Chapter 6

Surface Realisation

In this chapter we look at the process of deriving a surface text from a more abstract specification of that text, where that specification is the output of a previous stage in the language generation process. We discuss the different kinds of abstract specifications that are found in NLG systems, and describe how a number of available SURFACE REALISATION COMPONENTS can be used to carry out this mapping from specification to surface form.

6.1 Introduction

In the previous chapter, we saw how a TEXT SPECIFICATION can be constructed from a DISCOURSE PLAN via the process of MICROPLANNING. The text specification provides a complete specification of the document to be generated, but does not in itself constitute that document. It is a more abstract representation whose nature is suited to the kinds of manipulation required at earlier stages of the generation process; this representation now needs to be mapped into a surface form that ultimately consists of words on a page.[1]

More specifically, a text specification describes how the text to be generated is made up of elements such as headings, paragraphs and lists, and how these are made up of sentences and phrases. Figure 6.1 shows the text specification corresponding to the fragment of text in Figure 6.2, with the details of the phrase specifications omitted. Generally speaking, we can view the component parts of a text specification as being of two kinds. There are LOGICAL ELEMENTS such as list introductions and list elements which are realised by means of specific formatting constructs in the text to be generated; and there are elements—the PHRASE SPECIFICATIONS—which correspond to grammatical objects such as sentences and noun phrases, and require the application of some knowledge of the syntax of the target natural language before they can be realised. In

[1] Similar concerns arise when considering the production of a surface form which consists of phonological material in a speech stream; however, as elsewhere in the book, we focus here on the production of text rather than speech.

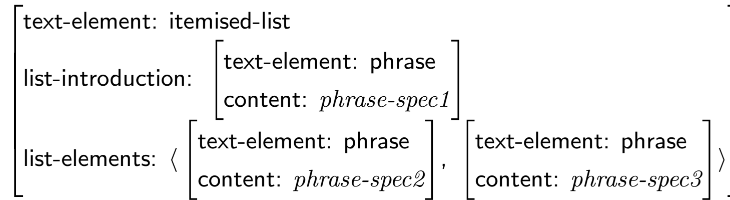

Figure 6.1: A text specification for a simple list structure

The Elephant has the following subtypes:

- the African Elephant; and
- the Indian Elephant.

Figure 6.2: A text corresponding to the text specification in Figure 6.1

reality, the distinction between formatting issues and grammatical issues is not so clear cut as this might suggest: so, for example, in Figure 6.2 the fact that the first phrase specification has the logical status of being a list-introduction means that it is terminated by a colon rather than a full stop; and the fact that the second phrase specification is the penultimate element in a list means that it is followed by the word *and*, with a preceding semi-colon. Ultimately, this means that we need to have some way of handling interactions between the two domains.

To keep things relatively simple, however, we will assume that there are two distinct aspects to the processing of text specifications. One is concerned with the mapping of logical constructs into appropriate document formatting, so that, for example, divisions between paragraphs are indicated typographically; the other is concerned with the application of grammatical knowledge to phrase specifications so that they can be realised as linguistic objects.

The mapping of logical constructs into typographic properties is not a question that has received much attention in the NLG literature; it is more often seen to be the province of work in document processing. In Section 6.2 we discuss briefly how a number of existing document processing mechanisms can be used to achieve what we need to do here. The bulk of the current chapter, however, is concerned with the mapping of phrase specifications into syntactically-defined surface forms.

The processing required to map phrase specifications into sequences of words

depends on how the phrase specifications are represented. As we indicated earlier, there are a number of alternatives here, varying in the degree to which they abstract away from the eventual surface form to be generated: the more abstract the representation, the more sophisticated the processing needs to be to derive appropriate surface forms. We provide a survey of the options and some attendant discussion in Section 6.3.

A great deal of work has been published on the problem of realising abstract phrase specifications. In many cases, the approaches adopted to the linguistic realisation task are necessarily based on the encodings of grammatical information suggested by specific linguistic theories. To understand this work often requires considerable familiarity with the relevant underlying theory. However, one consequence of the sheer bulk of work in this area is that a number of reusable software packages which carry out linguistic realisation are now widely available. In principle, this means that NLG system developers and researchers working in other areas of NLG do not need to have a deep understanding of what is involved in linguistic realisation: it is sufficient to know what the various packages do from a functional perspective (including the types of phrase specification representations they expect to see as inputs), and to have an understanding in general terms of how the packages work.

Accordingly, it is not the purpose of this chapter to explain how to build a linguistic realisation component. Rather, we discuss three existing realisation systems:

- KPML/NIGEL, a system based on Systemic Functional Grammar which accepts abstract MEANING SPECIFICATIONS as input;
- FUF/SURGE, a unification-based linguistic realiser which takes Systemic Functional Grammar as its theoretical base and accepts LEXICALISED CASE FRAMES as input; and
- REALPRO, a linguistic realiser based on Meaning-Text Theory, which takes what we will call ABSTRACT SYNTACTIC STRUCTURES as input.

In each case we describe the overall functionality of the system, we look at the nature of the inputs required, and we describe how the system works. The three systems are discussed in Sections 6.4, 6.5, and 6.6 respectively.

Section 6.7 discusses some of the issues that need to be considered when deciding which type of realisation component to use in a particular NLG system. In Section 6.8 we discuss the idea of BIDIRECTIONAL GRAMMARS: characterisations of the mappings between some underlying representation—typically some kind of logical form—and surface structure that can be used for both natural language generation and natural language analysis. Finally, Section 6.9 ends the chapter by providing pointers to the literature on surface realisation.

6.2 Realising Text Specifications

The idea behind our notion of a text specification is that the document planning and microplanning stages of the language generation process should be concerned with *what* the elements of a text are, but not with *how* they will appear on the page. The approach embodies what is sometimes referred to as a distinction between LOGICAL STRUCTURE and PHYSICAL STRUCTURE. In order to realise a text specification, we need, therefore, to map from logical constructs into physical constructs. For example, a fragment of text which has the logical status of being a heading may be shown on the page in bold face; an item within an itemized list might be typographically signalled by means of a preceding bullet and appropriate surrounding white space. Usually there are several presentational mechanisms which can be used for a given logical structure, with the choice often being dictated by some governing style or convention: for example, a publisher's house style might require that headings be indicated by the use of an italic typeface or a larger point size.

This distinction between logical structure and physical structure is now so well-established in the field of document processing that we do not, as authors, usually need to concern ourselves with low-level formatting issues. The earliest document processing systems removed the need for human authors to be concerned about line-end and page break decisions; within the last 20 years, most document processing systems have moved towards the use of markup conventions that further relieve the burden on the author, so that he or she needs simply to indicate, via some special symbols, what the nature of a textual construct is. The document processor then renders this in an appropriate manner on the printed page. So, for example, in LaTeX, we can indicate the logical type of the heading we require, and the formatting system will determine the appropriate typographic properties: the LaTeX construct in Example (6.1) produces the heading at the beginning of the section you are reading now.

(6.1)
```
\section{Realising Text Specifications}
```

The same ideas are now common in word processors through the use of style sheets, although the WYSIWYG nature of most word processors encourages a tendency to fiddle directly with aspects of physical structure.

For a real NLG system that is intended to produce anything more than disembodied fragments of text, the appropriate use of these presentational mechanisms becomes an issue. It is clear that such devices convey meaning; it is less clear how this meaning can be represented in a way that allows it to be reasoned with. We will return to this point in Chapter 7; however, for our present purposes we can sidestep many of the issues.

Given the availability of document processing systems whose purpose is to provide mappings from logical structures to presentational devices, it is sufficient for our NLG systems to know how to map from the logical representations they use to the kinds of markup required by these document processing systems as input. Thus, we can rely on rendering mechanisms such as LaTeX, and the capabilities of HTML browsers such as Internet Explorer and Netscape, to serve

```
<p>
⟨list introduction⟩
<ul>
<li> ⟨first list element⟩
<li> ⟨second list element⟩
</ul>
</p>
```

Figure 6.3: The logical structure specification in HTML form

```
⟨list introduction⟩
\begin{itemize}
\item ⟨first list element⟩
\item ⟨second list element⟩
\end{itemize}
```

Figure 6.4: The logical structure specification in LaTeX form

as post-processors that will produce the physical documents we need to generate. Provided the elements of logical structure in our text specifications correspond in a straightforward way to the logical elements supported by the rendering mechanism, this is a relatively trivial task.

Consider again the text specification shown in Figure 6.1. If our target platform is an HTML-based Web browser, we should convert this text specification into the structure shown in Figure 6.3; if the target is a version of the LaTeX document formatting system, we should convert the text specification into the structure shown in Figure 6.4.

There are practical situations where the NLG system may need to take responsibility for mapping logical specifications into lower-level rendering instructions. So, for example, it may be necessary for an NLG system to produce output in a proprietary formatting languages such as Microsoft's RTF (Rich Text Format), which tend more towards explicit physical characterisation; this might be appropriate, for example, in a system which generates tailored letters. In some contexts it may even be appropriate for an NLG system to be responsible for producing PostScript output that can be sent directly to a printer or display device. These kinds of outputs require the NLG system to know much more about low-level typographic issues; a suitable modularity will separate these issues from the questions of logical structure we are concerned with here, so that the lower-level details can be encapsulated in a separate post-processing component. Generally speaking, it is preferable to make use of a higher-level rendering mechanism if one is available.

So, from the point of view of surface realisation, we can view the problem of how to realise the logical structure of a text as a reasonably straightforward process of mapping from constructs in one logical specification language to another. The resulting formatting commands are, however, wrapped around fragments of text to be presented on the page; and these fragments of text are derived on the basis of the phrase specifications that populate the text specification. We turn, then, to consider the issues to be considered in realising phrase specifications. Within the literature, this is generally viewed as the domain of linguistic realisation proper.

6.3 Varieties of Phrase Specifications

In the literature, we find a wide range of approaches to characterising the input to linguistic realisation. Sometimes the differences between the representations used in different systems are simply nothing more than notational variations. In other cases, however, the content of the representations vary, signalling deeper differences between the views taken of what is involved in the process of linguistic realisation.

In Chapter 5, we took the view that the input to linguistic realisation should be in the form of what we called there a LEXICALISED CASE FRAME. This is a structure where the both the full semantic content and the specific lexical items to be used in realising that content have already been decided, but the particular syntactic form that the realised utterance should take has not been determined, other than perhaps in terms of the top level syntactic category of the grammatical constituents to be used. The linguistic realiser is then left with some decisions to make with regard to the fine syntactic details of the output utterance.

It is important to realise that there is no one right answer to the question of what the input to surface realisation should be. The approach we adopted in Chapter 5 corresponds to one point on a spectrum of abstractions; we could have chosen to use a more abstract, and thus less specified, form of representation; or we could have chosen a less abstract, nearer-to-surface-form representation. In the first case, the realiser would then be required to do more work in order to generate the text; in the latter case, to take an extreme, the microplanner might specify a specific text string to be used, so that the realiser has no work to do at all.

The three linguistic realisers surveyed in this chapter each expect inputs at different levels of abstraction. In this section, we prepare the ground for discussing these systems by looking more closely at the range of abstractions that can be considered. We start out with a very abstract representation of the content of a sentence, and then progressively make this less abstract.

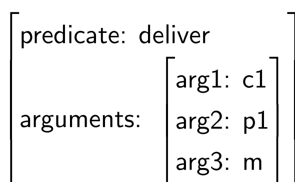

Figure 6.5: A skeletal proposition as an AVM

6.3.1 Skeletal Propositions

The most abstract representation we will consider is what we will call a SKELETAL PROPOSITION. This is similar to the representation a logician might use to represent the content of a natural language sentence. As an example, consider the following sentence:

(6.2) The courier delivered the green package to Mary.

In logic, we might represent this by means of the following proposition:

(6.3) $deliver(c1, p1, m)$

This representation says nothing about the content of the individual noun phrases in the sentence. It abstracts away from detail of this kind, to represent only the essence of the sentence: it indicates that a delivering event took place, and identifies the three participants in this event as being entities with the indices c, $p1$ and m respectively. Note that the symbol *deliver* here is not intended to be a lexical item: the logician would use the same symbol regardless of the source natural language of the sentence that the proposition is intended to represent, and there may well be other lexical items in the same source language that map into this same symbol; again, the representation is intended to abstract away from these differences.

This form of representation is often convenient for logical inference, and corresponds to what we might think of as a single fact in the domain. It is also very close to the notion of a message as discussed in Chapter 4. Such a representation could be used as the input to a linguistic realisation process. Note that this would place certain burdens on the realiser; most obviously, the realiser would be responsible for determining how the participants in the event should be referred to.

To allow easy comparison of the various representations we will consider, we will re-express the proposition in Example (6.3) using an attribute–value matrix which explicitly labels each of the elements, as shown in Figure 6.5.

6.3.2 Meaning Specifications

It is clear that the representation provided in Figure 6.5 does not represent all the information expressed in the sentence in Example (6.2). For example, the

representation does not tell us that the object delivered was a package, and it does not tell us that it was green. This, and the other information omitted from the skeletal propositional representation, could be added by conjoining some additional predicates to the representation in Example (6.3), as follows:

(6.4) $\ldots \wedge isa(p1, package) \wedge has\text{-}colour(p1, green) \wedge \ldots$

These additional predicates are very similar to the kinds of information we might expect to find in our knowledge base. A common view of linguistic realisation assumes that a prior processing step—what we discussed under the idea of MICROPLANNING in Chapter 5—identifies these elements in the knowledge base and not only selects them for inclusion in the text to be generated, but makes some decisions about the structures into which the information will be placed. The result is then what we might think of as a more complete MEANING SPECIFICATION for a sentence. This is a representation where the semantic content to be used in referring to each of the entities in the sentence has been determined. Figure 6.6 shows an example of a representation of this kind. Note how this has been developed from the previous representation we discussed:

- We have replaced the predicate and arguments labels in the original structure with the more semantically-loaded terms process and participants. There is no great significance here, although these alternative labels will be more appropriate later.

- In line with the above, we could also have changed the arg$n$ labels to have adopted from case grammar, such as agent, patient and so on. However, there is considerable variation in the literature with regard to the most appropriate set of case roles to use, and different case roles are appropriate for different verbs; so, for the moment we will retain the rather more neutral labels used here.

The above changes are arguably no more than notational. More significant are our extensions to the representation of the arguments:

- We have elaborated the representation of each argument so that it now specifies both an INDEX and a SEMANTICS. This corresponds to a distinction found in formal logic and the philosophy of language between EXTENSION and INTENSION, or between REFERENCE and SENSE [Frege ref, Carnap ref]. The index is a symbol that uniquely identifies the entity in some real or imagined world; the semantics is a representation of information that can be used to describe that entity. Note that the semantics is not unique to the entity in question: in the case of our example, there may be many other objects in the world that can be referred to as *the green package*. However, there is only one entity labelled $p1$: this name serves as a unique identifier.

- We have imposed some structure over the semantic information, identifying that semantic element that is the HEAD, in contrast to other elements which are viewed as MODIFIERS.

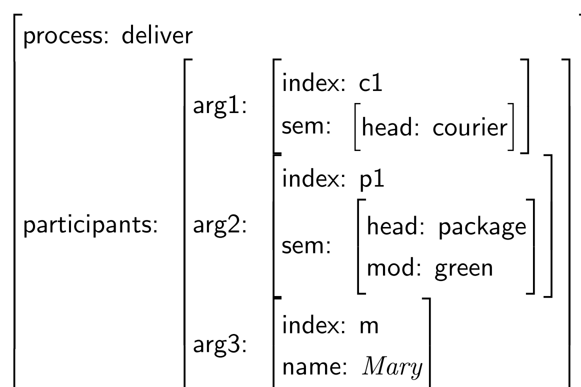

Figure 6.6: A meaning specification

- We have introduced a special element called NAME. Names are somewhat special in this representation: they provide an alternative to semantic expressions as a way of getting to a referent.

Note that this representation is still not lexicalised; in principle, each of the semantic elements here could be mapped into different lexical forms, with the possible exception of *Mary*. We can think of this representation as being something like a STRUCTURED LOGICAL FORM, since it indicates how the information is grouped in a way that a simple list of predicates does not.

6.3.3 Lexicalised Case Frames

The structure just presented is still more abstract than most approaches to linguistic realisation expect as input. In practice, many realisers expect the previous stage of processing to have also selected the base lexemes to be used in expressing the semantic content. Once these decisions have been made, the content of the index and sem features has exhausted its usefulness, and so we will omit them from our representations from here on.[2] Figure 6.7 shows such a LEXICALISED CASE FRAME. Note that the base lexemes are not yet words: to achieve this status they need to be subjected to morphological processing, so that, for example, the base lexeme *be* becomes the word *is* in those cases where the syntactic context requires that a third person singuler present tense form is used.

[2]Note, however, that in theories like HPSG, one might maintain a structure that encodes all these levels of representation simultaneously.

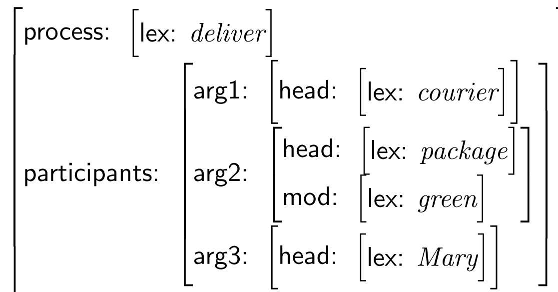

Figure 6.7: A lexicalised case frame

6.3.4 Abstract Syntactic Structures

Representations of the kind just discussed are quite common as inputs to linguistic realisation processes. They are sufficiently abstract that they allow the linguistic realiser to carry out what we might think of as SYNTACTIC INFERENCE. For example, given a structure like that in Figure 6.7 that has been augmented by the addition of some information about which argument of the process is to be placed in focus, the realiser can reason about the syntactic resources that should be used to achieve this effect. If the input structure requires that the second argument be the focus, then the realiser might produce the following sentence:

(6.5) The green package was delivered to Mary by the courier.

Sometimes, however, it is deemed appropriate for processing carried out prior to the invocation of the linguistic realiser to make such decisions about grammatical structure. In such cases, the role of the realiser is simply to encode knowledge about the grammatical minutae of the language in question, applying these to an ABSTRACT SYNTACTIC STRUCTURE as input. Figure 6.8 shows an example of such a representation. Here, the realiser converts this into a surface sentence by choosing the correct inflection of the content words, adding appropriate function words, and deciding the order in which the constituent parts of the sentence should appear. A realiser which operates on the basis of inputs like this thus conveniently hides some of the idiosyncracies of syntax from the rest of the NLG system.

6.3.5 Canned Text and Templates

Sometimes the general form of sentences or other constructions in a text is sufficiently invariant that they can be predetermined and stored as text strings. This can be particularly useful for elements of a text for which there is no obvious compositional treatment. So, for example, the closing salutations in

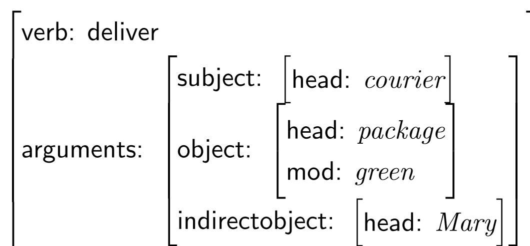

Figure 6.8: An abstract syntactic structure

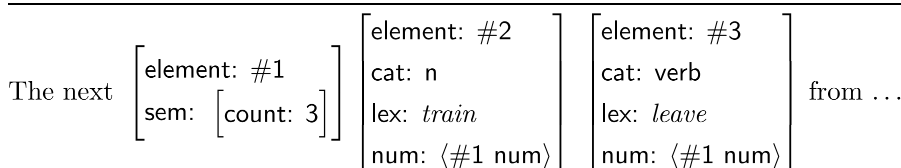

Figure 6.9: A simple template

a business letter generator—forms like *yours sincerely* and *with best wishes*—might be stored as predetermined text strings for inclusion in the document plan. We will refer to such structures as CANNED TEXT.

More often, substantial amounts of text may be predetermined in this way, but specific elements within otherwise fixed sentences may only be able to be determined at run-time. The input to the realiser in such cases will be a construct that contains information at more than one level of representation: it will be made up of text strings interspersed with representations that require a realisation process to be applied. These representations could be at any of the levels we have discussed in the preceding sections; most typically they will be specifications for individual words, consisting of a base lexeme and some morphological information which can be used by the realiser to produce a surface form.

We will refer to inputs of this kind as TEMPLATES. Figure 6.9 shows a simplified representation of a template that might be passed to a realiser. Such representations allow the earlier stages of the NLG process to ignore the idiosyncracies of morphology for those parts of the output text which need to be worked out as part of the generation process. Templates are, in effect, hybrid representations where some elements are specified as canned text and other elements are specified as phrase specifications.

6.3.6 Text and Orthography

Earlier, we drew a distinction between base lexemes and words: the latter are produced by applying morphological processing to the former. Here, we find it useful to draw a distinction between what we will call TEXT WORDS and ORTHOGRAPHIC WORDS. Just as the string representation of a word is still abstracted away from its phonological form, and needs further processing to be realised as sounds, it is also abstracted away from its ORTHOGRAPHIC FORM. So, for example, even once the morphological form of all the words in a sentence has been decided, there are still decisions to be made about casing and punctuation:

- If the phrase in question is a complete sentence, typically the first letter of the sentence will be presented in upper-case.
- In certain contexts, if the phrase in question is being used as a title, it may all be upper-cased.
- Particular elements of the text may be rendered in bold or italic face to indicate emphasis, or for some other purpose.
- Sentence-final punctuation will depend on whether the sentence is a statement, a question or an imperative, or whether it serves as the lead-in to an itemised list.

Again, sometimes prior stages in the NLG system may predetermine these orthographic aspects of the document; but often these will be left to the realisation component. Correspondingly, in our representations we will draw a distinction between TEXT and ORTHOGRAPHY: text elements are those to which orthographic processing is still to be applied.

6.3.7 Summary

We have surveyed a number of different levels of abstraction at which the inputs to surface realisation can be specified:

- SKELETAL PROPOSITIONS, in which the basic content of a sentence—who did what to whom—is determined, but the specific linguistic forms required to identify the participants in the represented eventuality remain to be worked out;
- MEANING SPECIFICATIONS, in which the semantic content of the referring expressions to be used in identifying the participants in the eventuality has been decided;
- LEXICALISED CASE FRAMES, in which the semantic content has been mapped into particular base lexemes in the target language;
- ABSTRACT SYNTACTIC STRUCTURES, in which the basic grammatical properties of the sentence to be generated have been specified;

- TEXT, in which the surface form of the words and phrases to be used has been determined; and
- ORTHOGRAPHY, in which the final rendered form of the text, taking into account typographic issues, has been decided.

For systems which produce speech, this last is replaced by PHONOLOGY, in which the particular sounds to be used in realising the text have been determined.

Given the number of different levels of abstraction we might use to specify the content of a generated text, it is not surprising that different NLG realisation components accept different kinds of input. In the remainder of this chapter, we look at three widely available surface realisation components: KPML/NIGEL, FUF/SURGE and REALPRO. We begin in Section 6.4 with KPML/NIGEL, since of the three systems this accepts the most abstract representation of phrase specifications as input; the input required by this system is close to what we have called here a MEANING SPECIFICATION. Then, in Section 6.5 we look at FUF/SURGE, whose input is close to what we have called here a LEXICALISED CASE FRAME. Finally, we look at REALPRO, whose input corresponds to what we have called here a ABSTRACT SYNTACTIC STRUCTURE.

6.4 KPML

6.4.1 An Overview

Komet-Penman Multilingual—the KPML system—is a linguistic realisation component based on the grammatical theory known as Systemic Functional Grammar [ref halliday]. The artifact that is popularly known as KPML in fact contains three elements:

- a computational engine that traverses grammatical resources;
- the collection of grammatical resources that comes with the system; and
- a development environment for writing and debugging new grammars.

We will use the term 'KPML' to refer to all three elements combined.

KPML is a complex and powerful system. Its grammar of English, which is called NIGEL, is probably (as of late 1997) the largest generation grammar of English ever built; it has been under development since the late 1970s. KPML also includes moderately-sized grammars of German and Dutch, and small illustrative grammars of French and Japanese; grammars for other languages, including Greek and Finnish, are under development. One of the long-term goals of the KPML project is to make multilingual generation easy by allowing a single input (in our terms, a phrase specification) to be realised in different languages simply by changing the active grammar being used by the system. [Bateman ??] provides a discussion of what is involved in carrying out multilingual generation from a common input.

```
(S1 / generalized-possession
  :tense past
  :domain (N1 / time-interval
             :lex march
             :determiner zero)
  :range  (N2 / time-interval
             :number plural
             :lex day
             :determiner some
             :property-ascription
             (A1 / quality :lex rainy)))
```

Figure 6.10: An input to KPML

In addition to its grammars, KPML also includes lexicons (for instance, an English lexicon which defines 1000 common English words), and a sophisticated and powerful development environment for writing and debugging new grammars. This environment includes graphical visualisation tools, static integrity checkers, and testing and verification systems. Again as of late 1997, the development environment distributed with KPML is much richer than the development tools distributed with the two other realisers we discuss in this chapter.

6.4.2 The Input to KPML

KPML can accept several types of input. The most common, and the only one we will discuss here, is in the form of SENTENCE PLANS expressed in SPL, the Sentence Planning Language.

The form of SPL expressions is best demonstrated by example. Figure 6.10 shows the SPL required to produce the sentence *March had some rainy days*. Figure 6.11 re-expresses this construct as an explicitly-labelled attribute–value matrix, to make it easier to compare with the various forms of phrase specification we have introduced in this chapter. Figure 6.12 shows an SPL for the more complex sentence *the month was cool and dry with the average number of rain days*.

Some of the details here warrant some elaboration.

- Every constituent starts off with an id and type. In SPL these are written as a pair of the form id / type, but we have shown them as distinct features in the AVM representation. The id simply associates a name with a constituent; this allows elements of constituents to refer to other constituents where necessary. The type specifies a class in KPML's UPPER MODEL, which we will return to below.

- The type generalised-possession in Figure 6.10 tells KPML that the sen-

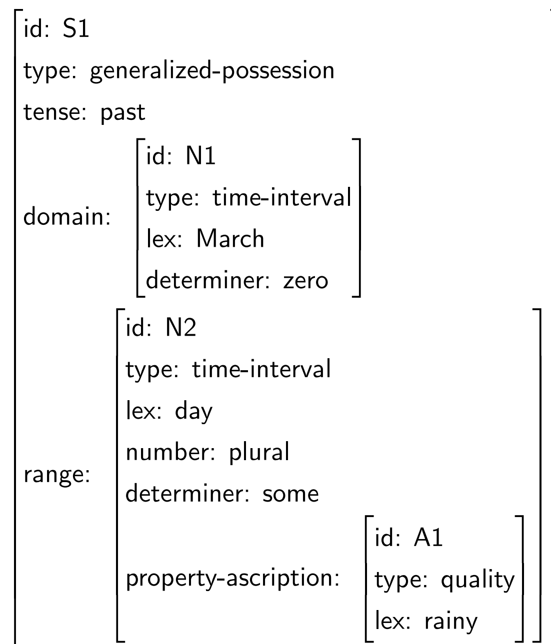

Figure 6.11: An AVM representation of the structure in Figure 6.10

```
(l / property-ascription
  :tense past
  :domain (m / one-or-two-d-time
             :lex month :determiner the)
  :range ((c / sense-and-measure-quality :lex cool)
          (d / sense-and-measure-quality :lex dry))
  :inclusive
    (ard / quality
         :lex number
         :determiner the
         :property-ascription
           (av / quality :lex average)
         :part-of (rd / one-or-two-d-time
                      :lex day
                      :number plural
                      :property-ascription
                        (r / quality :lex rain)))))
```

Figure 6.12: A more complex SPL expression

tence to be generated asserts that one entity possesses, in some general sense, another entity or set of entities. Note that the verb *have* is not explicitly specified in the SPL; its use will be inferred from the presence of the generalized-possession type. Similar considerations apply to the type property-ascription.

- The attribute–value pair [tense: past] indicates that the clause should be in past tense.
- The domain attribute describes the entity that is doing the possessing. The type time-interval asserts, as might be expected, that this entity is a type of time interval; the attribute–value pair [lex: March] tells KPML to use the word *March* in describing this concept, and the attribute–value pair [determiner: zero] tells KPML not to insert a determiner before the word *March*.
- The range feature describes the entity that is doing the posessing. This is also of type time-interval. In a similar fashion to that just described for the domain entity, the features specified here tell the system to use the word *day*, to make this plural, and to use the determiner *some*.

The most important thing to note about KPML is that its input structures are essentially semantic—the various elements are expressed as being concepts or entities of specific types—with attached grammatical information. Of the levels

⟨Here there should be a fragment of the Upper Model: probably best to show quite a large fragment and have it displayed landscape mode over a whole page.⟩

Figure 6.13: A fragment of the Upper Model

of abstraction we discussed earlier, it is therefore closest to what we have called a meaning specification.

The semantic types used in KPML are defined in terms of a linguistically-oriented ontology called the UPPER MODEL. Figure 6.13 shows a small fragment of the Upper Model. The hierarchical nature of this structure means that information can be inherited by subordinate nodes from superordinate nodes. This avoids the need to specify information repeatedly, so that, for example, information about the realisation of temporal concepts like months and days can be specified once only as properties of the one-or-two-d-time concept.

As we noted above, the SPL expressions we have examined here include both specifications of semantic types and elements which are more grammatical in nature. These grammatical features are a convenience for the SPL author: they are in effect macros which expand to more semantic features. So, for example, the grammatical attribute–value pair [determiner: some] in Figure 6.10 is a macro which expands into a set of six semantic features, including [singularity-q: nonsingular] and [amount-attention-q: minimalattention]. This set of features strongly biases the grammar towards selecting *some* as a determiner, unless for some reason this is grammatically impossible.

The Upper Model provides a repository for information about linguistic realisation that can be shared amongst applications. Allowing authors to specify the inputs to realisation in terms of semantic types from this ontology means detailed grammatical specifications can be omitted. For particular applications, further benefit can be achieved by extending the Upper Model by adding a MIDDLE MODEL: this is a set of concepts from the application's domain, subordinated to the elements of the Upper Model in such a way as to allow the appropriate inheritance of information to take place. So, for example, if we augment the Upper Model by adding March as an instance of the concept month, and subordinate the concepts month and day to the Upper Model concept time-interval, then we can generate the sentence *March had some rainy days* using the simplified SPL expression in Figure 6.14. We do not need to explicitly specify time-interval as a type in the SPL expression, since this will be picked up from the augmented Upper Model.

This mechanism is very powerful; however, it comes with a cost. In order to make use of these resources, one needs to have a good understanding of the structure of the Upper Model and the kinds of conceptualisations that underly KPML's notion of semantic types.

```
(S1 / generalized-possession
  :tense past
  :domain (N1 / March)
  :range  (N2 / day
             :number plural
             :determiner some
             :property-ascription
             (A1 / quality :lex rainy)))
```

Figure 6.14: A simpler SPL expression

Systemic Functional Grammar

KPML is based on the approach to linguistics known as Systemic Functional Linguistics, and in particular Systemic Functional Grammar (SFG; [ref halliday]). Systemic Functional Grammar is quite different in orientation to the approaches to linguistic description often adopted in work on natural language analysis (or, more precisely, natural language parsing), where researchers have tended to look to the world of generative grammar for ideas and theories about linguistic structure. Chomsky's initial work on transformational grammar began a line of development that was able to provide characterisations of linguistic structure precise enough for embodiment in computational models of parsing; within this broad area of linguistics, it is now *de rigeur* for a linguistic theoretician to be concerned about the computational aspects of the grammatical theory he or she is developing. These considerations are at their height in the development of approaches such as Head-driven Phrase Structure Grammar (HPSG), where the formalist's concern with precision means that computational aspects are often developed in lock-step with linguistic descriptions.

That such approaches, with an early emphasis on the autonomy of syntax, should be used in parsing should be of no surprise. These approaches to linguistic description are essentially concerned with cataloging the surface forms available in the language, and with describing these using structures that generalise across instances. Such a view of language is very appropriate when the known is the surface form, and the goal is to derive some underlying structure from this surface form.

The task in NLG is different, however; we start with the underlying representation, and our goal is to map this into a surface form. The questions to be dealt with are therefore rather different in nature: rather than ask 'What is the structure of this utterance and what is it being used to do?', we ask 'What resources does the language offer to present the meaning I want to convey?' This leads to a predisposition to categorise the resources of language in terms of their function (more precisely, FUNCTIONAL POTENTIAL) rather than their form. Such a categorisation is precisely what Systemic Functional Grammar

⟨Figure to be included: the English sentential mood system, but not including realisation statements; just the system network.⟩

Figure 6.15: The mood system for the English clause

provides.

In Systemic Functional Grammar (as in many aspects of NLG), the central notion is one of choosing from available alternatives. A surface form is viewed as the consequence of selecting a set of abstract functional features from a description of the resources of the language as a whole, In SFG, these resources are represented in the form of a SYSTEMIC NETWORK (in effect, a taxonomy), and linguistic realisation consists in traversing this network; each divergence in the network represents a choice between minimal grammatical alternatives, and results in the addition of small constraints on the final form of the utterance. The information collected as a result of these choices amounts to the interpolation of an intermediate abstract representation. This allows the specification of the text to accumulate gradually, and in principle means that the realisation process does not need to backtrack, since the constraints on the final form of the utterance added at each point are small enough to avoid unwarranted overcommitment.

As just noted, a systemic network is in essence a taxonomy: it categorises the different elements of the language, but it does this in a functionally-motivated way. Technically, a systemic grammar is composed of CHOICE SYSTEMS. Each system is a set of simultaneous alternatives; each alternative is named by a GRAMMATICAL FEATURE, and corresponds to a minimal grammatical alternation. Each system has an ENTRY CONDITION; during traversal, this determines which systems may be entered from a given point. From a more declarative point of view, the connectivity of systems expresses the dependencies between linguistic elements. The selection of one alternative determines what further systems may be entered; as we move from left to right in a systemic network, the systems are said to be more DELICATE.

To make all this somewhat more concrete, Figure 6.15 shows a systemic fragment that represents the choices available for mood in the English clause. This indicates that clauses in English can be major or minor; that a major clause can be declarative, interrogative or imperative; and so on through the chain of dependent characteristics to those shown at the rightmost edge of the network. Any path through this network thus provides a description of a clause type in English by selecting a feature from each choice system.

Of course, this is only a very small part of a complete grammatical description of the English language. A printed version of the complete NIGEL grammar will easily cover the wall of an average-sized office.

6.4.3 Using Systemic Grammar for Linguistic Realisation

So far we have seen how a particular notation—the systemic network—in conjunction with a terminology that represents a functional view on language can be used to represent the space of meanings that a language makes available. If we are to make use of this to produce surface utterances, we need to add two further elements.

First, we need a mechanism that allows us to traverse the network, guiding decision-making at each choice point in the network. Second, each choice should result in some consequences of that choice which contribute to the specification of the final surface form.

The general algorithm adopted in KPML for systemic network traversal is quite simple:

- Start with rank of least delicacy (typically the clause network).
- Make choices until the maximally delicate distinctions offered have been drawn.

The result of this is a complete description of a linguistic unit at that rank. The process then repeats recursively for the elements at the next rank (typically the noun groups that will realise the participants in the sentence).

Whenever we reach a disjunction—a choice point—in the network, a decision must be made. In the KPML framework, this is done using a general approach called INQUIRY SEMANTICS. The bridge between semantics and the choices that affect surface form is provided by specialist pieces of code attached to each choice point that know how to choose between the options available on the basis of various sources of information.

These specialist pieces of code are called CHOOSERS; each is a small parcel of program code—effectively a decision tree—that, by posing queries of the input SPL expression or of the wider 'environment' in which the system is running, can determine which feature should be chosen. In theory, the answers to these queries could be encoded in the input SPL expression: this is function of those features whose names end in '-q', such as as command-offer-q. However, for many queries, it would be unreasonable to expect the constructor of the SPL expression to be able to predict which queries might be relevant. It is generally more convenient for the grammar to determine what information is required at run-time, and to obtain this information by posing queries to the environment.[3] Figure 6.16 shows the chooser that decides which form of determiner should be used in referring to a given entity.

Choosers, and their underlying inquiry semantics, provide us with a reasoned way of traversing through the systemic network. We still require some means by which the choices made result in the generation of a surface form. In the KPML system, this is achieved by means of REALISATION STATEMENTS. These are low

[3] In fact, the earliest implementation of this model of generation, the PENMAN system, used no explicit input structure at all; traversal of the network was guided completely by an interactive process during which queries were put, one after another, to the user.

⟨Figure to be inserted: the chooser for picking determiners⟩

Figure 6.16: A chooser

+X	Inclusion of function X
X=Y	Conflation of functions X and Y
X/A	Classification of function X with feature A
X,Y//S	Agreement of functions X and Y for choice system S
X>Y	Ordering of function X before function Y
+X=Y	combines +X and X=Y
+X/A	combines +X and X/A
+X>Y	combines +X and X>Y

Table 6.1: Realisation Statement Operators

granularity partial specifications of surface structure attached to the features at each choice point in the grammar; the choice of features during network traversal thus results in the accumulation of a large number of realisation statements each of which adds minimal constraints to the utterance being constructed.

The kinds of operations permitted in realisation statements are shown in Figure 6.1; Figure 6.17 shows the English mood system we saw earlier, but with realisation statements added.

The fact that realisation statements allow the determination of surface structure in such small increments means that it is much easier to avoid over-commitment, thus reducing the likelihood that backtracking will be required. Once the entire network has been traversed, the collected realisation statements serve as a set of constraints that determine the surface form of the utterance.

6.4.4 Summary

We suggested earlier that the function-oriented approach of SFG made it more useful for work in NLG than phrase-structure approaches to linguistic descrip-

⟨Figure to be inserted: The English mood system complete with realisation statements.⟩

Figure 6.17: Realisation Statements in a Network

tion. We end this section by noting the specific advantages that have been claimed by proponents of SFG.

A significant claim of work in SFG is that it may be more natural and economical to state syntactic regularities in a functional framework: a constituent framework may require additional levels of structure to capture functional similarity. This is perhaps most apparent where cross-language generalizations are desired. These may be better stated in functional terms—by and large, all languages provides ways of doing the same things, but the ways in which these functions are achieved varies. As noted earlier, grammars for languages other than English have been deveoloped for KPML, and some of the current research here explores how portions of the grammatical description can be shared between languages.

One of the characteristic properties of SFG as a linguistic theory is that it attempts to encompass aspects of meaning other than the propositional (or, as it is termed in SFG, the IDEATIONAL). Systemics also considers two other dimensions of meaning: the INTERPERSONAL and the TEXTUAL. Interpersonal meaning is concerned with the relationship between the speaker and his or her audience, and the effect this has on the language used; it is this that conditions, for example, the use of honorifics in Japanese, the choice between formal and informal pronouns in French, and in English the firmness with which requests are expressed. Textual meaning is concerned with the way in which information is packaged in a text and how the message conveyed is structured thematically; this aspect of meaning covers, for example, the choice between different syntactic structures which convey the same propositional content.

In SFG, these three aspects of meaning are all represented within one space of grammatical description. This allows an interplay between the dimensions—called in SFG the three METAFUNCTIONS of language—that is not typically addressed by other linguistic theories, and in particular those that concentrate on the notion of truth conditions as a way of characterising semantics.

6.5 Fuf/Surge

6.5.1 An Overview

The Functional Unification Formalism (FUF: see [Elhadad 1992, 1993]) is in effect a programming language tailored to the needs of grammatical processing, and in particular, to the needs of linguistic realisation in a unification-based framework. FUF has its origins in Functional Unification Grammar [?], and uses graph unification to combine an input structure that corresponds to a sentence specification with a grammar of the output natural language, the result being a syntactically-specified structure which is then linearised to produce the required sentence.

SURGE—the Systemic Unification Realisation Grammar of English—is a large broad coverage grammar which, in conjunction with FUF, has been used by a number of research and development groups around the world. Like NIGEL,

```
((cat clause)
 (proc ((type possessive)))
 (tense past)
 (partic ((possessor ((cat proper) (head ((lex "March"))))
          (possessed ((cat common) (head ((lex day)))
                      (describer ((lex rainy)))
                      (selective yes) (number plural)))))))
```

Figure 6.18: An input to SURGE

SURGE embodies a view of language that is based on Systemic Functional Grammar, although the grammar also borrows from HPSG [?] and descriptive linguistic works [?]. FUF can be used in conjunction with other grammars, although it is most commonly used with SURGE.

The framework permits phrase specifications to be specified at a variety of levels of abstraction, but most commonly the input expected by SURGE is what we have called a LEXICALISED CASE FRAME. This level of representation permits the grammar to make certain decisions about realisation that can be inferred on the basis of the input. An example of this is THEMATISATION. English has several mechanisms which allow the theme or focus of a sentence to be changed, of which perhaps the most common is PASSIVISATION. For example, *John told Mary* and *Mary was told by John* convey the same information, but in the first sentence the focus is on *John*, while in the second it is on *Mary*. SURGE will generate both sentences from what is essentially the same basic input structure, the difference between the two being determined by the value of a focus attribute.

6.5.2 The Input to SURGE

Input to SURGE is in the form of a FUNCTIONAL DESCRIPTION or FD. Formally, this is a collection of attribute–value pairs that together provide a specification of the utterance to be generated. Each such FD must contain a cat feature, which indicates the syntactic category of the form to be produced; the other attributes present will then depend on the information required by the grammar in order to generate a structure of that category.

Figure 6.18 shows the FD required to produce the sentence *March had some rainy days*; Figure 6.19 re-expresses this FD in a form closer to the AVM notation we introduced earlier. Figure 6.20 shows an FD for the more complex sentence *the month was cool and dry with the average number of rain days*.

Just as in the case of SPL, the input specification language used in conjunction with KPML/NIGEL, some of the details here deserve a little elaboration.

- The initial [cat: clause] feature states that this FD should be expressed as a clause (rather than, for example, a noun phrase).

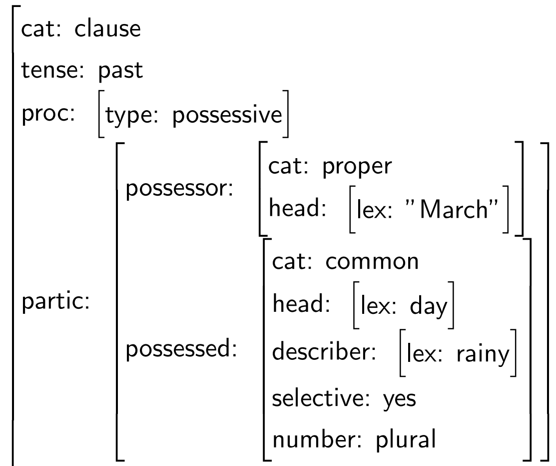

Figure 6.19: An AVM representation of the structure in Figure 6.18

- The [tense: past] feature states that the clause should be in the past tense
- The proc feature describes the type of PROCESS that the clause describes. In this case, the [type: possessive] feature tells SURGE that the clause is describing a possessive process; this means that the clause will have two participants, a possessor and a possessed, and that in the eventuality to be described by the clause, the possessor participant possesses the possessed participant. Note that the FD does *not* explicitly state that *have* should be used as the verb of the clause; SURGE infers this from the type of process.
- The partic feature describes the participants in the process, in this case the possessor and the possessed. The possessor is to be realised linguistically by the term *March*, which is a proper noun (indicated by [cat: proper]); the possessed is to be realised by the term *day*, which is a common noun ([cat: common]). The [number: plural] feature tells SURGE to use the plural form, [selective: yes] indicates that determiner *some* should be used, and [describer: rainy] tells SURGE to add *rainy* as an adjective which will modify the head noun *days*

In contrast to the inputs required by KPML/NIGEL for the same sentences, note that the inputs required by SURGE do not contain explicit references to the semantic entities which take part in the eventualities being described by the sentence. For this reason, we categorise the structures used here as LEXICALISED CASE FRAMES.

```
((cat clause)
 (tense past)
 (proc ((type ascriptive) (mode attributive)))
 (partic
   ((carrier ((cat common) (lex "month")))
    (attribute
      ((cat ap) (complex conjunction)
       (distinct
         ~(((cat adj) (lex "cool"))
           ((cat adj) (lex "dry"))))))))
 (circum
   ((accompaniment
      ((cat pp)
       (np ((cat common) (head === "number")
            (classifier === "average")
            (qualifier
              ((cat pp) (restrictive yes)
               (np ((cat common) (definite no)
                    (number plural) (head === "day")
                    (classifier === "rain")))))))
       (accomp-polarity +))))))
```

Figure 6.20: A more complex input to SURGE

6.5.3 Functional Unification Grammar

Although the grammatical theory underlying SURGE is, like NIGEL, based on Systemic Functional Grammar, the linguistic resources are represented in a quite different way, in a manner that has its origins in Martin Kay's early work in FUNCTIONAL UNIFICATION GRAMMAR [Kay 1979].

Kay developed the idea of functional unification grammar (FUG) as a means of representing grammatical information that was intended to be neutral between generation and analysis. Although the idea of specifying grammatical knowledge declaratively is now widely accepted, at the time Kay was writing it was still commonplace to embed linguistic knowledge in procedural mechanisms. Functional unification grammar brought together two ideas which are not in fact necessarily related. First, it embodied the idea that grammatical information should include functional aspects: Kay took the view that purely formal descriptions of language were not particularly useful, and that primary status should be given to functional aspects of language. This entailed making reference to notions about information structure like 'given', 'new' and 'focus', and concepts such as speech acts that are more traditionally considered part of pragmatics. Second, FUG made use of a notion of UNIFICATION as a process by means of which minimal, conceptually-derived functional descriptions could be fleshed out by merging with the information contained in an appropriately represented grammar. By insisting on a declarative representation of the grammatical knowledge, no specific demands would be imposed on the control structure used for parsing or generation.

Elhadad took these key ideas from Kay, and developed a formalism for expressing linguistic information in a unification-based framework; the result is FUF, which is in effect a programming language that is well-suited to manipulating grammatical information. SURGE is a broad coverage grammar based on systemic ideas, but encoded in a manner appropriate for use by a unification-based process.

6.5.4 Linguistic Realisation via Unification

The view of generation embodied in the SURGE grammar assumes that realisation covers the following tasks:

- the mapping of thematic structure onto syntactic roles;
- the control of syntactic paraphrasing and alternations;
- the provision of defaults for syntactic features;
- the propogation of agreement features;
- the selection of closed class words;
- the imposition of linear precedence constraints; and
- the inflection of open class words.

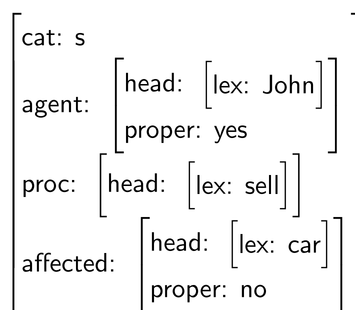

Figure 6.21: An FD for a simple sentence

We noted earlier that the input to SURGE is expressed as a functional description. The grammar itself is also expressed as functional description; realisation is achieved by unifying the functional description that corresponds to the input expression with this second, very large, functional description. The input FD then represents the semantic content of the message to be realised; the grammar FD describes the space of grammatical alternatives and the correspondences of these to semantic elements, and contains annotations that provide some control over how the grammar is used. The structure that results from this unification process is then linearised; the output of the realisation process is then a surface sentence expressing the specified meaning according to the grammatical constraints of the language.

In order to give some idea of how this works, we will discuss the process by reference to a much simpler grammar than SURGE itself. Figure 6.22 shows a modified version of one of the example grammars distributed with FUF; we will explore how this grammar is used in conjunction with the simple input FD shown in Figure 6.21, which results in the generation of the sentence *John sells the car*.

The grammar consists of four ALTERNATIONS: one for sentences ([cat: s]), one for oun phrases ([cat: np]), one for verb phrases ([cat: vp]), and one for articles ([cat: art]). When this structure is unified with the input FD, the following happens:

- First, the input FD as a whole is compared to the different sections of the grammar. Because the input FD specifies that a sentence is required—this being specified by the [cat: s] feature–value pair in the input FD—the first branch of the grammar is chosen.
- This part of the grammar is unified with the input FD: that is, the features in the input FD are merged with the features in the selected grammar segment, resulting in the combined FD shown in Figure 6.23. The net effect of this is that the grammar adds some information to the originally

```
((alt top (

   ;; First branch of the alternative
   ;; Describe the category S.
   ((cat s)
    (agent ((cat np)))
    (affected ((cat np)))
    (proc ((cat vp)
           (number {agent number})))
    (pattern (agent proc affected)))

   ;; Second branch: NP
   ((cat np)
    (head ((cat noun) (number {^ ^ number})))
    (alt (
      ;; Proper names don't need an article
      ((proper yes)
       (pattern (head)))
      ;; Common names do
      ((proper no)
       (pattern (det head))
       (det ((cat article)
             (lex "the")))))))

   ;; Third branch: VP
   ((cat vp)
    (pattern (head))
    (head ((cat verb))))

   ;; Fourth branch: ARTICLE
   ;; doesn't do anything
   ((cat article)))))
```

Figure 6.22: A simple grammar

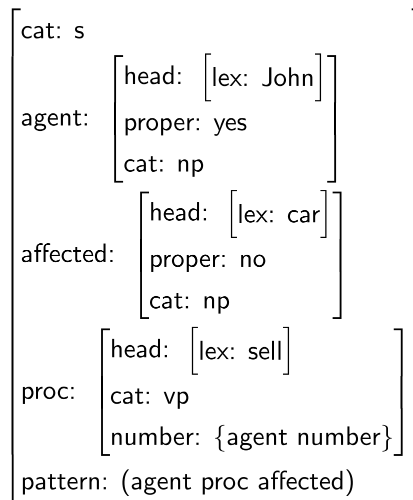

Figure 6.23: The result of initial unification of the input FD with the grammar

input structure: we gain the additional detail that agent will be a noun phrase, that the proc will be a verb phrase, that the proc and the agent will have the same grammatical number, and that the affected is also to be realised using a noun phrase. The pattern feature tells FUF that when this FD is linearised, the agent should come first, followed by the proc, followed by the affected.

- Once the top-level FD has been unified with the grammar, FUF tries to unify all CONSTITUENTS of the FD with the grammar. In simple cases (such as in this case), the constituents are all features mentioned in the pattern feature: here, agent, proc, and affected. The result of this is unification is the FD shown in Figure 6.24. Note that the section of the grammar that deals with noun phrases has two alternations: the first of these (for noun phrases which have the feature [proper: yes]) is used for the agent *John*, and the second of these (for noun phrases with the deature [proper: no]) is used for the affected *the car*.
- After all unifications have been completed and the final FD is produced, FUF linearises this FD. This is done by recursively expanding the pattern feature until FDs are reached which do not contain a pattern. At this point, the particular word to be used is inserted into the output sentence by retrieving the lex feature, and morphologically inflecting it according to features such as number and person. In our simple example this results in the sentence *John sells the car*.

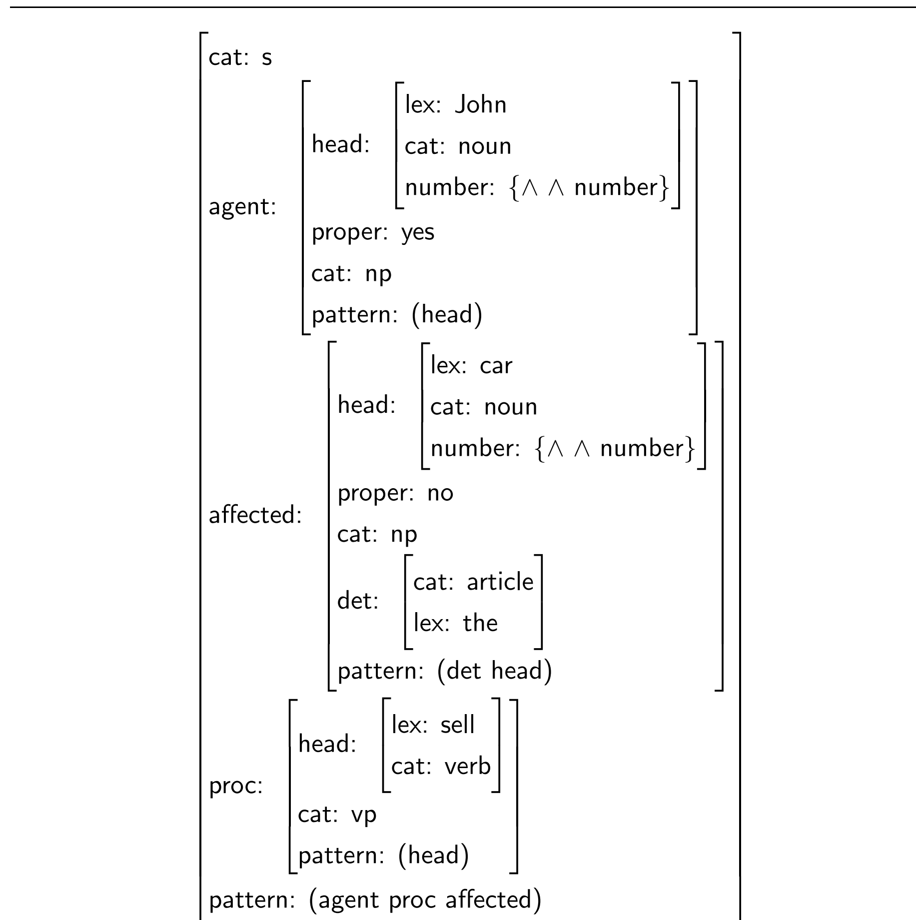

Figure 6.24: FD in Figure 6.21 after unification with the grammar in Figure 6.22

The processing carried out by FUF is generally more complex than this simple example suggests; the reader is encouraged to explore the examples discussed in the FUF system documentation. As with KPML/NIGEL, anyone who expects to use SURGE in an NLG system will have to spend a non-trivial amount of time learning how to use the system.

6.5.5 Summary

KPML/NIGEL and FUF/SURGE expect inputs at different levels of abstraction, with the consequence that the microplanner needs to do different amounts of work in each case.

From the point of view of system use, FUF/SURGE is self-contained, while KPML/NIGEL allows the user to connect lexical concepts to external knowledge sources such as the Upper Model. While this means that the microplanner has to do less work in constructing an input for KPML, using FUF/SURGE requires correspondingly less of a commitment to the theoretical underpinnings of the system. The more abstract semantically-oriented input required by KPML/NIGEL means that the microplanner has to do less work in building sentence plans, whereas the more syntactically-oriented input required by FUF/SURGE means that the microplanner has to know something about syntactic possibilities and has to be able to map elements of the document plan into these. This might seem like a distinct disadvantage to using FUF/SURGE; on the other hand, using a higher level of abstraction for input means that one has to be familiar with the vocabulary of that representation. This becomes an issue when one is faced with the inevitable limitations in grammatical coverage. Dealing with such lacunae in the case of KPML/NIGEL can mean that one has to subvert the proper use of semantic constructs in order to achieve the results required; doing this, of course, requires knowing what the syntactic effects of using those semantic constructs will be, whereas in FUF/SURGE it is often possible to adopt a workaround which is more directly grammatical in nature.

6.6 RealPro

6.6.1 An Overview

Of the three systems we discuss in this chapter, REALPRO is functionally the simplest. REALPRO looks to Mel'cuk's MEANING-TEXT THEORY (MTT) for its theoretical linguistic base. The system takes as input a DEEP SYNTACTIC STRUCTURE, termed a DSYNTS in MTT. Oversimplifying to some degree, a DSyntS can be thought of as a parse tree which does not specify function words or word ordering, but does specify content words, syntactic roles (for example, the subjects and objects of verbs), and syntactic features (for example, tense). REALPRO's function is to produce a surface sentence from such a structure by adding appropriate function words, deciding on the the appropriate inflections to use for content words, and specifying the order in which content and function

```
help
(I (John)
  II (Mary))
```

Figure 6.25: The DSyntS for a simple sentence

1	help [aspect:cont] (I (John) II (Mary))	*John is helping Mary*
2	help [polarity:neg] (I (John) II (Mary))	*John doesn't help Mary*
3	help [aspect:cont, polarity:neg] (I (John) II (Mary))	*John is not helping Mary*

Figure 6.26: Controlling the realisation of DSyntSs

words should appear in the output sentence.

6.6.2 The Input to RealPro

As just noted, the input to RealPro is an expression called a DSyntS. Figure 6.25 shows the DSyntS that corresponds tp the simple sentence *John helps Mary*, in a context where the words *help*, *John*, and *Mary* have been appropriately defined in RealPro's lexicon. This structure states that the head verb of the sentence to be generated is *help*, that the subject of the verb is *John*, and the the object is *Mary*.

So far this is not particularly interesting. The example we have just seen assumes a number of default values for features that affect the output; by making these features explicit and giving them values other than the defaults, we can begin to see the power that comes even from this simple abstraction away from the surface level of the sentence. Figure **??** shows a number of variations on the basic form used above. In the first example, we add the feature–value pair [aspect: cont] to the verb in order to generate the progressive form of the original sentence. The second and third examples show our first two sentences can be negated simply by adding the feature [polarity: neg]. The different syntactic consequences of negation—in the case of the non-progessive form, the *do* auxiliary verb needs to be added—are handled transparently by the realiser, so that the earlier stages of the generation process do not need to be concerned with these idiosyncracies.

Figure 6.27 shows the DSyntS corresponding to the sentence *March had some*

```
HAVE1 [tense:past]
(I  March [class:proper_noun]
 II day [class:common_noun number:pl]
      (ATTR rainy [class:adjective]))
```

Figure 6.27: A DSyntS input to REALPRO

```
[head: have
 tense: past
 subject: [head: March
           class: proper-noun]
 object: [head: day
          class: common-noun
          number: pl
          attr: [head: rainy
                 class: adjective]]]
```

Figure 6.28: An AVM representation of the structure in Figure 6.27

rainy days; Figure 6.28 re-expresses this in our by-now familiar AVM notation; we have taken the liberty here of replacing MTT's I and II labels with the terms subject and object. Figure 6.29 shows a DSyntS for the sentence *The month was cool and dry with the average number of rain days.*

6.6.3 Meaning-Text Theory

Meaning Text Theory is a form of DEPENDENCY GRAMMAR: the analyses the theory provides of sentences do not consist of phrase structure constituency descriptions, but instead identify heads and modifiers that are dependent on those heads. Modifiers themsevels may have dependent elements, so that quite complex dependency structures can be constructed. These are represented as trees where the nodes are words and the arcs connecting words are the labelled dependencies. This means that, for example, in the analysis of a typical sentence the verb will be the root of the tree, with its arguments being its daughters, connected by arcs labelled with the grammatical roles of those daughters (subject, object and so on).

```
BE1 [tense:past]
(I  month [class:common_noun article:def]
 II cool [ class:adjective]
    (COORD AND2 [ ]
          (II dry [class:adjective]))
  ATTR WITH1
     (II  number [class:common_noun article:def]
          (ATTR  average [class:adjective]
           ATTR  OF1
             (II day [class:common_noun number:pl article:no-art]
                 (ATTR rain [class:common_noun])))))
```

Figure 6.29: A more complex DSyntS input to RealPro

I	the subject of a clause
II	the object of a clause, or of any other type of phrase which requires an object, such as a prepositional phrase
III	the indirect object of a clause
ATTR	a phrase modifier—an adjective, adverb, prepositional phrase or relative clause
COORD	a conjunction

Figure 6.30: Common relations in MTT's DSyntS expressions

MTT views the generation of language as being a process of mapping iteratively through seven levels of representation. By taking DSyntS structures as input, RealPro effectively assumes that prior stages in the NLG process have already performed some of these mappings.

As already indicated, a DSyntS is essentially a tree whose nodes represent content words, and whose arcs represent the deep-syntactic relations that hold between content words. Features can be added to nodes to indicate that a particular word should be put into the plural form, put in past tense, and so on. The arcs between words are drawn from a finite set defined in the theory; the most common of these are shown in Figure 6.30.

In the light of this background, we can now further elaborate on some of the elements in the DSyntS for the sentence *March had some rainy days*, as shown in Figure 6.27:

HAVE1: This is the sentence verb *have*, and the root node of the DSyntS. The feature [tense: past] tells RealPro to put the sentence in past tense; the

realiser knows that *have* is an irregular verb and uses the irregular form *had*. Because *have* is a common auxiliary verb in English, it is included in RealPro's built-in lexicon, under the identifier HAVE1.

March: This is the subject of the sentence, as indicated by the I relation from the verb HAVE1 to this node. The feature [class: proper-noun] tells RealPro that March is a name; the realiser therefore knows that a determiner is not required.

day: This is the object of the sentence, as indicated by the II relation from the verb HAVE1 to this node. The feature [class: common-noun] tells RealPro that day is a common noun. The feature number:pl indicates that the plural form *days* should be used; this information is also used when selecting the determiner *some*. Note that *some* is not represented explicitly in the DSyntS; because it is a function word rather than a content word, it is the realiser's job to include it automatically.

rainy: This modifies day, as indicated by the ATTR link from day to this node. The [class: adjective] feature indicates that this is an adjective; this causes the realiser to place the modifier before the noun. Other types of modifiers, such as prepositional phrases, would be placed after the noun.

The class features in this DSyntS could be omitted if the words in question—March, day, and rainy—were present in RealPro's lexicon.

6.6.4 How RealPro Works

As noted earlier, from the perspective of NLG, MTT divides the generation process into seven steps. Some of these steps are not relevant for systems which generate written English (as opposed to, say, spoken Japanese), and in fact RealPro only implements the following stages:

Deep Syntactic Component: This converts the DSyntS into a SURFACE SYNTACTIC STRUCTURE or SSyntS. An SSyntS includes function words (such as *some*), and also uses more specialised arc labels.

Surface Syntactic Component: This LINEARISES an SSyntS, that is it decides in which order words should appear. Its output is a DEEP MORPHOLOGICAL STRUCTURE or DMorphS.

Deep Morphological Component: This applies morphological processing to produce appropriately inflected words from the contents of the DMorphS. Its output is a SURFACE MORPHOLOGICAL STRUCTURE or SMorphS.

Graphical Component: This carries out orthographic processing on the SMorphS. Its output is a *deep graphical structure* or DGraphS, which essentially is a complete representation of the sentence.

```
DSYNT-RULE:

    [(X I Y)]               |       [(X [class:verb])]
        <-->
    [(X predicative Y)]

SSYNT-RULE:

    [(X predicative Y)]     |       [(Y [number:?n person:?p])
                                     (X [class:verb])]
        <-->
    [(Y < X)]               |       [(X [number:?n person:?p])]
```

Figure 6.31: Some simple RealProgrammar rules

There is also an initial pre-processing stage which adds default features (for example, if no number is specified for a noun, it is assumed to have number:sg); and a final post-processing stage which converts the DGraph into a standard mark-up language such as HTML.

RealPro's grammar is the set of rules which convert DSyntSs to SSyntSs, SSyntSs to DMorphs, and so on. These are essentially pattern-matching rules which in many ways are similar to the production rules used in many expert systems. Two examples of RealPro rules are shown in Figure 6.31. The first rule is a DSyntS-to-SSyntS rule which replaces the DSyntS I relation with an SSyntS predicative (that is, subject) relation if the source (governer) of the relation is a verb. The second rule is an SSyntS-to-DMorphS rule which states that if a predicative relation occurs between a verb x and some other constituent y, then y should precede x in the sentence, and furthermore that x should have the same person and number features as y. These constraints are reminiscent of the realisation statements found in the NIGEL grammar discussed in Section 6.4.

In Figure 6.32, we show how these rules are used to realise a simple DSyntS which corresponds to the sentence *Sam sleeps*. This example is over-simplified, and we have not, in particular, shown the other rules that would be needed to process this DSyntS, such as the rules which select the correct inflection of the verb.

6.6.5 Summary

Of the systems we have surveyed, RealPro is the simplest: it takes as input what we have earlier termed an ABSTRACT SYNTACTIC STRUCTURE, and applies rules of grammar and morphology to this to produce an output sentence. Of the three systems, it correspondingly requires the microplanner to do the most work in constructing input specifications.

```
System input (DSyntS)

    sleep [class:verb]
    (I  Sam [class:proper-noun])

With defaults added (DSyntS)

    sleep [class:verb]
    (I  Sam [class:proper-noun person:3rd number:sg])

After DSYNT-RULEs applied (SSyntS)

    sleep [class:verb]
    (predicative  Sam [class:proper-noun person:3rd number:sg])

After SSYNT-RULEs applied (DMorphS)

    [ (Sam   [person:3rd number:sg])
      (sleep [person:3rd number:sg])

Final output

    Sam sleeps.
```

Figure 6.32: Stages of realisation in RealPro

REALPRO is also different from KPML and SURGE in that it is based on MTT instead of SFG. Advocates of MTT from a natural language processing perspective (for example, [Goldberg et al 1994]) claim that it is well-suited to NLG realisation because it modularises the realisation process into well-defined substages. Although REALPRO's input is an abstract syntactic structure, this is not required by MTT; indeed MTT includes a semantic representation level which has been used by other MTT-based realisers, such as ALETHGEN/GL (ref).

It is difficult to compare the pros and cons of SFG and MTT, not least because SFG and MTT advocates mostly compare their theories to the phrase-structure linguistic models which dominate natural language understanding research. In practice, engineering issues such as speed, reliability, documentation quality, and cost may be more important in choosing a realiser for an NLG system than the grammatical theory on which it is based.

As with the other systems we have described, making use of REALPRO requires a significant investment of time spent learning about both MTT in general as well as the realisation engine itself. Once more, it is important to emphasise that all of these systems require the user to undertake specific theoretical commitments in order to use the machinery in a consistent fashion.

6.7 Choosing a Realiser

We have taken the view in this chapter that it makes sense to use an existing realiser, rather than build one from scratch. How do you decide which realiser to use? Of course, the answer to this question depends on the particular application you are developing.

As we have demonstrated, the three systems we have described differ on two basic dimensions:

- the level of abstraction at which the input structures to the realiser are specified; and
- the particular grammatical theory embodied in the realiser.

There is a third dimension which we have only alluded to in passing: differences in grammatical coverage. The particular coverage offered by any system depends historically on its developmental path. It is a reality of using packages like these that, inevitably, one finds that there are gaps in the systems' grammatical coverage: there will always be grammatical structures that one wishes to build that are not catered for by the existing grammars. This is, of course, no different to the situation that occurs with the use of existing grammars for parsing: no matter how broad coverage such a resource is, one always finds there are structures that it does not cater for. From this point of view, it is best to take the view that the systems described here are are still under development. We have found the developers of both systems extremely helpful in assistance with the making of grammatical extensions, but clearly this kind of support is not sustainable on a wider basis unless resources are specifically allocated to the task.

Of course, as time goes on and the development of these systems continues, the effect of this difference will be reduced. The first two differences we identified remain. Unless one has a specific pre-existing theoretical commitment that causes one to choose one system rather than the others, the most important issue to consider is thus the level of abstraction of the input specifications the realisers expect. In some systems phrase specifications are produced from scratch in the microplanner by doing deep reasoning on the information that needs to be communicated and the linguistic resources available to communicate this information. In such systems, it may be best to use a realiser whose input is expressed in semantic terms; KPML or SURGE) might then be appropriate in such circumstances, since these will minimise the amount of work required by the microplanner. In other systems, in contrast, phrase specifications may originally be written in skeleton form by human developers, and then instantiated or otherwise modified by the document planner and microplanner. In such systems it is important that skeleton phrase specifications be easy for human authors to write. This may suggest using a phrase specification representation that is expressed in terms closer to the lay-linguistic terminology one might expect users of the system to be familiar with, and in such cases a realiser like RealPro which deals in terms of abstract syntactic structures may be most appropriate.

There is one context where it may make sense to build a realisation component from scratch: that is, in situations where the NLG system uses templates as defined in Section 6.3.5. In such contexts, it would be extravagant to bring the power of a full-blown realiser to bear; if all that is required is relatively straightforward morphological and orthographic processing, then it may be appropriate to build a bespoke software component to carry out these tasks. Such a component should, of course, be sophisticated enough to handle a broad range of morphological irregularities; see Ritchie *et al*[ref] for a discussion of many of the issues here.

In summary, to repeat our position at the outset of this chapter: given the availability of these components, each of which comes with a reasonably broad-coverage grammatical resource, it is not clear that there is much to be gained from building one's own realisation component. The development effort required to build grammars with the coverage of the systems described here should not be underestimated. At the same time, the effort required to come to terms with any of these systems is not insignificant. It should also be realised that these tools are very powerful—perhaps more powerful than many applications require. Indeed, for many NLG scenarios, a simple template-based realisation component maybe suffient, where the only grammatical processing required is morphological processing to produce inflected forms of nouns and verbs, and orthographic processing to ensure correct punctuation and capitalisation.

6.8 Bidirectional Grammars

In each of the systems we have described so far, a distinction can be drawn between, on the one hand, the grammatical and lexical resources used to char-

acterise information about language, and on the other, the engine that makes use of those resources. This raises the question of whether the grammar or lexicon in each case is necessarily something that can only be used for generation, or whether it can be viewed as a rather more declarative description that could in principle be used for either natural language generation or natural language analysis.

In the case of the particular systems we have surveyed, the grammatical resources are not specified in a manner that makes them neutral with respect to their direction of use; in each case there are assumptions made with regard to the processing of the representations that effectively makes them all generation-oriented grammars. However, the idea that one might specify a grammatical resource as a something that is independent of this aspect of its use is not new; indeed, as we saw in Section 6.5.3, this very possibility was one of the motivating factors in Kay's original development of Functional Unification Grammar, the progenitor to Elhadad's FUF. The idea has also received attention within work on Machine Translation, where there is a desire to economise on the construction of grammatical resources by allowing a grammatical description of a language to be used both for the analysis of the source language and for the generation of the target language.

In more recent years, the idea of a bidirectional grammar has surfaced in the context of discussion of algorithms for head-driven generation (see, in particular, Sheiber [ref] and Van Noord [ref]). The basic idea here is that a grammar provides a declarative specification of the relationships that hold between meaning and surface form. Some processing engine—a surface realiser or a parser—can then take an expression that is an instance of one of the relata and, given the grammar, produce an instance of the other relatum.

The simplest variants of this idea can easily be explored using Definite Clause Grammars in Prolog, where Prolog's top-down backtracking processing regime can be used to perform a mapping from an appropriately expressed surface form to a representation of semantic content or *vice versa*. More sophisticated work in the area is concerned with issues of the control of search which arise as soon as grammatical descriptions of any complexity are considered.

The ideas here are clearly appealing. It seems redundant to specify grammatical information once for generation and then again for analysis. However, bidirectional systems have not been widespread in the larger context of the kinds of NLG systems we describe in this book.

It is instructive to consider why this is the case. If we look at current experiments in the use of bidirectional grammars, it is immediately apparent that the kinds of representations used as input to realisation are not the same as the representations (such as SPL or DSyntS expressions) that are generally provided by microplanning components. The mappings encoded in these bidirectional grammars tend to focus on correspondences between logical forms and surface strings, where those logical forms express the propositional content of the corresponding sentences. This is generally deemed appropriate as far as natural language parsing is concerned. However, in NLG systems, there is generally a concern with kinds of information that can be ignored from the point of view

of the parsing process. A parser might produce the same logical form for the following three sentences:

(6.6) a. Mary gave John a ball.
b. Mary gave a ball to John.
c. John was given a ball by Mary.

The input to a realiser, however, must explicitly provide a means of choosing between these forms; the alternative is to choose at random.

This point relates to our discussion in Section 6.4.4 of aspects of meaning other than the propositional. Viewed simply from the point of view of propositional content, there are many ways of saying the same thing. Winograd [ref], for example, lists all the following as providing valid descriptions of the same state of affairs in the world:

(6.7) a. Jon bought a painting for Vina.
b. Jon bought Vina a painting.
c. Vina was bought a painting by Jon.
d. What Jon bought Vina was a painting.
e. What Vina was bought by Jon was a painting.
f. It was a painting that Jon bought for Vina.
g. It was Jon that bought Vina a painting.
h. It was Vina that Jon bought a painting for.

Although these may all be equivalent in propositional meaning, the sentences are not mutually interchangeable. This is easily demonstrated by identifying a larger discourse context in which one is appropriate and then attempting to substitute the others.

How then do we characterise the differences between these sentences? Generally this is viewed in terms of notions like FOCUS, GIVEN *vs* NEW INFORMATION, and so on; just those phenomena we earlier referred to as making up textual meaning. This is not to say that there are no random selections to be made between different ways of saying things; however, there is the perception in the NLG community that most choices are motivated in some way. The input specification should therefore include whatever information is required in order to generate these different outputs; in a true bidirectional resource, these elements would also be part of the output of analysis.

In practice, this is not the case. There are two related reasons for this. First, much of the work in the area of bidirectional grammars has its origins in the parsing world, bringing with it a focus on concerns like the control of search strategies. Second, the linguistic models adopted in these approaches tend not to highlight these non-propositional aspects of meaning.

This is not a necessary property of attempts to build bidirecitonal grammatical resources: provided we could characterise the correspondences between

surface forms and a richer notion of 'meaning' appropriately, we could conceivably have analysis systems which would use such a resource to produce these richer outputs, and at the same time the current source of indeterminacy that arises when these resources are used for generation would be removed. At the current time, however, we do not have appropriate characterisations of the relationships between meaning and surface form.

There is also a less theoretically motivated and more pragmatic reason for the lack of use of bidirectional grammars. For generation we only want to construct grammatical utterances; in real natural language analysis systems, however, we need to be able to cope with any sentence which a human user might produce. Grammars for use in parsers therefore also be annotated with information intended to help resolve ambiguities or robustly handle errors in human-entered text. This kind of information is not required for surface realisation.

6.9 Further Reading

The literature on surface realisation is considerable. Just as a significant amount of work in natural language understanding has focussed on syntactic analysis and parsing, it is possible that more has been written on linguistic realisation than any other aspect of NLG. Many grammatical formalisms and theories that have been developed to describe natural languages have been used in the construction of realisation components. Although we have focussed here the use systemic functional grammar and meaning text theory in linguistic realisation, other linguistic formalisms which have been used in parsing frameworks have also been used in realisation: for example, Tree Adjoining Grammars, Categorial Unification Grammar, Lexical Functional Grammar, Government and Binding theory, and of course Head-driven Phrase Structure Grammar.

One aspect of surface realisation that has not been explored very much in the literature is the issue of converting the logical structure found in text specifications into physical presentational mechanisms. To date, most systems have simply converted logical structure into a markup language such as HTML or LaTeX, in a very straightforward fashion, and relied on an external postprocessor (such as a Web browser) to perform the actual conversion. This reflects a tradition in the field of NLP as a whole whereby the focus is on what we have called TEXT, abstracted away from specific physical renderings. Nunberg [ref] provides some interesting discussion on the difference between what he calls LINGUISTIC GRAMMAR and TEXT GRAMMAR, but the most appropriate characterisation of the relationship between these levels of representation remains to be worked out.

Mellish (??) provides a good theoretical overview of different approaches to realising phrase specifications; and Reiter (1995) discusses at a very high level the pros and cons of strings *vs* more abstracted grammatical representations.

Numerous papers have been published on KPML, on its predecessor system PENMAN, and on the NIGEL grammar of English. One introductory paper is Bateman (1998?); Matthiessen and Bateman (1990) is also valuable. Halliday

(??) is the standard introduction to systemic functional linguistics.

The best introduction to SURGE is Elhadad and Robin (1998??); more detailed information about both FUF and SURGE can be downloaded from `http://www.cs.bgu.il/fuf`. Elhadad *et al* (1997) describe a very sophisticated use of SURGE, which illustrates the power of the system.

A general introduction to REALPRO is provided in Lavoie and Rambow (1997); more details are available in the system documentation, available from `http://www.cogentex.com/systems/realpro.html`. Mel'cuk (1988) describes Meaning Text Theory, the linguistic theory underlying REALPRO.

The most well-known algorithm for realisation in a bidirectional grammar framework is called SEMANTIC-HEAD-DRIVEN GENERATION; see Shieber et al (1990) and van Noord (1990) for an introduction to these ideas. Busemann (1996) discusses some of the problems with using 'inverse parsers' as realisers in NLG systems.

Although we would recommend that those interested in building complete NLG systems make use of one of the existing realisation components, if the aim is to construct a simple template-based system, then it may make sense to build a morphological processor. Ritchie *et al* (1992) discuss morphology in depth; Quirk and Greenbaum (1973) is a general source on grammar with some useful information on morphology.

Nunberg (1990) describes some of the linguistic rules behind the correct use of punctuation; Dale [ref] and White (1995) discuss how these can be used in the context of NLG systems.

Relatively little has been written discussing under what circumstances particular approaches to realisation should be preferred, but see Reiter (1995) and Reiter and Mellish (1993) for some comments on this. Geldof (1997) describes a 'structured strings' systems, and Jakeway and DiMarco (1996) discuss a simple authoring tool for SPL.