Generating Natural Language Descriptions of Project Plans

Margaret Wasko and Robert Dale

Language Technology Group
Division of Information and Communication Sciences
Macquarie University, Sydney, NSW 2109
Australia
{mwasko|rdale}@ics.mq.edu.au

Abstract. We often resort to graphical means in order to describe non-linear structures, such as task dependencies in project planning. There are many contexts, however, where graphical means of presentation are not appropriate, and delivery either via text or spoken language is to be preferred. In this work, we take some first steps towards the development of natural language generation techniques that seek the most appropriate means of expressing non-linear structures using the linear medium of language.

1 Introduction

Natural language generation—the use of natural language processing techniques to create textual or spoken output from some underlying non-linguistic information source—is an area of practical language technology that shows great potential. Various natural language generation (NLG) systems have been constructed which produce textual output from underlying data sources of varying kinds: for example, the FoG system [3] generates textual weather forecasts from numerical weather simulations; IDAS [5] produces online hypertext help messages for users of complex machinery, using information stored in a knowledge base that describes this machinery; ModelExplainer [4] generates textual descriptions of information in models of object-oriented software; and PEBA [7] interactively describes entities in a taxonomic knowledge base via the dynamic generation of hypertext documents, presented as World Wide Web pages.

The present work represents the first steps in exploring how NLG techniques can be used to present the information in complex, non-linear data structures. In particular, we focus on project plans of the kind that might be constructed in an application such as Microsoft Project. These software tools make it easy to present the content of project plans via a number of graphical means, such as PERT and Gantt charts. However, they do not provide any capability for presenting the information in project plans via natural language. We pursue this possibility for two reasons.

Firstly, we are interested in exploring the extent to which language can be used to express complex non-linear structures. We might hypothesise that language is not a good means for expressing this kind of information, since language requires us to linearise the presentation of the material to be expressed. However, some recent work in NLG has explored the use of mixed mode output, where graphics and text are combined; see, for example, [1]. A key question, then, is how best to apportion material between the two modalities. We intend to build on some of this recent research to see what kinds of information are best conveyed using language, and what elements are best conveyed graphically. We also aim to explore how sophisticated use of typography—indented structures, graphs containing textual annotations, and so on—can overcome some of the inherent limitations in purely 'linear' text.

Secondly, we are interested in determining the extent to which the information in a project plan might be conveyed to a user via speech. Suppose a project manager is driving to a meeting, and needs a report on the current status of some project whose internal structure is complex. Assuming that we do not have sophisticated heads-up displays or other similar presentation technologies, there is no possibility here that the information can be presented visually. In such a context, speech is the most plausible medium for information delivery, and so we are particularly interested in how the information available in a non-linear structure can most effectively be presented in a linear speech stream. Similarly, speech may be the delivery medium of choice for users who are vision-impaired.

In this paper, we present some first steps towards achieving these goals. Our focus here is on a specific but particularly important sub-problem: how do we produce descriptions of parallel structures in such a way as to avoid ambiguity in interpretation? We have implemented a simple NLG system, PLANPRESENTER, that takes project plan information as input, and produces from this information a text that describes the dependencies between the project plan elements. Section 2 presents an overview of the system, describing the key components. Section 3 shows how PLANPRESENTER generates text from a simple input project plan, and Section 4 shows how a more complex example is dealt with using our intermediate level of representation to allow the required flexibility in the generation process. Section 5 summarises the state of the work so far and sketches our next steps in this research.

2 System Overview

The system we describe here takes as its starting point earlier work described in [6], but departs from the system described there by adopting more recent ideas regarding the decomposition of the natural language generation process and the intermediate levels of representation that are required, as described in [8]. The input to our system is approximately equivalent to the information that can be extracted from a combination of the interchange formats provided by Microsoft Project, and which is likely to be available for any such project management tool. More particularly, we assume that we will be provided with a set of constructs corresponding to the basic undecomposable tasks in the plan—we will call these ATOMIC ACTIONS—and a set of dependency links that indicate which tasks must be completed before other tasks.

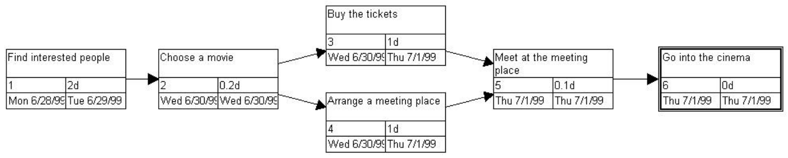

Fig. 1. A project plan for going to see a movie

It is likely that most project management systems will also make available information about other aspects of a project plan, such as the resources allocated to specific tasks, and hard constraints over temporal attributes such as start and end dates and task durations. In some contexts, we may also have access to information regarding the hierarchical relationships between plan components. We intend to make use of these elements of data as our work on project plan description proceeds; at this early stage, however, our primary concern is the linearisation of the information present in this essentially non-linear, networked structure.

The PLANPRESENTER system consists of three principal components:

– an INFORMATION STRUCTURER, which reconstructs the given plan information in a form more suited to textual description;
– a DESCRIPTION STRUCTURER, which assigns specific structural categories to all components of the plan; and
– a SURFACE REALISER, which works out how to express the content of the description structure linguistically.

The result is a set of English instructions instructions for performing that plan, written in such a way as to avoid ambiguities of understanding when parallelism occurs in the plan. The system is implemented in Prolog.

3 A Simple Worked Example

In this section we present a simple worked example that shows how PLANPRESENTER generates a description of a project plan from a symbolic representation of that plan.

3.1 The Project Plan

Figure 1 shows a PERT chart that indicates the relationships between a number of tasks within a larger project. In the example here, each task includes a start date, an end date and a duration; we will not make use of these for the moment, restricting ourselves to the standard elements of the temporal dependencies between the tasks.

The project is for a group of people to go to see a movie at the cinema together. The plan consists of six atomic actions, labelled here `a1` through `a6`:

```
action(a1, [find,people1]).           % Find interested people.
action(a2, [choose,movie1]).          % Decide what movie to see.
action(a3, [buy,tickets1]).           % Buy the tickets.
action(a4, [arrange,place1]).         % Arrange the meeting place.
action(a5, [meet,place1]).            % Meet at arranged place.
action(a6, [enter,cinema1]).          % Go into the cinema.

precedes(a1, a2).       precedes(a2, a3).         precedes(a2, a4).
precedes(a3, a5).       precedes(a4, a5).         precedes(a5, a6).
```

Fig. 2. The input representation corresponding to the project plan shown in Figure 1

first, we have to find the group of people who are interested in going, then we have to decide which movie to see, then we have to buy the tickets and arrange where to meet, and then we have to meet and go into the cinema. Note in particular that the actions of buying the tickets and of arranging a place to meet beforehand can be carried out in any order, or even in parallel.

The temporal dependencies here are indicated in the PERT chart by means of arrows. This information is presented to our system as a collection of symbolic constructs as shown in Figure 2. Here, for each action we have some additional information that will be used in describing this action: this is a pair of the form ⟨ActionType, Entity⟩, where the ActionType is drawn from an inventory of actions that the system knows how to express linguistically, and the Entity is a symbol that corresponds to some entity in the domain.[1] Given inputs of this kind, then, our goal is to generate a coherent text describing the plan in question.

3.2 Producing an Output Text

The present example is a very simple case of plan description; however, it allows us to demonstrate some of the essential elements of our method.

Building the Information Structure: First, we transform the given symbolic structures into a representation more suited to textual description. The key observation here is that language provides us with a variety of mechanisms for indicating both sequence and parallelism, so we re-express the input information in a form that highlights these relationships. Applying this process to the input data shown in Figure 2 results in the following structure:[2]

[1] There are clearly issues of specific versus non-specific reference here which complicate matters; however, our present focus is on describing the overall structure of a plan, so we will sidestep for the moment many of the issues regarding the fine-grained modelling of the entities that participate in the plan.

[2] There exist plans whose structure does not readily map to the form described here. Consider for example Figure 7 with an arrow added from action 8 to action 5. Generating descriptions of plans such as these is a topic of future work.

```
sequence([number_elements: 5],
[elements: [[1, atomic, a1],
            [2 ,atomic, a2],
            [3, simple_branch, [number_elements: 2],
                [elements: [[1, atomic, a3],
                            [2, atomic, a4]]]],
            [4, atomic, a5],
            [5, atomic, a6]]]).
```

Fig. 3. Representing sequence and simple parallelism

```
    sequence([a1, a2, parallel([a3, a4]), a5, a6]).
```

It is easy to see how, in general terms, such a structure might be mapped directly into a text:

- given a sequence of elements as in this case, we might simply express each element in the sequence by means of a sentence;
- if an element in the sequence is a parallel structure, then we might indicate explicitly that all the actions in this structure can be carried out in parallel.

Such a simple mapping mechanism will not, however, produce appropriate results in the case of more complex plans. In particular, if we have parallel structures that contain embedded parallelism or other complexities, then a direct mapping approach along the lines just sketched will result in unwieldy sentences.

Building the Initial Description Structure: In order to overcome this problem, instead of mapping the plan structure directly into text, we construct an intermediate representation which we call a DESCRIPTION STRUCTURE. This serves as an updateable repository for all the information we might need in making decisions as to how best to describe the plan. We can then perform reasoning operations over this structure to determine the best output, before committing ourselves to text. In the remainder of this section we show how the description structure is constructed and used in the present example; in Section 4 we show how this accommodates a more complex case of parallelism.

Figure 3 shows the initial description structure for our plan. Notice that here we have made explicit a number of properties of our original plan structure:

- We have explicitly indicated how many elements are present at each level in the plan structure.
- We have explicitly numbered each constituent element.
- We have explicitly indicated whether substructures in the plan are made up of atomic actions or are more complex in nature, as in the simple_branch element.

It is by virtue of this last step that our approach provides us with more sophisticated control over the description process. In essence, we identify different kinds of structural patterns in plans, where these different patterns correspond to different mechanisms for description. Thus, a simple branching structure is one where the elements within the parallelism are themselves atomic actions. Such a structure is amenable to the direct-mapping form of description suggested informally above, but more complex structures will require the use of more sophisticated linguistic mechanisms.

Determining Semantic Content: We now have to augment this description structure with additional information about the actions to be described. This is carried out as a sequence of two related processing steps. First, we incorporate information about how the actions themselves are to be described. The Action-Type in our input representation corresponds to the semantics of the predicate that will be used to describe that action, and the Entity serves as the index of the argument to the predicate. The next stage determines how the entities that participate in the plan will be referred to. For our present purposes, we do not make use of a sophisticated referring expression mechanism; essentially, we use simple table lookup to determine how a given entity should be described in a plan. At a later date we intend to incorporate more sophisticated algorithms for the generation of referring expressions along the lines described in [2]. The process of determining semantic content results in the output shown in Figure 4.

Applying Structure Realisation Strategies: Once we have determined the relevant aspects of the description of each of the actions in the plan, we are in a position to decide how to realise the overall description structure. We do this by means of STRUCTURE REALISATION STRATEGIES, which can be summarised in general terms as follows.[3]

– An action not immediately involved in a parallel description is described in a separate sentence, with appropriate adjuncts.
– Actions involved in simple parallelism are combined in a single sentence.
– actions involved in more complex parallelism are described in terms of the groupings assigned by the information structurer, with each group in a separate paragraph and signalled by appropriate adjuncts. See Section 4 for an example. Parallelism that is more embedded is signalled by means of the same strategy together with indentation.

The results of this process are shown in Figure 5. Here, we can see that the structure realisation rules for atomic actions have determined various aspects of the sentential forms to be used. By taking account of the number of the elements in the sequence, appropriate adjuncts for *first*, *then*, and *finally* are added; an alternative realisation rule might decide to use the adjuncts *second*, *third* and so on instead of *then*. The realisation rules have also determined that the imperative forms of the verbs should be used.

[3] There are additional realisation strategies available, including textual ones such as numbered lists, and the use of multiple modalities. These topics are a subject of future work.

```
sequence([number_elements: 5],
[elements: [[1, atomic,
                [index: a1,
                 predicate: [sem: find],
                  argument: [index: people1,
                   syn: [category: np],
                  text: [some,interested,people]]]],
            [2, atomic,
                [index: a2,
                 predicate: [sem: choose],
                  argument: [index: movie1,
                           syn: [category: np],
                          text: [a,movie]]]],
            [3, simple_branch, [number_elements: 2],
                [elements: [[1, atomic,
                                [index: a3,
                                 predicate: [sem: buy],
                                  argument: [index: tickets1,
                                           syn: [category: np],
                                          text: [the,tickets]]]],
                            [2, atomic,
                                [index: a4,
                                 predicate: [sem: arrange],
                                  argument: [index: place1,
                                           syn: [category: np],
                                          text: [a,meeting,place]]]]]]],
            [4, atomic,
                [index: a5,
                 predicate: [sem: meet],
                  argument: [index: place1,
                           syn: [category: np],
                          text: [the,meeting,place]]]],
            [5, atomic,
                [index: a6,
                 predicate: [sem: enter],
                  argument: [index: cinema1,
                           syn: [category: np],
                          text: [the,cinema]]]]]])
```

Fig. 4. Adding semantic content and referring expressions to the description structure

```
sequence([number_elements: 5],
[elements: [[1, atomic,
                [index: a1,
                 syn: [category: s, pre_adjunct: [first,',']],
                 predicate: [sem: find,
                             syn: [category: v,
                                   vform: imperative]],
                  argument: [index: people1,
                         syn: [category: np],
                         text: [some,interested,people]]]],
            [2, atomic,
                [index: a2,
                 syn: [category: s, pre_adjunct: [then,',']],
                 predicate: [sem: choose,
                             syn: [category: v,
                                   vform: imperative]],
                  argument: [index: movie1,
                         syn: [category: np],
                         text: [a,movie]]]],
            [3, simple_branch, [number_elements: 2],
                [syn: [category: s],
                 pre_adjunct: [then,','],
                 conjunct: [and],
                 post_adj: [',',doing,these,in,any,order,you,like],
                [elements: [[1, atomic,
                                 [index: a3,
                                  syn: [category: s],
                                  predicate: [sem: buy,
                                              syn: [category: v,
                                                    vform: imperative]],
                                   argument: [index: tickets1,
                                          syn: [category: np],
                                          text: [the,tickets]]]]
            [...]
```

Fig. 5. The result of applying realisation strategies to the description structure

```
[paragraph(1, [sentence([first,',',find,some,interested,people]),
               sentence([then,',',choose,a,movie]),
               sentence([then,',',buy,the,tickets,and,arrange,
                         a,meeting,place,',',doing,these,in,
                         any,order,you,like]),
               sentence([then,',',meet,at,the,meeting,place]),
               sentence([finally,',',go,into,the,cinema])])]
```

Fig. 6. The final set of sentence specifications

Surface Realisation Our description structure now contains enough information to be able to determine the final lexical content of our plan description. Information about the realisation of different verb forms is encoded in the system lexicon by means of entries like the following:

```
lex([category: verb, sem: find, vform:imperative,lex:find]).
lex([category: verb, sem: find, vform:progressive,lex:finding]).
```

The result of incorporating this information is a final specification for the text to be generated, as in Figure 6. These specifications are passed to a rendering module which, at present, simply uppercases the first character of the first word of each sentence and appends a full stop at the end of each sentence, and wraps the entire paragraph within appropriate HTML tags:

```
<p>
First, find some interested people.
Then, choose a movie.
Then, buy the tickets and arrange a meeting place,
doing these in any order you like.
Then, meet at the meeting place.
Finally, go into the cinema.
<\p>
```

Clearly, some improvements to the overall fluency are possible here, in particular with regard to the use of appropriate forms of subsequent. However, the key element of the system's behaviour we wish to focus on here is the use of the intermediate level of representation—the description structure—in enabling us to create textual realisations whose overall structure is coherent. In the next section we look at how this is used in a more complex example.

4 Dealing with Embedded Parallelism

4.1 The Project Plan

Figure 7 shows a PERT chart of a section of a plan dealing with housework. This part of the plan deals with cleaning the kitchen and dining area. After the task of ensuring that one has the required equipment, the tasks involved follow two main parallel branches. On one branch, we have the task of washing the dishes, followed by cleaning the stove, followed by wiping the benches, followed by mopping the kitchen floor. On the other branch we start with picking up rubbish and dusting the furniture, related to each other in simple parallelism. These two tasks are followed by vacuuming the floor, which is followed by taking out the garbage.

This plan provides an example of a structure we call EMBEDDED PARALLELISM: this occurs when the plan contains two or more collections of actions, where the ordering between these collections of actions does not matter, and where there is also parallelism within at least one of these collections of actions.

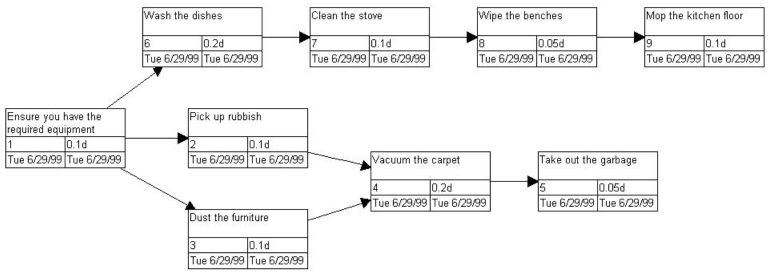

Fig. 7. A more complex plan fragment

The case shown here is also more complex than our first example above in that the two top-level parallel structures each contain more than one action in a sequence.

This plan information is provided to PlanPresenter in a form similar to that shown for our earlier example.

4.2 Producing an Output Text

Given the above input, PlanPresenter produces the following output text:

> First, ensure you have the required equipment. You are now ready for two main parts of the cleaning, which may be done in any order, or alongside each other.
> The first part is as follows. First, pick up any rubbish and dust the furniture, doing these in any order you like. Then, vacuum the carpet. Finally, take out the garbage.
> The second part is as follows. First, wash the dishes. Then, clean the stove. Then, wipe the benches. Finally, mop the kitchen floor.

Note that the parallel relationships that exist in the plan are preserved in the text.

As before, in order to generate this text we first construct an information structure as follows:

```
sequence([a1,parallel([sequence([parallel([a2,a3]),a4,a5]),
                                    sequence([a6,a7,a8,a9])])]).
```

Figure 8 shows part of the description structure that is then constructed from this representation. Note that this structure differs from our previous example in that, in building the description structure, we have recognised the presence of a complex branch.

The realisation strategies then augment this structure with relevant syntactic and semantic information as before; in this case, the presence of the complex branch results in a realisation decision that the two parts of this branch should

```
sequence([number_elements: 2],
[elements: [[1, atomic, a1],
            [2, complex_branch, [number_elements: 2],
                [elements:
               [[1, sequence, [number_elements: 3],
                    [elements: [[1, simple_branch, [number_elements: 2],
                                    [elements: [[1, atomic, a2],
                                                [2, atomic, a3]]]]
                                [2, atomic, a4],
                                [3, atomic, a5]]]]
                [2, sequence,[number_elements: 4]
                    [elements: [[1, atomic, a6],
                                [2, atomic, a7],
                                [3, atomic, a8],
                                [4, atomic, a9]]]]]]]]])
```

Fig. 8. Representing sequence and embedded parallelism

be realised by means of separate paragraphs, and that the entire text should be preceded by a sentence that indicates the overall structure of the plan.

Once complete, this description structure is then passed on to the surface realisation component, which produces the output specification shown in Figure 9.

5 Conclusions and Next Steps

In this paper, we have presented a NLG system that addresses the problem of generating English natural language descriptions of plans that contain non-linear elements. As a first step in this exercise, we have focussed on the problem of how to express parallelism at different levels of complexity. We have demonstrated how the use of an intermediate representation that encodes information about the overall structure of the plan can serve both as a updateable repository of information regarding the text to be generated, and as a structure that supports reasoning about the best ways to present that information. The resulting texts present instructions for performing the input plans in a way that makes attempts to remove potential ambiguities in the structures described.

So far we have only scratched the surface in this exploration of how to describe non-linear structures. There are three major directions in which we intend to extend the current work.

First, in many cases, a plan can be described hierarchically in terms of a number of high-level actions, each of which can consist of other high-level actions or atomic actions. We aim to incorporate this hierarchical information into our descriptions.

A second avenue of development concerns the means of expression that are available to PlanPresenter. So far we have only used simple typographic mechanisms, such as paragraph structuring, to indicate the underlying structure

```
[paragraph(1, [sentence([first,ensure,you,have,the,required,equipment]),
               sentence([you,are,now,ready,for,two,main,parts,
                         of,the,cleaning,which,may,be,done,in,
                         any,order,or,alongside,each,other])]),
 paragraph(2, [sentence([the,first,part,is,as,follows]),
               sentence([first,pick,up,any,rubbish,and,dust,the,
                         furniture,doing,these,in,any,order,you,like]),
               sentence([then,vacuum,the,carpet]),
               sentence([finally,take,out,the,garbage])]),
 paragraph(3, [sentence([the,second,part,is,as,follows]),
               sentence([first,wash,the,dishes]),
               sentence([then,clean,the,stove]),
               sentence([then,wipe,the,benches]),
               sentence([finally,mop,the,kitchen,floor])])]
```

Fig. 9. The final set of sentence specifications

of the plan. We aim to extend the range of realisation strategies available to the system so that more sophisticated outputs can be achieved.

Finally, so far we do not make use of a significant amount of other information regarding durations and resources that is available to us; we intend to incorporate this information to provide more complete descriptions of plans.

References

1. Elizabeth André and Thomas Rist. Generating coherent presentations employing textual and visual material. *Artifical Intelligence Review*, 9:147–165, 1994.
2. Robert Dale. *Generating Referring Expressions: Constructing Descriptions in a Domain of Objects and Processes.* MIT Press, Cambridge, MA, 1992.
3. Eli Goldberg, Norbert Driedger, and Richard Kittredge. Using natural-language processing to produce weather forecasts. *IEEE Expert*, 9(2):45–53, 1994.
4. Benoit Lavoie, Owen Rambow, and Ehud Reiter. Customizable descriptions of object-oriented models. In *Proceedings of the Fifth Conference on Applied Natural-Language Processing (ANLP-1997)*, pages 253–256, 1997.
5. John Levine, Alison Cawsey, Chris Mellish, Lawrence Poynter, Ehud Reiter, Paul Tyson, and John Walker. IDAS: Combining hypertext and natural language generation. In *Proceedings of the Third European Workshop on Natural Language Generation*, pages 55–62, Innsbruck, Austria, 1991.
6. Chris Mellish and Roger Evans. Natural language generation from plans. *Computational Linguistics*, 15(4):233–249, 1989.
7. Maria Milosavljevic, Adrian Tulloch, and Robert Dale. Text generation in a dynamic hypertext environment. In *Proceedings of the 19th Australasian Computer Science Conference*, pages 417–426, Melbourne, Australia, 31 January–2 February 1996.
8. Ehud Reiter and Robert Dale. *Building Natural Language Generation Systems.* Cambridge University Press, 2000.