Incremental Generation for Real–Time Applications

Anne Kilger and Wolfgang Finkler

RR-95-11
Research Report

Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

**Research
Report**
RR-95-11

Incremental Generation for Real–Time Applications

Anne Kilger and Wolfgang Finkler

July 1995

**Deutsches Forschungszentrum für Künstliche Intelligenz
GmbH**

Postfach 20 80
67608 Kaiserslautern, FRG
Tel.: + 49 (631) 205-3211
Fax: + 49 (631) 205-3210

Stuhlsatzenhausweg 3
66123 Saarbrücken, FRG
Tel.: + 49 (681) 302-5252
Fax: + 49 (681) 302-5341

Incremental Generation for Real–Time Applications

Anne Kilger and Wolfgang Finkler

DFKI-RR-95-11

This work has been supported by a grant from The Federal Ministry of Education, Science, Research and Technology (FKZ ITWM-01 IV 101 K/1).

ISSN 0946-008X

Incremental Generation for Real–Time Applications

Anne Kilger and Wolfgang Finkler
German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3, D–66123 Saarbrücken, Germany
E-mail: {kilger|finkler}@dfki.uni-sb.de

Abstract

The acceptance of natural language generation systems strongly depends on their capability to facilitate the exchange of information with human users. Current generation systems consider the influence of situational factors on the content and the form of the resulting utterances. However, the need to time their processing flexibly is usually neglected although temporal factors play a central part when directly addressing a human communication partner. A short response time of a system is crucial for its effective use. Furthermore, some applications — e.g., the simultaneous description of ongoing events — even necessitate the interleaving of input consumption and output production, i.e. the use of an incremental processing mode.

We claim that incremental processing is a central design principle for developing flexible and efficient generators for speech output. We discuss the advantages of parallel processing for incremental generation and several aspects of control of the generator. An extension of Tree Adjoining Grammar is introduced as an adequate representation formalism for incremental syntactic generation.

We present the system VM–GEN — an incremental and parallel syntactic generator based on Tree Adjoining Grammars. It offers flexible input and output interfaces that are adaptable to the requirements of the surrounding system by coping with varying sizes of input and output increments. The system's ability to produce fluent speech is a step towards approximating human language performance.

1 Introduction

The acceptance of software systems in natural language processing, e.g., human–computer interfaces, strongly depends on their capability to facilitate the exchange of information with human users. This article focuses on aspects of intelligent interfaces that provide means that facilitate the automatic generation of spoken language. There are many scenarios where speech I/O is worthwhile and effective. Research in human–computer interaction (hereafter, HCI) has explored a subset of these scenarios and evolved the term of a user in "hands busy or eyes busy environments." In addition, a professional typist would produce spoken language as input into a system much more faster than possible by entering written language on a keyboard. For tasks where the phone provides the most suitable form of communication channel speech output is still necessary (cf. [Booth 89]).

The design of systems for the automatic generation of spoken language requires the observation of models of performance that are adequate for the human user at both representation and algorithmic level (see [Marr 82]). Besides the consideration of desirable properties of the output of such a system, for example grammatical well–formedness, adequacy with respect to the contents, and coherence, there is a need to pay attention to the timeliness of the generation process.

1.1 Real–Time Constraints for Natural Language Generation

As research results in the field of HCI indicate, speech systems are subject to strong real–time constraints (cf. [Rubinstein & Hersh 84]):

- The most important requirement is that *the delay of the output shouldn't be too long.* Speech output of a system demands the human addressee to preserve steady attention in order to avoid missing some part of the transitory output. Noticeable delays get unacceptable as soon as they start to degrade human performance. Even a cooperative human interlocutor would be unwilling to continue a dialogue with a computer system if it took too long to produce a dialogue contribution.

- Conversational analysis discloses another time constraint of a dialogue that should be simulated in human–computer interaction. *The delays of the output shouldn't differ too much.* Typically, participants in a dialogue are expected to start the articulation of their contributions after a certain time span irrespective whether there are different degrees of difficulty in producing specific utterances. Otherwise, the course of the dialogue becomes incoherent and it is unclear whether a possibility for turn–taking should be utilized.

Therefore, reasoning about **When–to–Say**, i.e., the timing of output production, should be implemented in interactive applications considering high time pressure. In such applications the demand for real–time processing may be even more important than the need for a precise and stylish output. This demand affects all subtasks of the natural language generation process in two ways. Time pressure may force the system to restrict the search space during its choice processes, e.g., leading to a less sophisticated lexical choice. On the other hand, a generation component has to produce constructions that can be easily and quickly analyzed by the human addressee in a time–critical situation.

The demands for real–time processing can be fulfilled in several ways. A traditional sequential system is said to be well-suited for a time–critical application if it has the ability to

- produce complete utterances fast enough and
- drive (with respect to time) a speech synthesis component so that it articulates the utterance in a natural way.

However, there are some time–critical applications that do not allow for this type of processing. They are often characterized by the fact that the input elements to be verbalized arrive in a stream of specifications. The generator is expected to produce fragments of an utterance even before the input is complete. Examples of such applications are systems which simultaneously interpret natural language or report about ongoing events. Furthermore, for highly interactive applications it might be important to utilize the addressee's reaction to the past output in the continued production of the current utterance. Such applications, that incorporate components for the generation of speech, should use an *incremental processing* mode.

1.2 Incremental Natural Language Generation

In the course of time, the term 'incremental' has been used for the description of a specific processing strategy in different fields of computer science. [Lock 65, p. 462f] defines an **incremental compiler** as being "characterized by its ability to compile each statement independently, so that any local change in a statement calls only for recompilation of the statement, not the complete program. ... statement insertions, deletions or modifications are handled by adding, removing or replacing some elements in the program structure with appropriate changes in structure pointers." This definition specifies the ability to *independently* process the input increments as a central feature of incremental computation.

Incrementality has later on been discussed in relation to the task of natural language analysis. [Wirén 92, p. 1f] states that **incremental analysis** "is carried out bit by bit as the text is read or heard, rather than in one go when the text has come to an end. A new analysis (or a new information state) is thus computed after each new word or phrase by re–using as much as possible of the previous analysis." He distinguishes two kinds of incrementality:

- "Analysis of piecemeal, *left–to–right* extensions of a text. We call this **left–to–right incrementality**, or **LR incrementality** for short."
- "Analysis of *arbitrary* piecemeal changes (insertions, deletions and replacements) of a text. We call this **full incrementality**."

While the discussion of types of modifications of the input corresponds to the definition of incremental compilation, the description of incremental computation has been adapted to the needs of natural language processing: Input increments are expected to be related to each other so that global recomputations may be necessary.

In the field of natural language generation, [Kempen & Hoenkamp 82, p. 151] have introduced the term **incremental generation** from a psycholinguistic point of view: "Human speakers ... can start speaking having in mind only a fragmentary idea of what they want to say, and while saying this they refine the contents underlying subsequent parts of the utterance." This definition emphasizes the point that incrementality means a simultaneous working of the concerned modules on different parts of the utterance. As for natural language analysis, the problem of dependencies among choices also arises during generation and has been characterized by [De Smedt 90a, p. 10]: "This mode of generation, which is called *incremental* generation, seems to serve a system whose major purpose is to articulate relatively fluent speech, even if it is imperfect or incomplete. Once a partial sentence has been constructed, the generator will try to complete the sentence in a maximally grammatical way." First approaches to define incremental generation from a computational point of view resulted in rather vague descriptions such as "A cognitive model of generation should be incremental, or in other words, "on–line". That is, most of the processing relating to the output of a word should be done near the time when that word is output." ([Ward 89]). Obviously, there is need for a more formal definition of the term 'incremental generation.'

We begin by giving an application–independent definition of **incremental processing** which we use for the specification of an incremental natural language generator. Our definition of incremental processing describes the relationships between the consumption, internal processing, and output of increments by a module:

1. Each given increment triggers its immediate processing in the module so that from a global point of view:
   - processing starts before the input is complete,
   - first output increments – the so–called **output prefix** – are produced before processing is complete and if possible,
   - even before the input is complete.
2. The output increments must result from the processing of the past input.

The first requirement implies that the consumption of input increments and the production of output increments take place in an interleaved manner. The second constraint interdicts a system's designer from calling a natural language generator incremental, which starts with producing an "ups" unrelated to any input increment, then waits until processing is finished and outputs the real utterance. A system is called an **incremental system** if it can work in an incremental processing mode.

We formally define the different parts of the description given above by generalizing the definition of [Amtrup 95]: Let I(t), O(t), and Z(t) be functions of time describing the state of the input, the state of the output, and the internal state of a system, respectively. The system starts up in an initialization state which we describe by the predicate 'init': init(I(t)) holds if no input is given yet, init(O(t)) holds if no output is produced yet, and init(Z(t)) holds if there has been no processing yet. We define the starting time t_{start} as the maximum of all t such that init holds:

$$t_{start} = \max\{\mathrm{t} \mid \mathrm{init(I(t))} \land \mathrm{init(Z(t))} \land \mathrm{init(O(t))}\}$$

The system stops with a completion state which we characterize by the predicate 'compl': compl(I(t)) holds if the input has been completely provided, compl(O(t)) holds if the output has been completely produced, and compl(Z(t)) holds if processing has been finished. We define the ending time t_{end} as the minimum of all t such that compl holds:

$$t_{end} = \min\{\mathrm{t} \mid \mathrm{compl(I(t))} \land \mathrm{compl(Z(t))} \land \mathrm{compl(O(t))}\}$$

An incremental system is defined as combining incremental input consumption and incremental output production:

- Processing starts before input is complete, i.e., $\exists$ t: $\neg$ init(Z(t)) $\land$ $\neg$ compl(I(t))
- Output starts before processing is complete, i.e., $\exists$ t: $\neg$ init(O(t)) $\land$ $\neg$ compl(Z(t))
- Output possibly starts before input is complete, i.e., $\exists$ t: $\neg$ init(O(t)) $\land$ $\neg$ compl(I(t))

Each state of the system is characterized by a function F_Z the result of which depends upon the time, the input given and the initial state of the system:[1]

$$\mathrm{Z(t)} = F_Z(\mathrm{t}, \mathrm{I(t)}, \mathrm{Z}(t_{start})) \quad \text{for } t > t_{start}$$

Each new state of the output is constructed by applying a function F_O to the current input and the current state of the system:

$$\mathrm{O(t)} = F_O(\mathrm{t}, \mathrm{I(t)}, \mathrm{Z(t)})$$

[1]We here adapt the definition of [Schade et al. 91, p. 25] who describe the dynamics of a system in the context of a discussion about the process–related aspects of coherence.

The functions F_Z and F_O may be simple, if for example the tasks allow for pipelined processing, or very complicated due to dependencies among choices. Concerning temporal relations, the results of applying F_O to later given input increments may outstrip those of applying F_O to previously given elements. Furthermore, the results may depend upon temporal aspects of the input such as the order of input increments or pauses between them, since these aspects influence both I(t) and Z(t). F_O is also a function of time because reasoning about When–to–Say influences the output behavior.

The terms **input increment** and **output increment** are generally defined as the differences between states of input and output for two points in time. Let Δ_I be a function computing the difference between two states of input, e.g., $\Delta_I(t_1, t_2)$ is the difference between states of input at times t_1 and t_2, whereby $t_1 < t_2$. Let Δ_O be a function computing the difference between two states of output, e.g., $\Delta_O(t_1, t_2)$ at times t_1 and t_2, $t_1 < t_2$.

There are two ways of defining incremental interfaces. **Qualitative incrementality** is realized by having one component repeatedly hand over complete input information to another, increasing the quality of the produced output each time. In that case, the receiving component either has to start from scratch each time or incorporate a function Δ_I which helps to make the differences explicit and use them to trigger specific processing. **Quantitative incrementality** is realized by having the sending component locally compute several Δ_O and hand them over to the subsequent component in a stream. This kind of incremental processing is referred to by most of the literature cited above, so we will use it as starting–point for our discussion.

Complex interdependencies among decisions play an important role in **incremental syntactic generators** that are defined by their ability to realize incremental processing as introduced above. Enriching previous research on incremental natural language generation (see beginning of Section 2), we explicitly study two interrelated aspects of this processing mode for syntactic generation: the consumption of incremental input, i.e. F_Z, and the production of incremental output, i.e. F_O.

1.2.1 Incremental Input Consumption

In contrast to a non–incremental system, an incremental generator starts computing on fragments of the input which are handed over to the generator in a piecemeal way. This can be utilized in order to achieve parallelism. The back–end system providing the input may run along with the generator, which leads to a reduction of runtime of the whole system thereby approaching *real–time behavior*.

Another advantage of an incremental processing strategy lies in the dependencies among choices made by distinct modules. In a non–incremental system a component stops working after a complete result has been computed and handed over to the next component. This doesn't allow for the consideration of the results of a subsequent component during the computation of the current component. An incremental system allows for the consideration of such results, e.g., via processing feedback information, thereby increasing its *flexibility*. During natural language generation, there may be dependencies among the choices made by the word choice module and the microplanner. Decisions about syntactic structures, e.g. passivization, depend on features of the chosen words. The other way round, words have to fit into the resulting syntactic structure to avoid fusion errors.

In an interactive environment, incremental processing can be exploited to *efficiently* handle user interrupts: The user may signal that it is not necessary to complete the current sentence while the generator is still in the process of producing an utterance. This signal could lead to an immediate stop of processing within the generator which saves computing time as compared to a non–incremental system, that must have finished its output before starting to utter it.

By adapting the terms used by [Wirén 92] for incremental analysis, incremental systems can be characterized according to the kind of input specifications they accept. If new specifications may only be added in a piecemeal way we end up with so–called **left-to-right incrementality**. In this case, I(t_1) is always a part of I(t_2) for $t_1 < t_2$. The status of each input increment is thereby stable. In **fully incremental** systems it is possible to use input increments to modify or even delete previously specified input increments. Here, there may be t_1 and t_2, whereby $t_1 < t_2$ and I(t_1) is not enclosed in I(t_2). Such a system has to obviously provide for some kind of internal revision handling.

The design of the input interface for a syntactic generator depends on the features of the components that deliver the input, e.g., the microplanner. Since there are numerous approaches for the definition of the components of a natural language generator and for the design of the interfaces, no general characterization of I(t) can be given. The specification of I(t) for our system VM–GEN is described in Section 3.3.1.

1.2.2 Incremental Output Production

The output interface of an incremental component has to allow for handing over of partial results to the succeeding components. This feature is a prerequisite for building up cascades of incremental components that can work almost simultaneously. If incremental output is not realized the system is called **partially incremental** (cf. [De Smedt 90a]). For a natural language generator, the 'succeeding component' is the human user forcing the system's designers to consider features of human communication.

For applications in the field of spoken language production, O(t) can be further idealized as describing a *sequence of words*: O(t) = $(w_1 \ldots w_n)$ for $t > t_{start}$. It is important to distinguish between a *set* and a *list* of output increments, since a list encodes part of its meaning by the order of its elements. This obviously applies to natural language.

Furthermore, for spoken output, a word w_i that has been uttered cannot be withdrawn from the list because it has been perceived by the human addressee. This means that an incremental system for spoken language generation has to realize quantitative incrementality in its output interface, i.e., for all $t_1 < t_2$ $O(t_1)$ is a prefix of $O(t_2)$. Each output increment Δ_O is a list of at least one word.

The main advantage of incremental output production lies in the possibility to shorten pauses before and inbetween output increments. This ability is vital for the production of spoken output. In Section 1.1 we discussed the importance of guaranteeing for both short as well as regular delays when dealing with the task of When–to–Say. Incremental output production is an important means of helping a system to flexible fulfillment of these strong real–time constraints. Since the utterance can be started before the output has been completely computed, the system can try to *adequately spread its output increments over the response time interval.*

When time pressure forces the system to start the output even before the input is complete, the risk of overt repairs is increased. The output prefix results from decisions that have been made on the basis of the input given so far. They may conflict with input increments given later on (see [Kempen & Hoenkamp 87], [Ward 91], [De Smedt 91], [Finkler & Schauder 92]). Therefore, an incremental generator must be able to realize three forms of incremental output production:

- The least complicated type of incremental output production is called **right–concatenation**. It occurs when the verbalization of every incoming part is articulated following the uttered prefix without any internal or external restructuring. This means: $\forall$t, the list O(t) grows monotonically at its end and always encodes a syntactically correct prefix of a sentence.[2] For example, if the output prefix "the" has already been uttered, and the input increments leading to the modifier "first" and the noun "meeting" are given and processed in time, they may be uttered in the sequence "first meeting".
- If input increments are given too late to be correctly integrated into the sentence, **overt repair** may take place leading to utterances like "The meeting ... I mean the first meeting". In that case, there is a point in time t such that $O(t_i)$ is no longer a valid prefix of a sentence for all $t_i > t$.
- Sometimes it is possible to hide the repair, e.g., if changes of syntactic constructions don't lead to changes in the already uttered prefix. **Hidden repairs** can sometimes be recognized by pauses or a bad style, as in "The ...... first meeting". When starting the utterance the system might have planned to immediately utter the noun when new input necessitated to further specify it by means of the modifier "first". The delayed integration of the modifier may lead to an unnatural pause after the determiner.

[2]This can be seen as a 'valid prefix property' for natural language generation.

In a way, the problem of repair during incremental output production decreases the flexibility of the system, since starting an utterance reduces the set of possibilities to integrate new input increments into the sentence itself. It also decreases efficiency because of the extra costs of producing repairs or the processing of alternative solutions. In order to both fulfill the time constraints and avoid overt repair, incremental output production should be used to utter succeeding parts of the sentence as soon as *necessary* (see Section 1.1), rather than uttering them as soon as *possible*.

1.2.3 Incremental Generation of Spoken Language

To sum up, there are pros and cons for using incremental processing in natural language generation. Incremental processing in general lacks the ability to utilize a global view of the given input and therefore often leads to a non–elaborated style. Further disadvantages are ascribable to the strong interrelationships between the input increments of a natural language system: related elements cannot be processed independently – as would be the case in an optimal incremental approach. They therefore cause delays during output construction. Forcing incremental output may lead to overt repair and requires its adequate handling. Obviously, incremental processing is not expected to be usable in an optimal way.

Nevertheless, incremental processing leads to gains in efficiency if the input increments arrive in a suitable order. The most important advantage of incremental processing is the increased flexibility of the generator, which allows for simulation of characteristics of spontaneous speech; incremental output contributes to the fluency of utterances. Modification, addition, and deletion of input increments can be processed and enlarge the bandwidth for the system usage.

We conclude that for speech production it is more suitable to produce output that fulfills the time constraints while suffering from a few imperfections, than to produce perfectly formulated sentences that can be uttered only after an unacceptable delay.

1.3 The Incremental Syntactic Generator VM–GEN

In this work, we will introduce the incremental syntactic generator VM–GEN.[3] As part of the VERBMOBIL project [Wahlster 93] it was to fulfill the requirements of real–time communication in the framework of translation of face–to–face dialogues. VM–GEN differs from most known generators in its ability to combine incremental input consumption and incremental output production including simple kinds of overt repair. It uses Tree Adjoining Grammar (TAG) for syntactic representation.

[3]VM–GEN was developed on the basis of TAG–GEN, an incremental syntactic generator that was used in WIP (cf. [Wahlster et al. 93]), a system for multimodal presentation of information.

The rest of this paper is organized as follows. In Section 2, we discuss design principles for systems realizing incremental input consumption and incremental output production. In Section 3, we describe some features of VM–GEN illustrating how these design principles can be realized.

2 Design Principles for incremental syntactic generation

In this section we provide the motivation for several prominent design principles of incremental syntactic generation. We explicitly distinguish between requirements on the realization component of a generator that arise from dealing with incremental input, i.e., designing F_Z, and requirements that originate from the need to produce incremental output, i.e., designing F_O.

The main emphasis in developing our syntactic generator was laid on accomplishing the difficult task of interleaving the processes of input consumption and output production. Our approach differs from two research traditions in natural language generation. At one hand, an increasing number of generation systems working in an incremental fashion has been developed during the last decade. The systems KAMP by [Appelt 85], MUMBLE by [Meteer et al. 87], IPG by [Kempen & Hoenkamp 87], SOCCER by [Herzog et al. 89], POPEL–HOW by [Finkler & Neumann 89] and [Reithinger 91], IPF by [De Smedt 90a], the generation component in ΦDM DIALOG by [Kitano 90], FIG by [Ward 91], and SYNPHONICS–Formulator by [Abb et al. 93] exploit a processing mode that is at least partially incremental. These systems represent impressive innovations and realize steps on the way towards high performance natural language interfaces. However, they altogether do not cope with the problem of incremental output in its full extent. On the other hand, our approach differs from the big number of systems emerging from research in the field of Artificial Intelligence and Computational Linguistics that provide solutions to manifold phenomena of linguistic competence but do not pay attention to time–critical aspects of language use.

Basically, our incremental generator pursues the following fundamental subtasks of syntactic generation:

- The increments of the input to the generator – roughly consisting of lemmas and semantic relations – have to be translated into a hierarchical phrase structure obeying dominance relations and reflecting syntactic properties of the selected lexical elements.
- The parts of the structure must be linearized in a grammatically correct way and the linearized list of terminals must be handed over to the succeeding component where the utterance is provided for the user.

During processing at the hierarchical and the positional level several situational influences on syntactic choices have to be considered.

The input interface to our generator currently uses a specification of lemmas that were selected by another component. Therefore, the generator does not have much freedom during the creation of the hierarchical structure as well as during linearization for the target language English since word order is rather restricted as compared to other languages like German. However, the attempt to automatically produce fluently spoken language output brings a wide range of process–related complications with it.

We are currently working on two components which provide input to the syntactic generator – a component for lexical choice and a so–called microplanning component that determines the use of anaphora, propositional format of utterances, and nominalization decisions besides others. We provide an interactive architecture that enables the consideration of the effects of decisions (made inside the components) on the inner working of the syntactic generator and vice versa.

2.1 Consumption of Incremental Input

2.1.1 Distributed Parallel Model for Flexible Processing of Increments

As mentioned in Section 1.2.1, the processing of several incremental modules overlaps in time. Therefore, there is an innate relation between incrementality and parallelism. A combination of several incremental modules forms a cascade of components working in parallel[4] as soon as they start to exchange data. This kind of parallelism is called coarse–grained parallelism. While one component is engaged in consuming and processing its input, the preceeding component may provide further data. The preceeding component may even be influenced by partial results of the subsequent component which leads to an incremental architecture with feedback (see, e.g., the system POPEL ([Reithinger 91]) or the interactive architecture proposed for VERBMOBIL ([Görz 92])).

In addition to coarse–grained parallelism, we follow the approach proposed by [Finkler & Neumann 89] and suggest

fine–grained parallelism (Design Principle 1)

for incremental syntactic generation. Each component of a cascade may itself be conceived as a parallel model.[5] Some of the properties that are prerequisites for the use of an

[4]Besides engineering–oriented considerations, there are psycholinguistic studies that provide some evidence for the "...assumption that the human language processing apparatus is capable of carrying out different tasks in parallel" ([De Smedt 90a]).

[5]Again, from a psycholinguistic point of view, there is evidence for 'computational simultaneity' during syntactic processing. [Garrett 80] assumes that several types of speech errors, e.g., some exchange, fusion, and omission errors, emanate from the simultaneous processing of several units of grammatical encoding.

incremental processing mode do at the same time favor distributed parallel processing, thereby setting up a close relationship between the two modes.

For a component using explicit representation structures, incremental processing necessitates a suitable segmentation of those structures. In terms of our formal definition of incremental computation, the function F_Z — which makes use of the representation structures — must be carefully designed: Partial structures should correspond to input increments in such a way that they can be chosen on the basis of those input increments and be composed without retracting too much of previously built structures. The various interdependencies among the decisions made at the syntactic level make their exact identification vital in order to allow for

independent processing of all independent tasks. (Design Principle 2)

The existence of units that may be computed independently from each other suggests their distribution among separate parallel processes.[6] In a cascade of distributed parallel components, several partial results are computed simultaneously within a component and are forwarded to the next component to be handled by another set of parallel processes. The runtime of the overall system can thereby be reduced. In this way, an incremental component benefits from the use of a distributed parallel model as an operational base (see [Finkler 89], [De Smedt 90a]). We therefore suggest that components for grammatical encoding should be designed using object–oriented concurrent programming techniques (cf. [Yonezawa & Tokoro 87]).

However, exploiting parallelism is only useful if the gain from distributing the computation among several processes surpasses the costs of synchronization and communication. It is difficult to estimate the extent to which the task of natural language generation can fulfill this demand. Building a single structure for an utterance does not allow for a completely independent processing of partial structures. Additionally, the order of output increments embodies semantic, syntactic, and pragmatic information and forces the generator to collect and line up several of the parallely computed parts of the sentence before uttering them. On the other hand, [Kempen & Hoenkamp 82] require from incremental generators the ability to simultaneously construct parallel branches of syntactic trees in cases where there are no "cross–branch computational dependencies forcing a systematic order upon the construction of branches" (cf. [Kempen & Hoenkamp 82, p. 153]). Realizing a distributed parallel system makes it possible to explicitly state the independency of subtasks whereas serial approaches realize a predetermined order of decisions.

From a technical point of view, four main points have to be considered when realizing a distributed parallel approach (cf. [Bond & Gasser 88]):

- The given problem has to be partitioned adequately into partial problems. As mentioned above, this means dividing the task of grammatical encoding among a set of objects, which deal with partial syntactic structures. The objects integrate their

[6][Ward 91] uses the term 'part–wise parallelism' for this kind of processing.

local structures into a global representation of the current utterance. The definition of an adequate segmentation of syntactic structures is strongly coupled with the selection and design of the underlying representation formalism (see Section 2.1.3 for some of the requirements on the syntactic representation formalism).

- The bandwidth of possible communications and interaction between the processes has to be determined. It depends on the interdependencies among the specific tasks of the objects. For a syntactic generator, the objects have to cooperate at the hierarchical and at the positional level in order to access parts of the syntactic environment that are relevant for their local processing. In so doing, they may utilize synchronous as well as asynchronous communication depending on the circumstances prevailing (see Sections 3.3.2, 3.4.3, and 3.4.4).
- Interdependencies between objects necessitate a coordination of the associated processes. During syntactic processing, it may happen that an object has to stop its work to wait for relevant information (e.g., agreement information) from a cooperating object.[7] This kind of coordination requires synchronization of the objects. A global view (see Section 3.1) or a central component can be used for controlling complex cases of cooperation, e.g., during production of self–repairs of the system ([Finkler 95]).
- The combination of locally computed results into a globally consistent solution constitutes the central task of distributed parallel processing. In the domain of syntactic generation global consistency is reached when the resulting utterance is comprehensible. This is a weaker requirement than grammatical correctness. Especially the incremental production of spontaneous speech often leads to utterances that are not grammatically correct but obey certain well–formedness conditions (cf. [Levelt 83]; see Sections 3.4.3 and 3.4.4).

2.1.2 Lexical and Syntactic Guidance

An important aspect of incremental generator control deals with bidirectional dependencies between decisions during lexical and syntactic choice. Lexical items in the input to the syntactic generator may affect the generation of certain syntactic structures due to their grammatical properties, e.g., subcategorization constraints. This influence of lexical elements on syntactic representations is evident in many modern linguistic theories (for example, HPSG [Pollard & Sag 94] or GB [Chomsky 81]). It is unreasonable to design a generator which creates syntactic structures in a purely top–down fashion without paying attention to the concrete content words at hand, and which tries to incorporate lexical elements into the structure later on. Instead, [Levelt 89] postulates that the formulation process should be lexically driven.

[7][Eikmeyer et al. 91] describe a production model where 'lack of input' for one of four main processes of the natural language generator is used to trigger the production of some kind of covert repairs such as hesitation markers or repetitions.

On the other hand, the constructions chosen for the words must fit together to form correct sentences thereby paying attention to syntactic properties, e.g., co–occurrence restrictions of lexical heads. Furthermore, situational factors of the generation process may influence syntactic decisions, e.g., the choice of passive voice which in turn may exclude lexical items from the result of word–choice. In an integrated system, syntactic properties can be used during word choice. A modular approach should provide a flow of information from the syntactic level to the word–choice component.

The handling of such bidirectional dependencies[8] gets more complicated in case of incremental generation, since temporal aspects and the order of decisions have to be considered. The demand for the immediate processing of incrementally given lexical elements forces the generator to build partial syntactic structures without knowing about the whole contents to be incorporated into the current utterance. The partial structures embody subcategorization frames that (for example) restrict the category of new lemmas to be integrated into the ongoing utterance. Generally, the word–choice process for the new lemmas can be characterized as syntactically guided. Summing up the discussion above,

lexical and syntactic guidance (Design Principle 3)

should be realized within the generator.

2.1.3 Requirements on the Syntactic Representation Formalism

A natural language generation system working in an incremental mode imposes several constraints on the shape of its underlying representation formalism and the operations related to elementary structures.

In general, input increments may arrive in steps and in an arbitrary order. Therefore, neither pure top–down nor bottom–up processing are suitable for the efficient expansion of syntactic structures. As discussed by [De Smedt & Kempen 87], syntactic structures may grow in upward or downward direction, or be modified by insertion of additional elements. Upward expansions for the syntactic structure may be suitable, e.g., when a head is specified as input after some of its arguments. Filling the subcategorization frame of a lexical head with a structure of an argument is an example of downward expansion. Insertion can be illustrated by means of a subsequent specification of a modal or a demand for negation of a predicate, depending on the realization of such phenomena in the grammar formalism and theory. The combination operations of the representation formalism must support the kinds of expansion mentioned above. In Section 3.2 we present an extension of Tree Adjoining Grammar that allows for

flexible expansion operations. (Design Principle 4)

[8]There are similar interdependencies between microplanning and syntactic realization. [Hovy 88] suggests the interleaving of ‘prescriptive planning’ and ‘restrictive planning’ to enable a system to guide grammatical encoding by both the conceptual input and by language–specific properties.

As mentioned at the beginning of Section 2, syntactic generation comprises the construction of hierarchical and positional structures. If there is a separate description of the syntactic rules that define hierarchical and positional constraints, different serializations of nodes of a hierarchical structure can be encoded without necessitating a duplication of the structures. As a consequence, hierarchical structures can be chosen during syntactic generation without considering decisions about their linearization. These decisions might be made dynamically while the input is processed incrementally, reflecting, e.g., the input order of the concerned increments, as realized in the approach of [De Smedt 90a]. We suggest the

> **separation of the representation of hierarchical and positional constraints (Design Principle 5)**

primarily for reduction of redundancy during processing.

2.2 Production of Incremental Output

2.2.1 Coping with Temporarily Incomplete Input Information

In the course of incremental input consumption, the input information that is used for generation is temporarily incomplete. As long as the input is expected to be completed later on, decisions can principally be postponed until the missing information is available. For the production of incremental output, arbitrary delays are not acceptable because of the time constraints on the output interface. Therefore, decisions must be made regardless of the incomplete state of the input information thus opening up a search space of alternative continuations that can be explored by using breadth–first or best–first search.

During breadth–first search, the generation process has to deal with a number of competing alternatives (cf. [Ward 91], [Finkler & Neumann 89]). Such a model allows for the simultaneous creation of paraphrases, that can be utilized in case the processing of the favored alternative runs into a failure. The disadvantage of this kind of parallelism is the high processing effort caused by the branching factors that function as multipliers. Furthermore, the production of incremental output changes the state of the global system so that some of the paraphrases built previously cannot be used to correctly continue the utterance. In other words, 'classical' backtracking to intermediate states of the system is no longer possible if some output increments have already been articulated. The uttered prefix cannot be withdrawn because the hearer may already have perceived it and eventually even have reacted to it. In addition, humans frequently refer to the reparandum while producing a self–repair (cf. [Kempen 91]). Such results also cannot be obtained by simple insertion of paraphrases.

In a best–first approach only one working hypothesis can be considered as the basis of further processing. In case of failure previous decisions have to be withdrawn and another solution has to be computed taking the output prefix so far into account. The degree of efficiency strongly depends on the quality of the choice mechanism used to select the most favorable alternative. A best–first processing strategy performs early decisions between competing alternatives thereby improving the chance of shortening the delay between the articulation of single parts of the current utterance. Furthermore, it makes the need to continue the expansion of the alternative structures that haven't been selected for articulation obsolete.

Although both approaches can be used for incremental processing, we recommend the best–first approach for the reasons mentioned above. We therefore introduce

best–first search (Design Principle 6)

for our incremental natural language generator.

2.2.2 Effects of Incremental Output Production on Syntactic Processing

Resuming Design Principle 5, the representation formalism used in an incremental syntactic generator should allow for the separation of hierarchical and positional rules. There are several alternative approaches that fulfill this requirement and agree with the overall goal of incremental output production in a more or less beneficial way. Most important are their differences with respect to the size of syntactic structures that can be hierarchically composed and linearized on the basis of the input information at hand; this influencing the size of output increments and the length of inter–sentential delays. If linearization constraints refer to relatively large hierarchical structures — as is the case for context–free rules like 'VP $\rightarrow$ Subj V Obj IndirObj' — information about all concerned elements must be given before the structure can be chosen and its output can start. Alternatively, those structures can be chosen on the basis of incomplete information, thereby running the risk of overt repairs.

In our work we have concentrated on studying effects of incremental output production on syntactic processing, i.e., the design of F_O, and we are therefore interested in examining approaches that allow for the production of rather small output increments with short delays. To this end, the best candidates seem to be syntactic representation formalisms that allow the handling of minimal structures on the hierarchical and the positional level so that input increments can be almost directly mapped onto output increments. This is a step towards the 'independent processing of independent tasks' in syntactic generation (see Design Principle 2). There are principally at least two ways of fulfilling this requirement:[9]

- LD/LP–style approaches — e.g., 'VP $\rightarrow$ {Subj, V, Obj, IndirObj} + ((Subj<V, V<Obj, V<IndirObj) ...)' — allow for choosing hierarchical and positional rules depending on the difference in amount of input information. This decoupling makes it possible to isolate parts of hierarchical structures that can be linearized and

[9]For a detailed discussion see [Kilger 95].

uttered (e.g., subject and verb) while postponing the handling of the rest (object and indirect object).

- Some representation formalisms (e.g. MC–TAGs, see [Weir 88]) allow the definition of sets of small hierarchical structures that have to be combined in the course of natural language processing. The combination operations determine the final hierarchical structure which additionally defines the word order — e.g., the set of rules $\{VP \rightarrow \text{Subj VP}, VP \rightarrow \text{Obj VP}, VP \rightarrow \text{V VP}, \ldots\}$ plus constraints for the derivation.

Both approaches allow for a

> **separate processing of hierarchical and positional rules. (Design Principle 7)**

For incremental syntactic generators, the interdependencies among choices are further increased if incremental output is to be realized. Incremental output production introduces the uttered prefix of the current sentence as an additional constraint for further syntactic processing ([Finkler & Schauder 92]). If new input increments are consumed, they may be translated into expansions of hierarchical and positional structures that have effects on the prefix. However, during the production of spoken output, the prefix has already been 'handed over' to the human addressee and is therefore not available for local change any more. Therefore, an incremental module should have the following capabilities:

- to be able to preserve the output prefix by avoiding the necessity of changes (by exploiting alternatives or by realizing hidden repair). If this is not possible then it should have the ability
- to inform the succeeding component about necessary modifications using an appropriate method (by realizing overt repair).

Obviously, overt repair impedes communication with the human addressee. Firstly, there is a higher effort involved in understanding repaired utterances because each repair must be unravelled by the addressee. Secondly, a high frequency of overt repairs diminishes the reliability of the uttered prefix. Therefore, the uttered prefix should be preserved. This requirement influences the design of incremental generators. In addition to the bidirectional dependencies between word choice component and syntactic generator, as stated in Section 2.1.2, the production of incremental output causes additional interdependencies that are due to the timeliness of decisions. Decisions that are made during generation might be correct in structure but inappropriate with respect to the progress in time, that is manifested by the output prefix. For example, the uttered prefix "We will meet in the conference room" plus the linearization constraint 'ADJ < N' should influence the hierarchical level, and might lead to the realization of a relative clause instead of an adjective phrase for the input increment "small" ("We will meet in the conference room …which is small"). There are several ways of realizing these bidirectional dependencies. In a modular approach, they can be modelled by realizing feedback between components. Alternatively, an integrated approach facilitates the realization of interdependencies by sharing task–specific knowledge.

The second demand concerns all components of a cascade with incremental interfaces. The 'interface languages' must provide means of describing modifications in the output of the sending component, allowing the receiving component to react on those changes appropriately. It is more or less easy to realize language constructions that describe modifications in 'artificial' interfaces. The central problem of incremental output of natural language generators is that there is no artificial but a natural interface to a human hearer or reader, and that is the natural language. It predefines possible methods to convey overt repairs to the receiver. In a large corpus–based analysis of repair scenarios (see [Finkler 95]) we study overt repairs in human language use in order to develop methods that allow generators to imitate this behavior. Realizing

adequate repair strategies (Design Principle 8)

is important for incremental output production.

2.2.3 When–to–Say

The decision about When–to–Say, i.e., the timing of output production O(t), is crucial for the overall behavior of a generation system. One extreme position is to utter immediately – running the risk of having a prefix of the utterance that necessitates expensive processing for repair later on. The other extreme is delaying the output until the utterance has been completed. In this case, the system is only partially incremental and the long initial delay invalidates the quest for fluency in the output. In order to allow for adequate output behavior, the system has to preserve the incremental processing mode and reduce the number of repairs as well. This goal can be approached by applying some heuristics that describe the tactical use of delays in the output.

Research in HCI (see Section 1.1) confirms that output delays are acceptable and even natural as long as their length is limited and doesn't vary too much. For the production of spoken output, it seems useful to define a *threshold for output production* that can be set according to situational parameters such as time pressure and model of the hearer. The permitted delay may be used for building a reliable part of the utterance. Furthermore, the output increments, i.e. the Δ_O, should be oriented at constraints resulting from empirical studies in order to *avoid pauses at unnatural positions* in the utterance. For example, it might be practical to gather partial structures until a complete phrase is created and then handed over to the articulation component as a whole. In order to fulfill both aims described above we demand the

decoupling of surface generation and output production. (Design Principle 9)

If the time threshold is exceeded while the system is still waiting for input information in order to produce the next output increment,

default handling (Design Principle 10)

should be used to allow for adequate output behavior. It has to be combined with suitable repair strategies for the cases where input increments given later on contradict with the chosen defaults. Since this situation is comparable to fully incremental processing, that includes the modification of increments, it should be possible to use the same repair strategies for both cases.

2.2.4 Monitoring and Self–Repairs of the System

During incremental generation the utilization of delays cannot completely exclude cases where the constraints resulting from previous decisions lead to contradictions during the verbalization of successive input increments.[10] In order to cope with these problems, they first of all have to be detected by the system. This task is called monitoring. The effectiveness of controlling the correction of such failures supervising depends on the amount of available information. For both tasks, monitoring and supervising, there should be direct access to the internal state of the generator. This allows for the localization of the source of the failure, as well as the reuse of partial structures during repair. We therefore introduce

production–oriented monitoring and control. (Design Principle 11)

In contrast to this approach, the so–called 'perceptual theory of monitoring' ([Levelt 83]) describes monitoring as merely analyzing the inner or overt speech without direct access to the processes during grammatical encoding.

Several strategies for repair, e.g., overt or hidden repair, can be utilized in systems to manage the continuation of a sentence. In each case of a retracing repair, decisions must be withdrawn to perform a repair, i.e., structures have to be replaced. This can affect other structures that are related to them. There is a need for fast identification of the affected structures and the access to them. Storing the course of development of the current syntactic structure in a reason maintenance system supports such revisions (see [Finkler 95]). Another way to identify affected structures and to find syntactic alternatives consists of parsing the current output. This necessitates the use of an incremental parser that allows for the analysis of the prefix of an utterance (see, e.g., [Poller 93]).

The whole process of repair must be globally synchronized with the ongoing computation of the generator in order to guarantee consistency. To give an example: a modifying

[10] These phenomena naturally occur in spontaneous speech when humans 'talk themselves into a corner.'

adjective may be entered after the modified noun has just been uttered. The system can try to realize a hidden repair by triggering the selection of a relative clause. However, this solution can only be achieved if further output is delayed until the relative clause has been inserted and uttered.

3 VM-GEN

3.1 Architecture of the Syntactic Generator of VERBMOBIL

The system VM–GEN is designed for the incremental generation of natural language utterances. As will be explained later in this section, the generator fulfills most of the design principles discussed in Section 2. Its architecture is illustrated in Figure 1. Since we realize lexical guidance, the input (*goal of the utterance*) consists of content words and the semantic relations between them. These entities and relations can be forwarded to the generator regardless of their order and length of pauses contained thus providing a flexible interface to the component which calls VM–GEN. One object in a distributed parallel system is created for each input increment. Each of these objects runs through four processing phases that are illustrated as generator components in the figure. In the *Input Interface*, an adequate syntactic rule[11] is chosen according to the respective input information. Objects in the *Phrase Formulator* try to combine their locally managed syntactic structures by exchanging relevant data. In the *Linearization Component*, they serialize their structures and compute the inflected forms of the lemmas. According to Design Principle 9 — 'decoupling surface generation and output production' — incremental output is realized in the *Output Interface* by synchronizing the output activities of the single objects. Since each object rather changes its state than maps some represented structure into the next processing level, it still belongs to the pool of active objects which can co–operate with objects at other levels. This characteristics of the system makes it a partially integrated approach. The *Monitor* watches over the communications of the objects and is also used for purposes of controlling the system's progress.

The following sections deals with the representation formalism for syntactic structures and the components of VM–GEN in detail and illustrates the internal processing using a basic example.

[11]With the term 'syntactic rule' we refer to an elementary tree of the Tree Adjoining Grammar we use (see Section 3.2).

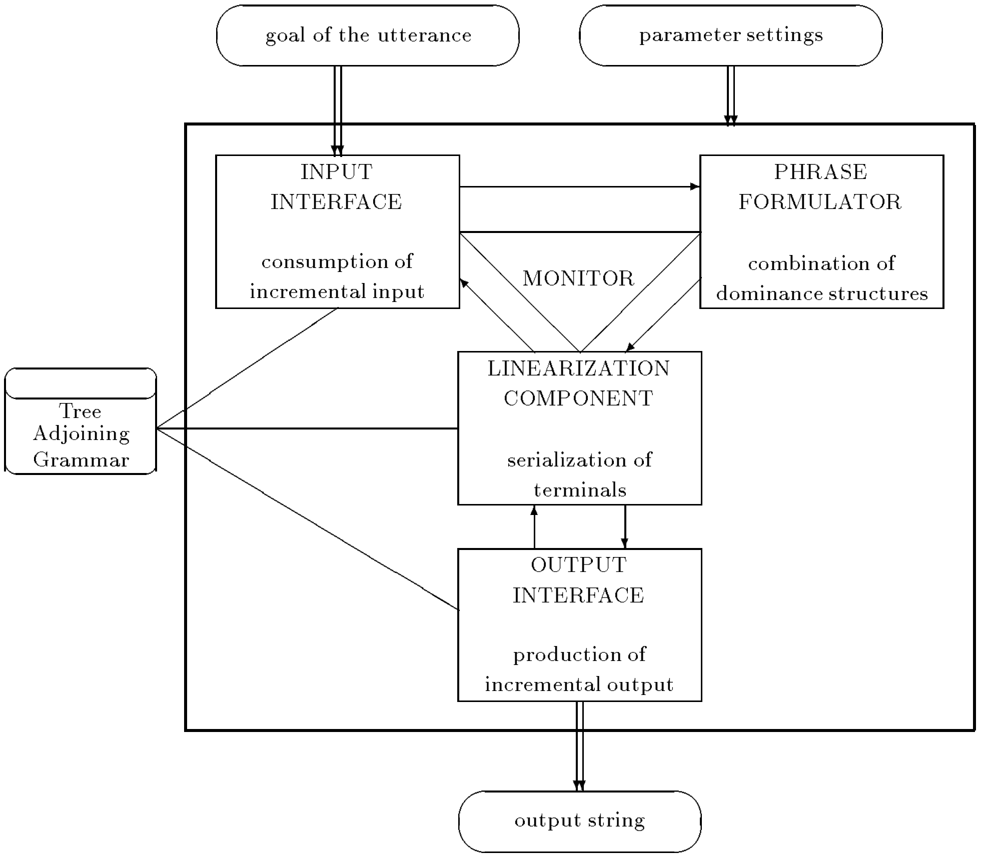


Figure 1: Architecture of VM–GEN

3.2 Fulfilling the Requirements on the Representation Formalism

As pointed out in [McDonald & Pustejovsky 85], [Joshi 87a], [Shieber & Schabes 90], and [McCoy et al. 90], **Tree Adjoining Grammars** (hereafter, TAGs, cf. [Joshi et al. 75, Joshi 85b]) are a suitable means of implementing a representation formalism in natural language generation systems. This is mainly due to their ability to provide an *extended domain of locality* to state syntactic dependencies within elementary structures. Each TAG defines a set of **elementary trees** that is split into **initial trees** and **auxiliary trees**. Initial trees describe phrase structures, their internal nodes being associated with nonterminal labels, all their leaves with terminal labels. Auxiliary trees represent recursive constructions by defining a specific kind of leaf – the **foot node** – that is labeled with the same nonterminal as the root node of the tree. Elementary trees of a TAG can be combined using the operation **adjunction**, which replaces an internal node X of one tree with an auxiliary tree whose root and foot nodes are labeled with the same nonterminal X (see Figure 2). The operation adjunction together with the depth of the concerned trees which forms the context for the combination makes TAG a mildly context–sensitive formalism. These characteristics make it an adequate candidate for the description of natural language[12] (cf. [Joshi 85a], [Weir 88]).

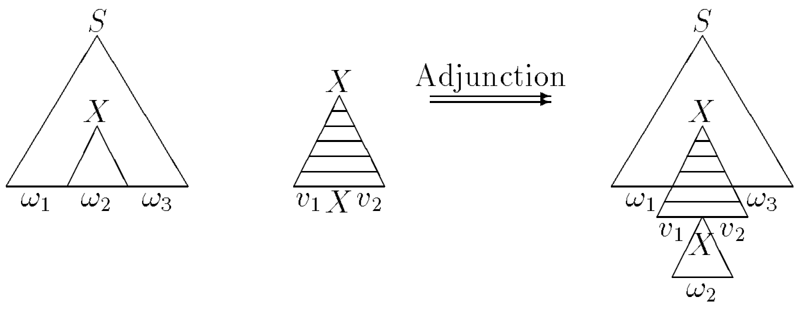


Figure 2: Elementary Trees and Adjunction in TAGs

The relevance of TAG to natural language processing results from the fact that its extended domain of locality allows for the specification of various related units within one elementary structure. For example, a predicate and all its arguments can be defined within a single elementary tree. TAGs can localize syntactic constraints such as agreement information or subcategorization constraints, thereby realizing adequately sized syntactic units.

Nodes of TAG trees have been associated with feature structures that lead to **TAGs with unification** (UTAGs, see [Kilger 92]).[13] Complex relations between syntactic structures can be encoded into compact units, which reduce redundancy in the grammar and increase the generative power of the formalism. For UTAGs, trees are interpreted as pre–combined context–free rules that easily allow for the adaption of PATR–II–style unification (cf. [Shieber et al. 83]). Each node is associated with a specification list that describes its features and those of its sons. During adjunction with unification, the feature structure of the adjunction node is transferred to the auxiliary tree. The feature structures of the root node and the foot node of the auxiliary tree are connected with the feature structures of the supertree and the subtree of the adjunction node. For a detailed definition and a discussion of different approaches to implementation see [Kilger 92].

Several extensions of the TAG formalism have been developed some of which are relevant to the task of incremental generation:

The combination operations of the representation formalism must support the kinds of expansion required for Design Principle 4 — upward expansion, downward expansion,

[12]Indeed, discussion is going on about the kind of extension of TAGs that is needed for the representation of various syntactic phenomena.

[13]Another approach to the association of TAGs with feature structures – FTAG – has been defined by [Vijay-Shanker & Joshi 88].

and insertion. In addition to adjunction we use the substitution operation as defined for **TAGs with Substitution** (cf. [Schabes et al. 88]). TAGs with Substitution define a third kind of leaf (called **substitution node**) that is associated with a nonterminal label and marked with a downward arrow in order to distinguish it from foot nodes. During a derivation, it has to be replaced by an initial tree the root of which is labeled with the same nonterminal.[14] For UTAGS, Substitution with unification is defined as unifying the feature structures of the substitution node and those of the root node of the substitution tree. The two combination operations adjunction and substitution support flexible combinations of partial syntactic structures by realizing all three kinds of expansions (see also [Finkler 90], [Schauder 92]).

Using a lexicalized grammar directly supports Design Principle 3 — 'lexical and syntactic guidance.' Each tree of **Lexicalized TAGs**[15] (cf. [Schabes et al. 88]) specifies a head element and its syntactic features. Substitution nodes are used to define details of the realization of its complements within separate trees. The head of a tree serves as anchor in the lexicon, making it possible to select the tree when its anchor is specified in the input to the syntactic generator.

The separation of hierarchical and positional structures as required by Design Principle 5 is reflected by another extension of the TAG formalism that has been especially designed for the demands of our incremental generation system: **TAGs with Context–Dependent Disjunctive Linearization Rules** (CDL–TAGs, cf. [Kilger 95]) define hierarchical relations within elementary domination structures and restrict possible orderings of their nodes by linearization rules.[16] The rules are headed by keys that refer to the syntactic context in which they may be used. The left part of Figure 3 illustrates a VP–node the subtree of which represents an English verbal phrase.

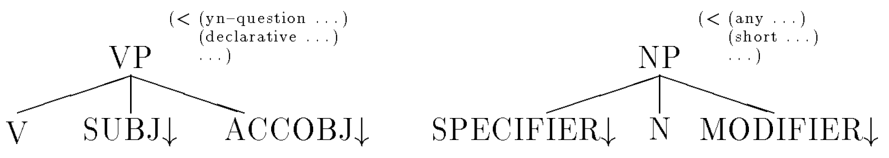


Figure 3: Examples for English Linearization Rules

Its linearization rules include statements about word order in declarative sentences, yes/no–questions, and others. The context of an object is inherited by means of unification. Other keys (like 'any' or 'short' at the NP–node in Figure 3) distinguish word order rules that differ with respect to their suitability for specific combinations of generation parameters.

[14]This operation is the same as the derivation operation for Context–Free Grammars. Therefore it does not extend the generative power of the TAG formalism.

[15]Lexicalized TAGs have also been used in the fields of natural language parsing and machine translation (see [Abeillé & Schabes 89], [Abeillé et al. 90]).

[16]This approach is based on the definition of LD/LP–TAGs by [Joshi 87b].

Each word order rule of a group is encoded as a regular expression that may contain numbers and symbols. The numbers refer to the sons of the node (i for the i–th son), the symbols to optional elements that can be incorporated into the phrase. The following expression might be associated with the VP–node of Figure 3:

```
(< ...
    (declarative ((advp)^{0|1} 2 1 ... (advp)* ... 3 ...)
                  ...)
    ...)
```

The linearization rule shows two alternatives of filling the first position of a verbal phrase in the declarative linearization context: either the subject (referred to by '2') or exactly one adverbial phrase (referred to by 'advp'). The symbols that denote adjuncts must distinguish the elements and be detailed enough to express all aspects that influence word positions. They have to be associated with the auxiliary trees that insert the modifying structure and may be inherited by the VP–node, e.g., through the feature structures that result from adjunction.

Because of the power of the linearization rules, the depth of TAG trees — that has been traditionally used to determine word order by guiding combination operations — is only used for a restricted set of applications. The linearization rules refer to modifier auxiliary trees (cf. [Schabes & Shieber 92]), a sequence of which can be adjoined into one and the same internal node. They only allow the introduction of compact sequences of nodes between two sons of the adjunction node by ignoring the depth of auxiliary trees, i.e., the path from root node to foot node.[17] To realize cross–serial dependencies, for example, we use predicative auxiliary trees in combination with the depth of initial trees.

In addition to the formal properties of the representation formalism, the *design of the grammar rules* strongly influences the behavior of the system. For VM–GEN we have designed the grammar in a way that is strongly influenced by criteria of performance, sometimes leaving aside criteria of linguistic theory. As a prerequisite for Design Principle 2 — 'independent processing of all independent tasks' — elementary trees should be adequately sized to cope with fragmental information of incrementally given input. For example, the syntactic structure of a noun phrase can be built and extended by some modifiers without knowing its syntactic function within the phrase, at least as long as the need for inflection can be postponed. Thus, the elementary trees that are chosen on the basis of a lexical element should not specify the syntactic function. Instead, their root nodes should be associated with phrasal names (e.g., 'NP').

Regarding the downward expansion of structures, substitution nodes of TAG trees should not be labeled with phrasal names since a verb often permits several alternatives

[17]Our usage of adjunction of modifier auxiliary trees is similar to the operation of 'furcation' as provided by Segment Grammar, cf. [De Smedt 90b].

for realizing its complements. The verb “to allow”, for example, accepts both a nominal phrase (“he allows [the party]”) and a subordinate clause (“he allows [that they have a party]”) as one of its complements. During incremental generation it might often be the case that a tree is to be chosen for a lexical item before its complements are realized. We have decided to associate substitution nodes with labels that reflect the syntactic function of the represented complement, e.g., ‘SUBJ’ or ‘ACCOBJ.’

The gap between substitution nodes labeled with syntactic roles and root nodes labeled with phrasal names is filled by a set of ‘intermediate structures’[18] representing the relation between head and argument. The small initial trees represent relations between head and complement, e.g., SUBJ–NP$\downarrow$. Relations between head and adjunct are handled by auxiliary trees. Root node and foot node are used to refer to the phrase of the modified head, a substitution node is used to introduce the modifying phrase. These trees do not define a lexical head but can be interpreted as anchored in a lexicon of semantic relations. The combination of elementary structures within our syntactic generator is described in Section 3.3.2, examples for intermediate structures are illustrated in Figure 6, Page 30.

Finally, redundancy in the grammar is reduced by encoding part of the syntactic knowledge within underspecified grammar trees and storing some word–specific information in a lexicon which is dynamically compiled into an instance of the currently chosen tree.

3.3 Consumption of Incremental Input

We use one basic example for the discussion of the generator components which illustrates the central features of our system. Figure 4 visualizes the input information for the target sentence “We'll meet in the small conference room at eleven o'clock.” Depending on the timing of input increments, the result might be a sentence with a less adequate word order — “We'll meet at eleven o'clock in the small conference room.” or even an utterance containing a self–repair.

3.3.1 The Interface

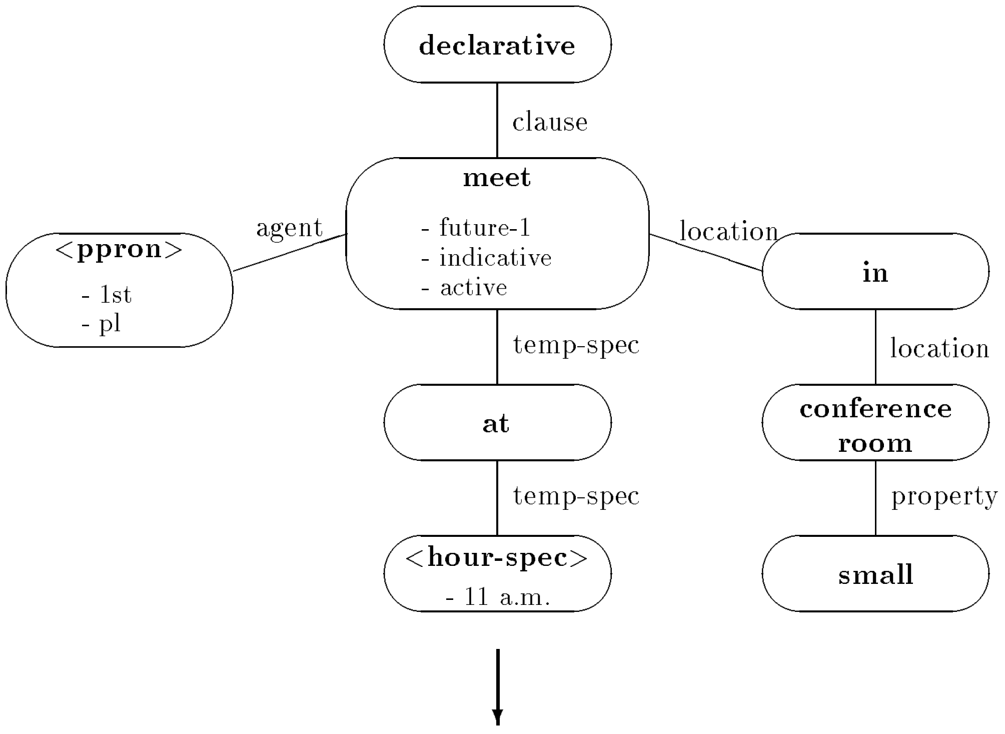
We defined the language of the input interface of VM–GEN in a way that ensures a high flexibility of specifying input information thus making the system suitable for a wide range of applications. The following properties of the input language fulfill several of the design principles as stated in Section 2.1 and contribute to the flexibility of the system:

- It is possible to specify input increments independent of each other and with a varying amount of information enclosed. This means that I(t) is defined as a sequence

[18] Intermediate structures are similar to the ‘intercalating segments’ used by [De Smedt 90a, p. 61].

of information packages A_1, A_2, ..., A_n yet to be composed.[19] Δ_I can easily be identified as the distinctive A_i that make the time dependent modification of the input explicit. This gives the calling component the freedom to hand over known data (from underspecified elements to groups of input increments) at any time.

- There is no need for a predefined order of the input increments when encoding the input. Our interface offers the flexibility of handing over input increments in any possible order and with arbitrarily long pauses between them. VM–GEN does not interpret the order of input increments as encoding information about, e.g., topic and focus.[20] Since the system is to be usable in situations where the input can be dynamically extended or changed, our interface allows for the modifications of previously given input increments, i.e., change of features or deletions of elements, thereby realizing full incrementality in the sense of [Wirén 92].



"We'll meet [in the small conference room] [at eleven o'clock]."
"We'll meet [at eleven o'clock] [in the small conference room]."

Figure 4: Example for VM–GEN

Let's take a closer look at the aspects of the input interface of VM–GEN mentioned above. Packages of information for content words and semantic relations are realized

[19]The notation is used according to our definitions at Page 5.
[20]Contrarily, [De Smedt 90a] uses the input to reflect the degree of conceptual accessibility of parts of the message.

as suitable increments of the input to VM–GEN. They correspond to the nodes and edges of the input network as shown in Figure 4 and are used as starting points for the syntactic generation processes. Their specification starts either with the keys 'entity' or 'relation' (cf. Figure 5). The packages are labeled with unique reference names that carry no meaning but can be mnemonic as in our example. It is possible to hand over underspecified input and to complete it later on, using the same reference name each time. The specifications of entities and relations may be combined to larger input increments by grouping them together in a list. Those input increments provide a larger context for the generator that can be used to avoid revisions. Increments may be subsequently changed by defining differing features in a package with the key 'changed-entity.' Finally, the interface is able to cope with any order of input increments by storing given increments and managing the addition of missing information using the unique reference names.

A subset of the input increments for our basic example is given in Figure 5. For reasons of clarity, the input increments in the example are sorted according to the dominance relations between the respective lemmas. They can be given in another order, as well. For each specification of an utterance, a technical category ('UTT-PAR-1' in the figure) has to be specified as the uppermost element of the hierarchy, providing some global settings for the utterance. It is related to the phrase that constitutes the utterance, e.g., a verbal phrase for a sentence, or a nominal phrase for a caption to be verbalized. At first sight, the input to the syntactic generator seems to encode a structure which already encodes too many surface–related decisions. However VM–GEN is designed to interact with the microplanning component in cases of input inconsistencies identified at the syntactic level of processing.

```
;; parameter setting for the utterance
(entity UTT-PAR-1 ((intention declarative)))
;; verb as linguistic head of the utterance
(entity VP-1 ((cat v) (head "meet") (tense future-1) (voice active)
     (mood indicative)))
(relation UTT-PROP-1 ((func clause) (regent UTT-PAR-1) (dep VP-1)))
;; noun phrase as agent
(entity NP-1 ((cat ppron) (num pl) (pers 1)))
(relation REL-VP-1 ((func agent) (regent VP-1) (dep NP-1)))
...
```

Figure 5: Fragment of the Input Specification for VM-GEN

Each input increment that is handed over to VM–GEN triggers off the creation of one or more objects in a distributed parallel system (realization of Design Principle 1 — 'fine–grained parallelism'). For each input increment of the type 'entity' there is an object responsible for the verbalization of the included content word. An example for the creation of additional objects is the specification 'definite' for a noun phrase that leads to the creation of a determiner object. A 'relation'–increment in the input triggers the creation of an object which is responsible for finding a syntactic relation for realizing a semantic role. These objects serve as mediators between the 'entity'–originated objects.

All objects are organized in a hierarchy that is also determined by dependency relations specified in the input.

In the Interface, the first task of an object is to choose an adequate syntactic structure considering the specified information. Referring to the lexical guidance aspect of Design Principle 3 — 'lexical and syntactic guidance' — objects which manage the verbalization of content words select an initial tree from the lexicalized TAG, using given syntactic features and the current parameter settings to choose between alternatives. The selection is performed by traversing a systemic network distinguishing various values of different attributes such as 'voice' or 'focus.' The outcome of the initial tree selection yields the locally best valued elementary tree for the input increment, thereby fulfilling Design Principle 6 — 'best–first search.'

Semantic relations specified in the input language are handled by separate objects. They are translated into small structures realizing the syntactic relation between heads and arguments (see Section 3.2). In order to do so, the objects inspect a larger set of possible intermediate structures on the basis of the given semantic relation and filter out inadequate trees by communication with the two objects that manage the related lemmas. As result of filtering, one most favored structure remains as in the case of objects realizing content words.

After the initialization phase, the objects change their state by entering the Phrase Formulator.

3.3.2 The Phrase Formulator

Objects in the Phrase Formulator[21] have to solve the global task of constructing the hierarchical representation of the current utterance while preserving the local management of their structures. The latter requirement is the presupposition for the effective use of fine–grained parallelism. The structures that are locally managed by single objects shouldn't be collapsed into larger shared structures by combination operations since this kind of processing would depart from parallel computation.

Each object maintains a local context containing information about relevant partners for communication in order to facilitate cooperation. As mentioned in the previous section, this information is extracted from the input specification of relations. Objects managing the transfer from semantic to syntactic relations communicate with both related objects, thereby extending the local contexts of those objects.

Depending on the type of tree that is operated on by an object in the distributed parallel model, there are different methods of combining syntactic structures. Our approach favors a prescribed direction of activity during combination operations which prompts the

[21]The design of the Phrase Formulator has been influenced by the experience that we gained from the incremental and parallel handling of structure combination in POPEL–HOW (see [Neumann & Finkler 90]).

dominated object to take the initiative. This helps in avoiding technical synchronization problems originating from mutual initializations of the combination operations. An object with an initial tree triggers a substitution operation, an object with an auxiliary tree tries to adjoin its structure into the structures of the dominating object. The feature structures that are associated with the nodes of the trees to be combined must not be destructively unified during substitution or adjunction in order to preserve local structures.

We defined the **distributed adjunction** of an auxiliary tree with root node X_R and foot node X_F of object O_1 at an internal node X of object O_2 as follows:

- O_1 sends copies of the feature structures associated with the root node X_R and the foot node X_F to object O_2.
- O_2 tests the compatibility of the feature structures at X_R, X_F and X. O_2 then splits X into a quasi–node[22] with X_T being the top–part and X_B the bottom part and stores a reference to the partner object O_1 at the so–called interface nodes X_T and X_B. Finally, O_2 sends back copies of the feature structures associated with the quasi–node to the partner object O_1.
- O_1 stores a reference to the object O_2 at its interface nodes X_R and X_F.

The second combination operation, **distributed substitution**, is defined in a similar way. It doesn't necessitate the splitting of a node and merely makes use of two interface nodes, i.e., the root node of the substitution tree at object O_1 and the substitution node at object O_2.[23] Figure 6 outlines some of the communications that enable the exchange of copies of feature structures during distributed substitution and adjunction. Four rectangular boxes represent the cooperating objects. Each box depicts the selected tree of the respective object. The nodes of the trees are not yet linearized, i.e. their order is not related to the order of output increments. Double arrows between objects 'NP-2' and 'REL-NP-3' connect the interface nodes during the adjoining operation that is necessary to incorporate the modifier "small" into the NP for "conference room". The double arrow between 'REL-PP-1' and 'NP-2' indicates the communications during the substitution operation to integrate the whole structure into a prepositional phrase.

An important aspect of the combination operations is that the effects of distributed substitution and adjunction are not limited to the two objects which are directly involved. Since each substitution or adjunction can modify the feature structures of the concerned trees, it has to be checked whether the combination operation leads to changes in the feature structures of the interface nodes. Those changes have to be propagated to the respective objects. Thus, a distributed combination of structures influences all objects that are connected with the basic pair by path equations within their feature structures.

[22]The notion of quasi–nodes and quasi–trees has been introduced by [Vijay-Shanker 92]. It simplifies the definition of the combination of TAGs with a feature structure based mechanism.

[23]See [Kilger 94] for details about both combination operations.

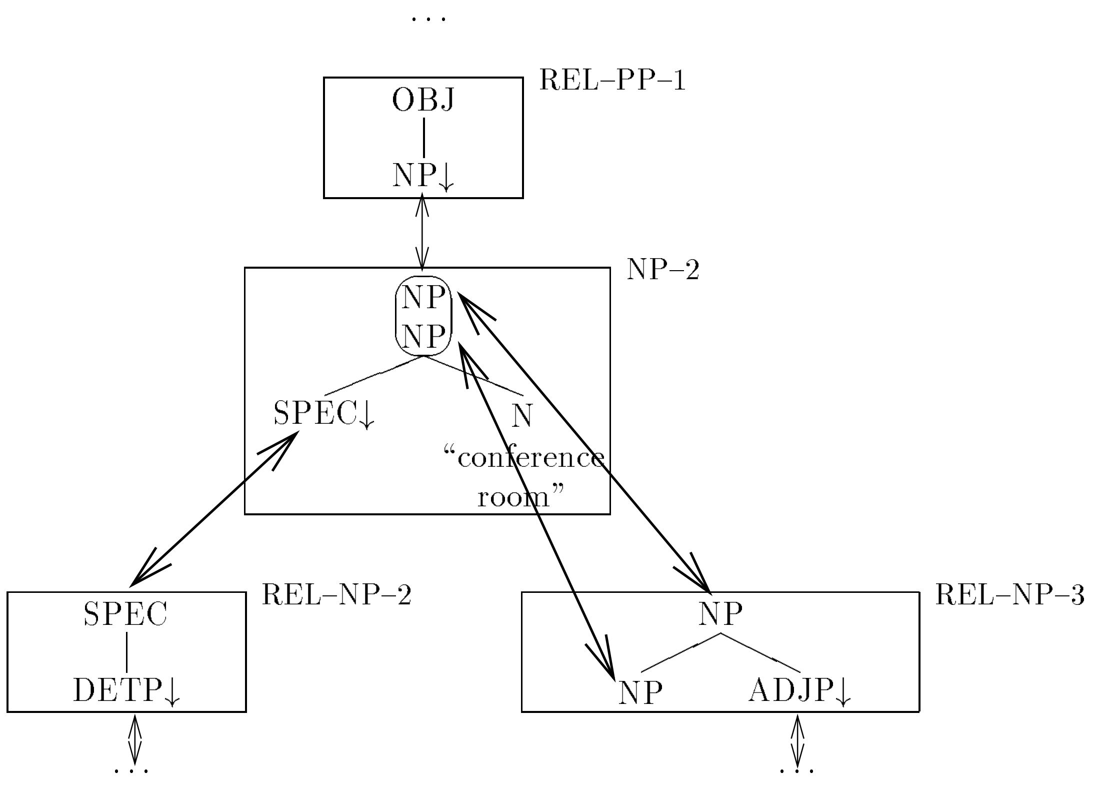


Figure 6: Communications during Distributed Substitution and Adjunction

The efficiency of this kind of realization depends highly on the design of the grammar since the flow of information stops as soon as the rule being examined specifies no further agreement of feature structures.

Currently, the presupposition for an object to change its state again and enter the Linearization Component is its structural combination with its dominating object. After this attachment the object is provided with contextual information that is needed for linearization and inflection.

3.3.3 Linearization Component

For an incremental syntactic generator the design of the linearization component is a central task. Language-specific word order rules restrict the possibilities of lining up the parts of an utterance under construction. They often forbid the simple addition of results of computation for input increments at the right of the current utterance (right concatenation), which is an important reason for self–corrections and other phenomena of spontaneous speech. This observation made us interleave the linearization task of *computing an adequate order of groups of words* with the controlling task of *deciding when to hand over successive parts to the articulator*. This interleaving is illustrated in the architecture (see Figure 1) by forward and backward arrows between the Linearization Component and the Output Interface.

Contrary to the approach of [Neumann & Finkler 90], we do not create a new process and local structures to handle the output of the hierarchical level at the linearization level. Instead, the objects at the hierarchical level merely change their state and use the linearization rules associated with the hierarchical structures to solve their new tasks. The main motivations for this *integrated* approach are:

- Using a separate representation, i.e., local structures, during linearization is partially redundant, since a lot of information from the hierarchical structures is also used as the basis for linearization.
- While a structure is linearized, it may be expanded simultaneously at the hierarchical level because of incrementally given fragments in the input to the generator. The state of linearization and output might be used to find out whether the new elements can be integrated into the utterance without risking overt repair (see [Finkler & Schauder 92]). If this is not possible, the system could trigger the choice of another syntactic tree or even request the microplanner to withdraw decisions (see [Wahlster et al. 93] for an example).

As mentioned at the end of the previous section, an object starts working at the linearization level after having integrated its syntactic structure into the one managed by its dominating object. The attachment to a more global structure supplies the object with contextual information that is advantageously used for selecting the appropriate linearization rules.[24] Especially, the linearization of verbal phrases depends on the type of the sentence, e.g., imperative or declarative. This information is inherited from the dominating object.

The linearization process is carried out by the objects of VM–GEN in a distributed way. When an object starts to linearize its local structure it first chooses the group of linearization rules that is suitable with respect to the inherited linearization context and the current parameter settings (benefiting from Context–Dependent Linearization Rules, [Kilger 95]). It then traverses its local structure, starting with the root node and interpreting the word order rules that are associated with internal nodes to guide the traversal. Figure 7 illustrates the distributed linearization of the objects managing the verb "to meet", the personal pronoun realizing, e.g., the Speaker and the Hearer during a face–to–face dialogue, and the prepositional phrase "at eleven o'clock".

If there are several candidates for the next position, the decision may be influenced by additional input information or by the dynamic behavior of the objects competing

[24]Another reason for the attachment is that there are cases where the elements of a phrase are not locally ordered into a continuous block but mixed with elements of the dominating phrase, e.g., for scrambling phenomena (see [Becker et al. 91]). Our system cannot handle these cases yet.

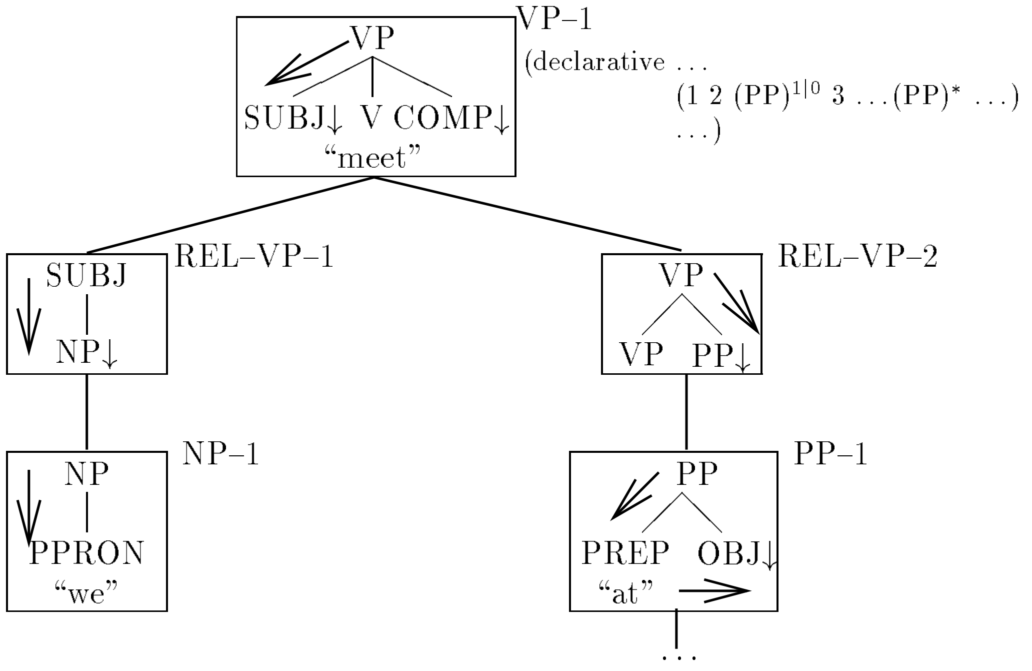


Figure 7: Local Traversal of Dominance Structures in the Linearization Component

for continuing the utterance. A specific 'topic'–attribute can be associated with an input increment and – if possible – leads to the topicalization of the element referred to. Furthermore, the interpretation of the regular expressions of the linerization rules is influenced by the fact whether the element referred to has already entered the Linearization Level. Figure 7 shows a linearization rule associated with the VP–node that allows either an optional prepositional phrase ($(\mathrm{PP})^{1|0}$) or the complement of the verb "to meet" ('3') to fill the third position in the verbal phrase. If the complement has not yet been specified, there is the chance for a PP, e.g., the temporal specification "at eleven o'clock", to fill the slot. In this way, the utterance can be continued rapidly with a slightly worse style. That illustrates the use of our grammar for spoken language allowing flexible formulation.

Each terminal node that is reached during the linearization traversal is inflected. When an interface node is reached, the traversal is interrupted and traversal in the connected object is triggered. The synchronization of utterances by the single objects and its interleaving with linearization is explained in Section 3.4.3.

3.4 Incremental Output Production

Producing incremental output imposes demands on all components of a generation system. The output prefix has to be handled as an additional constraint during computation allowing the system to realize right–concatenation, hidden, or overt repair in an adequate way.

3.4.1 Phrase Formulator

Operations within the Phrase Formulator may lead to conflicts with the output prefix if the concerned objects have already been engaged in the Output Interface. Substitution is less problematic since the substitution nodes guarantee for the consideration of the respective elements during linearization. Adjunction may influence the process of linearization in two ways. First of all, it leads to the integration of a new element into the syntactic structure that has to be adequately inserted into the sequence of terminals. Furthermore, it may change the feature structures of several connected TAG–trees which may be used for tests during linearization. Currently, adjunction in an object in the Linearization Component triggers a test and – if necessary – a recomputation of the local word order.

When an object receives a call for adjunction, it might already have entered the Output Interface and handed over some parts of its local phrase to the articulator. We realized a simple strategy for revision that copes with these cases by repeating the whole prefix of the current utterance. Figure 8 shows a snapshot of our example illustrating a possible sequence of processing steps when building the noun phrase "the small conference room". In this example, the modifying adjective is given too late to be included into the noun phrase without a repair. The left part of the figure shows a distributed adjunction. The object managing the 'property'–relation during Phrase Formulator processing contacts its dominating object. In this case, it has already entered the Output Interface. The system therefore triggers a repair (which is discussed below).

3.4.2 Linearization Component

Incremental output production requires decision making on the basis of incomplete information at the positional level. The system has to be able to skip references to optional elements within the linearization rules even if there is no guarantee that further input would correspond with that decision. For the worst case, repair facilities must be provided to 'backtrack' to an earlier state (see Section 3.4.4). In Figure 8 the NP–node of object NP–2 is associated with a linearization rule of the kind

$$(1\ (\mathrm{ADJP})^{*}\ 2\ \ldots)$$

In order to continue the utterance rapidly, NP–2 has skipped the (ADJP)*–entry and has uttered the noun in the Output Interface, assuming that no modifying adjective will be given later on.

As a second consequence of incremental output production, the output prefix has to be considered when choosing between linearization alternatives. Alternatives that contradict with the prefix have to be ruled out.

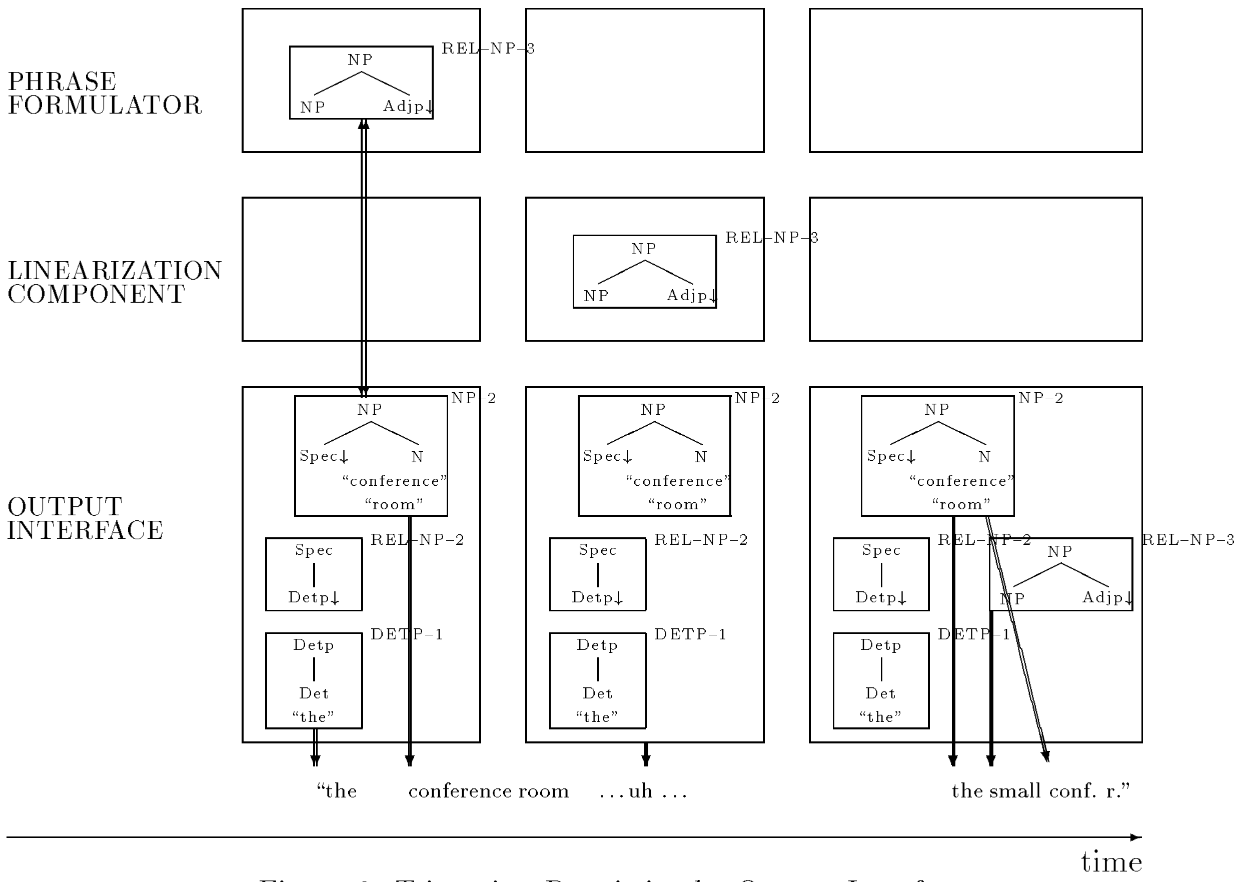


Figure 8: Triggering Repair in the Output Interface

3.4.3 Output Interface

For VM–GEN, we studied incremental output production particularly with respect to intra–sentential effects. Design Principle 7 — separate processing of hierarchical and positional rules — makes it possible to minimize the size of output increments since the single terminals of the hierarchical structures can be linearized even if the hierarchical structures are not yet complete.

The 'best case' of incremental output production, i.e., right–concatenation, is realized in the distributed parallel system by synchronizing the output activities of the objects by means of two types of messages. Each object locally decides, whether to provide an element of the utterance for articulation. It sends a message to its dominating object, declaring its *readiness for output* and expecting to get the allowance to feed the articulator. At all times, there is exactly one object of the distributed parallel system that is responsible for guiding the output activities. Initially, the highest object in the hierarchy has the *license for output*. In the course of time, this license is handed over from one object to the other. The object that has the license for output, locally traverses its structure (see Section 3.3.3). When an interface node is reached, the object hands over the license for output to the respective partner object. The local traversal is interrupted until the partner object has finished its contribution to the utterance and has returned the license.

Figure 9 shows some objects in our example which are engaged in linearization and output production. The adjective here has been given in time to be introduced into the noun phrase. The arrows inside of the objects' boxes illustrate the local traversal of syntactic structures. Arrows pointing from interface nodes to distinct objects depict the transfer of the license for output. The resulting fragment of the utterance consists of the sequence: determiner, adjective, and noun "the small conference room".

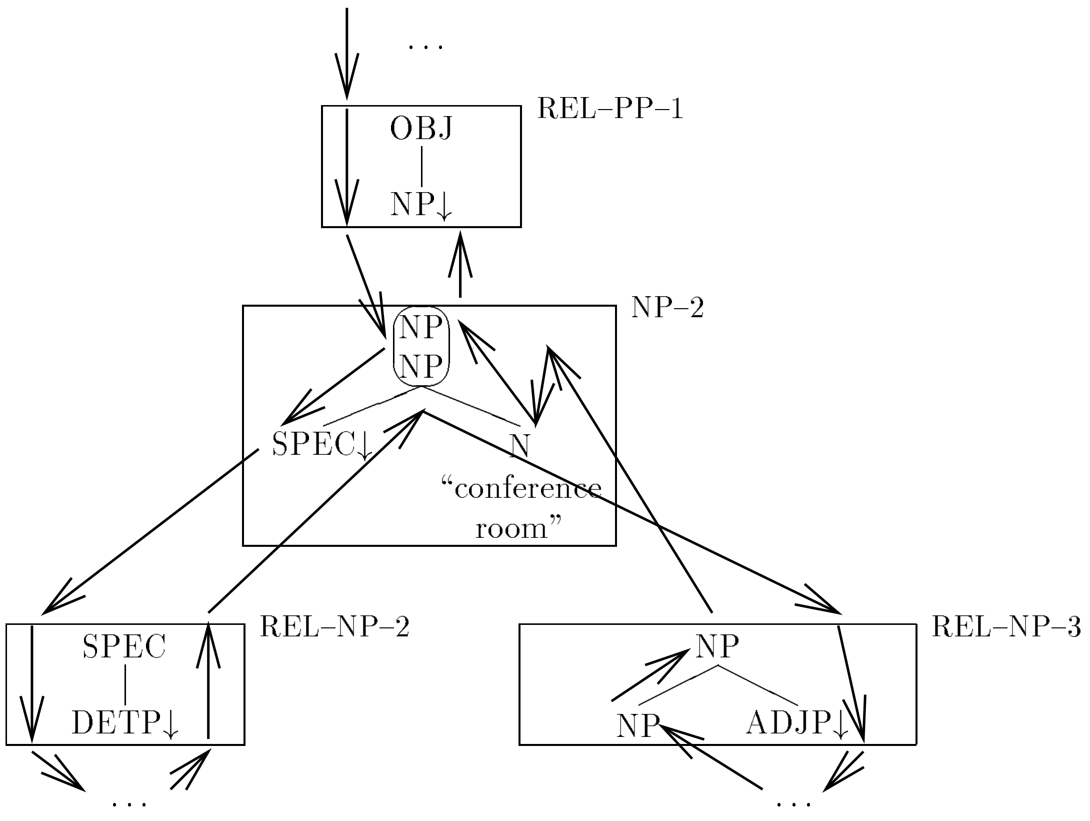


Figure 9: Extract from the Example: Distributed Linearization and Output Production

In the distributed parallel system VM–GEN, the global results of generation — the syntactic structure and the complete utterance — are not stored in a compact way but remain distributed among the single objects. Each object keeps its local history by storing the chosen syntactic structure, the results of exchanging features in the Phrase Formulator, the local path of linearization and the locally computed parts of the utterance. This information can be used for realizing repair as described below.

3.4.4 Repair

Currently, VM–GEN only partially fulfills Design Principle 8 — 'adequate repair strategies.' It realizes one simple form of overt repair by repeating parts of the articulated sentence. New input increments trigger the creation of new objects which try to combine their local syntactic structures with the already existing syntactic structure. These combination operations in the Phrase Formulator may change the basis for decisions in the Linearization Component, either by modifying the inherited feature structures that are used for linearization tests or by introducing optional elements that were skipped previously. Therefore, each object in the Linearization Component or the Output Interface that receives a call for a combination operation afterwards checks its linearization state for consistency. Whenever a contradiction is found, another kind of message is used to synchronize repair. The object sends the message 'output stop' to its dominating object, and in upward direction through the object hierarchy until an object is found that currently holds the 'license for output' or just handed it over to an object on another path in the hierarchy. In the second case, the object waits until the license is returned. It then sends a repair marker (e.g., 'uh' or 'sorry') to the articulator, reinitializes its local state of linearization and makes all its descendants do the same. Then linearization is started again, leading to the repetition of all concerned terminals including the newly inserted ones.

In the middle of Figure 8 the Output Interface has produced the repair marker "uh" while the conflicting object already has entered the Linearization Level. Object NP–2 reinitializes the utterance of the noun phrase, using local information about the output produced by REL–NP–2 to repeat the determiner "the", and handing over the license for output to REL–NP–3 to include the adjective at the correct position.

3.4.5 When–to–Say

In order to realize a threshold for output production, as discussed in Section 2.2.3, VM–GEN includes a counter that keeps track of the duration of inter– and intra–sentential pauses. Whenever this counter exceeds a predefined time–limit for delay intervals, time–pressure is interpreted as being rather high. VM–GEN currently reacts by triggering a default–handler thereby realizing Design Principle 10.

The default handler uses a set of default descriptions that can be matched with the situation at hand to find suitable heuristics for reaction as its knowledge base. The default body consists of the specification of additional input increments for VM–GEN, thereby allowing the system to handle default–caused and input–licensed values homogeneously. Take for example an input state where information about the relation between a verb and a noun is missing. Under certain conditions, it is likely to assume an 'agent'–relation that is specified as default–licensed input. Contradictions with input increments specified later on can be handled by VM–GEN since we designed the input interface to cope with modifications and deletions of input increments (see Section 3.3.1). For more details on default handling in VM–GEN, see [Harbusch et al. 94].

3.5 Effects of Incremental Processing on the Output

The order and the timing of input increments for VM–GEN are important determinants for the output of the generator. We have developed the graphical interface VIIO (**V**isualization of **I**ncremental **I**nput and **O**utput) that illustrates the relations of input increments and output increments with respect to progress in time (see, e.g., Figure 10). The x–axis shows progress in time. Points in graph at the 'INPUT' level mark the creation time of objects for input increments. Points at the 'OUTPUT' level indicate the points of time when the respective parts of the utterance are articulated. The points representing the insertion and the articulation of an element are connected by an arrow. Crossed arrows illustrate the interleaving of computations in the generator. Objects which manage a syntactic structure without any terminal node are merely represented as dots at the 'INPUT' level. The objects are numbered in the order of their creation, the numbers are associated with the arrowheads to simplify orientation. In the upper right corner of the screen, a legend shows the association of numbers to objects.

We present three examples of input variations for the sentence discussed above that illustrate facets of I/O–behavior of VM–GEN. If the input increments are given as VP–1, NP–1, PP–1, NP–2, ADJP-1, PP–2, HOUR–1, they will normally be uttered as subject, verb, object-loc, object-temp: "[We'll] [meet] [in the small conference room] [at eleven o'clock]" (see Figure 10).

When PP–1 and PP–2 are exchanged in the input and if the next position in the verbal phrase is to be filled before PP–1 is available at the Positional Level, then PP–2 is uttered irrespective of the slightly worse style: "[We'll] [meet] [at eleven o'clock] [in the small conference room]" (see Figure 11).

If there is a long pause between the input of NP–2 and its modifier ADJP-1, it may happen that the noun is uttered before object NP–2 knows about the existence of the modifier. Now the adjective is introduced into the noun phrase by recomputing the linearization and output production of NP–2: "[We'll] [meet] [in the conference room] ... [we'll] [meet] [in the small conference room] [at eleven o'clock]" (see Figure 12).

4 Conclusion and Future Work

The automatic generation of spoken language imposes strong real–time constraints on the system. The incremental processing paradigm is a promising approach to enable a generator to react to changes in the input as fast as possible and to time the output in

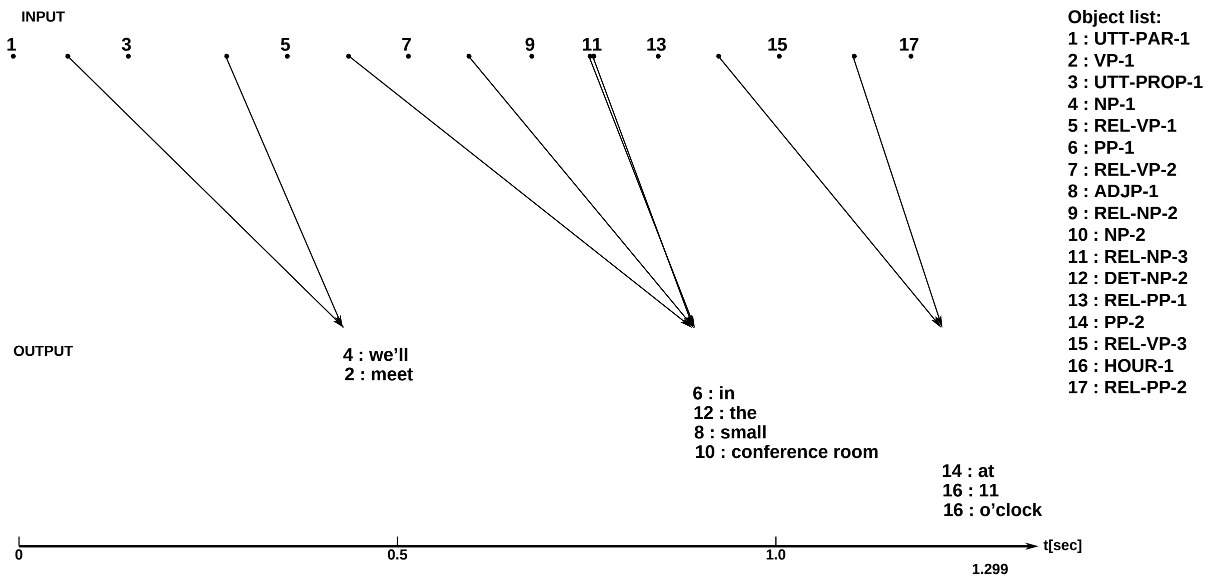


Figure 10: Screen Snapshot of VIIO: PP–1 before PP–2

a way that is adequate for the human hearer. An incremental generator must cope with contradictions between the assumptions it makes to continue its output and the input it receives later on. It must be able to realize strategies for hidden and overt self–corrections that are acceptable for human dialogue partners.

VM–GEN is a syntactic generator that exploits the notion of incremental processing to allow for flexible real–time reactions on an evolving input. Input increments can be handed over in any order and with arbitrarily long pauses between them. They trigger their immediate processing, leading to the construction of new parts of the syntactic structure and to the fast production of a well–formed prefix of the utterance.

VM–GEN uses a distributed parallel model of active objects that co–operate in order to realize the given input increments in one utterance. Parallelism not only increases efficiency but also supports the processing of independent tasks during syntactic generation.

The formalism Tree Adjoining Grammar is well suited for the representation of syntactic structures because of the extended domain of locality of its rules. Its combination operations adjunction and substitution allow for the flexible expansion of the syntactic tree which is the presupposition for the incremental introduction of new elements.

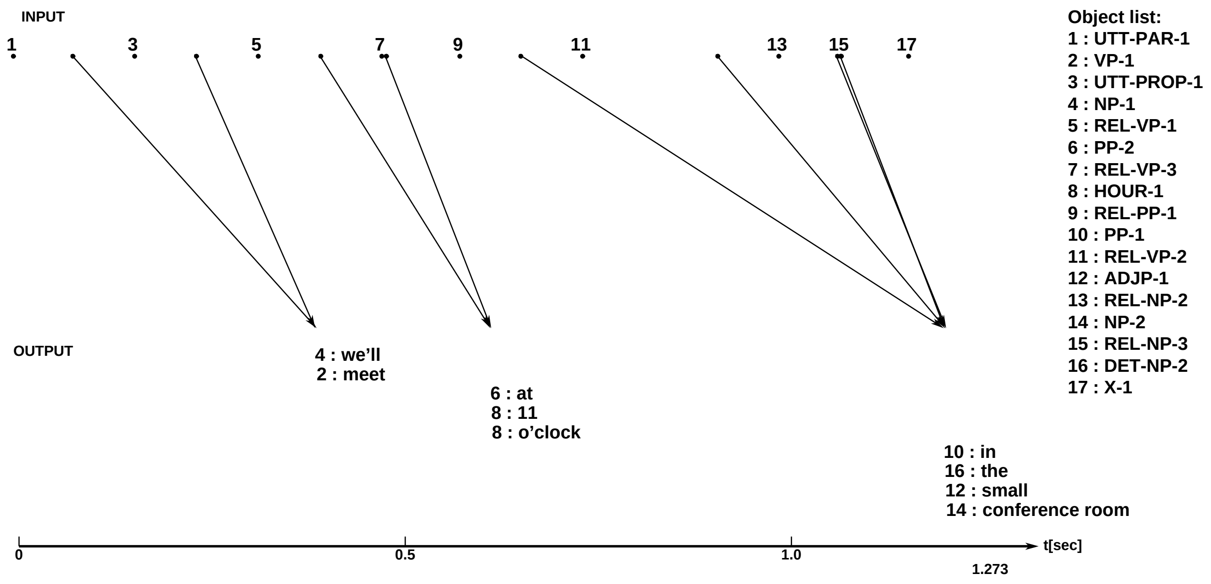


Figure 11: Screen Snapshot of VIIO: PP–2 before PP–1

The experience we gained from the design and implementation of VM–GEN shows that the benefits of incremental processing surpass its costs. It provides a good starting point for developing a system with an output interface that approximates human language performance. The applicability of the core generator has been substantiated by using it in various domains, namely a system for generating multimodal documents (WIP, [Wahlster et al. 92]), a spoken–language dialogue system for train schedule inquiries (EFFENDI, [Poller & Heisterkamp 95]), a dialogue system managing negotiations in the used car sales domain (PRACMA, [Jameson et al. 94]), the natural language description of simultaneously interpreted real world image sequences (VITRA, [Herzog & Wazinski 94]), a natural language interface to an autonomous mobile robot (KANTRA, [Längle et al. 95]), and a system for explaining machine–found proofs (PROVERB, [Huang 94]). Nevertheless, we have identified several shortcomings of the system, that are worth being examined in future research.

Since the input for VM–GEN is expected to contain syntactic specifications, the syntactic generator has to be combined with a sophisticated incremental component for microplanning and word choice. A suitable system that can cope with these tasks is VM–IMP (**I**ncremental **M**icro**P**lanning) developed as part of the VERBMOBIL project. We can therefore construct a bidirectional interface to VM–GEN and realize incrementality and

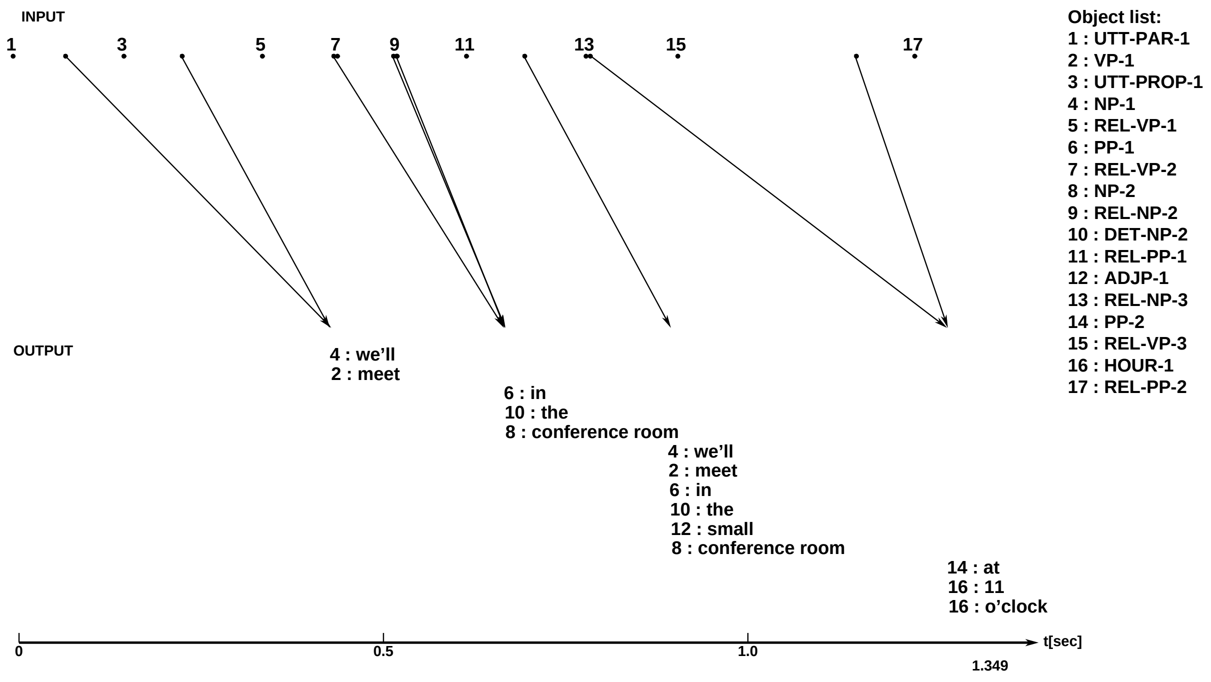


Figure 12: Screen Snapshot of VIIO: Overt Repair for delayed ADJP–1

interactivity.

The task of When–to–Say has long been neglected in natural language generation research, due to the fact that it has always been interpreted as a functional part of a speech synthesis component. Nevertheless, the borderline between natural language generation and synthesis is gradually becoming vague when realizing the concept–to–speech approach. A lot of work remains to be done in order to define an interleaved processing mode for generation and synthesis that allows for an adequate timing of output production.

Referring to the control of the incremental generation process – as suggested in Design Principle 11, 'production–oriented monitoring and control' – we currently develop a suitable system PERFECTION (**PERF**orming Self–Corr**ECTION**s) that bases on VM–GEN. It features a central component for monitoring and supervising activities distributed over the generation component and allows for the realization of more sophisticated repair scenarios. We apply and evaluate the usage of different types of reason maintenance systems for the identification of affected structures when decisions have to be withdrawn to perform self–corrections.

One of the objectives of the VERBMOBIL generation group is to utilize a grammar for English that is encoded using the HPSG framework. A compiler from HPSG to TAG is currently realized by [Kasper et al. 95]. They expect both formalisms to benefit from the compilation. HPSG, which has an elaborated principle–based theory could be processed more efficiently and TAG providing the means to localize dependencies in elementary trees could represent structures encoded in a linguistic theory.

References

[Abb et al. 93] B. **Abb**, C. **Günther**, M. **Herweg**, K. **Lebeth**, C. **Maienborn**, and A. **Schopp**. *Incremental Syntactic and Phonologic Encoding – An Outline of the SYNPHONICS–Formulator.* In: Fourth European Workshop on Natural Language Generation, Pisa, Italy, 1993.

[Abeillé & Schabes 89] A. **Abeillé** and Y. **Schabes**. *Parsing Idioms in Lexicalized TAGs.* In: Fourth Conference of the European Chapter of the ACL, Manchester, UK, 1989.

[Abeillé et al. 90] A. **Abeillé**, Y. **Schabes**, and A. K. **Joshi**. *Using Lexicalized TAGs for Machine Translation.* In: 13th International Conference on Computational Linguistics, pp. 1–6 (3), Helsinki, Finland, 1990.

[Amtrup 95] J. **Amtrup**. *Perspectives for Incremental MT with Charts.* In: Machine Translation and Translation Theory, Natural Language Processing. Mouton de Gruyter, 1995. to appear.

[Appelt 85] D. **Appelt**. *Planning English Sentences.* Cambridge: Cambridge University Press, 1985.

[Becker et al. 91] T. **Becker**, A. K. **Joshi**, and O. **Rambow**. *Long–Distance Scrambling and Tree Adjoining Grammars.* In: Fifth Conference of the European Chapter of the ACL, pp. 21–26, Berlin, Germany, April 1991.

[Bond & Gasser 88] A. H. **Bond** and L. **Gasser** (eds.). *Readings in Distributed Artificial Intelligence.* San Mateo, California: Morgan Kaufmann Publishers, Inc., 1988.

[Booth 89] P. **Booth**. *An Introduction to Human–Computer Interaction.* Hillsdale, USA: Lawrence Erlbaum Associates, 1989.

[Chomsky 81] N. **Chomsky**. *Lectures on Government and Binding.* Dordrecht: Foris, 1981.

[De Smedt & Kempen 87] K. **De Smedt** and G. **Kempen**. *Incremental Sentence Production, Self–Correction and Coordination.* In: G. Kempen (ed.), Natural Language Generation: New Results in Artificial Intelligence, Psychology and Linguistics, NATO ASI Series E 135, pp. 365–376. Dordrecht: Martinus Nijhoff, 1987.

[De Smedt 90a] K. **De Smedt**. *Incremental Sentence Generation: a Computer Model of Grammatical Encoding.* PhD thesis, Nijmegen Institute for Cognition Research and Information Technology, Nijmegen, 1990. NICI TR No 90–01.

[De Smedt 90b] K. **De Smedt**. *IPF: An Incremental Parallel Formulator.* In: R. Dale, C. Mellish, and M. Zock (eds.), Current Research in Natural Language Generation. Academic Press, 1990.

[De Smedt 91] K. **De Smedt**. *Revisions during generation using non–destructive unification.* In: Abstracts of the Third European Workshop on Natural Language Generation, 13-15 March, pp. 53–70, Judenstein/Innsbruck, 1991.

[Eikmeyer et al. 91] H.J. **Eikmeyer**, W. **Kindt**, U. **Laubenstein**, S. **Lisken**, Th. **Polzin**, H. **Rieser**, and U. **Schade**. *Kohärenzkonstitution im gesprochenen Deutsch.* In: G. Rickheit (ed.), Kohärenzprozesse, pp. 59–136. Westdeutscher Verlag, 1991.

[Finkler & Neumann 89] W. **Finkler** and G. **Neumann**. *POPEL-HOW – A Distributed Parallel Model for Incremental Natural Language Production with Feedback.* In: 11th International Joint Conference on Artificial Intelligence, pp. 1518–1523, Detroit, MI, August 1989.

[Finkler & Schauder 92] W. **Finkler** and A. **Schauder**. *Effects of Incremental Output on Incremental Natural Language Generation.* In: B. Neumann (ed.), 10th European Conference on Artificial Intelligence, pp. 505–507, Vienna, Austria, August 1992.

[Finkler 89] W. **Finkler**. *POPEL–HOW: Eine Komponente zur parallelen, inkrementellen Generierung natürlichsprachlicher Sätze aus konzeptuellen Einheiten, Teil 1.* Master's thesis, Department of Computer Science, University of the Saarland, Saarbrücken, 1989.

[Finkler 90] W. **Finkler**. *Incremental Natural Language Generation with TAGs in the WIP–Project.* In: W. Wahlster and K. Harbusch (eds.), First International Workshop on Tree Adjoining Grammars, pp. 64–70, Dagstuhl, Germany, 1990. IBFI.

[Finkler 95] W. **Finkler**. *Automatische Selbstkorrektur bei der inkrementellen Generierung gesprochener Sprache: Ein empirisch–simulativer Ansatz unter Verwendung von Truth Maintenance Systemen.* Dissertation, in preparation, 1995.

[Garrett 80] M.F. **Garrett**. *Levels of Processing in Sentence Production.* In: B. Butterworth (ed.), Language Production. Volume 1: Speech and Talk, pp. 177–220. London: Academic Press, 1980.

[Görz 92] G. **Görz**. *Kognitiv orientierte Architekturen für die Sprachverarbeitung.* Technical Report ASL–TR–92–39, Universität Erlangen–Nürnberg, Institut für mathematische Maschinen und Datenverarbeitung (IMMD), Erlangen, Germany, February 1992.

[Harbusch et al. 94] K. **Harbusch**, G. **Kikui**, and A. **Kilger**. *Default Handling in Incremental Generation.* In: 15th International Conference on Computational Linguistics, pp. 356–362, Kyoto, Japan, August 1994.

[Herzog & Wazinski 94] G. **Herzog** and P. **Wazinski**. *VIsual TRAnslator: Linking Perceptions and Natural Language Descriptions.* Artifical Intelligence Review, 8(2):175 – 187, 1994.

[Herzog et al. 89] G. **Herzog**, C.-K. **Sung**, E. **André**, W. **Enkelmann**, H.-H. **Nagel**, T. **Rist**, W. **Wahlster**, and G. **Zimmermann**. *Incremental Natural Language Description of Dynamic Imagery.* In: W. Brauer and C. Freksa (eds.), Wissensbasierte Systeme, pp. 153–162, Berlin, 1989.

[Hovy 88] E.H. **Hovy**. *Generating Natural Language Under Pragmatic Constraints.* Hillsdale, NJ: Lawrence Erlbaum Associates, 1988.

[Huang 94] X. **Huang**. *Planning Argumentative Texts.* In: 15th International Conference on Computational Linguistics, pp. 329–333, Kyoto, Japan, 1994.

[Jameson et al. 94] A. **Jameson**, B. **Kipper**, A. **Ndiaye**, R. **Schäfer**, J. **Simons**, T. **Weis**, and D. **Zimmermann**. *Cooperating to Be Noncooperative: The Dialog System PRACMA.* In: B. Nebel and L. Dreschler-Fischer (eds.), 18^{th} Annual German Conference on Artificial Intelligence, pp. 106 – 117, Berlin, 1994. Springer.

[Joshi et al. 75] A.K. **Joshi**, L. **Levy**, and M. **Takahashi**. *Tree Adjunct Grammars.* Journal of the Computer and Systems Science, 10(1):136–163, 1975.

[Joshi 85a] A.K. **Joshi**. *How much Context-Sensitivity is Required to Provide Reasonable Structural Descriptions: Tree Adjoining Grammars.* In: D. Dowty, L. Karttunen, and A. Zwicky (eds.), Natural Language Processing: Psycholinguistic, Computational and Theoretical Perspectives, pp. 206–250. Cambridge: Cambridge University Press, 1985.

[Joshi 85b] A.K. **Joshi**. *An Introduction to TAGs.* Technical Report MS-CIS-86-64, LINC-LAB-31, Department of Computer and Information Science, Moore School, University of Pennsylvania, 1985.

[Joshi 87a] A.K. **Joshi**. *The Relevance of Tree Adjoining Grammar to Generation.* In: G. Kempen (ed.), Natural Language Generation: New Results in Artificial Intelligence, Psychology and Linguistics, NATO ASI Series E 135, pp. 233–252. Dordrecht: Martinus Nijhoff, 1987.

[Joshi 87b] A.K. **Joshi**. *Word-order Variation in Natural Language Generation.* In: AAAI 87, 6th National Conference on AI, Seattle, USA, 1987.

[Kasper et al. 95] R. **Kasper**, B. **Kiefer**, K. **Netter**, and K. **Vijay-Shanker**. *Compilation of HPSG to TAG.* In: 33th Annual Meeting of the Association for Computational Linguistics, 1995.

[Kempen & Hoenkamp 82] G. **Kempen** and E. **Hoenkamp**. *Incremental Sentence Generation: Implications for the Structure of a Syntactic Processor.* In: J. Horecky (ed.), 9th International Conference on Computational Linguistics. North–Holland Publishing Company, 1982.

[Kempen & Hoenkamp 87] G. **Kempen** and E. **Hoenkamp**. *An Incremental Procedural Grammar for Sentence Formulation.* Cognitive Science, 2(11):201–258, 1987.

[Kempen 91] G. **Kempen**. *Conjunction Reduction and Gapping in Clause–Level Coordination: An Inheritance–Based Approach.* Computational Intelligence, 7:357–360, 1991.

[Kilger 92] A. **Kilger**. *Realization of Tree Adjoining Grammars with Unification.* DFKI Technical Memo TM–92–08, German Research Center for Artificial Intelligence (DFKI GmbH), 1992.

[Kilger 94] A. **Kilger**. *Using UTAGs for Incremental and Parallel Generation.* Computational Intelligence, 10(4):591–603, 1994.

[Kilger 95] A. **Kilger**. *Incremental Linearization with CDL–TAGs in VM–GEN.* Verbmobil–Memo 74, German Research Center for Artificial Intelligence (DFKI GmbH), Saarbrücken, Germany, 1995.

[Kitano 90] H. **Kitano**. *Incremental Sentence Production with a Parallel Marker–Passing Algorithm.* In: 13th International Conference on Computational Linguistics, pp. 217–221, Helsinki, Finland, 1990.

[Längle et al. 95] T. **Längle**, T.C. **Lüth**, G. **Herzog**, E. **Stopp**, and G. **Kamstrup**. *KANTRA – A Natural Language Interface for Intelligent Robots.* In: 4^{th} International Conference on Intelligent Autonomous Systems, Karlsruhe, Germany, 1995.

[Levelt 83] W.J.M. **Levelt**. *Monitoring and Self–Repair in Speech.* Cognition, 14:41–104, 1983.

[Levelt 89] W.J.M. **Levelt**. *Speaking: From Intention to Articulation.* Cambridge, MA: MIT Press, 1989.

[Lock 65] K. **Lock**. *Structuring Programs for Multiprogram Time–Sharing On–Line Applications.* In: AFIPS Conference Proceedings, Fall Joint Computer Conference, volume 27, London, 1965. Macmillan and Co Ltd.

[Marr 82] D. **Marr**. *Vision.* New York: W.H. Freeman, 1982.

[McCoy et al. 90] K. F. **McCoy**, K. **Vijay-Shanker**, and G. **Yang**. *Using Tree Adjoining Grammars in the Systemic Framework.* In: 5th International Workshop on Natural Language Generation, Dawson, PA, 1990.

[McDonald & Pustejovsky 85] D.D. **McDonald** and J. **Pustejovsky**. *TAGs as a Grammatical Formalism for Generation.* In: 23rd Annual Meeting of the Association for Computational Linguistics, pp. 94–103, Chicago, IL, 1985.

[Meteer et al. 87] M.W. **Meteer**, D.D. **McDonald**, S.D. **Anderson**, D. **Forster**, L.S. **Gay**, A.K. **Huettner**, and P. **Sibun**. *MUMBLE–86: Design and Implementation.* COINS Technical Report 87-87, University of Massachusetts, 1987.

[Neumann & Finkler 90] G. **Neumann** and W. **Finkler**. *A Head–Driven Approach to Incremental and Parallel Generation of Syntactic Structures.* In: 13th International Conference on Computational Linguistics, pp. 288–293, Helsinki, Finland, 1990.

[Pollard & Sag 94] C. **Pollard** and I.A. **Sag**. *Head–Driven Phrase Structure Grammar.* Chicago & London: The University of Chicago Press, 1994.

[Poller & Heisterkamp 95] P. **Poller** and P. **Heisterkamp**. *Hybrid Knowledge Sources for Generation in a Speech Dialogue System.* Applied Artificial Intelligence, 1995. in preparation.

[Poller 93] P. **Poller**. *Earley–Parsing von LD/TLP-TAGs.* Master's thesis, Department of Computer Science, University of the Saarland, Saarbrücken, 1993.

[Reithinger 91] N. **Reithinger**. *Eine parallele Architektur zur inkrementellen Generierung multimodaler Dialogbeiträge.* PhD thesis, Department of Computer Science, University of the Saarland, Saarbrücken, 1991.

[Rubinstein & Hersh 84] R. **Rubinstein** and H. **Hersh**. *The Human Factor: Designing Computer Systems for People.* Burlington, Mass.: Digital Press, 1984.

[Schabes & Shieber 92] Y. **Schabes** and S.M. **Shieber**. *An Alternative Conception of Tree–Adjoining Derivation.* In: 30th Annual Meeting of the Association for Computational Linguistics, pp. 167–176, Newark, DW, 1992.

[Schabes et al. 88] Y. **Schabes**, A. **Abeillé**, and A.K. **Joshi**. *Parsing Strategies with 'Lexicalized' Grammars: Application to Tree Adjoining Grammars.* In: 12th International Conference on Computational Linguistics, pp. 578–583, Budapest, Hungary, 1988.

[Schade et al. 91] U. **Schade**, H. **Langer**, H. **Rutz**, and L. **Sichelschmidt**. *Kohärenz als Prozeß.* In: G. Rickheit (ed.), Kohärenzprozesse: Modellierung von Sprachverarbeitung in Texten und Diskursen. Opladen: Westdeutscher Verlag, 1991.

[Schauder 92] A. **Schauder**. *Incremental Syntactic Generation of Natural Language with Tree Adjoining Grammars.* DFKI Document D-92-21, German Research Center for Artificial Intelligence (DFKI GmbH), Saarbrücken, FRG, 1992.

[Shieber & Schabes 90] S.M. **Shieber** and Y. **Schabes**. *Generation and Synchronous Tree Adjoining Grammars.* In: 28th Annual Meeting of the Association for Computational Linguistics, pp. 253–258, Pittsburgh, PA, 1990.

[Shieber et al. 83] S.M. **Shieber**, H. **Uszkoreit**, F.C.N. **Pereira**, J.J. **Robinson**, and M. **Tyson**. *The Formalism and Implementation of PATR–II.* In: B. Grosz and M. Stickel (eds.), Research on Interactive Acquisition and Use of Knowledge. Menlo Park, California: Artificial Intelligence Center, SRI International, 1983.

[Vijay-Shanker & Joshi 88] K. **Vijay-Shanker** and A.K. **Joshi**. *Feature Structure Based Tree Adjoining Grammars.* In: 12th International Conference on Computational Linguistics, pp. 714–719, Budapest, Hungary, 1988.

[Vijay-Shanker 92] K. **Vijay-Shanker**. *Using Descriptions of Trees in a Tree Adjoining Grammar.* Computational Linguistics, 18(4):481–517, 1992.

[Wahlster et al. 92] W. **Wahlster**, E. **André**, S. **Bandyopadhyay**, W. **Graf**, and T. **Rist**. *WIP: The Coordinated Generation of Multimodal Presentations from a Common Representation.* In: A. Ortony, J. Slack, and O. Stock (eds.), Communication from an Artificial Intelligence Perspective: Theoretical and Applied Issues, pp. 121–144. Heidelberg: Springer, 1992.

[Wahlster et al. 93] W. **Wahlster**, E. **André**, W. **Finkler**, H.-J. **Profitlich**, and T. **Rist**. *Plan-based Integration of Natural Language and Graphics Generation.* Artificial Intelligence, 63:387–427, 1993.

[Wahlster 93] W. **Wahlster**. *Verbmobil: Translation of Face–to–Face Dialogs.* Research Report RR-93-34, German Research Center for Artificial Intelligence (DFKI GmbH), Saarbrücken, FRG, 1993.

[Ward 89] N. **Ward**. *Capturing Intuitions About Human Language Production.* In: 11th Conference of the Cognitive Science Society, pp. 956–963, Ann Arbor, 1989. Lawrence Erlbaum.

[Ward 91] N. **Ward**. *A Flexible, Parallel Model of Natural Language Generation.* PhD thesis, Computer Science Division (EECS), University of California, Berkeley, California, 1991.

[Weir 88] D. **Weir**. *Characterizing Mildly Context-Sensitive Grammar Formalisms.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 1988.

[Wirén 92] M. **Wirén**. *Studies in Incremental Natural–Language Analysis.* PhD thesis, Department of Computer and Information Science, Linköping University, Linköping, Sweden, 1992.

[Yonezawa & Tokoro 87] A. **Yonezawa** and M. **Tokoro**. *Object–Oriented Concurrent Programming: An Introduction.* In: A. Yonezawa and M. Tokoro (eds.), Object–Oriented Concurrent Programming, pp. 1–7. Cambridge, Massachusetts: The MIT Press, 1987.