Constraint-Based Text Realization

Joakim Nivre
Växjö University
School of Mathematics and Systems Engineering

Torbjörn Lager
Göteborg University
Department of Linguistics

Abstract

This report introduces constraint-based text realization, a technique for text generation based on logic grammar, which differs from most mainstream approaches in being non-modular and structure-driven. It is argued that this technique provides a simple, flexible and efficient alternative to traditional methods when generating texts of limited variability and sensitivity to context.

1 Introduction

The business of natural language generation (NLG) is to construct computer programs capable of producing natural language texts or utterances from some underlying non-linguistic representation of information. As far as the generation of text is concerned, most present-day NLG systems divide the generation task into three functional steps along the following lines [Reiter 1994, Reiter & Dale 1997]:

1. Text planning
2. Sentence planning
3. Linguistic realization

This usually results in a methodology which can be described as *content-driven* — because the specification of content (the planning) precedes the specification of structure (the realization) — and *modular* — because earlier steps are assumed to be independent of later ones.

In this report, we describe an approach to text generation which, by contrast, can be characterized as *structure-driven* — because the specification of structure as it were precedes the specification of content — and *integrated* — because it collapses the different functional modules into a single process of *text realization*, converting a generation goal into a surface text without using any intermediate representations. We call this approach *constraint-based text realization* because the key component is a constraint-based grammar formalism, where sentence structure and text structure can be described by the same kind of rules, and where semantic constraints on the underlying domain of application can be

incorporated into the rules, making it possible to use these constraints to guide the realization process. Moreover, the formalism permits the integration of different realization techniques, such as canned text, templates and traditional grammar rules, in a flexible and efficient manner.

The technique of constraint-based text realization was originally developed for multilingual generation of job adverts on the World Wide Web.[1] It has since been applied to the generation of weather forecasts involving both text and graphics and to the generation of CVs in multiple languages, but in this report we will stick to the generation of job adverts as our example application. This application is introduced in section 2. In section 3, we present the grammar formalism and exemplify the different kinds of rules that can be expressed in the formalism, and in section 4 we discuss the algorithm used to apply grammar rules in order to convert a generation goal into an actual text.

Despite being more limited than traditional planning-based approaches to natural language generation, we believe there is a large class of practical applications where the integrated, structure-driven methodology of constraint-based text realization provides a useful, and in some cases even preferable, alternative to the more orthodox approaches, and part of the purpose of this report is to try to characterize this class of applications more carefully. In section 5, we therefore compare our approach to the more traditional three-step architecture outlined above, pointing out differences as well as similarities. This provides the background for a discussion, in section 6, of the limitations of constraint-based text realization and an attempt at characterizing the class of applications for which it is best suited. Section 7 summarizes our conclusions.

2 The Domain of Job Adverts

In the following presentation, we will assume a very simple application consisting of three main components:

1. A database of job vacancies, containing information about various properties of jobs, such as type of job, location, education requirements, salary, etc.

2. A search interface where the user can search the database for specific jobs that he/she is interested in.

3. A generator that prints out a short description of any job matching the user's query, describing the type of job, location, education requirements, etc. according to the information contained in the database.

The search interface will largely be ignored in this report, but before we continue we need to say a little more about the job database.

In a realistic application of this sort, the job database will allow a fine-grained specification of vacancies through the use of up to one hundred different attributes concerning such things as requirements of education and experience, working hours, type and duration of contract, age requirements, language skills,

[1]This work was carried out within the project TREE (Trans European Employment), Language Engineering project LE-1182 of the European Commission's Fourth Framework Programme (Telematics section). For a presentation of the project, see [Somers *et al* 1997].

salary, application procedure, etc. [Multari & Gilardoni 1997]. In what follows, we will restrict ourselves to a small subset of these attributes that will nevertheless be rich enough to allow us to illustrate our approach to generation. This simplification is only for expository convenience; the generation system that we have actually implemented for this application handles the full set of attributes used by real employment services (cf. [Ellman *et al* 1999]).

Moreover, the database entries will be given in a logical notation, which corresponds to the logical database interface of the generator, rather than to the underlying (relational) database itself. The point of having a logical database interface on top of the actual database is to make the generator independent of the specific implementation details of the application database and to allow the definition of more complex logical constructs in terms of the simple database concepts. For example, this allows us to introduce entities which are only implicit in the application database but which need to be made explicit in the semantic representations of the generator. This will be exemplified in section 3 below.

To exemplify the logical database representation, here is a very simplified representation of a vacancy for an architect in the city of Göteborg (Sweden):

(1)
```
job_type(173,120).
location(173,'Göteborg').
education(173,7).
experience(173,2).
salary_amount(173,20000).
salary_currency(173,'SEK').
salary_rate(173,3).
```

The number 173, occurring as the first argument in every clause, is the unique identifier of the vacancy in the database. The first two clauses specify that the vacancy concerns job type 120 ('architect') and that the job is located in Göteborg. The next two clauses state that the education requirements are of type 7 ('higher secondary education') and that at least two years experience is required. The last three clauses specify the salary with regard to amount (20,000), currency ('SEK') and pay rate ('per month').

3 Grammar Rules and Constraints

A key idea behind our approach to text generation is the assumption that information about text structure and information about phrase structure can be represented in the same formal framework, which in our case is a constraint-based grammar formalism. The formalism itself is presented in section 3.1, while sections 3.2–3.3 illustrate how it can be used to encode aspects of both phrase structure and text structure.

3.1 Formalism

A grammar rule has the following format:

(2) $C_0{:}S_0 \longrightarrow SS_1, \ldots, SS_n \ \#\ \mathit{Constraints}.$

where C_i is a syntactic category, S_i is a semantic representation, and each SS_i has the format C_i, the format C_i:S_i, or the format $[W_1, \ldots, W_m]$, where W_j is a word form.[2] The hash symbol (#) is used to separate the rule body from a set of domain constraints, which in our examples will take the form of logical conditions on the application database (mediated through the logical database interface). If the set of conditions is empty, the hash symbol may simply be omitted.

The grammar formalism used in our framework belongs to the tradition of logic grammar (cf. [Abramson & Dahl 1989]) and readers are sure to notice the similarity between our formalism and that of Definite Clause Grammar [Pereira & Warren 1980]. It is thus important to note that the categories and semantic representations used in grammar rules are really logical terms, often containing variables that will be instantiated through unification as a result of verifying domain constraints in the generation process.

3.1.1 Syntactic Categories

Besides traditional phrase structure categories, such as sentence, noun phrase, prepositional phrase, etc., extended with syntactic features to handle agreement and similar phenomena, our grammar makes use of text structure categories. These categories may be application-specific, such as the categories 'full ad' and 'one-liner', which we use in the generation of job adverts, or generic, such as the category 'text', which is our name for a plain paragraph of text with no special formatting requirements.

Text categories are similar to traditional syntactic categories in that they are used to classify contentful entities, i.e. entities with a semantic representation. However, we also permit syntactic categories which lack semantic representations and only refer to the formatting of texts, such as 'line break' and 'bold'. Some of these categories (e. g., bold) take the form of functors, which can be applied to other syntactic categories (with or without semantic representations).

3.1.2 Semantic Representations

The grammar formalism in itself does not impose any constraints on the format of semantic representations. In the examples below, however, semantic representations will normally take the following form:

(3) `sem(Type,Val)`

where `Type` is a semantic type, and `Val` refers to an object of that type.[3] The set of semantic types used in a particular application will normally be dependent on the underlying domain. For example, in generating job adverts most of the semantic types correspond to 'slots' in the database of job vacancies, such as `job_type`, `location`, `education_type`, and the semantic values are simply the codes used to fill those slots. Thus, a noun like *architect* will have a semantic representation like the following:

(4) `sem(job_type,120)`

[2] We represent surface strings as lists of words, using the Prolog list notation.

[3] Again, we follow the notational conventions of Prolog in letting variable names begin in uppercase and constant names in lowercase.

where 120 is the code for 'architect' in the database storing vacancies.

The semantic type system used for generating job adverts also includes functor types that may be used to construct complex types from simpler ones. For example, given the two simple types `job_type` and `location`, we can form a complex type where, e. g., the job type is restricted to a certain location:

(5) `rest(job_type,location)`

This type occurs in the semantic representation of a noun phrase like *architect in Göteborg*, where the structure of the semantic value mirrors the structure of the complex type:

(6) `sem(rest(job_type,location),rest(120,'Göteborg'))`

3.1.3 Generation Goals

A generation goal in this framework is simply a syntactic category with a suitable semantic representation. Thus, the goal of generating a full ad describing vacancy 173 is represented as follows:

(7) `full_ad:sem(vacancy,173)`

3.1.4 Domain Constraints

Domain constraints attached to grammar rules provide the necessary link between semantic representations and the underlying domain model (or database). The basic idea is that a grammar rule can only be applied if its associated constraints are true in the domain, which in turn requires that constraints can be evaluated during the generation process. Exactly how this will be done depends on the particular application, but in applications where domain constraints amount to database conditions, as in the job ad application, we define a *logical database interface*, i. e., a series of biconditionals stating the satisfaction conditions for constraints in terms of the underlying application database. For example:

(8) $\forall$`X,Y job_type(X,sem(job_type,Y))` $\Leftrightarrow$ `db(job_type(X,Y))`

This biconditional states that an object Y of type `job_type` is the job type of vacancy X iff the database relation `job_type` holds between X and Y. In the logical database interface, we also have to define satisfaction conditions for complex semantic types, such as the 'located job' type discussed earlier:

(9) $\forall$`X,Y,Z loc_job(X,sem(rest(job_type,loc),rest(Y,Z)))` $\Leftrightarrow$
`db(job_type(X,Y)` $\wedge$ `db(location(X,Z))`

This is a good illustration of the way in which the domain interface can be used to introduce objects that are only implicit in the application database, such as a "located job", but which are needed in the semantic representations of the generator, in this case to define the semantics of noun phrases like *architect in Göteborg*.

3.2 Phrase Structure

In the following sections we will now exemplify the use of the grammar formalism to encode different aspects of the syntax and semantics of texts. We will begin at the level of traditional grammar, i.e. the level of phrase structure. Consider the following rules:

(10) `n(N):sem(rest(HType,RType),rest(HVal,RVal)) ⟶ n(N):sem(HType,HVal), pp:sem(RType,RVal).`

(11) `pp:Sem ⟶ p:Sem, n(_):Sem.`

The first rule says that a nominal constituent[4] (with certain agreement features) whose semantic representation is of the type 'head + restriction' may consist of a nominal expression (with the same agreement features) denoting the 'head object', followed by a prepositional phrase expressing the restriction. The second rule says that a prepositional phrase may consist of a preposition followed by a nominal expression, where both constituents inherit the semantic representation of the mother. Now, suppose we add the following rules in order to provide lexical material for the phrases in question:[5]

(12) `n(sing):sem(job_type,120) ⟶ [architect].`

(13) `n(sing):sem(location,S) ⟶ [S].`

(14) `p:sem(location,_) ⟶ [in].`

These rules say that 'architect' is a singular noun denoting job type 120, that any string S is (potentially) a singular noun denoting location S, and that 'in' is a preposition (heading a phrase) of type location. Given these rules, together with the previous ones, the noun phrase in (16) can be generated from the goal in (15):

(15) `n(sing):sem(rest(job_type,loc),rest(120,'Göteborg'))`

(16) *architect in Göteborg*

3.3 Text Structure

As already mentioned, we assume that the same kind of rules that were used in the previous section to describe phrase structure can also be used to describe the structure of larger text elements, at least for the simple kind of texts that we have been dealing with so far. As a first and almost trivial example, we give the rule for a 'one-liner' job ad:

[4] Note that we do not make any categorical distinction between noun phrases and nouns (or 'n-bars') but instead rely on semantic information to ensure that rules do not overgenerate.

[5] The framework itself does not require that the lexicon be specified by means of terminal productions in this way. In fact, in the implemented system for generating job adverts, lexicalization requires lookup in a separate term database. But for ease of exposition, we will ignore the distinction between grammar and lexicon in what follows.

(17) `one_liner:sem(vacancy,Vac) ⟶`
`n(_):LocJob # loc_job(Vac,LocJob).`

This rule says that a one-liner describing the vacancy Vac (where the variable `Vac` will be instantiated to the unique identifier of a vacancy in the database) can consist of a nominal expression with the semantic representation `LocJob`, provided that it is true in the current state of the domain (database) that LocJob is the 'located job' of `Vac`. Suppose, for example, that the database contains the following facts:

(18)
```
job_type(173,120).
location(173,'Göteborg').
education(173,7).
experience(173,2).
salary_amount(173,20000).
salary_currency(173,'SEK').
salary_rate(173,3).
```

Given the generation goal `one_liner:sem(vacancy,173)`, we may use the rule above (together with the phrase structure rules from the previous section and the logical database interface from section 3.1.4) to generate the 'text' in (16).

As a second example, let us consider a greatly simplified version of the rule for a 'full ad':

(19) `full_ad:sem(vacancy,Vac) ⟶ heading1(n(_):LocJob),`
`heading2(['Specifications']), text:sem(spec,Vac).`

This rule says that a full ad, describing a particular vacancy `Vac`, consists of three elements, the first of which is the same kind of noun phrase that appears in a one-liner, but formatted as a first level heading. The second element is the word 'Specifications', formatted as a second level heading, while the third element is a text whose content is a specification ('spec') of the vacancy `Vac`. Let us now see how the specification text can be further expanded:

(20) `text:sem(spec,Vac) ⟶ text:sem(education,Vac),`
`text:sem(experience,Vac), text:sem(salary,Vac).`

The first element of the specification text concerns education requirements and is covered by the following rules:

(21)
```
text:sem(education,Vac) ⟶ bold(['Education:']),
comma_list(n(_)):Edus, line_break #
setof(Edu,education(Vac,Edu),Edus).
```

(22) `text:sem(education,Vac) ⟶ [].`

The database condition associated with the first rule checks whether there are any education requirements connected with the vacancy being specified. If this is so, the variable `Edus` gets instantiated to the list of requirements, expressed in terms of integer codes (cf. section 2), which becomes the semantic representation of a 'comma list', which is a text category that takes another syntactic category as argument — in this case the category `n(_)`. The intuitive idea is that a 'comma list' is a list of items of the argument category, separated by commas,

corresponding one by one to the items on the semantic representation list. The rules for expanding comma lists look as follows:[6]

(23) `comma_list(Cat):[Sem] --> Cat:Sem.`

(24) `comma_list(Cat):[Sem|[X|Y]] --> cat:Sem, [','], comma_list(Cat):[X|Y].`

The second rule for education requirements is needed for the case where there are no education requirements associated with a vacancy, in which case no text at all should be generated.

The second element of the specification is about experience requirements:

(25) `text:sem(experience,Vac) --> bold(['Experience:']), ['at least'], N, ['years experience'], line_break # experience(Vac,Exp).`

(26) `text:sem(experience,Vac) --> [].`

The first of these rules is a good illustration of the ease with which co-called 'canned text' and 'templates' can be integrated into the grammar. Since the pattern 'At least N years of experience' is invariable, except for the value of N, there is no need to use grammar rules to generate that phrase, although it would of course be possible to do so. Other examples of canned text occur in the headings for each item of the specification ('Education', 'Experience', etc.).

The third item in the specification is the salary specification:

(27) `text:sem(salary,Vac) --> bold(['Salary:']), pp:Rate, [','], n(_):Sal, line_break # salary_rate(Vac,Rate), salary(Vac,Sal).`

(28) `text:sem(salary,Vac) --> bold(['Salary:']), pp:Rate, [','], ['amount not specified'], line_break # salary_rate(Vac,Rate).`

(29) `text:sem(salary,_) --> [].`

The first rule generates a prepositional phrase indicating the 'salary rate' (e.g., 'per month'), a comma, and a noun phrase specifying the 'salary' (e.g. '2000 SEK'), where 'salary' is a logical construct composed of the database concepts 'salary amount' and 'salary currency' via conditions in the logical database interface. The second rule generates the same kind of prepositional phrase specifying the rate, but followed by a piece of canned text: 'amount not specified'. The third rule again covers the case when no specification can be generated.

Given the text structure rules outlined in this section, together with phrase structure rules and a suitable lexicon, we can now generate the following text from the generation goal `full_ad:sem(vacancy,173)` (given the database above).

[6] Note again the Prolog list notation where `[X|Y]` denotes a list with head `X` and tail `Y`.

(30)
```
<h1>Architect in Göteborg</h1>
<h2>Specifications</h2>
<b>Education:</b> higher secondary education<br>
<b>Experience:</b> at least 2 years experience<br>
<b>Salary:</b> per month, 20000 SEK<br>
```

In this example, formatting properties such as boldface and different heading levels have been encoded with HTML tags, since the ads are meant to be displayed in web pages. And while other applications may have other requirements, it is still the case that most realistic applications require some sort of text encoding. In keeping with the general philosophy of the integrated approach, we handle this aspect of generation using the same kind of grammar that is used to describe phrase structure and text structure. As an illustration, here are some of the rules needed to realize the text above:

(31) `line_break` $\longrightarrow$ `['<br>'].`

(32) `heading1(Item)` $\longrightarrow$ `['<h1>'], Item, ['</h1>'].`

(33) `bold(Item)` $\longrightarrow$ `['<b>'], Item, ['</b>'].`

Note that in the case of functor categories like `heading1` and `bold`, the formatting rules have to be integrated with the rest of the grammar in order to allow the expansion of whatever category occurs as the argument of the functor category. This means that, although we have presented phrase structure rules, text structure rules and formatting rules in separate sections, they are really different parts of the same integrated grammar.

4 Text Realization

Given a generalized text grammar of the kind outlined in the previous section, we still face the problem of choosing which rules to apply in a given situation.[7] For this we have developed the following simple algorithm:

- Given a generation goal G:
  1. Select the most specific rule R whose head unifies with G.
  2. Try to prove the domain constraints associated with R.
  3. If successful, apply R; else select the next most specific rule whose head unifies with G and go to 2.

Now, suppose we want to process the generation goal `text:sem(salary,173)`, i. e., we want to generate a piece of text describing the salary conditions of vacancy 173. In our example grammar, we have three rules that might be used to do this:

[7] In addition, we have to decide in which order to process sequentially ordered generation goals. This choice, however, does not affect the outcome of the generation process.

(34) `text:sem(salary,Vac) ⟶ bold(['Salary:']), pp:Rate, [','], n(_):Sal, line_break # salary_rate(Vac,Rate), salary(Vac,Sal).`

(35) `text:sem(salary,Vac) ⟶ bold(['Salary:']), pp:Rate, [','], ['amount not specified'], line_break # salary_rate(Vac,Rate).`

(36) `text:sem(salary,_) ⟶ [].`

These rules are ordered here with respect to the specificity criterion; the first rule is most specific because it gives information about both pay rate and salary level; the second is less specific because it only gives information about pay rate, but it is nevertheless more specific than the last one, which generates the empty text containing no information at all. Hence, according the realization algorithm, the rules should be tried in this order.

Now, suppose the state of the job database is as given below:

(37)
```
job_type(173,120).
location(173,'Göteborg').
education(173,7).
experience(173,2).
salary_amount(173,20000).
salary_currency(173,'SEK').
salary_rate(173,3).
```

Then the conditions associated with the first rule may be proven, and the variables `Rate` and `Sal` will be instantiated (via the logical database interface) to the terms `sem(salary_rate,3)` and `sem(rest(amount,currency),rest(2000,'SEK'))`, respectively, and the following text will eventually be generated:

(38) `<b>Salary:</b> per month, 2000 SEK<br>`

But suppose instead the database only contains the following facts (i. e., suppose there is no information about salary level or currency):

(39)
```
job_type(173,120).
location(173,'Göteborg').
education(173,7).
experience(173,2).
salary_rate(173,3).
```

Then the database conditions associated with the first rule cannot be satisfied (because there is no object `Sal` such that `salary(Sal,173)` is true) and the second rule will be used instead, producing the following text describing salary:

(40) `<b>Salary:</b> per month, amount not specified<br>`

Finally, suppose the database does not contain any information about pay rate either. In that case only the third rule can be used, which in actual fact means that the piece of text describing salary is suppressed in the resulting job advert.

This example illustrates an important aspect of the flexibility of constraint-based text realization, namely how the same generation goal processed with the same grammar and the same algorithm may result in different texts depending on the information available in the domain model. Moreover, the algorithm used may be motivated by an appeal to general pragmatic principles. Echoing Grice [Grice 1975, Grice 1989], we could say that trying rules in order of specificity amounts to the same thing as trying to make your contribution as informative as is required (Maxim of Quantity). And only applying rules whose domain constraints are satisfied of course means not saying that which you believe to be false (Maxim of Quality).

Finally, we would like to point out that the algorithm used in text realization imposes certain conditions on the form of the grammar:

- For every possible generation goal G (where G has the form C or $C{:}S$) there must be a non-empty set of rules $\{R_1, \ldots, R_n\}$ and a specificity relation $>$ such that:

  1. $\forall i, j[R_i{>}R_j \wedge R_j{>}R_i] \rightarrow i = j$
  2. $\forall i, j, k[R_i{>}R_j \wedge R_j{>}R_k] \rightarrow R_i{>}R_k$
  3. $\exists i \forall j[R_j{>}R_i \wedge Con(R_i) = \{\}]$

  where $Con(R_i)$ is the set of domain constraints associated with rule R_i.

Conditions 1–2 ensure that the specificity relation is a weak ordering relation, while condition 3 requires there to be at least one rule which is minimal with respect to this ordering (i. e., which is less specific than all other rules) and which has no domain constraints. The latter condition guarantees that the text realization process always succeeds, irrespective of the state of the domain model. Note also that we do not require the specificity relation to be a strict ordering, which means that there can be a non-deterministic choice between equally specific rules, a possibility that we have exploited, e. g., in the weather forecast application to randomly generate paraphrases in order to increase variation in the texts produced.

5 Comparisons

Having presented in some detail the approach of constraint-based text realization, we will now proceed to a comparison with other available methods for text generation. We will take as our main point of departure a recent state-of-the-art article by Ehud Reiter and Robert Dale [Reiter & Dale 1997].

5.1 The State of the Art in Natural Language Generation

According to [Reiter & Dale 1997], the most common architecture in present-day NLG systems is a three-stage pipeline with the following stages:

1. Text planning
2. Sentence planning
3. Linguistic realization

Text planning usually combines the tasks of content determination and discourse planning in order to construct a text plan from a suitably represented generation goal. The sentence planning step involves such tasks as sentence aggregation, referring expression generation, and sometimes lexicalization, and aims at converting text plans into sentence plans. Linguistic realization, finally, converts the sentence plans into surface text through syntactic, morphological and orthographic processing. Thus, the complete list of NLG tasks that Reiter and Dale discuss is the following (cf. [Reiter & Dale 1997], section 4.1):

1. Content determination
2. Discourse planning
3. Sentence aggregation
4. Lexicalization
5. Referring expression generation
6. Linguistic realization

Although the detailed architecture and the number of actual modules may differ from one system to the other, there seems to be two commonly shared assumptions here; first, that the problem of natural language generation requires a modular approach; and secondly, that content determination should precede structural realization in the generation process. The latter assumption was even more pronounced in the old traditional way of dividing the generation task into deep (strategic) generation, or determining 'what to say', and surface (tactic) generation, or determining 'how to say it' (see, e. g., [McKeown & Swartout 1988]).[8]

Nevertheless, as indicated in the introduction, we think that constraint-based text realization contrasts with the majority of current approaches along both these dimensions, i. e., in being non-modular and structure-driven. We will therefore focus on these two aspects in the following.

5.2 Modularization vs. Integration

One of the distinctive features of constraint-based text realization is that it merges the levels of text and sentence processing, which completely obviates the need for intermediate representations or interfaces between modules. Now, we are certainly not the first to propose a non-modular approach to natural language generation. Perhaps the most well-known example of this in the literature is Appelt's KAMP system [Appelt 1985], which uses a single knowledge representation formalism to encode all the knowledge needed to plan and produce utterances. Another point in common is that logical representations play a key role in both frameworks, but apart from that the approaches are rather different. While Appelt uses an AI planning paradigm and extends this to aspects of linguistic realization, we start from the grammars used in linguistic realization and extends them to typical planning tasks. So, in a way, you could say that our approach is the complete opposite of Appelt's.

[8] In fact, the priority of content seems to implied already by the anaphoric use of *it* in 'how to say it'.

Our approach to text planning has much more in common with the use of *schemas* [McKeown 1985, Kittredge *et al* 1991]. In fact, many of our text structure rules *are* schemas, specifying how a certain kind of text should be constructed in terms of smaller constituents. What is different, though, is that we have integrated these rules into the same framework that is used to describe sentence structure, using a single formal representation and a single process of text realization, which means that some of the traditional generation tasks, such as sentence aggregation, becomes superfluous.

5.3 Content-driven vs. Structure-driven Approaches

As noted earlier, almost all existing approaches to text generation assumes that content determination precedes structural realization. To a certain extent, this is of course true also of our approach, since the generation process always starts from a goal of the form C:S, where C is a syntactic category and S is a semantic representation of the text to be generated. In most cases, however, this is a very non-specific representation of content — in the job ad application it only contains the unique database identifier of the vacancy to be described — and most of the detailed content is actually determined as a side effect of applying grammar rules in order to realize a text.

Exaggerating somewhat, we could say that we have subsumed all the other NLG tasks under linguistic realization. That is, once we have a generation goal, we immediately start the realization process by applying grammar rules. Since most of the rules have domain constraints associated with them, this process will be guided by semantic conditions determining how the content will be articulated depending on the current state of the domain model. But the driving force is the structural realization of a text through the application of grammar rules, and in this sense the approach deserves to be called structure-driven, although it might be more accurate to say that structure and content are developed in parallel.

Regardless of how we describe it, however, the effect of this methodology is that the tasks of content determination, lexicalization and referring expression generation will not be performed prior to linguistic realization but rather in conjunction with it, through the application of rules guided by domain constraints. In this respect, our approach resembles more simplistic template-based approaches to generation, and some people might even argue that the former is only a variant of the latter. But we think there are important differences. First, although the grammar formalism allows easy integration of both canned text and templates, it is nevertheless a grammar formalism, which means that linguistic phenomena such as agreement and subcategorization can be handled in a systematic and principled fashion. Secondly, the systematic use of semantic representations in grammar rules, which is crucial in the evaluation of domain constraints, ensures that the system has a transparent semantics, which is usually not true of simple template-based systems. Nevertheless, it is clear that one of the attractions of constraint-based text realization lies precisely in the fact that it allows the integration of template- and grammar-based generation, a feature which has been pointed out as a key to efficient but flexible systems in the future [Hovy *et al* 1995]. In practice, this means that constraint-based text realization can preserve the simplicity and efficiency inherent in template-based methods while adding flexibility and reusability. We will have more to say about

this in the next section.

6 Limitations and Possibilities

Even though we believe that constraint-based text realization is a useful method for many text generation applications, it should be obvious that it is more limited in scope than systems adopting specialized techniques for all the tasks traditionally involved in text planning and sentence planning. This gives rise to two important and related questions:

1. What characterizes the class of applications for which constraint-based text realization is a useful method?

2. What exactly are the advantages of using the method for those applications?

Although a considerable amount of testing and research is still needed before we can give precise answers to these questions — partly because the usefulness of *any* NLG technique for a given application is not yet very well understood [Reiter & Dale 1997] — we will close the report by presenting our current thoughts on these issues.

First of all, we believe that constraint-based text realization is most useful when the texts to be generated can be broken down into syntactico-semantic sub-units (paragraphs, list items, table rows, sentences, phrases, etc.) that satisfy the following conditions:

1. **Pre-fixed order:** The order between sub-units is either completely fixed or subject only to small and predictable variation.

2. **Context-insensitivity:** The internal structure of sub-units is relatively insensitive to the presence or absence, as well as the structure and content, of other sub-units.

Note here that the condition of a pre-fixed order does not require that the same kind of text unit always consists of the same number of sub-units, only that when we have a particular set of sub-units they can be sequentially ordered without reasoning about their internal structure or content. In fact, allowing an expected sub-unit to be suppressed when relevant information is lacking is easily achieved, as we have seen in previous sections. The point is simply that this should not affect the ordering of remaining units.

Concerning the second condition, it may be pointed out that the insensitivity to context applies only to linguistic context and means, for example, that constraint-based text realization is not convenient for texts where anaphoric relations abound. Extra-textual context constraints, on the other hand, are handled by the domain constraints associated with grammar rules.[9]

Texts that satisfy these two conditions might be described as having a semi-rigid structure, which means that they consist of a more or less fixed sequence of elements, some of which may be missing, but where the internal structure of

[9] In principle, it would be possible to handle intra-textual context constraints in the same way, but we suspect that it would become very cumbersome very soon, although this is really an empirical question.

each element is not (heavily) dependent on the other elements in the sequence. The texts occurring in the applications we have been working with so far (job ads, weather forecasts, CVs) clearly satisfy these conditions to a high degree, and we believe there are many other practical applications where this is also the case. And this is the reason why we think the structure-driven approach has a lot to offer. When generating texts where the most predictable and stable properties reside in their (hierarchical and sequential) structure, it simply makes most sense to start from the structural side in order to articulate the content.

This brings us to the advantages of constraint-based text realization. We believe that, for applications satisfying the conditions outlined above, this approach can offer the following desirable properties:

1. **Simplicity:** For anyone familiar with constraint-based grammar (and semantics), the transparency of the formalism makes for easy and quick development, which means fewer errors and less development time.

2. **Efficiency:** The absence of pipe-lined modules, interfaces and intermediate representations makes the generation process very efficient, which is often important in large-scale practical applications where large amounts of text have to be generated with reasonable speed.

3. **Flexibility:** The possibility of integrating canned text and templates into full-fledged grammar rules gives great flexibility in grammar development, i. e., you can use the full power of unification-based grammars when you need it, but you are not forced to use it when something much simpler is quite sufficient.

4. **Reusability:** Because of the transparency of grammar rules, it is often possible to reuse portions of old grammars in developing new applications or in porting an old application to a new language and/or domain model.

The first two properties stand in contrast to the more ambitious state-of-the-art techniques, while the third and fourth distinguish our approach from the old 'quick and dirty' methods based on canned text and templates.

Reiter and Mellish have argued that what they call 'intermediate' generation techniques are appropriate as long as corresponding 'in-depth' approaches are poorly understood, less efficient or more costly to develop [Reiter & Mellish 1993]. Busemann and Horacek go one step further and claim that what they call 'shallow' techniques are not only useful as a substitute for in-depth methods, but are actually preferable for many small applications, provided they offer enough flexibility and reusability [Busemann & Horacek 1998]. We tend to share this view, and we think our own approach fits well into Busemann and Horacek's category of flexible shallow generation techniques. In fact, our methodology has many things in common with the approach presented in [Busemann & Horacek 1998], although their approach is more conservative with respect to the issues of modularity vs. integration and content-driven vs. structure-driven generation.

7 Conclusions

In conclusion, let us try to summarize what we take to be the weak and the strong points of constraint-based text realization. On the negative side, we

have observed that our approach is mainly suitable for applications where the generated texts have a semi-rigid structure and where text elements are fairly insensitive to (textual) context. On the positive side, we have claimed that there are substantial gains in using this approach when suitable, gains in ease and speed of development, efficiency of processing, flexibility in realization techniques, and reusability of resources. Finally, we have expressed our support for the view, stated in [Busemann & Horacek 1998], that there is a real need for what might be called 'quick and clean' methods of generation, i. e., methods that combine the simplicity and efficiency of the old *ad hoc* techniques with some of the flexibility and generality of the more theoretical approaches. We hope that our own work may contribute to the fulfilment of that need.

Acknowledgements

The work presented here was partly supported by the European Commission, DG-XIII, through the project TREE (Trans European Employment), Language Engineering project LE-1182 (Fourth Framework Programme, Telematics Section). We thank all the people who participated in the project and especially our colleagues in the Göteborg team: Jens Allwood, Anders Green, Sören Sjöström, and Kaarlo Voionmaa.

References

[Abramson & Dahl 1989] Abramson, H. and Dahl, V. (1989) *Logic Grammars*. Springer Verlag.

[Appelt 1985] Appelt, D. E. (1985) *Planning English Sentences*. Cambridge University Press.

[Busemann & Horacek 1998] Busemann, S. and Horacek, H. (1998) A Flexible Shallow Approach to Text Generation. In *Proceeding of the 9th International Workshop on Natural Language Generation*, Niagara-on-the-Lake, Canada, August 1998, pp. 238–247.

[Ellman *et al* 1999] Ellman, J., Goddard, A., Green, A., Nivre, J., Gilardoni, L., Wallington, A. and Walsh, S. (1999) The TREE Project — Final Report. Technical Report TREE, LE 1182-D-20.

[Multari & Gilardoni 1997] Multari, A. and Gilardoni, L. (1997) Data Structures Design and Specification 2. Technical Report TREE, LE 1182-WP3-T3.2.

[Grice 1975] Grice, H. P. (1975) Logic and Conversation. In Cole, P. and Morgan, J. (eds) *Syntax and Semantics. Vol. 3: Speech Acts*. Academic Press.

[Grice 1989] Grice, H. P. (1989) *Studies in the Way of Words*. Harvard University Press.

[Hovy *et al* 1995] Hovy, E., van Noord, G., Neumann, G. and Bateman, J. (1995) Language Generation. Chapter 4 of *Survey of the State of the Art in Human Language Technology*. URL: `http://www.elsnet.org/publications/hlt/index.html`.

[Kittredge *et al* 1991] Kittredge, R., Korelsky, T. and Rambow, O. (1991) On the need for domain communication knowledge. *Computational Intelligence* **7**, 305–314.

[McKeown 1985] McKeown, K. (1985) Discourse strategies for generating natural-language text. *Artificial Intelligence* **27**, 1–42.

[McKeown & Swartout 1988] McKeown, K. R. and Swartout, W. R. (1988) Language generation and explanation. In M. Zock and G. Sabah (eds.) *Advances in Natural Language Generation: An Interdisciplinary Perspective*, pp. 1–51. London: Pinter.

[Pereira & Warren 1980] Pereira, F. C. N. and Warren, D. H. D. (1980) Definite clause grammars for language analysis — a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence* **13**, 231–278.

[Reiter 1994] Reiter, E. (1994) Has a consensus NL generation architecture appeared, and is it psycholinguistically plausible? In *Proceedings of the 7th International Workshop on Natural Language Generation*, pp. 163–170.

[Reiter & Dale 1997] Reiter, E. and Dale, R. (1997) Building applied natural language generation systems. *Natural Language Engineering* **3**: 57–87.

[Reiter & Mellish 1993] Reiter, E. and Mellish, C. (1993) Optimizing the costs and benefits of natural language generation. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, Chambery, France, pp. 1164–1169.

[Somers *et al* 1997] Somers, H., Black, B., Ellman, J., Gilardoni, L., Lager, T., Multari, A., Nivre, J. and Rogers, A. (1997) Multilingual Generation and Summarization of Job Adverts: The TREE Project. *Proceedings of the Fifth Conference on Applied Natural Language Processing*, Washington D. C., March/April 1997, pp. 269–276.