Natural Language Generation for dialogue: system survey

Mariët Theune
Parlevink Language Engineering Group
University of Twente
theune@cs.utwente.nl

May 6, 2003

Contents

Introduction

Many natural language dialogue systems make use of 'canned text' for output generation. This approach may be sufficient for dialogues in restricted domains where system utterances are short and simple and use fixed expressions (e.g., slot filling dialogues in the ticket reservation or travel information domain); but for more sophisticated dialogues (e.g., tutoring dialogues) a more advanced generation method is required. In such dialogues, the system utterances should be produced in a context-sensitive fashion, for instance by pronominalising anaphoric references, and by using more or less elaborate wording depending on the state of the dialogue, the expertise of the user, etc. In the case of spoken dialogues, it is very useful if the natural language generation component can provide information that is relevant for determining the prosody of the speech output. Similarly, for use in embodied agents it is useful if the generation component can provide information about the facial and body movements that should accompany the language being produced by the agent. Clearly, it will be extremely difficult to achieve all this using simple string manipulation, so a more flexible and context-sensitive generation method is required.

This report discusses some of the possibilities for the sophisticated generation of system utterances in a dialogue system. The basic assumption is that this task is performed by a separate language generation module, which takes as its input a message specification produced by a dialogue planner and transforms this message into an expression in natural language. Part I of this report provides a general discussion of different methods for performing this task, and outlines some requirements on language generation systems that might be used for this purpose. Part II gives an overview of publicly available language generation systems, and discusses to what extent they meet the previously stated requirements.

Part I
NLG in dialogues

1 Introduction to natural language generation

Natural language generation (NLG) is the process of automatically creating natural language on the basis of a non-linguistic information representation. This may be, for instance, information from a database or an abstract message specification provided by the dialogue manager in a dialogue system. Most work on language generation is aimed at the production of written texts. For this, *text generation* would be a better name. Nevertheless, following common practice, throughout this report the more general term language generation will be used.

It is generally assumed that the language generation process can be split up into three stages: document planning, microplanning, and surface realisation [Reiter and Dale, 2000]. Below is an overview of the generation tasks that are performed within each stage.

1. Document planning
    - *Content determination* is the task of converting the raw system input to the specific kind of data objects (or 'messages') that serve as a basis for the subsequent generation tasks. An important subtask here is deciding which input information to express in the text.
    - *Document structuring* (or discourse planning) is the process of ordering the information to be expressed and determining the structure of the output text. The result is a text plan, often in the form of a tree structure representing the order and grouping of the messages, and the relations between them.
2. Microplanning
    - *Lexicalisation* is the task of choosing the right words to express the input information. Often, a concept can be expressed using different words or phrases; the task of the lexicalisation procedure is to choose the one that is most appropriate in the given context.
    - *Aggregation* is the process of deciding which information to put in one sentence. Pieces of information that form separate input messages may be joined together and expressed using one sentence which, for instance, contains a conjunction or a relative clause.
    - *Referring expression generation* is the task of creating phrases to identify domain entities. This involves choosing the type of expression (e.g., a pronoun or a definite description) and in the case of definite descriptions, choosing the properties to include in the description.
3. Surface realisation
    - *Linguistic realisation* is the task of creating grammatical sentences. This involves the application of syntactic and morphological rules that determine aspects like word order and agreement.
    - *Structure realisation*, finally, is the task of converting the output text into some required external format, for instance by adding HTML mark-up or LaTeXcodes. This is not a generation task *per se*, and in many language generation systems it is not included.

In general, it can be said that the document (or text) planning tasks of stage 1 are language-independent but domain-specific, whereas linguistic realisation (stage 3) is language-specific, but can in principle be done in a domain-independent fashion. This difference in domain-dependence of the different tasks largely explains why reusable software packages exist for linguistic realisation, but not for document planning. The tasks associated with stage 2, that of sentence planning, require both domain and language-specific knowledge. For instance, the generation of referring expressions requires knowledge about the application domain (information about domain objects and their properties) and about the application language (e.g., the syntactic properties of modifiers expressing certain properties).

Almost all NLG systems discussed in the literature are aimed at the generation of stretches of text to be read (or listened to) without interruption by the user. NLG is rarely used for the generation of system utterances within a dialogue system; this task is generally performed in a non-linguistic, ad-hoc fashion, using canned text or simple string manipulation.[1] Some reasons for this are suggested in the next section, which discusses the differences between language generation in a 'monologue' or a dialogue situation.

2 Dialogue versus monologue generation

In a dialogue system, the input for the language generation component (if any) generally consists of a message specification produced by the dialogue system's planning component (also called the *dialogue manager*). This message specification, which may have the form of a QLF [Alshawi and Crouch, 1992] or another kind of meaning representation, describes the content to be expressed in the system's next dialogue turn. This means that it is the dialogue manager which decides what to say and when to say it, thus performing the generation tasks of *content determination* and *discourse planning*. The responsibility of the language generation component is restricted to the *formulation* of the system utterances, i.e., the decision 'how to say it'. This process is sometimes referred to as 'tactical planning', and in principle it includes all tasks that make up the microplanning and surface realisation stages of the generation process.

Many dialogue systems operate in such a restricted domain that it is possible to perform most microplanning tasks in a rather limited way, or not at all. This holds in particular for the most common dialogue applications such as air travel information and reservation systems. Often, the utterances to be produced by such dialogue systems consist of one sentence only (either a question or a short answer to a question), so performing *aggregation* is hardly necessary. In addition, *lexicalisation* and the generation of *referring expressions* generally require only limited linguistic knowledge, because most systems operate in restricted domains where fixed expressions are common. This leaves *linguistic realisation* as the principal task to be performed for the generation of dialogue utterances. This task can often be carried out using a simple string manipulation approach, because the dialogue domain is usually so restricted that it is possible to list all desired output strings (after abstracting over variable information).

[1] Only recently, generation in dialogues has started to receive more widespread attention, cf. Freedman and Callaway [2003].

In sum, in many practical dialogue systems there appears to be no direct need for a full-fledged language generation component, since relatively simple techniques are sufficient to produce adequate language output. However, output generation by means of such simple methods is seriously lacking in flexibility and context-sensitivity. This may be a problem in application domains involving more sophisticated dialogues where it is less acceptable for the system to behave in a machine-like fashion, such as tutoring and social dialogues [Jordan et al., 2003, Mathews et al., 2003, Green and Davis, 2003].

Human speakers in a dialogue take different aspects of the dialogue context into account when formulating their utterances. For instance, information on the form and content of preceding dialogue utterances is used to produce elliptic utterances and pronominalised references to previously mentioned entities. Speakers also have a tendency to use the same wording as their dialogue partner (the 'priming' effect noted by, among others, Levelt and Kelter [1982] and Clark and Wilkes-Gibbs [1986]). In addition, the style of the dialogue utterances depends on the speaker's relation to the listener (e.g., salesperson versus customer, teacher versus student), on different characteristics of the listener (expert or novice, adult or child), and on the speaker's emotional and cognitive state (happy or sad, certain or uncertain). Finally, in situations where speaker and hearer share some visual domain, information on what is mutually visible may be used to produce deictic expressions such as *here, there, this* and *that*. Ideally, a system for the automatic generation of dialogue utterances should also take these contextual aspects into account. Clearly, doing so using a simple string manipulation approach will be difficult (if not impossible), so a more advanced form of language generation is needed. Different approaches to language generation can be distinguished, and these are discussed in the next section, paying special attention to their usability for the generation of dialogue utterances.

3 Generation approaches

It is a reasonable assumption that a message specification produced by the dialogue manager of a dialogue system will at least contain information about the speech act, predicate and arguments of the message to be expressed in the system's next dialogue contribution. To produce a natural language utterance on the basis of such a specification, some form of language generation is required. This section presents four possible approaches to the generation of dialogue utterances, corresponding to four types of NLG systems (or NLG system components) that might be used. An overview is given of the advantages and disadvantages of each approach. The currently available NLG-systems in which the different approaches are taken, are discussed in Part II of this report.

3.1 'Reversed parsing'

In language analysis, the surface form of a natural language expression is mapped to a representation of its meaning, using a grammar that encodes the relationships between meaning and surface form. Language generation (at least in the context of a dialogue system) starts with a meaning representation, and maps it to a surface form expressing this meaning. Because both analysis and generation deal with the relationship between meaning and surface form, only in opposite

directions, it seems an obvious step to treat generation as the reverse of parsing and to try using a single grammar for both purposes. A well-known generation algorithm based on this idea is the semantic head-driven generation algorithm [Shieber et al., 1990]. Important advantages of the 'reversed parsing' approach for language generation in a dialogue system are that in principle, it allows for (*i*) the use of one grammar for both parsing and generation, and (*ii*) the use of a single meaning representation format in all system components: the output of language analysis has the same format as the input for language generation. Other advantages are that it is psycholinguistically plausible, it allows for input/output consistency (the system can understand everything it can generate and vice versa), and that it allows for adapting the generated output to speech of others [Neumann, 1994].

Despite the attractive simplicity of the underlying idea, the 'reversed parsing' approach has received relatively little attention in the natural language generation community. An important reason for this is that in practice, parsing and generation make such different demands on a grammar that it is difficult[2] to use the same grammar for both purposes. As pointed out by Reiter and Dale [2000], a parser may return the same logical form for many different sentences that roughly express the same (propositional) meaning.[3] As an example they give the following sentences, which can all be mapped to a logical representation such as ∃x [give(m,j,x) & ball(x)]:

- Mary gave John a ball
- Mary gave a ball to John
- John was given a ball by Mary

When such a meaning representation is given as the input to a sentence realiser, a method is required to choose between the different surface forms that can be used to express this meaning. The choice of surface expression is not restricted to the choice between different syntactic constructs, as illustrated by the example sentences, but also includes lexical choice (*Mary handed John a ball*), choice of referring expressions (*She gave him a ball*) and aggregation (*Mary gave John something. It was a ball*). Reversed parsing does not provide the means to make these kinds of choices, and random selection is not likely to give good results. Note that most of the choice points listed above are situated at the level of *microplanning*, the second stage of language generation as discussed in Section 1. 'Reversed parsing' only covers the third and last stage, that of realisation.

Another, relatively minor, reason to avoid using the same grammar for both parsing and generation is that parsing grammars are usually set up in a robust fashion, so as to deal with partial or (slightly) ungrammatical input, whereas a generation grammar should of course only produce output which is complete and grammatically correct. Similarly, parsing grammars often include information intended to to help resolve ambiguities, which is not needed during generation.

Several of the problems mentioned above may be solved by using a grammar that is specifically tailored to the generation task, instead of trying to use one

[2]But not impossible, see Shieber et al. [1990], Gerdemann [1991], van Noord [1993], Neumann [1994], Erbach [1997], Goldwater et al. [2000], Lemon et al. [2003].

[3]For an in-depth discussion of this problem of 'logical-form equivalence' see Shieber [1993].

grammar for both parsing and generation. Such a generation-oriented grammar would only describe those expressions that the system should be capable of uttering. This is the approach taken in most grammar-based generation systems that are currently publicly available,[4] so from now on, we will use the term 'reversed parsing' in a loose sense, referring to grammar-based systems that take as input the kind of logical meaning representation that is generally produced by language analysis – that is, input which is not specifically aimed at language generation (unlike the detailed sentence plans required by most specialised linguistic realisers, to be discussed in the next section). These systems may or may not use a truly bidirectional grammar.

'Reversed parsing' appears to be a feasible approach to language generation in simple dialogue systems where the range of desired expressions is quite limited. For instance, many such systems only need to produce active sentences in the simple present tense. This means that choices concerning verb tense and aspect, and between active and passive voice may be avoided by simply including only one of these constructions in the grammar. (See also Bunt [1988].) In addition, the lexicon used for generation may be limited so that each input concept is associated with one word only. For instance, the neutral verb *give* may be chosen as the standard expression for the concept of 'giving', thus avoiding having to choose between alternatives such as *hand*, *donate*, etc. Similarly, fixed expressions may be used to refer to objects in the domain of discourse. Finally, in a dialogue system the input for generation is usually provided in sentence-sized chunks, so the aggregation issue may be safely ignored.

A serious drawback of this approach is that in the absence of a choice mechanism, the system's output cannot be adapted to the circumstances in which it is produced: the same meaning representation will always be mapped onto the same surface form. For instance, the generator cannot decide to use a pronoun after having referred to the same object in several subsequent turns, unless this is explicitly prescribed in its input - that is, this decision is the responsibility of the dialogue manager rather than the NLG component. (Or, alternatively, an additional planning component is required to enrich the message specifications produced by the dialogue manager, cf. Bratt and Dowding [2003].) All in all, it seems that the 'reversed parsing' approach does not offer much more flexibility than a simple string manipulation approach, although it is, of course, a far more elegant and robust solution to the generation problem. As discussed in Section 2, for the generation of natural, sophisticated dialogue utterances at least some context-sensitivity is required. This holds in particular for the generation of utterances that consist of multiple sentences.

In short, for tasks that are more demanding than the generation of isolated sentences, a richer message specification is necessary than would normally be produced by a parser or a dialogue manager. Such a meaning representation would have to represent additional non-propositional meaning aspects, e.g., relating to information structure and speech style. (This kind of 'rich' input is actually what is expected by most of the specialised linguistic realisers discussed in Sections 3.2, 6 and 7.) The disadvantage of this alternative is that an additional *sentence planner* will have to be used to produce this kind of rich message specification.

[4] A notable exception is Gemini, which is used with a bidirectional grammar for both parsing and generation in the CommandTalk and WITAS systems [Goldwater et al., 2000, Lemon et al., 2003].

3.2 Using a linguistic realiser

The term 'linguistic realiser' refers to a software system that has been specifically designed for the NLG task of linguistic realisation, i.e., determining the surface form of a system utterance. A linguistic realiser is set up specifically to deal with the problem of choosing the most appropriate surface form, if more than one is available. Several general, reusable linguistic realisers are publicly available for research purposes. These are aimed at the generation of single sentences, which may or may not be part of a larger text. As input they take some form of phrase specification or 'sentence plan', which is much richer than a standard logical meaning representation. To map this input to a grammatical sentence, one of two approaches can be taken: either using a specialised generation grammar or using some form of sentence templates, where parts of the output sentence to be produced are already hard-coded in the templates. These two different approaches are discussed in Sections 3.2.1 and 3.2.2 below.

3.2.1 Grammar-based realisation

Grammar-based linguistic realisers offer an elegant and theoretically attractive method for the generation of dialogue utterances. Generation is based on the use of a wide-coverage, theoretically motivated grammar of the target language. The fact that the grammar is tailored for natural language generation means that the problems with bidirectional grammars, discussed in Section 3.1, are avoided. All currently available realisers come with at least a generation grammar for English, and sometimes also for other languages. If the chosen linguistic realiser comes with a grammar for the application's target language, little effort needs to be spent on grammar development (although some customising to the application domain will always remain necessary).

Of course, the use of a grammar-based linguistic realiser for NLG in dialogues also has its disadvantages, which are mainly of a practical nature. First of all, the input of existing grammar-based realisers is much richer and more specific than the logical representations typically produced by a dialogue act planner. An example is given in Figure 1. It shows an example input to the KPML system [Matthiessen and Bateman, 1991, Bateman, 1997], which is used to produce the sentence *March had some rainy days.*

We see that the input to KPML contains several linguistic details, such as specifications of the lexical items to be used in the description of various concepts. To bridge the gap between this kind of rich sentence specification and, e.g., a logical meaning representation, some form of microplanning is needed. During microplanning, which is partially domain-specific, information about lexemes, tense, number and other properties of the utterance is added to the message specification. There are several ways to handle this task. The simplest way is to adopt a template-like approach, where the input to the microplanner is mapped directly onto a 'canned' phrase specification (see Marsi [2001] for a similar approach). Since only default phrase specifications will be produced, such an approach does not allow for much context-sensitivity, and therefore resembles the use of a simple, domain-specific generation grammar as discussed in Section 3.1. On the other hand, developing a 'real' microplanner, capable of making linguistically motivated, context-sensitive decisions involves a lot of work.

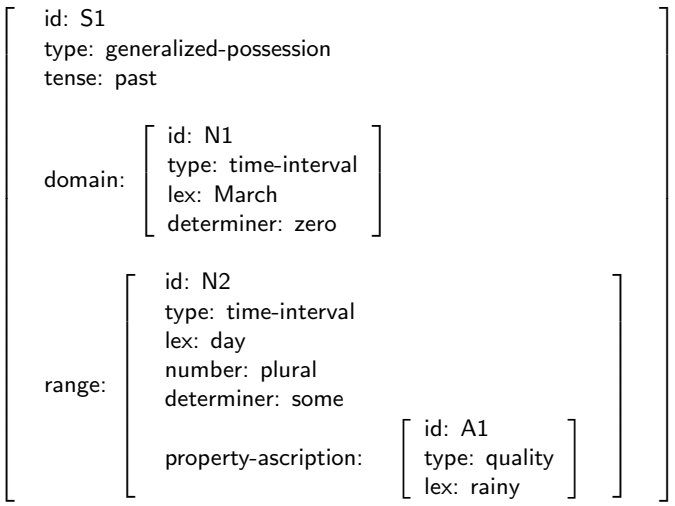

Figure 1: Example input to KPML, taken from Reiter and Dale [2000], p. 172.

For most dialogue systems the use of a broad coverage grammar, as supplied with many realisers, may be regarded as 'overkill'. Although such grammars are theoretically attractive, their sheer size and complexity often cause practical problems. For instance, linguistic realisers may be quite slow, in particular when relatively long utterances are generated [Marsi, 2001]. For use in a dialogue system, this is a rather unattractive property. Another related point is that existing grammar-based realisers are so complex that learning to work with them takes a lot of time and effort [Reiter and Dale, 2000]. As a consequence, it is not always clear (at least from a practical point of view) if much will be gained by using such a system.

3.2.2 Template-based realisation

An alternative method for linguistic realisation, which is more attractive from a practical point of view, is the use of templates. A template is "a predefined form that is filled in by information by either the user or the application at run-time" (McRoy et al. [2001], p. 23). This is a rather broad definition, and as we will see in Part II, the templates used by different template-based generation systems are quite different in nature.

Although theoretically less attractive than grammar-based realisation, the use of templates has an important practical advantage, which is that generation is much faster because the size and number of structures to be traversed is relatively small [Busemann and Horacek, 1998, McRoy et al., 2001]. In general, developing a new application is easier using a template-based system than using a grammar-based system, because the specification of templates requires less linguistic expertise than the addition or adaptation of grammar rules. Another advantage of template-based systems is that they can generally deal with input that contains less linguistic detail than is expected by grammar-based realisers. Because the templates are specified by hand, different templates can be specified for use in different contexts. Still, just like grammar-based realisers, template-based realisers expect all relevant contextual information to be provided by a

preceding planning component.

Template-based systems can be used for language generation in a wide range of applications; however, they are more suitable for some applications than for others. Because of their use of handmade templates, they are most suitable for use in domains that are limited in size and subject to few changes. Developing syntactic templates for a very large domain is very time-consuming, and when it is used to present information that is subject to many changes, maintenance of a template-based system may become a problem. Finally, an important practical disadvantage of template-based realisation is that it is almost completely domain- and application-specific. When a template-based system is to be used in a different application, or for another language, most of the templates will have to be replaced.

3.3 Using a full NLG system

The main NLG task to be performed when generating dialogue utterances is that of surface realisation. However, we have seen that – at least in more sophisticated dialogues – several microplanning tasks such as lexical choice and the generation of referring expressions are also involved. When using a full NLG system, these tasks are (in principle) all covered.[5] A full NLG system can take non-linguistic data as input; in general, any kind of input can be used as a basis for generation, as long as it has a predictable structure. Another advantage is that a full NLG system can be used for the generation of short dialogue utterances as well as longer stretches of text (for instance, presentations of information from a database or another external source). The latter task cannot be handled by a linguistic realiser alone, because that lacks the required document and microplanning components.

A disadvantage of using a full NLG system for the generation of dialogue contributions is that this task generally does not require the full power provided by the NLG system. As was pointed out previously, the generation of dialogue contributions does not involve document planning (this task is performed by the dialogue manager), and the role of microplanning is often fairly limited. This means that a full NLG system is over-equipped for the task at hand, and some of its components will have to be by-passed or disabled when the system is used in a dialogue context. Other adaptations may also be necessary, for instance to make the system maintain a discourse history across turns, and to make it accept its input in a piecemeal fashion instead of all at once at the beginning of the generation process, as is usually the case in text generation [Horacek, 2003, Theune, 2003]. All these modifications are related to the fact that NLG systems are generally used for the generation of texts (or spoken monologues), and not in a dialogue situation.

Finally, some customisation to the target application will also be necessary. To some extent, this holds for all approaches discussed here, but since a full NLG system has several components that are at least partly domain-specific (in particular in the area of microplanning), customisation may present more

[5] It should be noted that it is very difficult to build (efficient) NLG systems where each generation task is fully guided by linguistic principles. As a consequence, many applied generation systems can be characterised as *hybrid* systems, in the sense that some generation tasks are carried out on the basis of linguistic notions, whereas other tasks are performed using a non-linguistic method, e.g., by using templates for linguistic realisation.

work in this approach. Also, in NLG-systems that use templates for linguistic realisation, new templates must be constructed for each application.

4 System requirements

This section provides a general discussion of the properties which are relevant when judging whether a specific NLG system is suitable for use in a dialogue framework. Special attention will be paid to these properties in the system descriptions provided in Sections 5 to 8 below.

Obviously, the first requirement the NLG system should meet is that it is publicly *available* (at least for academic purposes). In this document, only systems are discussed which meet that requirement. However, availability in itself may not be sufficient. It is also important that the system is still actively supported or at least well-documented. Otherwise, learning to work with the system and customising it for the intended application may be very cumbersome.

Another important aspect, which has already been discussed above, is the kind of *input* which the system can deal with. Can it handle basic meaning representations or does it require sentence specifications that already contain a lot of linguistic detail? In the latter case, an additional microplanning component will have to be developed, which will add significantly to the work load involved in employing the system.

In the simplest dialogue systems only limited *context-sensitivity* is required of the NLG component. Still, even in these systems it is useful if the NLG system is capable of at least some context-sensitive behaviour, such as automatic pronominalisation of references to salient objects and (in the case of spoken dialogue systems) deaccentuation of previously mentioned items. In both cases, the NLG system must have some knowledge of the dialogue history, and must know how to use this knowledge [Theune, 2003]. Other kinds of context-sensitive behaviour may be relatively less important, but can still contribute considerably to the naturalness of the output and user-friendliness of the dialogue system. Examples of this are adapting the generated output to characteristics of the user, or to the 'personality' to be conveyed by the system. The latter applies in particular to dialogues where the system is represented by an *embodied* agent, which has a specific appearance and therefore is also expected to have a distinct personality. For embodied agents, preferably the language generation component should also be ably to specify any gestures or facial expressions that the agent should use to express (part of) the message content or to stress important parts of the utterance. The generation of nonverbal signals is context-sensitive, in that at least the dialogue history must be taken into account (see e.g., Cassell et al. [1994]).

Ideally, the NLG component should have a sufficient level of *generality* to be easily reusable across different domains and applications. For grammar-based realisers, the level of generality equals that of the grammar being used. When using a broad-coverage grammar, presumably only a few lexical items will have to be added when applying the system in a new domain, but when using a domain-specific grammar, many new lexical items (and, possibly, grammatical constructions) will have to be added. In a template-based system, nearly always a completely new collection of templates will be required.

Most existing NLG systems have English as their target language. Since

we need to generate dialogue utterances in *Dutch*, it will be quite useful if an NLG system comes with additional resources for Dutch, for instance, a Dutch grammar. If Dutch resources are not available, an important requirement is that it should be easy to develop them. Ideally, in a grammar-based system it should be possible to 'plug in' an existing grammar. For template-based (full) NLG systems, it should be examined if they contain any language-specific information outside the templates (which are necessarily language-specific).

The computational *efficiency* of the NLG system also influences its suitability, especially since the system is to be used in a dialogue situation, where it is quite undesirable if the user has to wait seconds for a system reply. When discussing the individual NLG systems in Sections 5 to 8 below, this property is only discussed in those cases where unacceptably long processing times are known to occur. For all other systems, we assume that the time needed for generation stays within reasonable limits. In general, complex systems employing very large, sophisticated grammars that have to be fully traversed will take more time for generation than simpler systems using either small, domain-specific grammars or templates for surface realisation.

Finally, it would be nice if all components in a dialogue framework would be written in the same *programming language* (at Parlevink, Java would be preferred). At least, to ensure portability, the NLG system would have to run on both Unix and Windows platforms.

Summing up, the following points have to be taken into account when examining different candidate NLG systems:

System availability, support and documentation
Kind of input required by the system
Context-sensitivity
Generality / domain independence
Availability of Dutch resources
Speed / efficiency
Programming language and platform

In Part II of this report, only those systems are discussed which meet the first requirement: availability. The systems are grouped together according to the approach they embody (as discussed in Section 3).

'Reversed parsing' (Section 5): ALE, ASTROGEN, Gemini
Grammar-based realisers (Section 6): KPML, FUF, RealPro
Template-based realisers (Section 7): TG/2, YAG
Template-based 'full' NLG (Section 8): LGM, Exemplars
Grammar-based 'full' NLG (Section 9): SPUD, InDiGen

Part II
Systems overview

5 'Reversed parsing'

ALE and ASTROGEN, the first two systems discussed in this section, take as their input basic meaning representations that are not especially geared towards generation. However, they do not use truly bidirectional grammars (i.e., grammars that can be used for both parsing and generation). Instead, they make use of grammars that are aimed at generation only. Of the two systems, ASTROGEN is a fairly small system that was designed specifically for the purpose of natural language generation. ALE is a general natural language processing framework, providing facilities for both parsing and generation. Although some comparable frameworks for parsing and generation used to exist (e.g., ALEP, the Advanced Language Engineering Platform [Simpkins, 1994]), these appear to be no longer supported. Gemini, the third system discussed here, is the only one that does use of a bidirectional grammar for both parsing and generation. As its input (for generation) it takes a logical form that has been enriched with information that is relevant for generation.

5.1 ALE

ALE, the Attribute Logic Engine, is an integrated phrase structure parsing and logic programming system. Its terms are typed feature structures that generalise those found in the linguistic programming systems PATR-II and FUG and the grammar formalisms HPSG and LFG (Carpenter and Penn [1999], p.1). ALE provides a grammar writing environment, a parser, a generator, and several grammars. Phrase structure grammars in ALE have the form of Definite Clause Grammars, allowing the use of arbitrary procedural attachments. The ALE generator is based on the Semantic Head-Driven Generation algorithm of Shieber et al. [1990]. This algorithm follows a mixed top-down/bottom-up control strategy, relying on the notion of a semantic head: a constituent which shares its semantics with the constituent of which it is the head. As argued in van Noord [1990] and Shieber et al. [1990], this algorithm avoids the problems associated with both top-down and Earley-based generation strategies. It is therefore widely acknowledged as the best algorithm for generation using a traditional phrase structure rule grammar.

The ALE system is available without charge for research purposes.[6] The system is extensively documented and actively supported. Its latest version, ALE 3.2.1, stems from December 2001.

As input for generation, ALE takes minimal semantic information, together with a syntactic goal category. An example input is shown in Figure 2. Based on this input, the generator tries to build a phrase corresponding to the specification; in this case, a sentence with semantics call_up(mary,john). Note that apart from syntactic category and basic semantics, the only further information required is the sentence type (declarative, interrogative etc.). Even a simple dialogue planner should be able to provide this information without any problems.

If the ALE generator is able to build more than one phrase matching the input, all matching phrases are returned one by one after prompting. So, given the example input of Figure 2, ALE's sample generation grammar (see below)

[6] Downloadable from `http://www.cs.toronto.edu/~gpenn/ale.html/`.

```
(sentence,
 sem: (pred: decl,
       args: [(pred: call_up,
               args: [pred: mary,
                      pred: john])]))
```

Figure 2: Example input to the ALE generator [Carpenter and Penn, 1999], p. 58.

returns the strings *Mary calls up John* and *Mary calls John up*. Of course, a larger grammar might return several additional strings. In any case, it is not clear how the most appropriate string can be selected when more than one is generated. Possibly, the procedural attachments to the ALE grammar rules might be used to specify additional constraints on those rules. A simpler, practical solution might be to use a domain-specific grammar which is so limited that for each input specification, only one string is returned (see Section 3.1). In such a situation context-sensitivity, generality and domain independence are minimal.

The ALE software package comes with a small sample generation grammar for English. A Dutch HPSG grammar for ALE has been developed in the DenK-project [Verlinden, 1999], where it was used for the analysis of user utterances. We are not aware of any Dutch *generation* grammars which might be used in ALE, and it is not clear how much work would be involved in adapting existing Dutch (analysis) grammars for this purpose. ALE provides a grammar writing environment with several debugging facilities to ease the development of new grammars.

The ALE system is implemented in SICStus Prolog and runs on both Unix and Windows platforms.

5.2 ASTROGEN

ASTROGEN (Aggregated deep and Surface naTuRal language GENerator) is a Prolog-based natural language generator developed as part of the thesis work described in Dalianis [1996]. In addition to a generation algorithm and grammar for surface realisation, ASTROGEN also includes a module for sentence planning (i.e., microplanning). The sentence planner is specialised in the task of aggregation. Other microplanning tasks are not performed, with the exception of a simple form of pronominalisation. ASTROGEN's surface realiser is a definite clause grammar where the terminals are the lexical items.

The ASTROGEN system can be freely downloaded for non-commercial purposes. It appears to be still actively supported by its developer (the latest changes on the ASTROGEN website are dated at September 2002). The documentation of the system is very limited, consisting of only a small number of webpages.[7] However, ASTROGEN appears to be a fairly small and simple system, consisting of only 15 files of Prolog code, including short comments. More extensive documentation may therefore not be necessary.

ASTROGEN's input consists of one or more so-called 'f-structures': simple

[7] Presumably, more information can be found in Dalianis [1996].

formulas specifying a predicate and its arguments plus the required verb tense. An example is shown below.

```
f(pres,isa,john,subscriber) & f(pres,isa,mary,subscriber)
& f(pres,state,john,busy) & f(pres,state,mary,idle)
```

The output which ASTROGEN generates on the basis of this input depends on the current settings of the system. The system includes several aggregation rules, as well as a pronominalisation rule, which can be individually switched on or off. With all rules switched off, the following output is generated:

John is a subscriber and
Mary is a subscriber and
John is busy and
Mary is idle.

With all aggregation rules switched on, the output is as follows:

John and Mary are subscribers and
John is busy and
Mary is idle.

The ASTROGEN system comes with a grammar and lexicon for a very small fragment of English, which can be used to test the aggregation rules included in the system. Resources in other languages are not available. To use the system for the generation of Dutch, and in another than the current (telecom) domain, a new grammar and lexicon will have to be written. It is not clear if the aggregation rules included in ASTROGEN hold for languages other than English, so at least some of these may have to be adapted as well.

The ASTROGEN system does allow for a very limited form of context-sensitivity in the sense that, depending on the characteristics of the input message (e.g., the presence of symmetrical relations, shared arguments or predicates), different aggregation rules are applied during microplanning. However, ASTROGEN does not model any other contextual aspects that may influence the surface form of the generated sentences. In order to perform other tasks in a context-sensitive fashion, e.g., lexical choice or the generation of referring expressions other than pronouns, additional rules as well as more domain knowledge will have to be added to the system. All in all, ASTROGEN is a relatively small and simple system that seems sufficient for the generation of simple dialogue utterances. It may be less suitable for the generation of longer texts or more complex linguistic constructions.

ASTROGEN is implemented in SICStus Prolog and runs on both Unix and Windows platforms.

5.3 Gemini

Gemini [Dowding et al., 1993] was developed at SRI International as a natural-language parsing and semantic interpretation system based on the unification of typed feature structures. In addition to a bottom-up parser, it also comprises a generator that employs the Semantic Head-driven Generation algorithm of Shieber et al. [1990], which was briefly discussed in Section 5.1. It makes use of a

bidirectional grammar for both parsing and generation. Gemini has been applied for parsing and generation in various spoken dialogue systems. These include CommandTalk [Goldwater et al., 2000], and WITAS [Lemon et al., 2003]. These are spoken-language interfaces to a battlefield simulator and a robot helicopter respectively. It is not clear whether Gemini is freely available for use by others than (past) affiliates of SRI. The system is not downloadable and no on-line documentation is available. Available publications focus mainly on the parsing side of the Gemini system.

No example is available of Gemini's input (for generation), but according to Bratt and Dowding [2003], it consists of logical forms that are enhanced with information that is relevant for generation. This includes details about number agreement, definite/indefinite determiner choice and referring expression generation. In CommandTalk, these details are provided by an utterance planner that receives high-level logical forms from the dialogue manager as its input, and converts these to input that is suitable for the Gemini generator. The utterance planner itself is not part of Gemini. The WITAS system [Lemon et al., 2003] incorporates a similar approach to generation as CommandTalk. Here, some additional tasks performed by the utterance planner are the selection of relevant messages and aggregation. One additional context-sensitive generation task carried out in CommandTalk, is the enrichment of the generated utterances with prosodic markers for speech synthesis. For this task, use is made of the syntactic parse trees produced by the Gemini generator, and of speech act information from the logical form. The latter information is used, for instance, to identify alternative questions, which have a distinctive intonation. Responsible for adding the prosodic mark-up is a highly system specific prosody component, which is used as an add-on to Gemini. In other words, all context-sensitive generation decisions in CommandTalk are made by components outside Gemini.

The Gemini system itself is highly reusable, but the grammars it is used with tend to be application and domain specific. Dutch resources are not available. Gemini has been specifically developed for use in spoken dialogue systems, with special attention for robustness, speed and efficiency. Concrete data on processing times are not available, however. Gemini has been written in Prolog.

5.4 Comparison

The ALE system is a general DCG grammar writing environment, which comes with both a parser and a generator. The system is mostly aimed at parsing. This also holds for Gemini, which was designed first and foremost with robust parsing in mind, but has also been used for generation (i.e., surface realisation) in several dialogue systems. ASTROGEN, on the other hand, is a dedicated generation system which not only performs surface realisation but also includes a basic planning component. Planning in ASTROGEN is mostly aimed at aggregation, providing a very limited form of context-sensitivity. Generation in ALE is not context-sensitive at all, and is purely aimed at the production of isolated sentences. In applications that use Gemini for realisation, context-sensitive generation decisions are made by modules that are external to Gemini. For all three systems, only English grammars are available (and these are only small example grammars in the case of ALE and ASTROGEN). This means that using them for generation in Dutch will require the development of a new, Dutch grammar for the intended application. However, because the systems all make

use of fairly general grammar formalisms, the (partial) reuse of existing Dutch grammars may be an option. In contrast to ASTROGEN, ALE is extensively documented and actively supported. For Gemini, this is not clear.

The main advantages of the 'reversed parsing' approach taken by the systems discussed here, are (*i*) that the input for generation consists of relatively simple semantic sentence representations that are easy to produce in a dialogue system and (*ii*) that generation is grammar-based, which is theoretically more attractive than the use of templates containing canned text (see Section 3.2.2). In the case of Gemini, an additional advantage is that the same grammar is used for both parsing and generation. A problem with the 'reversed parsing' approach is that it offers no context-sensitive choice between different ways of expressing the same basic proposition. For this, additional mechanisms must be added. Still, it would be interesting to investigate the limits of this approach for dialogue generation. The use of Gemini in various spoken dialogue systems is promising in this respect.

6 Grammar-based linguistic realisers

The first two grammar-based linguistic realisers discussed below, KPML and FUF, are the most well-known and widely used realisers currently available. The third system, RealPro, is slightly less known, mainly because it is more recent than the other two. The Dutch grammar-based realiser developed by Marsi [2001] is not described here, because it is not sufficiently portable and well-documented for general use (Marsi p.c.).

6.1 KPML

The KPML (Komet-Penman Multilingual) linguistic realiser [Matthiessen and Bateman, 1991, Bateman, 1997] comprises a generation engine, a large collection of grammatical resources and a rich development environment. Its grammatical engine is based on Systemic Functional Grammar (SFG, [Halliday, 1985]). Unlike traditional linguistic theories, SFG distinguishes linguistic categories in terms of their function (or function potential) rather than their form. In SFG, the central notion is that of choosing from available alternatives. This property makes it quite suitable as a basis for natural language generation. Linguistic realisation in SFG corresponds to the traversal of a systemic network, which corresponds to a taxonomy of linguistic elements in terms of their function. Each divergence in the network represents a choice between minimal grammatical alternatives, by adding small constraints on the final form of the generated utterance. When the network has been traversed, all details of the utterance have been specified (Reiter and Dale [2000], p. 175).

KPML is freely available and very well documented.[8] For beginning users, an online tutorial is provided as well as a package of simple introductory example grammars. KPML is actively supported and under constant development, its latest version dating from March 2003.

An example input to the KPML system is shown in Figure 3. Although this input is somewhat less rich in detail than that of other linguistic realisers (see Sections 6.2 and 6.3), it is still some steps removed from the kind of basic, logical

[8] Downloadable from `http://www.fb10.uni-bremen.de/anglistik/langpro/kpml/`.

meaning representation produced by most dialogue act planners. This means that probably, an additional (but possibly quite simple) planning component will be needed to achieve the kind of input required by KPML.

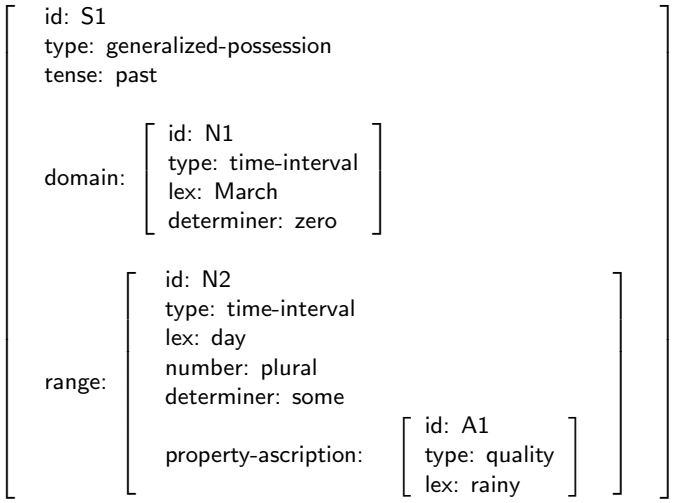

Figure 3: Example input to KPML, taken from Reiter and Dale [2000], p. 172.

In principle, the KPML system allows for highly context-sensitive generation choices. When determining the most suitable output utterance, not only propositional, but also interpersonal and textual meaning aspects are taken into account. Interpersonal meaning is concerned with the relationship between the speaker and his audience, affecting for instance the level of formality of the generated utterances. Textual meaning is concerned with discourse structure and information status, affecting for instance the choice between different syntactic constructs that convey the same (propositional) meaning. Information about all these meaning aspects determines how the systemic network is traversed during generation. The refinement of the choice system depends on the grammar that is used. However, all information on which the generation choices are based must be specified in the input structure; otherwise, default choices are made. This means that ultimately, the extent to which the generated utterances are context-sensitive, depends on the sophistication of the planning component that constructs (or enriches) the realiser's input specifications.

The KPML generation engine is quite general; it can (and has been) used for generation in several different languages. Apart from a very large English grammar (NIGEL), KPML also provides moderately-sized grammars for several other languages, including Dutch. This means that when using KPML, relatively little effort will have to be spent on grammar development. Still, as pointed out by Reiter and Dale [2000], KPML is so complex that simply learning to work with it already takes a 'non-trivial' amount of time and effort.

KPML was written in ANSI standard Common Lisp, and runs both on Unix and PC. Stand-alone executables, requiring no additional software, are available for different Windows versions. According to the on-line system documentation (dating from 1996), the fastest average generation time achieved by using different speed-up methods is 2.8 seconds on a Sun Sparc with 48 Mb RAM.

6.2 FUF

FUF, the Functional Unification Formalism, is an interpreter for a functional unification based language specifically designed to develop text generation applications. Generally, it used in combination with a large grammar for English called SURGE [Elhadad and Robin, 1997, Elhadad et al., 1997]. Although FUF is supposed to be usable for all generation stages [Elhadad, 1993], it is most widely used for linguistic realisation in combination with a generation grammar. Linguistic realisation using a FUF-grammar is done by unifying an input structure with the structures in a grammar. The result is a syntactic structure that is then linearised to produce a sentence string. The unification grammars written in FUF are based on ideas from Systemic Functional Grammar (i.e., traversal of different choice systems, at each step enriching the structure under construction), but also borrow from other grammar formalisms such as HPSG [Pollard and Sag, 1994].

The freely available FUF package[9] contains the source for the interpreter, documentation and a tutorial on how to write generation grammars in FUF. The system documentation is extensive and there is a wealth of related scientific publications.

An example of the input to a FUF-based generator is shown in Figure 4. The message it specifies corresponds to that of Figure 3, and is expressed by the sentence *March had some rainy days*. The input to FUF is slightly more detailed than that to KPML, in that it must additionally specify the syntactic categories to be used (e.g., cat:proper specifies that a proper noun must be used to express the first participant in the process). To produce this kind of detailed input, a specialised microplanner will have to be used in addition to the FUF-generator. Such a microplanner could be written in the FUF formalism, cf. Elhadad [1993] and others.

```
[ cat: clause
  tense: past
  proc: [ type: possessive ]
  partic: [ possessor: [ cat: proper
                         head: [lex: March] ]
            possessed: [ cat: common
                         head: [lex: day]
                         describer: [lex: rainy]
                         selective: yes
                         number: plural ] ] ]
```

Figure 4: Example input to FUF, taken from Reiter and Dale [2000], p. 181.

As in KPML, the context-sensitivity of the generated utterances depends both on the grammar and the microplanner that are used. A simple grammar written in FUF may be less sensitive to different meaning aspects than a more sophisticated one. Regardless of the grammar's level of sensitivity, all semantic features to which it is sensitive must be specified in the input (i.e., they must

[9] Downloadable from `http://www.cs.bgu.ac.il/research/projects/surge/index.htm`.

be provided by the preceding planning component). Otherwise, default features are assumed.

As in KPML, and all grammar-based realisers, the generality of the generator depends on the grammar that is used. FUF comes with a large English grammar (SURGE), but grammars for other languages are not provided. A Dutch grammar using FUF was developed by Marsi [1998], but due to several problems this work was abandoned (see Marsi [2001]). On the linguistic side, these problems mainly concerned the development of a morphological component for Dutch (for which FUF provides no facilities) and the treatment of word order in Dutch verb phrases. In addition, the system was very slow when generating long sentences and running it on a Windows PC proved to be problematic. FUF is implemented in Common Lisp, and the FUF/SURGE website states that it was only tested on Unix workstations.

6.3 RealPro

The RealPro linguistic realiser is based on a grammatical theory called Meaning-Text Theory [Mel'čuk, 1988], which views the generation of language as a process of mapping iteratively through seven levels of linguistic representation. In RealPro, only four of these levels are implemented. Of the realisation systems discussed in this section, RealPro is functionally the simplest, but it also requires the most detailed input specification.

Although RealPro has been developed by a commercial company,[10] it is available for research purposes, and documentation is provided. RealPro has been developed relatively recently and is currently being applied in systems such as the Why2-Atlas tutoring system [Jordan et al., 2003] and DIAG-NLP3, a tool for teaching troubleshooting in a heating system [Di Eugenio et al., 2003]

The input to RealPro consists of a so-called deep syntactic structure. Simplifying, this is a syntactic structure that specifies content words, syntactic roles and features, but does not contain any function words or word ordering information. An example input corresponding to the example sentence *March had some rainy days* is given in Figure 5. At the first level of generation, the deep syntactic structure is mapped to a surface syntactic structure, among other things by adding function words to it. Then, this structure is linearised, i.e., word order is determined. The output of this level is called a deep morphological structure. The next step is to apply morphological processing, resulting in a surface morphological structure. Finally, orthographic processing returns the completed sentence.

The input to RealPro is quite detailed; most syntactic and lexical choices have already been made at this stage. The tasks performed by RealPro are largely limited to word ordering and morphology. This means that, even more than for KPML or FUF, any context-sensitivity has to reside in the planner that constructs RealPro's input structures.

RealPro includes a general-purpose English lexicon and grammar. It is not clear how extensive these resources are. RealPro does not come with resources for any other languages, but is claimed to be customisable to generate languages other than English. Presumably, the relative simplicity and modularity of the system make RealPro more suitable for such adaptation than the other realis-

[10] Cogentex, see `http://www/cogentex.com/technology/realpro/index.shtml`.

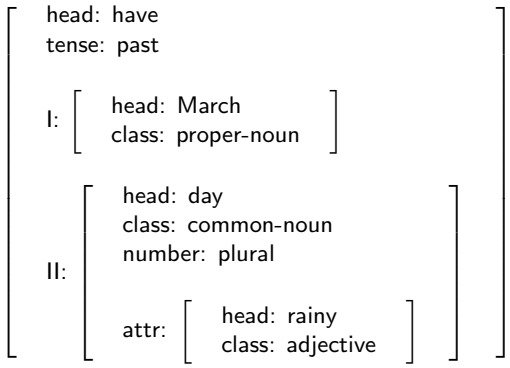

Figure 5: Example input to RealPro, taken from Reiter and Dale [2000], p. 188.

ers. However, according to Reiter and Dale [2000], working with RealPro still requires a significant investment of time spent in learning about both the realiser and the theory underlying it.

An important feature of RealPro is that compared to other realisers, it is quite fast. Average execution times range between 0.11 (five word sentences) and 0.72 seconds (50 word sentences) on a 150 Mhz Pentium PC with 32 Mb RAM [Lavoie and Rambow, 1997]. RealPro has been implemented in Java, running on both Unix and PC.

6.4 Comparison

In the previous sections we have described three general, reusable components for grammar-based linguistic realisation: KPML, FUF/SURGE and the more recent RealPro. The two older realisers appear to be roughly similar with respect to complexity (high), learning curve (steep), and efficiency (low). However, KPML has a Dutch grammar available, whereas using FUF for realisation in Dutch proved to be somewhat problematic ([Marsi, 2001]). Also, it seems that KPML is the more actively supported of the two. These features make KPML somewhat more suitable for our purposes than FUF. As for RealPro, its modularity, its relative simplicity, and its efficiency make it more attractive for generation in a practical system than the older realisers. A drawback is that RealPro comes with few resources. Only an English grammar is included with the system, so for use in Dutch a Dutch grammar will have to be developed. In addition, RealPro expects even more linguistically detailed input than FUF and KPML, and therefore makes even higher demands on the preceding planner. In general, all realisers must be preceded by a planning system, which has to provide them with linguistically detailed, system-specific sentence plans. This is in contrast to most template-based systems, which do not expect linguistic input. Such systems will be discussed in the following section.

7 Template-based linguistic realisers

In this section, two template-based linguistic realisers are discussed: TG/2 and YAG. These systems are more recent than most grammar-based realisers, be-

```
[(COOP THRESHOLD-EXCEEDING)
 (LANGUAGE FRENCH)
 (TIME [(PRED SEASON) (NAME [(SEASON WINTER) (YEAR 1996)])])
 (THRESHOLD-VALUE [(AMOUNT 600) (UNIT MKG-M3)])
 (POLLUTANT SULFUR-DIOXIDE)
 (SITE ‘‘V&o1lklingen-City’’)
 (SOURCE [(LAW-NAME SMOGVERORDNUNG)(THRESHOLD-TYPE VORWARNSTUFE)])
 (DURATION[(HOUR 3)])
 (EXCEEDS [(STATUS NO)(TIMES 0)])]
```

En hiver 1996/1997 à la station de mesure de Völklingen-City, le seuil d'avertissement pour le dioxide de soufre pour une exposition de trois heures (600.0 μ/m^3 selon le decret allemand "Smogverordnung") n'a pas été dépassée.

Figure 6: A sample GIL structure and the corresponding report from a TG/2 application called TEMSIS (Busemann and Horacek [1998], p. 242).

cause only in the last few years researchers have become increasingly convinced of the fact that template-based NLG systems do not only offer practical advantages (such as speed and efficiency), but can also be scientifically interesting.

7.1 TG/2

TG/2 is a template-based realisation system that allows the developer to vary modelling granularity according to the requirements posed by the application at hand. The system was developed to be more flexible and to achieve better quality output than simple template-based systems, but without the costs of in-depth syntactic realisation [Busemann and Horacek, 1998]. The TG/2 system can be obtained from Dr. Stephan Busemann at DFKI Saarbrücken.[11] The system is actively supported, and there are several scientific publications on TG/2 [Busemann and Horacek, 1998] and applications using TG/2 ([Busemann and Horacek, 1997] and others). It is not clear whether any system documentation is available.

TG/2 can deal with different kinds of input representations, converting them to a structure in its internal representation language GIL (Generator Internal Layer). Flat input, such as a list of domain-specific attribute-value pairs, can be accommodated as well as structured logical form representations of sentence semantics. An example of a GIL structure for the TEMSIS[12] application, which generates air-quality reports [Busemann and Horacek, 1997], is shown in Figure 6. The format of the original input data, on which the GIL structure is based, is not known.

An important characteristic of the TG/2 system is that it allows for grammatical modelling at different levels of granularity, combining the use of templates and grammar fragments for surface realisation. Templates may be used for the generation of application specific, fixed text fragments, while subgrammars may be used for the generation of phrases that are more or less domain-independent (e.g., time expressions). This property of the system makes it somewhat less domain-dependent than NLG systems that only use sentence

[11] See `http://www.dfki.de/pas/f2w.cgi?lts/tg2-e`.
[12] Transnational Environmental Management Support.

```
(defproduction threshold-exceeding ‘‘WU01’’
  (:PRECOND (:CAT DECL
             :TEST ((coop-eq ’threshold-exceeding)(threshold-value-p)))
   :ACTIONS (:TEMPLATE (:OPTRULE PPtime (get-param ’time))
                       (:OPTRULE SITEV (getparam ’site))
                       (:RULE THTYPE (self))
                       (:OPTRULE POLL (getparam ’pollutant))
                       (:OPTRULE DUR (getparam ’duration))
                       ’’(‘‘ (:RULE VAL (getparam ’threshold-value))
                             (:OPTRULE LAW (getparam ’lawname))’’)’’
                       (:RULE EXCEEDS (getparam ’exceeds))‘‘.’’
            :CONSTRAINTS (:GENDER (THTYPE EXCEEDS) :EQ))))
```

Figure 7: The TEMSIS sentence template for threshold exceeding statements, used to generate the report in Figure 6 (Busemann and Horacek [1998], p. 244).

templates for realisation, since the grammar fragments may be reused in different applications, whereas templates that specify a full sentence must nearly always be completely replaced. Besides replacing the templates, developing a new TG/2 application also requires specifying a new mapping from input data to GIL structures.

The context-sensitivity of language production in TG/2 resides in the preconditions associated with the production rules (templates and grammar rules). A rule is only applicable if its preconditions are met. Busemann and Horacek [1998] only give an example of preconditions relating to the application's input data: a specific sentence template from the TEMSIS system for 'threshold exceeding statements' (see Figure 7) can only be applied if a threshold has actually been exceeded and if information about the threshold value is available. However, it is also possible to have preconditions that place restrictions on other contextual aspects, as is discussed by Geldof [2000] for the COMRIS[13] application, a mobile device that presents context-sensitive advice to the user in the conference domain. In COMRIS, the surface realisation of the presented messages is adapted to different context dimensions: linguistic context, physical context, and user preferences. This is achieved by using context annotations (*i*) in the preconditions for the firing of production rules, and (*ii*) in template-internal conditions for variation. In COMRIS, information about the current context is specified in the GIL structures that form the input to the TG/2 production rule base. This contextual information is derived from context models that were added specifically for COMRIS, and that are external to TG/2. In other words, TG/2 can be adapted so that it reacts appropriately to the information specified in its input, but the system itself does not keep a record of relevant contextual information.

TG/2 has been used for language generation in English, French, German, and Portuguese.[14] Applying it to Dutch would involve the specification of new templates in Dutch, and (optionally) the specification of Dutch grammar fragments.

The aim in developing TG/2 was to achieve a system offering efficiency both

[13] Co-habited Mixed Reality Information Spaces.

[14] An on-line demonstration can be found at `http://www.dfki.de/services/nlg-demo`.

```
((M2 (AGENT B4)
     (ACT (M1 (ACTION ‘‘want’’)
          (DOBJECT B6))))
 (M5 (OBJECT B4)
     (PROPERNAME ‘‘Jack’’))
 (M11(CLASS ‘‘dog’’)
     (MEMBER B6))
 ((form decl)
  (attitude action)))
```

Figure 8: A YAG knowledge structure input (McRoy et al. [2001], p. 28).

at run-time and in the development of new applications, while still being more flexible than simpler template-based systems. As reported in Busemann and Horacek [1998], the development of the TEMSIS generator based on TG/2 took an effort of about eight person months. On the TG/2 website it is claimed that accommodating TG/2 to new tasks takes an average effort of approximately three person-months. The system's average generation time is less than a second. TG/2 is implemented in Allegro Common Lisp and runs on Unix and PC platforms.

7.2 YAG

YAG (Yet Another Generator [McRoy et al., 2001]) is a general-purpose template-based realiser that extends the more traditional template-based approach by allowing templates to embed several types of control expressions, in addition to simple string values. Also, templates can contain other templates. In YAG, the users can specify canned text, general templates, and a grammar with the same formalism. YAG was developed for use in real-time applications, and contains many reusable parts.

The YAG system can be freely used for non-commercial purposes; its latest version can be obtained from Dr Susan McRoy at the University of Wisconsin-Milwaukee.[15] An installation guide and a tutorial are available,[16] and in addition, there are several recent, scientific publications on YAG [Channarukul et al., 2000, McRoy et al., 2000, 2001].

YAG accepts two types of input: feature structures and 'knowledge structures'. The latter are based on the knowledge representations of SNePS (The Semantic Network Processing System [Shapiro, 2000]), a widely used knowledge representation and reasoning system. YAG comes with a component that maps these representations to feature structures. Applications that use other kinds of knowledge representations are required to implement their own component that would do the same thing. Examples of the two types of input, a knowledge structure and the corresponding feature structure for the sentence *Jack wants a dog*, are shown in Figures 8 and 9. Note that the feature structure already specifies which templates are to be used to express the message content.

Like other realisers, YAG cannot make context sensitive decisions on its own, but depends for this on the control features in its input specification. For

[15] See `http://tigger.cs.uwm.edu/ nlkrrg/yag/distribution.html`.
[16] They can be downloaded from `http://tigger.cs.uwm.edu/ songsak/`.

```
((template CLAUSE)
 (process-type MENTAL)
 (process ‘‘want’’)
 (processor ((template NOUN-PHRASE)
             (head ‘‘’Jack’’)
             (np-type PROPER)))
 (phenomenon((template NOUN-PHRASE)
             (head ‘‘dog’’))))
```

Figure 9: A YAG feature structure input, corresponding to the knowledge structure in Figure 8 (McRoy et al. [2001], p. 28).

```
((EVAL processor)
 (TEMPLATE verb-form
           ((process ^process)
            (person (processor person))
            (number (processor number))
            (gender (processor gender))))
 (EVAL phenomenon)
 (PUNC ‘‘.’’  left))
```

Figure 10: A (domain-independent) YAG clause template, adapted from McRoy et al. [2001], p. 32.

instance, to express an object YAG uses a full noun phrase by default, unless it is specified in the input that a pronominal form should be used. Other examples of control features are `form = decl` and `attitude = action` (see Figure 8). Based on the information in the control features, specific templates (and options within templates) are selected by YAG. A simplified example template is shown in Figure 10. The **processor** and **phenomenon** slots in the template are filled using two noun-phrase templates (referred to in the input feature structure).

There are two types of knowledge bases in YAG; domain dependent and domain independent. The domain independent knowledge base includes a library of templates that are used as a grammar of English, such as clause, noun-phrase, and pronoun template. They can be embedded in other templates to form more complex structures. The domain independent knowledge base also contains a lexicon, including several lexical functions to inflect a given verb according to verb features (e.g., tense, person, and aspect), and to generate the singular or plural form of a noun. The lexicon and the syntactic template library can be reused across applications. The domain dependent parts of YAG are the table used for mapping knowledge representations to the associated templates, and the library of templates that are specific to a particular application. These parts will have to be replaced when using YAG for a new application. To facilitate this process, YAG provides a development tool for authoring and testing templates in a graphical environment.

So far, YAG has only been used for linguistic realisation in English, and no resources for other languages are available. This means that when using the system for Dutch, the domain independent parts of YAG (the ‘grammar’ templates and the lexicon) will have to be replaced or adapted, since these are

specific for the English language.

An important characteristic of YAG is that, unlike most grammar-based realisers, it works in real-time. This means that it is suitable for use in applications where generation speed is crucial (such as dialogue systems).

Different versions of the YAG system are available for DOS/Windows and Linux platforms. YAG has originally been implemented in Lisp, but there also exists an implementation in Java called JYAG.

7.3 Comparison

Both template-based realisers that have been discussed, TG/2 and YAG, are faster and more efficient than grammar-based realisers, because they do not require the traversal of a large grammar. Of the two, YAG is claimed to be more efficient than TG/2, because unlike in TG/2 there is never any backtracking (which might slow down the generation process). In TG/2, execution of the rules included in a template can fail, in which case the system backtracks and chooses another applicable template. In YAG, there is a one-to-one mapping between input and templates, and the mapping of a template to a sequence of words can never fail,[17] so there is no need for backtracking. In addition, the developers of YAG claim that YAG is more declarative than TG/2, and also more powerful, because the system offers more extensions to 'standard' templates than TG/2 does [McRoy et al., 2001]. On the other hand, it seems that YAG's very direct mapping of input to templates makes the system less flexible than TG/2. Also, YAG is clearly more language-dependent than TG/2.

8 Template-based 'full' NLG systems

Of the many full NLG systems that have been developed in the past years of NLG research, only few have been made publicly available. Probably, this is partly because a full NLG system necessarily contains a substantial application- or at least domain-specific component, which must be adapted when the system is to be used for other applications or in other domains. This holds in particular if the system makes use of templates, like the systems discussed in this section. Making such a system publicly available is only useful if it comes at least with adequate documentation on how to change its application or domain-specific parts; better still, the system should be made available as a general NLG 'shell' in which the user can fill in the information that is specific to the needs of the intended application. Apparently, most developers of NLG systems do not regard this as a worthwhile effort, possibly because their systems are insufficiently general to be easily portable to other applications or domains. Two exceptions, the LGM developed at the University of Eindhoven and EXEMPLARS developed at CoGenTex, are discussed in the current section. Both systems make use of templates for syntactic realisation. Some grammar-based 'full' NLG systems are discussed in Section 9.

[17] However, it is possible for YAG to produce an ungrammatical result.

```
Template DepTime

TREE s [np [det 'de'
            n 'trein']
        vp 'vertrekt'
        pp [p 'om'
            <time>]]
TOPIC route
CONDITION notknown(current_subpart.departure.time) AND
          known(current_subpart.departure.station)
<time> ∈ expresstime(current_subpart.departure.time)
```

Figure 11: A simplified LGM template for the OVIS system.

8.1 LGM

The LGM (Language Generation Module) is a full NLG system that performs document planning, microplanning, and realisation. (As discussed in section 2, this makes the LGM more powerful than may be needed in most dialogue systems.) Planning is done locally, in an incremental fashion, and for surface realisation so-called 'syntactic templates' are used. These are syntactic sentence structures with slots in them for variable information. Specialised 'express' functions are used to construct different types of slot fillers such as definite descriptions, time expressions, etc. Each syntactic template has a set of conditions on its use that check whether the template is applicable in the current context. A simplified example template is shown in Figure 11. This template can be used to generate the sentence *De trein vertrekt om negen over zes* ('The train leaves at nine past six'), using information from the data structure shown below in Figure 12.

The LGM has been designed for use in combination with speech synthesis. For this purpose, it enriches the generated texts with markers that indicate which words in the text should be pronounced with a pitch accent and where phrase boundaries (pauses) should occur. Using this mark-up, a spoken version of the input text can be generated which has a higher speech quality than can be achieved by using a plain text-to-speech system. The LGM is (presumably?) freely available for research purposes and comes with full documentation.[18] Also, several research papers are available explaining the ideas behind the system [Odijk, 1995, van Deemter and Odijk, 1997, Klabbers et al., 1998, Theune et al., 2001]. Finally, Krahmer et al. [1997] guide the user step-by-step through the process of using the LGM for a new application.

The LGM can deal with any kind of input (in ascii format) that has a fixed structure and therefore can be automatically analysed. The system contains a simple parser which automatically converts the input to an internal data representation in the form of a feature structure. Figure 12 shows an example from the OVIS[19] system, a public transport information system in which the LGM has been used for the generation of both dialogue utterances and route descriptions. The top part of the figure shows the results from a train planner, which

[18] For more information on obtaining the LGM, contact Polderland Language & Speech Technology (`http://www.polderland.nl`).

[19] Openbaar Vervoer Informatie Systeem (Public Transport Information System).

are used as input for the generation of a route description. The corresponding feature structure is shown below it.

The LGM is for the most part domain and language independent. Although the LGM was initially developed for language generation in English, it has also been successfully applied to Dutch[20] and German.[21] Using the LGM for a new application mainly involves (*i*) constructing a new set of syntactic templates, (*ii*) designing a structure representing the input data, (*iii*) adapting the parser to map the raw input onto this internal data structure, and, optionally, (*iv*) adding a knowledge base with domain specific information.

An important feature of the LGM is that it was designed to achieve variation in the generated texts. This makes it attractive for information presentation in applications where the user is likely to hear a number of generated texts in succession, and where the goal is not just to inform, but also to entertain the user. This focus on variation makes the LGM slightly less suitable for some other types of applications. For instance, for feedback messages and prompts in dialogue systems, variation does not seem beneficial, since users may interpret the variation as meaningful even though it is not. Unlike most full NLG systems, which create a global text plan before starting on later generation stages, the LGM uses a form of local, reactive planning by means of the conditions on the syntactic templates. This makes it suitable for dialogue generation, where the input is provided in a piecemeal fashion by a dialogue manager, making global text planning impossible. On the other hand, the template selection algorithm has to be adapted for use in dialogue generation mode: to ensure the well-formedness of the generated dialogue turns, the algorithm must always pick the most specific applicable template, and only one question may be generated per turn. The advantages and disadvantages of using the LGM in a dialogue system are discussed in more detail in Theune [2000] and Theune [2003].

The LGM contains several rules to ensure that different aspects of the linguistic context are taken into account during generation. These include rules for the computation of prosody and the generation of appropriate (anaphoric) referring expressions. Based on information about the discourse history, which the system collects during generation, the LGM automatically computes the information status (new, contrastive, or given) of the items that are being referred to. Since the production of nonverbal signals is linked to information status as well [McNeill, 1992, Cassell et al., 1994], the ability of the LGM to automatically compute information status also makes it attractive for use in an embodied agent. It is important to note that the relevant discourse information does not have to be specified in the system's input (as is the case with the linguistic realisers discussed in the previous sections), but is taken from a representation of the preceding discourse that is maintained by the LGM itself.

The LGM is written in the functional programming language Clean, developed at the University of Nijmegen.[22] Clean (which is compiled into C) runs efficiently on both Unix and PC/Windows platforms. Since surface realisation is performed using templates rather than a grammar, language generation by the LGM is quite fast.

[20] Within OVIS and GoalGetter; see Theune [2000].

[21] Within VODIS, the Voice-Operated Driver Information System [Krahmer et al., 1997, Pouteau and Arévalo, 1998].

[22] `http://www.cs.kun.nl/ clean/`

Input from train planner	**Informal explanation**
OK	Query successful
5	Found five connections
1	First part of the closest matching connection
STRING intercity	Intercity
INT 1564	Train number 1564
STRING eindhoven	From Eindhoven
INT 1809	Departure at nine past six p.m.
STRING breda	To Breda
INT 1846	Arrival at a quarter to seven p.m.
END_OF_PART OK	End of this part
2	Begin of the second part
STRING stoptrein	Slow train
INT 3663	Train number 3663
STRING breda	From Breda
INT 1850	Departure at ten to seven p.m.
STRING roosendaal	To Roosendaal
INT 1907	Arrival at seven past seven p.m.
END_OF_PART	End of second part
END_OF_RESPONSE	End of connection

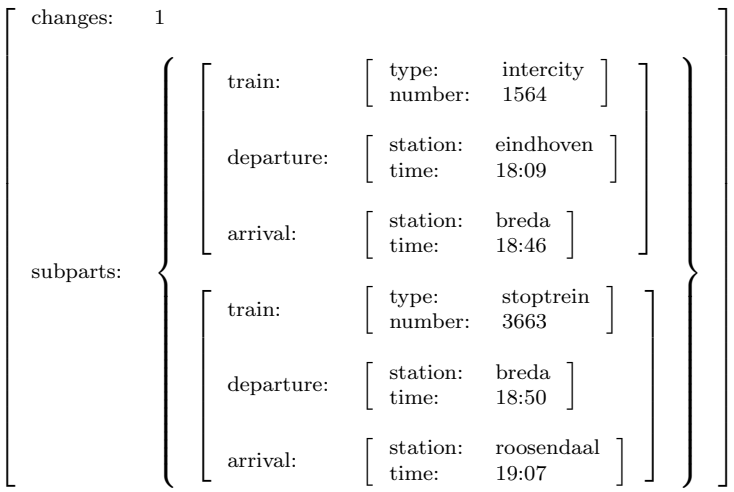

Figure 12: Example input and corresponding LGM data structure

8.2 EXEMPLARS

EXEMPLARS [White and Caldwell, 1998, 1999] is a template-based framework for the dynamic generation of text and hypertext. It is claimed to provide high performance, scalability, and ease of integration with other application components. The basis of the framework is the notion of an exemplar, a template-like text planning rule which represents an exemplary (or expert) way of achieving a communicative goal in a given communicative context. Exemplars are both recursive and object-oriented, making it possible to generate a wide variety of texts. Like RealPro (Section 6.3), EXEMPLARS has been developed at CoGenTex. It is in active use and is continuously being extended and improved. EXEMPLARS is freely available for research purposes. No on-line documentation is provided.

The EXEMPLARS framework requires Java objects as input. In principle, any XML data can serve as input after being read into Java application objects, or if they are accessed in Java via XML APIs. Figure 13 shows some example XML input data, taken from White and Caldwell [1999].

```
<?xml version=‘‘1.0’’>
<xml>

<person id=‘‘heather’’ gender=‘‘female’’>
  <name>Heather</name>
  <department>Accounting</department>
  <meetings>
    <meeting>ref=‘‘401k-switchover’’</meeting>
  </meetings>
</person>

<!-- ...  etc.  ...  -->

<meeting id=‘‘new-water-cooler’’>
  <title>New Water Cooler</title>
  <date>November 18, 1999</date>
  <participants>
    <person ref=‘‘frank’’/>
    <person ref=‘‘alice’’/>
  </participants>
<meeting>

</xml>
```

Figure 13: Example XML input data from White and Caldwell [1999], p. 2.

Figure 14 shows two of the exemplars that can be used to convert the data from Figure 13 to an HTML page containing a section entitled “People”, which gives a short description of each person in the input data together with a list of his or her meetings. One of the entries on the generated page would be *Heather works in the Accounting department. She is involved in only one meeting, 401k Switchover (November 18, 1999).* Output statements in exemplars are delimited by <<+ ... +>>. In the output statements, HTML or XML elements can appear, as well as string substitution expressions (delimited by curly braces) and calls to other exemplars (delimited by double curly braces). The first exemplar in Figure 14, `DescribeEverything`, is a schema for the entire HTML page to

```
exemplar DescribeEverything( Data data )
{
   void apply()
   {
      <<+
         <html>
           <head><title> Who, What, When </title></head>
           <body bgcolor=‘‘white’’>
             <h2> People </h2>
             <hr size=1>
             <> ::  PeopleSite
             // ...etc.  ...
           </body>
         </html>
      +>>
      for (int i = 0; i < data.people.length; i++) {
         PeopleSite <<+
           {{ DescribePerson(data.people[i]) }}
           <hr size=1>
         +>>
      }
      // ...  etc.  ...
   }
}
exemplar DescribePerson( Person person )
{
   void apply()
   {
      <<+
         { person.name } works in the { person.department } department.
         {{ ListMeetings(person) }}
      >>+
   }
}
```

Figure 14: Exemplars used to process the input data from Figure 13 (White and Caldwell [1999], p. 6).

be generated. To fill in the People section of the page (for which `PeopleSite` is a label), it calls the `DescribePerson` exemplar. `DescribePerson` in its turn calls `ListMeetings` (an exemplar not shown in Figure 14).

To achieve context-sensitive variation, exemplars are arranged in specialization hierarchies, with more specialized exemplars augmenting or overriding the more general ones they specialize. The EXEMPLARS text planner performs a decision-tree style traversal of the specialization hierarchy, successively evaluating the applicability conditions of each exemplar in the hierarchy to find the most specific one that is applicable in the current context. The conditions on the exemplars may be discourse-sensitive. For example, the general exemplar `IdentifyDate` is used to create date references such as *April 1* and is applicable in any context. It is specialized by `IdentifyDateInFocus`, which is applicable when the date being referred to is identical to the date that was last mentioned. `IdentifyDateInFocus` returns the phrase *the same day*, thus overriding `IdentifyDate`. Other exemplars that specialize `IdentifyDate` do not override the more general exemplar, but augment the phrase it returns. For instance, `IdentifyDateWithDifferentYear` adds a year expression to the phrase created

by `IdentifyDate`, returning a phrase like *April 1, 1999*. In the DIAG-NLP2 system [Di Eugenio et al., 2003], EXEMPLARS was extended with an alternative (and more principled) way of generating referring expressions, using the GNOME algorithm of Kibble and Powers [2000]. It was also extended to model a few rhetorical relations such as *contrast* and *concession*. These extensions are not inherent to EXEMPLARS, but they do show that some context-sensitivity and discourse structuring principles can easily be added to the system.

Most exemplars are application-specific and thus not reusable. However, there are some types of text that re-occur across applications, and which may be handled using generic exemplar libraries that make no assumptions about particular types of input data. White and Caldwell [1999] give the example of `MakeList`, a general-purpose exemplars for creating lists which takes an application-specific "list-describing" exemplar as its argument (e.g., `ListMeetings`). Generic exemplars specializing `MakeList` are `MakeEmptyList`, `MakeSingletonList` and `MakeVerticalList`, which in its turn is specialized by `MakeBulletedList` and other kinds of vertical lists. Presumably, also the `IdentifyDate` exemplar and its specializations could be re-used across applications. In addition, EXEMPLARS is supposed to allow integration with RealPro (Section 6.3), making it possible to vary modelling granularity according to the requirements posed by the application at hand (as in TG/2, discussed in Section 7.1). However, no information is available on how this is done.

EXEMPLARS has been used to develop several applications, including Project Reporter, a web-based project report generator, EMMA [McCullough et al., 1998], a tool for managing software requirements and design evolution, CogentHelp [Caldwell and White, 1997], a prototype help-authoring system based on text "snippets", and DIAG-NLP1 and DIAG-NLP2 [Di Eugenio et al., 2002, 2003]. All these applications involve the generation of written monologues in English. Since for any new application, new exemplars will have to be written anyway, using EXEMPLARS for generation in languages other than English is not expected to give rise to any additional work. (Except for the adaptation of any general-purpose exemplars that are available.) Di Eugenio et al. [2002] state that it took a graduate student six months to implement the language generation module of DIAG using EXEMPLARS (and they state that most of the effort was devoted to integration with the encompassing tutoring environment). It is not clear how easy it will be to adapt the EXEMPLARS framework for use in a (spoken) dialogue system. The fact that EXEMPLARS (presumably) makes use of a discourse model, in combination with local conditions, makes it sound at least as suitable as the LGM, discussed in the previous section.

Exemplars are defined using a superset of Java, and may contain loops, local variables, auxiliary methods, etc. A source file normally contains a set of related exemplars, each of which is compiled into a separate, pure Java class file (though all in the same package). One of the practical benefits of EXEMPLARS' Java basis is that "with just-in-time compilers, the compiled Java code supports the performance demands of interactive web applications" (White and Caldwell [1998] p. 274). In other words, EXEMPLARS is supposed to be quite fast.

8.3 Comparison

Both the LGM and EXEMPLARS make use of a kind of templates (referred to as 'syntactic templates' and 'exemplars' respectively). Templates have the draw-

back of being largely domain and application-dependent, but the advantage of allowing for efficient generation (helped in both cases by the use of an efficient programming language). Another shared property is that both the LGM and EXEMPLARS are 'full' NLG systems that are able to stand on their own, unlike linguistic realisers which require the presence of a preceding planning component. The advantage of this is that both the LGM and EXEMPLARS do not require linguistically detailed input. The LGM basically allows for any kind of structured input, which is automatically converted to an LGM internal data representation by a simple application-specific parser. EXEMPLARS requires Java objects as input, and can deal with any XML data via XML APIs. Finally, the LGM and EXEMPLARS have in common that the applicability conditions on their templates are sensitive to the linguistic context (and, possibly, to other contextual aspects). The EXEMPLARS planner always picks the most specialized template (exemplar) that is applicable in a given context. Such a mechanism is lacking in the original, monologue version of the LGM's template selection algorithm, and had to be added for the dialogue version.

The LGM has been used for natural language generation in a spoken dialogue system [Theune, 2000, 2003]. This experience revealed that in combination with an advanced dialogue planner, some of the LGM's own planning capacities become redundant or even cumbersome, and must be worked around in some way. For instance, the template selection mechanism had to be adapted so as to achieve conciseness rather than variation. It is not clear how EXEMPLARS would fare when used as part of a dialogue system. At least, its use of exemplar specialization hierarchies seems quite suitable for the generation of dialogue prompts.

Compared with EXEMPLARS as described in White and Caldwell [1998] and White and Caldwell [1999], the LGM appears to use more general linguistic knowledge, which allows it to do principled generation of referring expressions and of prosodic mark-up for use with speech output. However, future versions of EXEMPLARS as described by White [2001] are envisaged to perform more sentence planning tasks, including referring expression generation, by means of general revision rules. Possibly, EXEMPLARS could be extended in the same fashion to produce prosodic mark-up. In addition, more sophisticated linguistic knowledge may be added to EXEMPLARS through integration with RealPro (Section 6.3).

Finally, the programming language used for EXEMPLARS' (Java) is much more common than that used for the LGM (Clean), making easy integration with other application components more likely and making it more attractive for new users (developers).

8.4 Discussion of template-based systems

EXEMPLARS and the LGM are like TG/2 and YAG, discussed in Section 7, in that they make use of templates. What all four approaches have in common is that they offer more than the kind of simple string manipulation which is usually associated with the term 'templates' [Reiter, 1995]. Still, the kind of templates and the way they are used, are different for each of these systems. In YAG, there is a direct, one-to-one mapping between input and templates via a mapping table, which may be efficient but does not allow for any variation. In the three other systems, templates are selected on the basis of applicability

conditions associated with the templates, and more than one template may match a given input. From the set of applicable templates, EXEMPLARS always picks the most specialized one,[23] the LGM makes a random choice, and TG/2 simply returns the first one (the ordering can be set according to preferences).

For filling the verb slots in the templates, both TG/2 and YAG make use of general rules for verb inflection. The LGM has no such rules; here, the correct verb form is determined in a more ad hoc fashion within the templates. The exemplars given in White and Caldwell [1998] and White and Caldwell [1999] do not deal with inflection either; here, the verb tense is hard-coded in the exemplars. However, in the case of integration with RealPro this should be different.

The LGM templates contain a fully specified syntax tree of the sentence to be generated (hence the name 'syntactic templates'). This information is used for computing prosody and for checking binding constraints on referring expressions. The templates of TG/2 and YAG do not contain detailed syntactic information; these systems neither compute prosody nor decide on the form of referring expressions (the latter is specified in the input). The exemplars described in White and Caldwell [1998] and White and Caldwell [1999] contain no syntactic information either (although in the case of integration with RealPro, which is grammar-based, syntactic information will of course be available through the latter). However, White [2001] contains a proposal to include extensive mark-up in the exemplars, encoding (partially) rhetorical, referential, semantic and morpho-syntactic structure. This mark-up is intended for use by revision rules dealing with aggregation, referring expression generation and discourse marker insertion (examples are given of aggregation only).

Besides efficiency, an advantage of the use of templates for realisation is that it is easy to design specific templates for use in specific contexts. However, in YAG and TG/2 (as in other realisers) the context information needed for template selection has to be provided from outside the system. In the LGM and (supposedly) in EXEMPLARS, most of this information is produced by the system itself.[24] In a dialogue situation, it should therefore be possible to use them in combination with even the simplest dialogue planner (producing only very basic meaning representations), and still make context-sensitive generation decisions.

9 Grammar-based 'full' NLG systems

In this section, two closely related systems are discussed: SPUD and INDIGEN. These systems perform both microplanning (including aggregation, lexical choice, and referring expressions generation) and grammar-based syntactic realisation using a Lexicalized Tree Adjoining Grammar (LTAG). As input they take a specification of a 'communicative goal' as may be produced by a dialogue manager. This specification takes the form of a basic semantic representation, without any syntactic details. So, although these systems do not deal with

[23] It is not clear if EXEMPLARS allows for several equally specialized exemplars to be applicable in a given context.

[24] This holds only for information about generated system utterances, whereas in a dialogue situation other information (e.g., concerning user utterances) may also be relevant. The latter kind of information still has to come from outside the generation system.

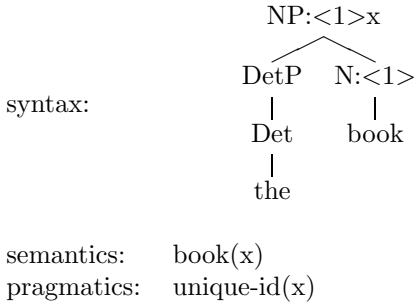

Figure 15: An LTAG tree with a semantic and a pragmatic specification.

content determination, they are much less restricted than those discussed in Sections 5 to 7, and therefore, here they are classified as 'full' NLG systems.

9.1 SPUD

SPUD (Sentence Planning Using Descriptions, Stone and Doran [1997]) combines the generation steps of microplanning and surface realisation by using a lexicalized grammar to construct the syntax and semantics of a sentence simultaneously. The core of the system is a Lexicalized Tree Adjoining Grammar (LTAG, Joshi and Schabes [1997]). An LTAG grammar consists of a set of elementary trees which can be combined using substitution and adjoining operations. Each tree contains at least one lexical item. The LTAG trees used in SPUD are extended with semantic and pragmatic constraints referring to a rich discourse model, which ensure that the generation algorithm only selects contextually appropriate trees to achieve the communicative goal. An example tree for the NP *the book* is given in Figure 15. The semantic constraint associated with this tree is BOOK(X), and the pragmatic constraint is that the entity x must be uniquely identifiable in the discourse model (UNIQUE-ID(X)); otherwise, an indefinite article should be used.

The system takes two kinds of goals as input. Goals of the form *describe* x instruct the system to create a description of entity x; goals of the form *communicate* P instruct the system to express the proposition P (expressed in modal logic). To find the best way to achieve these communicative goals, SPUD makes use of a rich context model which includes a conversation record, information about the common ground, and general world knowledge. In combination with the pragmatic constraints on the LTAG trees, this information is used to make the contextually most appropriate syntactic and lexical choices. In Cassell et al. [2000] it is shown how this approach can also be used to select appropriate gestures in combination with language generation for embodied conversational agents.

SPUD version 0.01 (February 2, 1999) is provided on the web[25] "to evaluate SPUD for research purposes only". In addition, a new 'lightweight' (i.e., limited) implementation of SPUD is available, which stems from June 2002 and was developed for educational purposes (TAGLET, Stone [2002]). The SPUD system has been described in several research papers that highlight different aspects of

[25] `http://www.cs.rutgers.edu/~mdstone/nlg.html#software`

the system such as the generation of collocations [Stone and Doran, 1996], the generation of referring expressions [Stone and Webber, 1998, Stone, 2000], verb choice [Stone et al., 2000], and the combined generation of language and gestures [Cassell et al., 2000]. An outline of the system is presented in Stone and Doran [1997]; a detailed description in Stone et al. [2001]. Also, example data files (grammar fragments, lexicons) are available for download.

The SPUD generation algorithm and the principles of LTAG are very general, and domain independent. However, the grammar (tree set) and lexicon used by SPUD are for a large part domain-specific. When SPUD is used in a new application, many new data files will have to be created. In this respect, the system resembles the template-based systems discussed in the previous section. Currently, in the XTAG project[26] a wide-coverage LTAG grammar for English is being developed, and possibly this could be used as a basis for a general SPUD grammar for English, which might be reused across different applications. For Dutch, however, no LTAG grammars are available.

SPUD is written in SML of New Jersey, a free fast functional programming language. It runs on a Unix platform. TAGLET is written in Prolog (code is available for different Prolog versions). No data are available concerning speed or efficiency.

9.2 InDiGen

The InDiGen-generator has been developed at the university of Saarbrücken as part of the German national research project InDiGen (Integrated Discourse Generation, Gardent and Striegnitz [2001], Striegnitz [2000b, 2001]). The system can be obtained by directly contacting its main developer, Kristina Striegnitz. A limited version of the system can be tried out via the web.[27]

The InDiGen-generator is similar to SPUD, discussed in the previous section. Therefore, only the main differences between the two systems are listed here. First, unlike SPUD, InDiGen uses a chart-based generation algorithm [Striegnitz, 2000a]. In addition, given some input communicative goal, InDiGen generates all possible output trees instead of just the one that best matches the input. In InDiGen, semantic and pragmatic information is represented in first order logic instead of modal logic, and for making context-sensitive generation decisions the system makes use of off-the-shelf first order reasoning tools. Like SPUD, the system focuses on the context-sensitive generation of referring expressions, but in addition, it also aims at the generation of elliptic utterances and dialogue markers. Finally, the InDiGen-generator is written in a different functional programming language than SPUD, i.e., MoZart Oz.[28]

9.3 Discussion

In SPUD and InDiGen, language generation is carried out in a theoretically attractive manner, as it is based on well-founded syntactic (LTAG) and semantic principles. In addition, the extended LTAG trees used in SPUD and InDiGen are very suitable for context-sensitive language generation, because they specify not only surface structures but also semantic and pragmatic constraints on their

[26] http://www.cis.upenn.edu/ xtag/

[27] `http://www.coli.uni-sb.de/cl/projects/indigen/software.html`

[28] `http://www.mozart-oz.org`

use, which refer to a rich context model. In this respect, they somewhat resemble the templates discussed in TG/2, the LGM and EXEMPLARS.

Dutch resources are available for neither of the systems, and it is not clear how much work will be involved in adapting them for a new application, or how efficient they are. On the SPUD/TAGLET homepage, SPUD is described as being 'old and clunky'. Its lightweight version TAGLET has been developed for educational purposes, and it is not clear if it is suitable for use in more serious applications.

10 General discussion

In the previous sections, a number of publicly available generation systems have been described that embody four different approaches to natural language generation: 'reversed parsing', grammar-based realisation, template-based realisation, and 'full' NLG using either a grammar or templates for realisation. The general advantages and disadvantages of these different approaches have been presented in Part I, Section 3. Here we give a final overview of the pros and cons of the different systems, from the perspective of using them for generation in a Dutch (spoken) dialogue system.

The two most important general requirements on NLG in a dialogue system are that it should be fast and produce contextually appropriate output. With respect to the first requirement, we have seen that the two most well-known grammar-based realisers, KPML and FUF, do not perform very well, because very large grammars need to be traversed for generation. 'Reversed parsing' using a small, domain-specific grammar is probably much faster, but also more limited. In general, the fastest performance can be expected from NLG systems that do not use a grammar, but templates for realisation.

The second requirement is that the NLG system should be able to adapt its output to different contexts. In most existing dialogue systems, this is currently not done: the system is can only produce a few (hard-wired) utterances, and these are used in any context. Such systems appear rigid and unnatural, and are not very user-friendly. The use of 'real' natural language generation instead of fixed output strings may offer a solution to this. In fact, most of the NLG systems that have been discussed are capable of taking some contextual information into account during generation. However, in most available systems, such information has to be provided by a preceding planning component. This means that their ability to produce contextually appropriate output depends on the richness of the input provided by the dialogue planner or some other component mediating between the dialogue planner and the generation component. Exceptions to this are the 'full' NLG systems discussed in Sections 8 and 9, which maintain their own context model[29] and consult this for context-sensitive generation decisions such as the generation of referring expressions. Of these systems, the template-based systems from Section 8 (LGM and EXEMPLARS) are more practically oriented, whereas the LTAG-based systems from Section 9 are aimed at more theoretical research.

Finally, for use in a Dutch dialogue system the NLG component should be able to generate grammatical Dutch utterances. For grammar-based NLG (using either a specialised realiser or a 'reversed parsing' system), this means that a

[29] In the case of EXEMPLARS (Section 8.2) this is not entirely clear.

grammar for Dutch is needed. Of the systems that have been discussed, only KPML comes with a Dutch grammar (which no doubt will have to be extended or adapted for use in a new application). For all other grammar-based systems, a Dutch grammar will have to be developed. In the case of 'reversed parsing' systems that use a fairly standard grammar formalism, the effort involved in developing a small, domain-specific grammar may be limited, and especially in the case of Gemini it may be worthwhile, because here the same grammar may be used for both parsing and generation. For template-based systems, matters are somewhat different. Since new templates will have to be written for each application anyway, writing them in a new target language does not give rise to extra work. Still, all template-based systems also have a few components that are domain-independent but language-specific. This holds in particular for YAG, which makes use of a special store of templates for syntactic categories such as clauses and noun phrases. Since these are specific for English, they will have to be replaced if YAG is to be used for another language. In the other template-based systems, TG/2, the LGM and Exemplars, the language-specific parts are largely limited to the use of specialised functions for expressing times, dates etc. Of the four template-based systems that have been discussed, the LGM is the only one that has previously been used for language generation in Dutch.

11 Conclusion

We have seen that with respect to generation speed, template-based systems are generally more suitable for use in a dialogue system than grammar-based systems (at least those that make use of a wide-coverage grammar). With respect to context-sensitive generation, which can make the system utterances more natural and more user-friendly, only a full NLG system is able to make context-sensitive generation decisions on its own. Systems which are only aimed at realisation (the last step of the generation process), depend on a preceding planning component to provide them with the required information. In dialogue generation, as opposed to monologue generation, such a preceding component is available in the form of the dialogue manager. However, it is doubtful whether the dialogue manager will always be able to provide the NLG component with sufficiently detailed input. The use of a full NLG system may therefore seem the safest choice for dialogue generation, but an important disadvantage of this is that many of the NLG system's capabilities will remain unused and, in the worst case, may even be somewhat cumbersome (see Theune [2003] for a discussion).

In conclusion, the best option for developing a context-sensitive generation component for a dialogue system, based on existing resources, might be to create a slimmed-down version of an available full NLG system, retaining only those capabilities that are required for dialogue generation (see Section 2). In doing this, special attention should be paid to the task division between the dialogue manager and the language generation component. Also, the possibility of sharing contextual knowledge sources must be considered. It seems that either the LGM or Exemplars, discussed in Section 8, could be a suitable starting point for such an exercise. An alternative that would be interesting to explore is the use of a bidirectional grammar generation system such as Gemini. Although this would involve the development of some additional generation components deal-

ing with microplanning tasks, the great advantage is that of grammar reuse. The feasability of this approach has already been shown by several dialogue systems using Gemini [Goldwater et al., 2000, Lemon et al., 2003].

References

H. Alshawi and R. Crouch. Monotonic semantic interpretation. In *Proceedings of the 30th Annual Meeting of the Association for Computational Linguistics (ACL'92)*, pages 32–39, Newark, USA, 1992.

J. Bateman. Enabling technology for multilingual natural language generation: The KPML development environment. *Natural Language Engineering*, 3:15–55, 1997.

E.O. Bratt and J. Dowding. Syntactic and semantic input to prosodic input in CommandTalk. In *Proceedings of the AAAI Spring Symposium on Natural Language Generation in Spoken and Written Dialogue*, pages 1–5, Palo Alto, CA, 2003.

H. Bunt. DPSG and its use in sentence generation from meaning representations. In M. Zock and G. Sabah, editors, *Advances in Natural Language Generation*. Pinter Publishers, London, 1988.

S. Busemann and H. Horacek. Generating air-quality reports from environmental data. In T. Becker, S. Busemann, and W. Finkler, editors, *Proceedings of the DFKI Workshop on Natural Language Generation*, Saarbrücken, Germany, 1997. DFKI Document D-97-06.

S. Busemann and H. Horacek. A flexible shallow approach to text generation. In *Proceedings of the 9th International Workshop on Natural Language Generation (IWNLG'98)*, pages 238–247, Niagara-on-the-Lake, Canada, 1998.

T. Caldwell and M. White. CogentHelp: A tool for authoring dynamically generated help for Java GUIs. In *Proceedings of the 15th Annual International Conference on Computer Documentation (SIGDOC'97)*, pages 17–22, 1997.

B. Carpenter and G. Penn. The Attribute Logic Engine: User's guide (version 3.2 beta), May 1999.

J. Cassell, M. Stone, B. Douville, S. Prevost, B. Achorn, M. Steedman, N. Badler, and C. Pelachaud. Modeling the interaction between speech and gesture. In A. Ram and K. Eiselt, editors, *Proceedings of the Sixteenth Annual Conference of the Cognitive Science Society*, pages 153–158. Lawrence Erlbaum Associates, 1994.

J. Cassell, M. Stone, and H. Yan. Coordination and context-dependence in the generation of embodied conversation. In *Proceedings of the First International Conference on Natural Language Generation (INLG 2000)*, Mitzpe Ramon, Israel, 2000.

S. Channarukul, S. McRoy, and S. Ali. Enriching partially-specified representations for text realization using an attribute grammar. In *Proceedings of the First International Conference on Natural Language Generation (INLG-2000)*, Mitzpe Ramon, Israel, 2000.

H. Clark and D. Wilkes-Gibbs. Referring as a collaborative process. *Cognition*, 22:1–39, 1986.

H. Dalianis. *Concise Natural Language generation from Formal Specifications*. PhD thesis, Stockholm University, 1996.

B. Di Eugenio, M. Glass, and M. Trolio. The DIAG experiments: NLG for intelligent tutoring systems. In *Proceedings of the AAAI Spring Symposium on Natural Language Generation in Spoken and Written Dialogue*, pages 120–127, 2002.

B. Di Eugenio, S. Haller, and M. Glass. Development and evaluation of nl interfaces in a small shop. In *Proceedings of the AAAI Spring Symposium on Natural Language Generation in Spoken and Written Dialogue*, pages 15–22, 2003.

J. Dowding, J.M. Gawron, D. Appelt, J. Bear, L. Cherny, R. Moore, and D. Moran. Gemini: A natural language system for spoken-language understanding. In *Proceedings of the 31st Annual Meeting of the Association for Computational Linguistics (ACL'93)*, pages 54–61, 1993.

M. Elhadad. *Using Argumentation to Control Lexical Choice: A Unification-Based Implementation*. PhD thesis, Columbia University, 1993.

M. Elhadad, K. McKeown, and J. Robin. Floating constraints in lexical choice. *Computational Linguistics*, 23:195–239, 1997.

M. Elhadad and J. Robin. SURGE: A comprehensive plug-in syntactic realisation component for text generation. Technical report, Computer Science Department, Ben-Gurion University, Beer Sheva, Israel, 1997.

G. Erbach. *Bottom-Up Earley Deduction for Preference-Driven Natural Language Processing*. PhD thesis, Universität des Saarlandes, Saarbrücken, 1997.

R. Freedman and C. Callaway, editors. *Natural Language Generation in Spoken and Written Dialogue: Papers from the 2003 AAAI Spring Symposium*, 2003. AAAI Press.

C. Gardent and K. Striegnitz. Generating indirect anaphora. In *Proceedings of the Fourth International Workshop on Computational Semantics (IWCS-4)*, pages 138–155, 2001.

S. Geldof. *Context-sensitivity in Advisory Text Generation*. PhD thesis, University of Antwerp, 2000.

D. Gerdemann. *Parsing and Generation of Unification Grammars*. PhD thesis, University of Illinois, 1991.

S. Goldwater, E.O. Bratt, J.M. Gawron, and J. Dowding. Building a robust dialogue system with limited data. In *Proceedings of the NAACL 2000 Workshop on Conversational Systems*, pages 61–65, Seattle, WA, 2000.

N. Green and B. Davis. Dialogue generation in an assistive conversation skills training. In *Proceedings of the AAAI Spring Symposium on Natural Language Generation in Spoken and Written Dialogue*, pages 36–43, 2003.

M. Halliday. *An Introduction to Functional Grammar*. Edward Arnold, London, 1985.

H. Horacek. Text generation methods for dialog systems. In *Proceedings of the AAAI Spring Symposium on Natural Language Generation in Spoken and Written Dialogue*, pages 52–54, 2003.

P. Jordan, M. Makatchev, and U. Pappuswamy. Extended explanations as student models for guiding tutorial dialogue. In *Proceedings of the AAAI Spring Symposium on Natural Language Generation in Spoken and Written Dialogue*, pages 65–70, 2003.

A. Joshi and Y. Schabes. *Tree Adjoining Grammars*. Springer-Verlag, Berlin, 1997. Chapter 2.

R. Kibble and R. Powers. Nominal generation in GNOME and ICONOCLAST. Technical report, Information and Technology Research Institute (ITRI), University of Brighton, UK, 2000.

E. Klabbers, E. Krahmer, and M. Theune. A generic algorithm for generating spoken monologues. In *Proceedings of the 5th International Conference on Spoken Language Processing (ICSLP'98)*, volume 6, pages 2759–2762, Sydney, Australia, 1998.

E. Krahmer, J. Landsbergen, and J. Odijk. A guided tour through LGM; how to generate spoken route descriptions? Report 1182, IPO, Eindhoven, The Netherlands, 1997.

B. Lavoie and O. Rambow. A fast and portable realizer for text generation. In *Proceedings of the 5th Conference on Applied Natural-Language Processing (ANLP-1997)*, pages 265–268, 1997.

O. Lemon, A. Gruenstein, R. Gullett, A. Battle, L. Hiatt, and S. Peters. Generation of collaborative spoken dialogue contributions in dynamic task environments. In *Proceedings of the AAAI Spring Symposium on Natural Language Generation in Spoken and Written Dialogue*, pages 85–90, Palo Alto, CA, 2003.

W.J.M. Levelt and S. Kelter. Surface form and memory in question-answering. *Cognitive Psychology*, 14:78–106, 1982.

E. Marsi. A reusable syntactic generator for Dutch. In *Computational Linguistics in the Netherlands 1997, Selected Papers from the Eight CLIN Meeting*, pages 171–194, Amsterdam, The Netherlands, 1998. Rodopi.

E. Marsi. *Intonation in Spoken Language Generation*. PhD thesis, University of Nijmegen, 2001.

E. Mathews, T. Jackson, A. Graesser, N. Person, and the Tutoring Research Group. Discourse patterns in Why/AutoTutor. In *Proceedings of the AAAI Spring Symposium on Natural Language Generation in Spoken and Written Dialogue*, pages 97–103, 2003.

C. Matthiessen and J. Bateman. *Text generation and systemic-functional linguistics: Experiences from English and Japanese*. Pinter Publishers, London, 1991.

D. McCullough, T. Korelsky, and M. White. Information management for release-based software evolution using EMMA. In *Proceedings of the Tenth International Conference on Software Engineering and Knowledge Engineering (SEKE'98)*, 1998.

D. McNeill. *Hand and Mind: What Gestures Reveal about Thought*. University of Chicago Press, Chicago, 1992.

S. McRoy, S. Channarukul, and S. Ali. Text realization for dialog. In *Proceedings of the International Conference on Intelligent Technologies*, Bangkok, Thailand, 2000.

S. McRoy, S. Channarukul, and S. Ali. Creating natural language output for real-time applications. *Intelligence: New Visions of AI in Practice*, pages 21–34, 2001.

I. Mel'čuk. *Dependency Syntax: Theory and Practice*. State University of New York Press, Albany, NY, 1988.

G. Neumann. *A Uniform Computational Model for Natural Language Parsing and Generation*. PhD thesis, Universität des Saarlandes, Saarbrücken, 1994.

J. Odijk. Generation of coherent monologues. In T. Andernach, M. Moll, and A. Nijholt, editors, *CLIN V: Proceedings of the 5th CLIN Meeting*, pages 123–131, Enschede, The Netherlands, 1995.

C. Pollard and I. Sag. *Head Driven Phrase Structure Grammar*. Studies in Contemporary Linguistics. The University of Chicago Press, Chicago, 1994.

X. Pouteau and L. Arévalo. Robust spoken dialogue systems for consumer products: A concrete application. In *Proceedings of the 5th International Conference on Spoken Language Processing (ICSLP'98)*, volume 4, pages 1231–1234, Sydney, Australia, 1998.

E. Reiter. NLG vs. templates. In *Proceedings of the 5th European Workshop on Natural Language Generation (EWNGL'95)*, pages 95–106, Leiden, The Netherlands, 1995.

E. Reiter and R. Dale. *Building Applied Natural Language Generation Systems*. Cambridge University Press, Cambridge, 2000.

S. Shapiro. SNePS: A logic for natural language understanding and commonsense reasoning. In L. Iwanska and S. Shapiro, editors, *Natural Language Processing and Knowledge Representation: Language for Knowledge and Knowledge for Language*, pages 175–195. AAAI Press/The MIT Press, Menlo Park, CA, 2000.

S. Shieber. The problem of logical form equivalence. *Computational Linguistics*, 19(1):179–190, 1993.

S. Shieber, G. van Noord, F. Pereira, and R. Moore. Semantic head-driven generation. *Computational Linguistics*, 16:30–42, 1990.

N.K. Simpkins. An open architecture for language engineering: The Advanced Language Engineering Platform (ALEP). In *Proceedings of the Linguistic Engineering Convention*, Paris, France, 1994.

M. Stone. On identifying sets. In *Proceedings of the First International Conference on Natural Language Generation (INLG 2000)*, pages 116–123, Mitzpe Ramon, Israel, 2000.

M. Stone. Lexicalized grammar 101. In *ACL Workshop on Tools and Methodologies for Teaching Natural Language Processing*, pages 76–83, 2002.

M. Stone, T. Bleam, C. Doran, and M. Palmer. Lexicalized grammar and the description of motion events. In *Fifth Workshop on Tree-Adjoining Grammar and Related Formalisms (TAG+ 2000)*, 2000.

M. Stone and C. Doran. Paying heed to collocations. In *Proceedings of the 8th International Workshop on Natural Language Generation (IWNLG'96)*, pages 91–100, 1996.

M. Stone and C. Doran. Sentence planning as description using tree-adjoining grammar. In *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics and the 8th Conference of the European Chapter of the Association for Computational Linguistics (ACL/EACL'97)*, pages 198–205, Madrid, Spain, 1997.

M. Stone, C. Doran, B. Webber, T. Bleam, and M. Palmer. Microplanning with communicative intentions: the SPUD system. Technical Report 65, Rutgers University Center for Cognitive Science, 2001.

M. Stone and B. Webber. Textual economy through close coupling of syntax and semantics. In *Proceedings of the 9th International Workshop on Natural Language Generation (IWNLG'98)*, pages 178–187, Montreal, Canada, 1998.

K. Striegnitz. A chart-based generation algorithm for LTAG with pragmatic constraints. Technical report, University of Saarbrücken, 2000a.

K. Striegnitz. Pragmatic constraints and contextual reasoning in natural language generation: a system description. Technical report, University of Saarbrücken, 2000b.

K. Striegnitz. Model checking for contextual reasoning in NLG. In *Proceedings of the Third Workshop on Inference in Computational Semantics (ICoS-3)*, pages 101–115, 2001.

M. Theune. *From Data to Speech: Language Generation in Context*. PhD thesis, Eindhoven University of Technology, 2000.

M. Theune. From monologue to dialogue: natural language generation in OVIS. In *Proceedings of the AAAI Spring Symposium on Natural Language Generation in Spoken and Written Dialogue*, pages 141–150, 2003.

M. Theune, E. Klabbers, J.R. de Pijper, E. Krahmer, and J. Odijk. From data to speech: A general approach. *Natural Language Engineering*, 7(1):47–86, 2001.

K. van Deemter and J. Odijk. Context modeling and the generation of spoken discourse. *Speech Communication*, 21(1/2):101–121, 1997.

G. van Noord. An overview of head-driven bottom-up generation. In *Current Research in Natural Language Generation*. Academic Press, London, 1990.

G. van Noord. *Reversibility in Natural Language Processing*. PhD thesis, University of Utrecht, 1993.

M. Verlinden. *A Constraint-Based Grammar for Dialogue Utterances*. PhD thesis, Tilburg University, 1999.

M. White. Text polishing: surface-oriented smoothing of generated text via mark-up based revision rules. Technical report, CoGenTex, February 2001.

M. White and T. Caldwell. EXEMPLARS: A practical, extensible framework for dynamic text generation. In *Proceedings of the 9th International Workshop on Natural Language Generation (IWNLG'98)*, pages 266–275, Niagara-on-the-Lake, Canada, 1998.

M. White and T. Caldwell. Beyond XSL: Generating XML-annotated texts with EXEMPLARS. Technical report, CoGenTex, November 1999.