Insert button by applying *this method.*
> 4 *More steps.*
>
> To open the Appointment Editor window:
> First, achieve *this precondition.*
> Then follow these steps.
>
> 1 Choose the Appointment option from the Edit menu by applying *this method.*
> 2 *More steps.*

Two points about this feedback text are worth noting. First, to avoid overcrowding the main paragraph, the text planner has deferred the sub-procedure for opening the Appointment Editor window, which is presented in a separate paragraph. To maintain a connection, the action of opening the Appointment Editor window is mentioned twice (as it happens, through different constructions). Secondly, no red (bold) anchors are left, so the knowledge base is potentially complete. (Of course it could be extended further, e.g. by adding more steps.) This means that the author may now generate an output text by switching the modality from 'Feedback' to 'Output'. The resulting instruction to the generator will be

```
generate(proc1, english, output)
```

yielding the output text shown at the beginning of the section. Further output texts can be obtained by switching to another language, e.g. French:

> Insertion du rendez-vous:
> Avant de commencer, ouvrir la fenêtre Appointment Editor en choisissant l'option Appointment dans le menu Edit.
> Exécuter les actions suivantes:
> 1 Choisir l'heure de fin du rendez-vous.
> 2 Insérer la description du rendez-vous dans la zone de texte What.
> 3 Cliquer sur le bouton Insert.

Note that in output modality the generator ignores optional undefined attributes; the method for opening the Appointment Editor window thus reduces to a single action which can be re-united with its goal in the main paragraph.

3 Significance of WYSIWYM editing

WYSIWYM editing is a new idea that requires practical testing. We have not yet carried out formal usability trials, nor investigated the design of feedback texts (e.g. how best to word the anchors), nor confirmed that adequate response times could be obtained for full-scale applications. However, if satisfactory large-scale implementations prove feasible, the method brings many potential benefits.

- A document in natural language (possibly accompanied by diagrams) is the most flexible existing medium for presenting information. We cannot be sure that all meanings can be expressed clearly in network diagrams or other specialized presentations; we can be sure they can be expressed in a document.
- It seems intuitively obvious that authors will understand feedback texts much better than they understand alternative methods of presenting knowledge bases, such as network diagrams. Our experience has been that people can learn to use the DRAFTER-II system in a few minutes.
- Authors require no training in a controlled language or any other presentational convention. This avoids the expense of initial training; it also means that presentational conventions need not be relearned when a knowledge base is re-examined after a delay of months or years.
- Since the knowledge base is presented through a document in natural language, it becomes immediately accessible to anyone peripherally concerned with the project (e.g. management, public relations, domain experts from related projects). Documentation of the knowledge base, often a tedious and time-consuming task, becomes automatic.
- The model can be viewed and edited in any natural language that is supported by the generator; further languages can be added as needed. When supported by a multilingual natural language generation system, as in DRAFTER-II, WYSIWYM editing obviates the need for traditional language localisation of the human-computer interface. New linguistic styles can also be added (e.g. a terminology suitable for novices rather than experts).
- As a result, WYSIWYM editing is ideal for facilitating knowledge sharing and transfer within a multilingual project. Speakers of several different languages could collectively edit the same knowledge base, each user viewing and modifying the knowledge in his/her own language.
- Since the knowledge base is presented as a document, large knowledge bases can be

navigated by the methods familiar from books and from complex electronic documents (e.g. contents page, index, hypertext links), obviating any need for special training in navigation.

The crucial advantage of WYSIWYM editing, compared with alternative natural language interfaces, is that it eliminates all the usual problems associated with parsing and semantic interpretation. Feedback texts with menus have been used before in the NL-Menu system (Tennant et al., 1983), but only as a means of presenting *syntactic* options. NL-Menu guides the author by listing the extensions of the current sentence that are covered by its grammar; in this way it makes parsing more reliable, by enforcing adherence to a sub-language, but parsing and interpretation are still required.

So far WYSIWYM editing has been implemented in two domains: software instructions (as described here), and patient information leaflets. We are currently evaluating the usability of these systems, partly to confirm that authors do indeed find them easy to use, and partly to investigate issues in the design of feedback texts.

References

AECMA. 1995. AECMA Simplified English: A guide for the preparation of aircraft maintenance documentation in the International Aerospace Maintenance Language. AECMA, Brussels.

E. Goldberg, N. Driedger, and R. Kittredge. 1994. Using natural-language processing to produce weather forcasts. *IEEE Expert*, 9(2):45–53.

Anthony F. Hartley and Cécile L. Paris. 1997. Multilingual document production: From support for translating to support for authoring. *Machine Translation, Special Issue on New Tools for Human Translators*, 12(1-2):109–129.

L. Iordanskaja, M. Kim, R. Kittredge, B. Lavoie, and A. Polguere. 1992. Generation of extended bilingual statistical reports. In *Proceedings of the 14th International Conference on Computational Linguistics*, pages 1019–1023, Nantes.

Y. Kim. 1990. Effects of conceptual data modelling formalisms on user validation and analyst modelling of information requirements. PhD thesis, University of Minnesota.

Cécile Paris, Keith Vander Linden, Markus Fischer, Anthony Hartley, Lyn Pemberton, Richard Power, and Donia Scott. 1995. A support tool for writing multilingual instructions. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 1398–1404, Montreal, Canada.

M. Petre. 1995. Why looking isn't always seeing: readership skills and graphical programming. *Communications of the ACM*, 38(6):33–42.

R. Power and N. Cavallotto. 1996. Multilingual generation of administrative forms. In *Proceedings of the 8th International Workshop on Natural Language Generation*, pages 17–19, Herstmonceux Castle, UK.

R. Power, D. Scott, and R. Evans. 1998. What you see is what you meant: direct knowledge editing with natural language feedback. In *Proceedings of the 13th Biennial European Conference on Artificial Intelligence*, Brighton, UK.

Ehud Reiter and Chris Mellish. 1993. Optimizing the costs and benefits of natural language generation. In *Proceedings of the International Joint Conference on Artificial Intelligence, Chamberry France*, pages 1164–1169.

D. Scott, R. Power, and R. Evans. 1998. Generation as a solution to its own problem. In *Proceedings of the 9th International Workshop on Natural Language Generation*, Niagara-on-the-Lake, Canada.

D. Skuce and T. Lethbridge. 1995. CODE4: A unified system for managing conceptual knowledge. *International Journal of Human-Computer Studies*, 42:413–451.

H. Tennant, K. Ross, R. Saenz, C. Thompson, and J. Miller. 1983. Menu-based natural language understanding. In *Proceedings of the Association of Computational Linguistics.*