Dependency-directed Unification of Functional Unification Grammar in Text Generation

INUI Kentaro, TOKUNAGA Takenobu and TANAKA Hozumi
Department of Computer Science
Tokyo Institute of Technology
2-12-1 Ôokayama Meguro Tokyo 152 Japan
{inui,take,tanaka}@cs.titech.ac.jp

Abstract

In text generation, various kinds of choices need to be decided. In the conventional framework, which can be called "one-path generation framework," these choices are decided in an order designed carefully in advance. However, many researchers have pointed out that the choices, generally, depend on one another and the one-path generation framework cannot handle these interdependencies sufficiently. Our previous paper proposed introducing a revision process into text generation for solving this problem. In our framework, the overall generation process consists of the initial generation process, followed by the revision process. The revision process gives us opportunities to change choices that have already been made. In general, a change in a choice point may cause changes in other choice points, and such dependencies can be managed by Truth Maintenance System (TMS). However, it is well known that dependency network management in TMS requires some computational overhead in general. We need an efficient implementation of network management to make our framework feasible. In this paper, we propose an efficient implementation of dependency network management in Prolog. In our implementation, arcs between dependent nodes are represented by bindings of logical variables, and efficient state propagation is realized by destructive argument substitutions.

1 Introduction

In text generation, various kinds of choices need to be decided. At each choice point, more than one alternatives satisfying various kinds of constraints may be available. In such a case, the system chooses one out of the alternatives by referring to heuristic rules, which assign preferences to them. We call such rules "preference rules" in this paper. In the conventional framework, which can be called "one-path generation framework", these choices are decided in an order designed carefully in advance. However the choices, generally, depend on one another. These interdependencies make it difficult to design the rules.

In the field of text generation, this issue has been discussed and several solutions have been proposed [1, 5, 8]. In our previous paper, we proposed a generation model that incorporates a revision component as illustrated in Fig. 1 [9]. In this model, the overall generation process consists of the initial generation process followed by the revision process. The revision process is realized by the repeated revision cycles each of which consists of evaluation of a draft, revision planning, and regeneration.

Since we focus on surface generation[1] at the present, we assume that the input to the system be a rhetorically organized semantic representation [13]. In the initial generation process and the regeneration process, the surface generator refers to the preference rules to decide the choices. In our model, however, these decisions are tentative and may be changed in the revision process. Thus the result of generation is called a "draft." A draft contains not only semantic information but also syntactic and lexical information. What is significant here is that all the necessary choices have been decided at the end of the initial generation process. In the revision process, if the evaluator finds a problem in the current draft, the

[1]In general, text generation can be decomposed into two phases, deep generation and surface generation. Deep generation decides the contents and the organization of a text, while surface generation makes choices on syntactic structure and lexical items [15].

revision planner refers to the revision rules to solve it. The revision rules suggest which choice should be changed to solve the problem. We call such a choice "culprit choice." In the regeneration process, the surface generator changes a culprit choice and generates another draft.

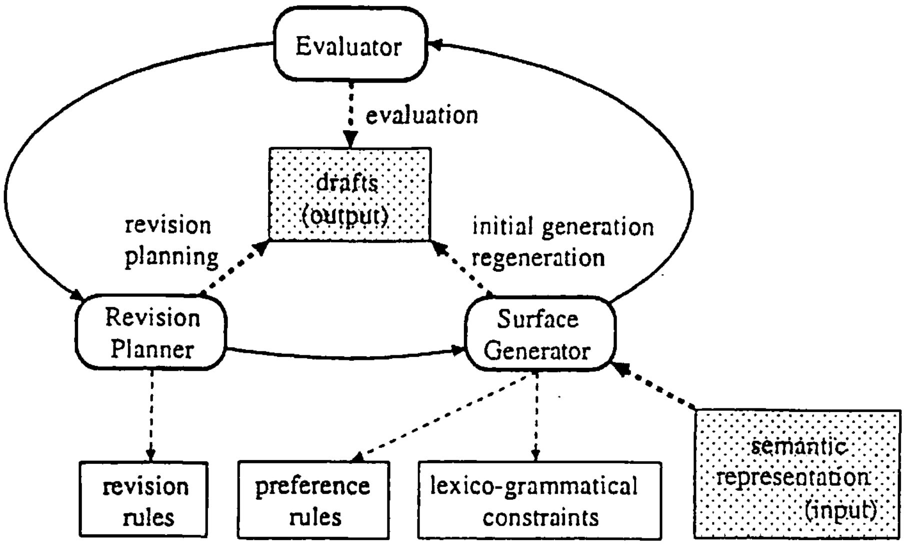


Fig. 1 Generation model with revision component

A major difficulty in handling the interdependencies among the choices is that at a certain choice point the system could not decide an appropriate choice unless the system had the information about the effects of the choice on the following decisions and the final text. In one-path generation, therefore, the system needs to anticipate the future decisions to decide the current choice. While, in our model, the evaluator can evaluate the decisions according to all the information necessary for the evaluation. This is because in the evaluation process all the necessary choices have already been made. In the previous paper, we considered structural complexity and ambiguity of each sentence as the evaluator's criteria.

As mentioned above, the system revises a draft by changing a culprit choice and regenerating another draft. In this sense, revision can be seen as backtracking. Our approach is, however, different from naive chronological backtracking. While, in chronological backtracking, the system would go back to the latest choice point, our backtracking is "dependency-directed." That is:

- the system directly goes back to a culprit choice point by referring to the revision rules,
- and the system reuses the results of the previous computation if possible when regenerating another draft.

To realize dependency-directed backtracking (DDB), the system needs to maintain the history of decisions and their effects on the current draft. In the previous paper, we proposed a method to use Justification-based Truth Maintenance System (JTMS) [2] to realize DDB in text generation. In this method, the system maintains the history of choices and the dependencies among them in a dependency network. It is, however, well known that dependency network management in JTMS requires some computational overhead. We need an efficient implementation of network management to make our framework feasible. In this paper, we propose an efficient implementation of DDB for text generation in Prolog. In our implementation, arcs between dependent nodes are represented by bindings of logical variables, and efficient state propagation is realized by destructive argument substitutions.

In the following sections, we first describe an overview of text generation in section 2. We adopt the formalism of Functional Unification Grammar (FUG) for representing linguistic knowledge. In section 3, we propose an efficient implementation of FUG unification with DDB. In section 4, we show a result of a preliminary experiment to evaluate our implementation. Finally, we conclude the paper with some future research directions in section 5.

2 Controlling FUG unification by DDB

We are now developing a Japanese grammar based on the framework of systemic-functional grammar [7]. SFG has desirable features for text generation and has been used by several text generation systems [3, 12, 14]. This is mainly because of the following respects [14].

- SFG organizes the linguistic information based on the "paradigmatic" perspective, which makes the choices in generation explicit.
- SFG describes the linguistic constraints in terms of functionality of language. Consideration of functionality is indispensable in goal-oriented text generation.

The first aspect is desirable not only for text generation but also for revision. Since SFG describes choices and their effects on the lexico-grammatical structure explicitly, it is not difficult to find a correspondence between a problem in a draft and candidates of its culprit choices. Therefore, design of the revision rules would be easier than in the case using the other framework.

As a computational tool to realize SFG, we use Functional Unification Grammar (FUG) [11], which has good properties for the implementation of DDB in the following respects.

- SFG can be represented in the FUG notation straightforwardly [10]. For example, each choice in SFG corresponds to a disjunction in FUG and each conflation in SFG [2] corresponds to a unification in FUG.
- FUG is based on unification and the order of choices is flexible. This property is desirable for our model since the order of choices may be changed during the revision process.

In this section, we first briefly describe text generation using FUG, and then explain the control of unification by DDB.

2.1 Text Generation using FUG

In the FUG formalism, the grammar is described in the form of functional descriptions (FDs), which we call "grammar functional descriptions (GFDs)" in this paper. The input to the system is also represented by FDs. To generate a text, the system unifies an input FD with a GFD. As unification proceeds, features in the GFD are added to the input FD; thus we refer to the input FD by the term "working functional description (WFD)" during unification. Fig. 2 shows an example of unification of a GFD and a WFD, generating the sentence "John loves Mary."

In our system, unification of a WFD and a GFD proceeds in a top-down and depth-first manner. In the example in Fig. 2, the GFD has three alternatives at the top level, where only the first alternative can unify with the WFD. The unification produces:

$$\left[\begin{array}{ll} \texttt{cat}: \texttt{s} & \\ \texttt{prot}: & \left[\begin{array}{l} \texttt{cat}: \texttt{np} \\ \texttt{n}: [\texttt{lex}: \texttt{John}] \end{array}\right] \\ \texttt{goal}: & \left[\begin{array}{l} \texttt{cat}: \texttt{np} \\ \texttt{n}: [\texttt{lex}: \texttt{Mary}] \end{array}\right] \\ \texttt{verb}: & \left[\begin{array}{l} \texttt{cat}: \texttt{vp} \\ \texttt{n}: [\texttt{lex}: \texttt{love}] \end{array}\right] \\ \texttt{pattern}: [\texttt{prot}, \texttt{verb}, \texttt{goal}] & \end{array}\right].$$

Then the system tries to unify recursively each constituent that is listed in the value of feature **pattern**. In this case, the value of **prot** is unified first with the GFD, the second alternative being chosen this time. After the completion of recursive unification of **prot** with the GFD, the system moves on to the

[2]Conflation is an important device to realize the multi-functionality of SFG. In SFG, a constituent can have more than one function. For example, "John" in the sentence "John bought a car." realizes not only the ideational function AGENT but the textual function THEME. In the FUG framework, the multi-functionality can be realized by conflating (unifying) constituents, a constituent labeled AGENT and another constituent labeled THEME in this example.

next feature verb. The final result is also shown in Fig. 2.[3] If unification fails during the process, the system goes back to the last disjunction to try another alternative. The system tries alternatives one by one at each choice point, which means that the order of the alternatives represents a static preference among these alternatives. Therefore, the information represented in GFDs can be considered both the lexico-grammatical constraints and the preferences among the alternatives (the preference rules in Fig. 1).

In text generation, the top-down and depth-first fashion will work efficiently.

$$\text{GFD} = \left\{ \begin{array}{ll} \boxed{1} & \left[\begin{array}{l} \texttt{cat : s} \\ \texttt{prot : [cat : np]} \\ \texttt{goal : [cat : np]} \\ \texttt{verb : [cat : vp]} \\ \texttt{pattern : [prot, verb, goal]} \end{array}\right] \\ \boxed{2} & \left[\begin{array}{l} \texttt{cat : np} \\ \texttt{n : [cat : noun]} \\ \texttt{pattern : [n]} \end{array}\right] \\ \boxed{3} & \left[\begin{array}{l} \texttt{cat : vp} \\ \texttt{v : [cat : verb]} \\ \texttt{pattern : [v}\cdots\texttt{]} \end{array}\right] \end{array}\right\}$$

$$\begin{array}{c}\text{Input FD} \\ \text{(WFD)}\end{array} = \left[\begin{array}{l} \texttt{cat : s} \\ \texttt{prot : [n : [lex : John]]} \\ \texttt{goal : [n : [lex : Mary]]} \\ \texttt{verb : [n : [lex : love]]} \end{array}\right]$$

$$\begin{array}{c}\text{Result} \\ \text{(WFD)}\end{array} = \left[\begin{array}{ll} \texttt{cat : s} & \\ \texttt{prot :} & \left[\begin{array}{ll} \texttt{cat : np} & \\ \texttt{n :} & \left[\begin{array}{l} \texttt{lex : John} \\ \texttt{cat : noun} \end{array}\right] \\ \texttt{pattern : [n]} & \end{array}\right] \\ \texttt{goal :} & \left[\begin{array}{ll} \texttt{cat : np} & \\ \texttt{n :} & \left[\begin{array}{l} \texttt{lex : Mary} \\ \texttt{cat : noun} \end{array}\right] \\ \texttt{pattern : [n]} & \end{array}\right] \\ \texttt{verb :} & \left[\begin{array}{ll} \texttt{cat : vp} & \\ \texttt{v :} & \left[\begin{array}{l} \texttt{lex : love} \\ \texttt{cat : verb} \end{array}\right] \\ \texttt{pattern : [v]} & \end{array}\right] \\ \texttt{pattern : [prot, verb, goal]} & \end{array}\right]$$

Fig. 2 An example of GFD and WFD

Since a text generation system does not have to give possible outputs all at once, and because our framework allows chances to revise a draft during the revision process, depth-first search is suitable for our purpose.

2.2 DDB in FUG unification

JTMS maintains the dependencies among assumptions using a dependency network in order to realize DDB. We also need to store dependencies among choices and features to realize DDB for FUG unification.

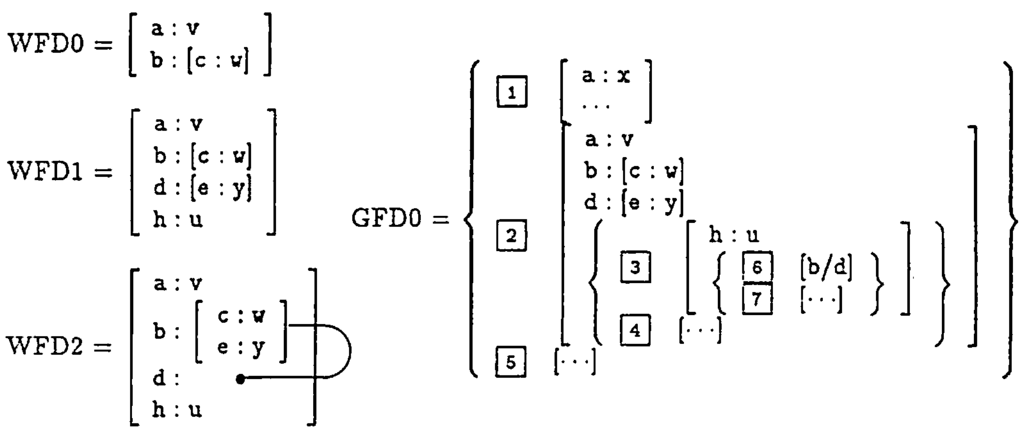
The system constructs a dependency network incrementally during the initial generation process and updates it during the regeneration process. For example, when the unification shown in Fig. 3 occurs, the system constructs the network shown in Fig. 4. Assume the system first try to unify WFD0 with GFD0. Since alternative $\boxed{1}$ is not unifiable with WFD0, the system tries the next alternative $\boxed{2}$. The first three features (a, b, and d) in $\boxed{2}$ are unifiable. WFD1 shows a snapshot of the instance when features a, b and d in alternative $\boxed{2}$ and h in $\boxed{3}$ have been unified.

A dependency network contains two kinds of nodes, feature nodes and choice nodes, which are linked with one another. A feature node corresponds to a feature in a WFD and a choice node represents which alternative was chosen at a choice point. In Fig. 4. feature nodes are denoted by f(_,_) and choice nodes are denoted by c(_,_).

Each time the system succeeds in unifying a disjunction in a GFD with a WFD, the system creates a choice node to store information about the choice. A choice node consists of the path from the root of a WFD to a constituent on which unification is performed, and the identifier of a chosen alternative of a disjunction in a GFD. At the same time, the system creates feature nodes, each of which corresponds to a feature that is newly added to the WFD by the choice. A feature node consists of the path from the root of a WFD to itself and its value. Furthermore, the system creates arcs between the choice node and the feature nodes. These arcs represent justification in terms of JTMS. The arcs are created as follows:

[3]To generate a sentence from this resultant WFD, a further process called "linearlization" is necessary, which we do not describe here.

1. Create an arc from a feature to a choice, if the feature was already present in both the GFD and the WFD before unification.
   If this feature is changed in the revision process, the validity of the choice needs to be checked again. Therefore this feature can be seen as a justification of the choice. In Fig. 3, WFD0 and alternative $\boxed{2}$ share features a and c before unification, and therefore arcs are created between feature a and choice $\boxed{2}$, and between feature c and choice $\boxed{2}$. They are denoted by arc (1) and (2) in Fig. 4.
2. Create an arc from a choice to each feature that is newly added to the WFD by choosing the alternative.
   If the choice is changed, the validity of these features should be checked again. Therefore this choice can be seen as a justification of these features. In Fig. 3, feature e in WFD1 is newly introduced by unifying WFD0 and choice $\boxed{2}$. So the arc is created between feature e and choice $\boxed{2}$, which is denoted by arc (3). Arc (5) is also created in the same manner.
3. Create an arc from a choice to each of its daughter choice(s).
   For instance, alternative $\boxed{2}$ includes a disjunction that has two alternatives $\boxed{3}$ and $\boxed{4}$. If the system chooses $\boxed{3}$ after $\boxed{2}$, the validity of choice $\boxed{3}$ is suported by choice $\boxed{2}$. Thus it is necessary to create an arc between these choices, which is denoted by arc (4).



WFD0 = [a : v, b : [c : w]]
WFD1 = [a : v, b : [c : w], d : [e : y], h : u]
WFD2 = [a : v, b : [c : w, e : y], d : •, h : u]
GFD0 = { 1 [a : x, ...]; 2 [a : v, b : [c : w], d : [e : y], { 3 [h : u, { 6 [b/d]; 7 [· · ·] }]; 4 [· · ·] }]; 5 [· · ·] }

Fig. 3 An example of unification

$$\begin{tikzcd}
\texttt{f([a],v)} \arrow[dr, "(1)"'] & & \texttt{f([b,c],w)} \arrow[dl, "(2)"] \\
& \texttt{c([ ],}\boxed{2}\texttt{)} \arrow[dl, "(3)"'] \arrow[dr, "(4)"] & \\
\texttt{f([d,e],y)} & & \texttt{c([ ],}\boxed{3}\texttt{)} \arrow[dl, "(5)"] \\
& \texttt{f([h],u)} &
\end{tikzcd}$$

Fig. 4 An example of dependency network

In each revision cycle, the system first identifies a culprit choice by referring to the revision rules. Then, the system removes all the features justified, directly or indirectly, by that choice from the current WFD. For example, if choice $\boxed{2}$ is changed, the system will remove features d: [e:y] and h:u from WFD1. The important point to note is that the system preserves all the features and the choices that are independent of the changed choice[4]. In the regeneration process, the system tries another alternative at the culprit choice point and resumes unification skipping the choices that have already been decided and still stay valid. Thus, our method prevents the system from unnecessary recomputation in the regeneration process. In this respect, our method is significantly different from chronological backtracking.

[4]This point is not shown clearly in the example.

3 Implementation

We use Prolog to implement the system, since Prolog intrinsically performs unification operation which makes it easier to realize unification of FDs. In addition, logical variables can be used as a versatile device in constructing dependency networks.

Our unification algorithm is based on Gazdar and Mellish's [6]. In their algorithm, an FD is represented by a Prolog list of feature-value pairs whose tail is an unbound variable. For example, WFD0 in Fig. 3 is represented as

```
[a:v,b:[c:w|_]|_].
```

Here a colon (:) is defined as a Prolog operator which conjoins a feature and its value.

Given FDs in this data structure, a Prolog predicate to unify two FDs is defined as follows:

```
unify(FD,FD) :- !.
unify([Feature:Value|FD1],FD) :-
    pathval(FD,Feature,Value,FD2),
    unify(FD1,FD2).

pathval([Feature:Value1|FD],Feature,Value2,FD) :-
    !,unify(Value1,Value2).
pathval([Feature1|FD1],Feature,Value,[Feature1|FD2]) :-
    pathval(FD1,Feature,Value,FD2).
```

Predicate pathval finds the value Value of a feature Feature in the FD given in the first argument, returning the remainder of the FD without the feature-value pair Feature:Value in the fourth argument.

Altough our unification is principlly the same as Gazdar and Mellish's, the data structure is slightly different. Since we need a dependency network to control backtracking, the data structure of FDs includes pointers to the network. In the following subsections, we explain the data structure of FDs, followed by descriptions of the algorithm for network construction, backtracking, and regeneration.

3.1 Data Structure

Fig. 5 illustrates the data structure of WFD0 shown in Fig. 3. Here the enclosure of a pair of square brackets denotes a Prolog cons cell. A vertical bar separates CAR and CDR of the cons cell. WFD0 is enclosed by the dashed box.

An FD is represented by a structure:

$fd(cycle\_id, list\_of\_features)$.

Cycle_id is an identifier of revision cycle which is updated in every revision cycle. This is used to avoid unnecessary recomputation in backtracking (see subsection 3.3). As in Gazdar and Mellish's algorithm, the tail of *list_of_features* is always an unbound variable. which is denoted by an underscore ("_") in Fig. 5. A feature value is either an atomic value or an FD. An atomic value is represented the a structure:

$atom(value, f(state, descendants))$.

Value denotes an atomic feature value and structure *f* denotes a feature node of a dependency network. Note that each atomic value has a feature node as its own argument. Thus, in our data structure. the FDs and the dependency network are integrated into a single structure. The first argument of a feature node, *state*, represents the feature's state. It is an unbound variable as long as the feature is valid. When the state propagation invalidates the feature, its *state* is bound to a special constant "*out*". The second argument *descendants* is a list of the nodes justified by this feature.

Choice nodes are indexed by another data structure which we call "choice history." The data structure of the choice history is defined as follows, where a choice node is represented by structure *c*:

$choice\_history ::= history(choices, constituent\_history)$.
$choices ::= [choice\_node|choices]|[]$.
$constituent\_history ::= [label : choice\_history|constituent\_history]|[]$.
$choice\_node ::= c(choice\_id, state, antecedents, descendants)$.

For example, in the case of the generation process shown in Fig. 2, the system constructs the following choice history.

```
history([c([1],_,_)],
        [prot:history([c([2],_,_)],[ ]),
         verb:history([c([3],_,_)],[ ]),
         goal:history([c([2],_,_)],[ ])]).
```

With the choice history, the system can get efficient access to a choice node by specifying the path of a constituent with which the choice is associated, and the identifier of this particular choice (*choice_id*). *Antecedents* is a list of states of choice nodes (see subsection 3.3).

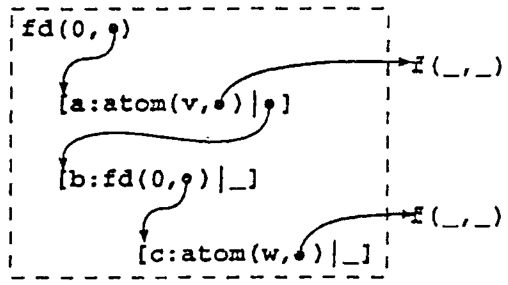


Fig. 5 Data structure of WFD0 from Fig. 3

3.2 Network construction

As described in subsection 2.2, the dependency network is updated each time an alternative is chosen. Fig. 6 shows a Prolog data structure corresponding to WFD1 from Fig. 3 and the dependency network from Fig. 4. WFD1 is enclosed by the dashed box, while the dependency network is shown outside the dashed box. Each atomic value in the WFD and its own feature node in the network are connected by a variable binding, and the dependency arcs in the network are realized by variable bindings as well. Each of the variable bindings (1) through (5) in Fig. 6 corresponds to the dependency arc that has the same number in Fig. 4.

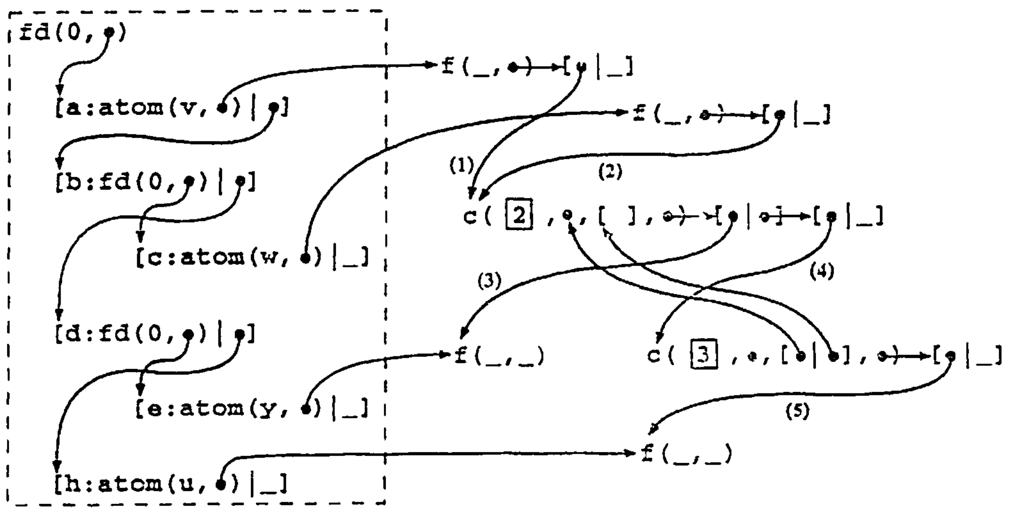


Fig. 6 Data structure of WFD1 from Fig. 3

The outline of unification algorithm is shown in Fig. 7 in the form of a Prolog program. Except for network handling, the most significant difference of our algorithm from Gazdar and Mellish's is that our algorithm does not treat two FDs symmetrically. In other words, features in a GFD can be added to a WFD, but not vice versa. GFDs are always kept intact during unification.

```
unify([],_,_):-!.
unify([Feature:Value|GFD],WFD,CNode):-!,
    featureVal(WFD,Feature,Value,CNode),
    unify(GFD,WFD,CNode).
unify([alt(Alternatives)|GFD),WFD,CNode):-!,
    unify_alt(Alternatives,WFD,CNode),
    unify(GFD,WFD,CNode).
unify([Feature1/Feature2|GFD],WFD,CNode):-
    conflate(Feature1,Feature2,WFD,CNode),
    unify(GFD,WFD,CNode).

featureVal(WFD,Feature,Value,choice(_,_,_,Descendants)):-
    var(WFD),
    atom(Value),!,
    WFD=[Feature:atom(Value,FNode)|_],
    tail_of([FNode|_],Descendants).
featureVal(WFD,Feature,Value,CNode):-
    var(WFD),!,
    WFD=[Feature:fd(_,FD)|_],
    unify(Value,FD,CNode).
featureVal([Feature:atom(Value1,f(_,Descendants))|_],Feature,Value,CNode):-!,
    Value=Value1,
    tail_of([CNode|_],Descendants).
featureVal([Feature:fd(_,FD)|_],Feature,Value,CNode):-!,
    unify(Value,FD,CNode).
featureVal([_|WFD],Feature,Value,CNode):-
    featureVal(WFD,Feature,Value,CNode).

unify_alt([Id:GFD|Alternatives],WFD,c(_,State,Antecedants,Descendants)):-
    unify(GFD,WFD,CNode),
    tail_of([CNode|_],Descendants),
    CNode = c(Id,_,[State|Antecedants],_).
unify_alt([_|Alternatives],WFD,CNode):-
    unify_alt(Alternatives,WFD,CNode).

conflate(Feature1,Feature2,WFD,c(_,_,_,Descendants)):-
    featureVal2(WFD,Feature1,Value1),
    featureVal2(WFD,Feature2,Value2),
    create_leaf_list(Value1,L0-L1),
    create_leaf_list(Value2,L1-[]),
    tail_of([conflation(L0)|_],Descendants),
    unify(Value1,Value2).
```

Fig. 7 Outline of unification algorithm

Predicate unify/3 assumes the first argument is a GFD, the second is a WFD, and the third is a dependency network to construct. unify/3 is defined in terms of four clauses. The first one is the termination clause for the case there is no more GFD fragment to apply. The second clause handles unification of a feature-value pair. This is actually performed by predicate featureVal/4. The third clause handles a disjunction in a GFD. A disjunction is represented by the following data structure:

$alt([id_1 : FD_1, id_2 : FD_2, \ldots])$.

The order of alternatives within a disjunction represents their preference. Predicate unify_alt/3 tries each choice one by one. The last clause of unify/3 handles conflation of features.

Given a feature and its value in a GFD, predicate featureVal/4 searches a WFD for the feature and unifies these two values. Unlike Gazdar and Mellish's algorithm, featureVal/4 keeps the GFD intact. featureVal/4 receives a WFD, a feature from a GFD and its value as an input in the first three arguments, and returns an updated choice node in its last argument.

featureVal/4 is defined by five clauses. The first two add into the WFD, the features that are contained in the GFD but not yet in the WFD. If the feature value is atomic, a dependency arc is created in line 15 since this feature is added to the WFD due to the current choice. Arcs (3) and (5) in Fig. 6 is created by this operation. Predicate tail_of/2 unifies its first argument with the last element of the second argument, which is supposed to be an unbound variable. If the WFD and the GFD share a unifiable feature, either the third clause or the fourth one is used. If the feature value is atomic, this feature can be a justification of the current choice. Therefore a dependency arc is created between the feature and the current choice. This is realized in line 21. Arcs (1) and (2) in Fig. 6 illustrate this case.

unify_alt/3 deals with a disjunction. When the system finds an alternative that is unifiable with the WFD, i.e. in the case of the first clause, the alternative comes to be justified by its mother choice. The arc to represent this justification is created in line 28. Arc (4) in Fig. 6 is an example of such an arc. Line 29 creates the antecedents list of the current choice, which is a list of backward pointers to the antecedent choices.

Predicate conflate/4 deals with conflation. First, the system extracts values of features to conflate in line 33 and 34. Predicate featureVal2/3 finds the value of the given feature in the WFD. Then create_leaf_list in lines 35 and 36 traverses the FD given in the first argument (i.e. the FD to conflate) in order to collect cons cells whose CDR is an unbound variable. This information is necessary to cancel the conflation in the revision process. Then the system creates a conflation node and puts it at the end of descendants (See line 37). The conflation node has a pointer to the list created by create_leaf_list. Finally, the system performs Gazdar and Mellish's symmetric unification unify/2 in line 38.

Suppose, for example, the system choose alternative $\boxed{6}$ and conflate features b and d in Fig. 6. In this case, the leaf node under feature b is [c:atom(w,f(_,•))|_] and that under d is [e:atom(y,f(_,•))|_]. Then the system creates a conflation node conflation with the pointers to these two nodes (arc (6) and (7)) as illustrated in Fig. 8. Also, the choice node corresponding to choice $\boxed{6}$ points to the conflation node (arc (8)).

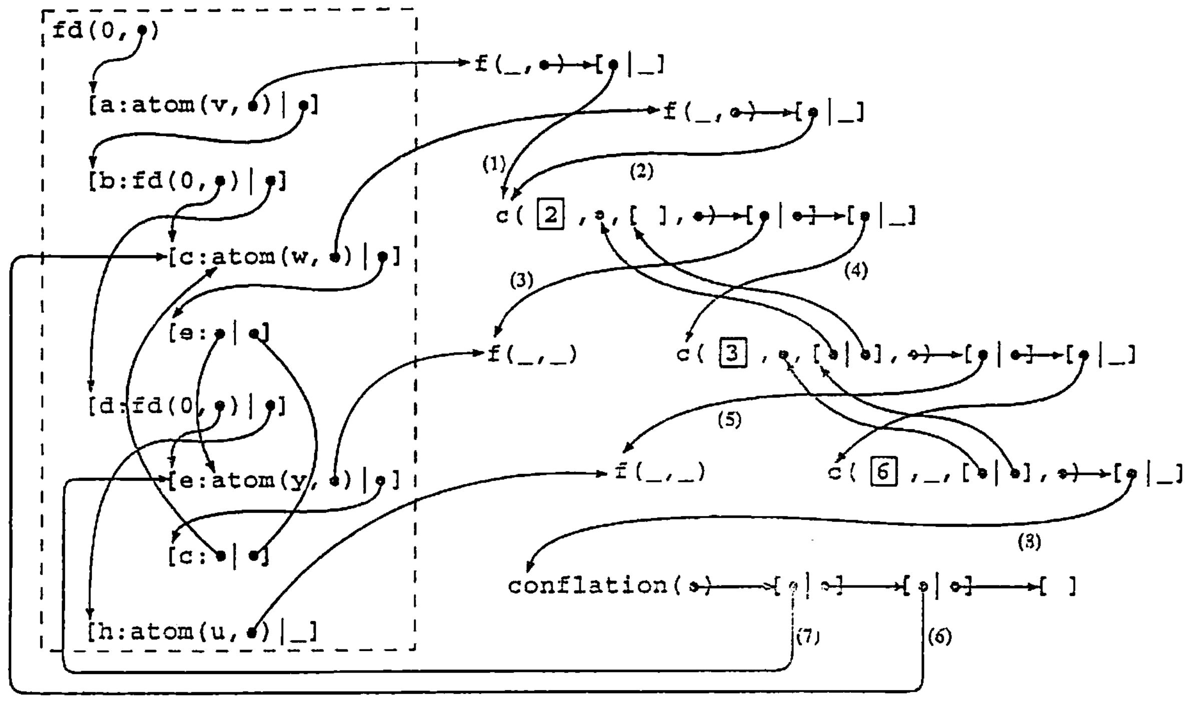


Fig. 8 Data structure of WFD2 in Fig. 3

An important point to note about this algorithm is as follows. In order to choose an alternative at a choice point, the system refers to features in the WFD that support the choice (e.g. [a:atom(v,•)|•] in the case of choice $\boxed{2}$ in Fig. 6). This access would be necessary even though the system did not construct the network. In our algorithm, since the system creates dependency arcs (e.g. arc (1)) simultaneously with the access to these features, the overhead of network construction can be reduced. Similarly, the system can efficiently create new feature nodes (e.g. [e:atom(y,•)|_] in Fig. 6) and their justification arcs (e.g. arc (3)) simultaneously with unification of these feature.

3.3 Backtracking

In each revision cycle, the revision planner suggests a culprit choice that should be changed in order to solve a problem detected by the evaluator. After that, in backtracking, the system removes all the features dependent on the culprit choice from the WFD. In our method, since dependencies are represented as variable bindings, the system has only to traverse these pointers to find the features to remove. This process, therefore, should be efficient. In backtracking, the system performs the following procedures.

1. Search the choice history for the choice node corresponding to the culprit choice.
2. Bind the state of the culprit choice to "*changed*".
3. Traverse the dependency network via descendant links starting from the culprit choice node, and perform the following procedures on each node.
   (a) If the node is a feature node, bind the state of the node to "*out.*"
   (b) If the node is a choice node, bind the state of the node to "*out*" and also bind each variable in its antecedents list to "*unknown.*"
   (c) If the node is a conflation node, substitute the CDR of each node pointed by the conflation node with an unbound variable. This substitution is performed destructively and cancels the conflation. This is realized by the Prolog built-in predicate setarg[5].
4. Remove all the features that are marked "*out*" from WFD by destructive substitution.

Suppose, for example, choice $\boxed{3}$ in Fig. 8 be identified as a culprit choice. The state of the node c($\boxed{3}$,...) is bound to "*changed*" and traverse starts from this node. The state of the antecedents, node c($\boxed{2}$,...) in this case, is changed to "*unknown*" and the state of the descendants, node c($\boxed{6}$,...) is changed to "*out.*" State "*out*" means that the node is invalid. On the other hand, node N is "*unknown*" when all the nodes supporting N are valid but some of N's descendant choices are invalid ("*unknown*" can be marked only on choice nodes). Let us see this difference according to the current example. Fig. 9 shows the snapshot after step 3 in the above procedure. Choice $\boxed{2}$ is now *unknown* because its descendant choices $\boxed{3}$ and $\boxed{6}$ are invalid. However, choice $\boxed{2}$ will be valid again if the system finds another unifiable alternative instead of $\boxed{3}$ at the choice point in regeneration. Since such cases occur frequently, it would be better not to invalidate choices like $\boxed{2}$ for efficiency. Therefore, while state "*out*" propagates via descendant links, "*unknown*" does not.

Step 3(c) is illustrated by the example shown in Fig. 8 and 9. When the system binds the CDRs of the nodes pointed by the conflation node ([c:atom(v,f(_,•))|•] and [e:atom(y,f(_,_))|•]) to unbound variables, the network from Fig. 8 comes to be that from Fig. 9. In Fig. 9. conflation between features b and d has been canceled.

Finally, the system removes feature [h:atom(u,f(*out*,_))|_] by substituting the CDR of node [d:fd(0,•)|•] with an unbound variable. In this process, the system searches the WFD for all the invalid features. Since a WFD is a DAG in general. a naive algorithm would cause unnecessary duplication of traverse. The system avoids unnecessary traverse using *cycle_*(subsection 3.1). When the system comes across an FD whose *cycle_id* has not been updated, the system updates it and traverses the FD. The system skips the FDs whose *cycle_id* has been updated.

[5] setarg is available in SICStus Prolog.

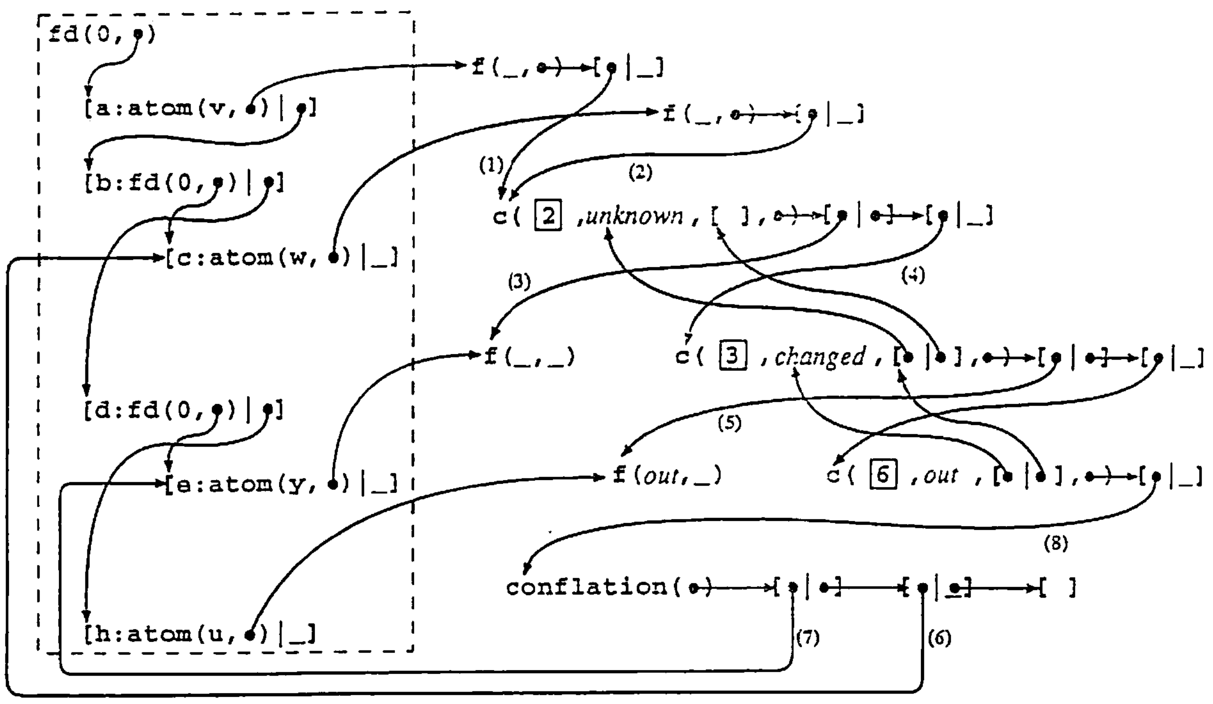


Fig. 9 The snapshot after step 3 in backtracking

3.4 Regeneration

After the system goes back to a culprit choice point and cancels the decisions dependent on that choice as described in the previous subsection, the system resumes unification from the culprit choice point to generate another draft. Unlike the initial generation process, the system now refers to the choice history and avoids unnecessary recomputation. At each choice point, the system performs the following operation.

1. If the choice point has a choice whose state is valid, accept the choice without recomputation.
2. If the choice point has a choice whose state is "*unknown*", try the choice again.
3. If the choice point has a choice whose state is "*out*" or has no associated choice node, try all the alternatives one by one.

For example, when the system starts regeneration from the situation shown in Fig. 9;

- the system does not try alternative $\boxed{1}$ again because the choice point including alternative $\boxed{1}$ has an *unknown* choice node (alternative $\boxed{2}$), and thus the system knows the unification with $\boxed{1}$ fails;
- the system tries alternative $\boxed{2}$ again because its state is "*unknown*";
- the system skips the unification of the first three features in $\boxed{2}$ because the system knows these features are already shared with the current WFD.

4 Experiment

In this section we show the result of a preliminary experiment to demonstrate the efficiency of our implementation. In the experiment, we use a small experimental Japanese grammar consisting of 39 grammatical disjunctions and 38 lexical entries. Fig. 10 shows an input WFD represented in terms of rhetorical structure [15]. In this figure, features n and s mean "nucleus" and "satellite" respectively. The drafts generated from this input are shown in Fig. 11. Draft (1) is the first draft, in which both the propositions keep and located are realized by one sentence. This choice, however, results an unexpected long noun modifier "*tonari-no tatemono-no 4kai-no ichiban oku-ni aru* (which is located in the most inner part on the fourth floor of the next building)." The evaluator detects this problem and solves it by changing the choice that realizes the two propositions as one sentence, generating draft (2) instead. Note that the first sentence in draft (1) is split into two sentences in draft (2).

```
[ cat : segment
  pred : elaborate#1
        [ pred : elaborate#2
              [ pred : keep#1
          n : [ topic : document#1
              [ agt : [ pred : library#1 ]
  n :         [ obj : [ pred : document#1 ] ]
              [ pred : located#1
          s : [ self : [ pred : library#1 ]
              [ value : [ pred : location#1 ] ] ]
        [ pred : elaborate#3
              [ pred : close_period#1
          n : [ topic : library#1
              [ self : [ pred : library#1 ]
  s :         [ value : [ pred : summer_vacation#1 ] ]
              [ pred : period#1
          s : [ self : [ pred : summer_vacation#1 ]
              [ from : [ pred : july21#1 ]
              [ to : [ pred : august31#1 ] ] ] ]
```

Fig. 10 An example of input structure

Draft (1) *sono syorui-wa tonari-no tatemono-no 4 kai-no itiban oku-ni aru siryôsitu-ni hokansareteimasu. tadasi, sirôsitu-wa 7 gatu 21 niti-kara 8 gatu 31 niti-made-no natuyasumi-no aida-wa tukaemasen.*
(That document is kept in the document room which is located in the most inner part on the fourth floor of the next building. The document room will be closed during the summer vacation from July 21 to August 31.)

Draft (2) *sono syorui-wa siryôsitu-ni hokansareteimasu. siryôsitu-wa tonari-no tatemono-no 4 kai-no itiban oku-ni arimasu. tadasi, 7 gatu 21 niti-kara 8 gatu 31 niti-made-no natuyasumi-no aida-wa tukaemasen.*
(That document is kept in the document room. The document room is located in the most inner part on the fourth floor of the next building. The document room will be closed during the summer vacation from July 21 to August 31.)

Fig. 11 An example of drafts

We compared our implementation with chronological backtracking, and the bk-class framework proposed by Elhadad [4]. If we apply bk-class to revision, bk-class plays a similar role to our revision rules. A bk-class is a pair of a name of feature and choice points that may cause failure of unification of the feature. When unification fails at a certain bk-class feature, the system goes directly back to the latest bk-class choice point, ignoring all the intermediate choices. The significant difference between the bk-class framework and our DDB framework can be summarized in the following three respects.

- In our method, a revision rule can identify candidates of a certain culprit choice by specifying both the path of a constituent with which that choice is associated, and the identifier of that choice. On the other hand, a bk-class identifies candidates of a culprit choice only by specifically the identifier of them. Since a revision rule describes candidates of a certain culprit choice more specificly than a bk-class does, the frequency of backtracking in revision process in our method would be lower than that in Elhadad's.

- In the bk-class framework, the system cancels all the chronologically intermediate decisions when backtracking. Therefore, the system may repeat the same computation as that done in the previous generation process. Since we maintain a dependency network, we can reuse the results of the computation that are independent of the change of the culprit choice.
- In this paper, we assume that a revision rule identify a culprit choice by specifing its identifier. But it is also likely that one would like to describe a solution of a problem by specifying a particular feature to remove from a WFD (a "culprit feature" as it were). In our method, since the system maintains the dependencies between the choices and the features, the system can identify the choice to change when given a culprit feature; therefore, a solution can also be described in terms of a culprit feature. In this sense, our framework is more general than the bk-class framework.

Table 1 CPU time to generate drafts

draft	1	2	3	···	17	Total
chronological backtracking	1.18	0.25	0.90	···	1.17	11.80
bk-class	1.18	1.15	1.36	—	—	3.70
DDB	1.23	0.91	—	—	—	2.14

(SONY NEWS 3860; [sec])

In this experiment, we do not take into account the cost of evaluating drafts and revision planning, but focus on the cost of network management. Table 1 shows CPU time to generate draft (2) from Fig. 11 for each case. Our framework solves the problem mentioned above by only a single backtrack; the bk-class framework requires two, while chronological backtracking requires seventeen. In general, since a bk-class does not identify candidates of a culprit choice so specifically as a revision rule, backtracks tend to occur more frequently. In this example, the bk-class algorithm generates draft (2′) below as the second draft, where the second sentence in draft (1) is split into two sentences. However, it does not solve the problem in the first sentence.

Draft (2′) *sono syorui-wa tonari-no tatemono-no 4 kai-no itiban oku-ni aru siryôsitu-ni hokansareteimasu. tadasi, sirôsitu-wa natuyasumi-no aida-wa tukaemasen. natuyasumi-wa 7 gatu 21 niti-kara 8 gatu 31 niti-made desu.*

(That document is kept in the document room which is located in the most inner part on the fourth floor of the next building. The document room will be closed during the summer vacation. The summer vacation is from July 21 to August 31.)

In addition, note that the CPU time to generate the second draft in our framework is less than the CPU time to generate the third draft in the bk-class framework. This is because the system can avoid unnecessary recomputation in the regeneration process.

Table 2 CPU time for network management

	CPU time [sec]	%
network construction	0.05	4.2
backtracking	0.04	3.4
total	0.09	7.6

Table 2 shows CPU time for network management. The CPU time for "network construction" denotes the cost of network construction in the initial generation process. It is calculated by the difference between the CPU time needed to generate the first draft in DDB and that in the bk-class method. In this experiment, the CPU time for the network construction is only 4.2 percent of that of the initial generation. The CPU time for "backtracking" denotes the cost of state propagation in the dependency network and undoing unification. This is calculated by the difference between:

- (for bk-class) the time for the completion of backtracking to the culprit choice after generating the second draft.
- and (for DDB) the time for the completion of state propagation and removing all the features marked "*out*" after generating the first draft.

The difference means the overhead of network management during backtracking in our system, which costs only 3.4 percent of the initial generation. The result shows that our method is worth introducing.

5 Conclusion

In text generation, various kinds of choices need to be decided. Since these choices depend on one another, the one-path generation framework makes it difficult to design a set of rules that can make appropriate decisions. Introducing the revision process can be a solution to this problem. Our previous paper proposed a framework in which revision is realized as dependency-directed backtracking (DDB).

In this paper, we proposed a method to realize DDB for text generation using Functional Unification Grammar (FUG) in Prolog. FUG is suitable for DDB because FUG is based on unification with flexible ordering of decisions.

In our method, the system constructs a dependency network to maintain dependencies among choices and features. As shown in subsection 3.2 and 3.3, the system realizes DDB in the following senses.

- The system directly backtracks to a culprit choice point by referring to the revision rules.
- The system reuses the previous result if possible in the regeneration process by referring to the dependency network.

Thus, the DDB mechanism enables the system to traverse the search space efficiently. However, we also need to consider the overhead of network management when we realize DDB in the framework of JTMS. We proposed a method to realize an efficient DDB by integrating a WFD and a network into a single data structure. In this data structure, dependencies are represented as bindings of logical variables, and the update of the network is realized with destructive substitutions. According to a preliminary experiment, the cost of network management is less than 8 percent of the total cost of the initial generation. This result shows that our method is worth introducing.

In addition to its efficiency, our framework allows grammar writers to specify a culprit choice in terms of features to remove as well as the identifier of the choice. This is possible because our method keeps which choice introduced the feature in question. Our framework provides a more general means to control the search process than that without maintaining dependencies such as the bk-class framework.

Our method may be applicable to other applications with DDB. However, we are not claiming that we have provided a general solution to the problems of implementing DDB. Our solution works efficiently due to the fact that features and choices never have disjunctive justifications in FUG.

In this paper, we mentioned neither on evaluating drafts nor on revision planning. To realize the overall generation system, we need further research on both the evaluation criteria and the revision rules. These issues depend on an actual grammar. In this context, we are now developing a fairly large Japanese grammar based on the systemic-functional theory. We believe the grammar will provide us with useful information to develop good evaluation criteria and revision rules.

Acknowledgements

We would like to thank the reviewers for their helpful comments on the early version of this paper.

References

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

[2] J. de Kleer, K. Forbus, and D. McAllester. Truth maintenance systems. the eleventh International Joint Conference on Artificial Intelligence, tutorial program, 1989.

[3] M. Elhadad. FUF: the universal unifier - user manual, version 5.0. Technical Report CUCS-038-91, Columbia University, 1991.

[4] M. Elhadad and J. Robin. Controlling content realization. In R. Dale, E. Hovy, D. Rösner, and O. Stock, editors, *Aspects of Automated Natural Language Generation*, pp. 89–105. Springer-Verlag, 1992. Lecture Notes in Artificial Intelligence Vol. 587.

[5] M. Emele, U. Heid, and R. Zajac. Interactions between linguistic constraints: Procedual vs. declarative approach. *Machine Translation*, Vol. 6, No. 4, 1991.

[6] G. Gazder and C. Mellish. *Natural Language Processing in Prolog.* Addison Wesley, 1989.

[7] M. A. K. Halliday. *An Introduction to Functional Grammar.* Edward Arnold, 1985.

[8] E. H. Hovy. *Generating Natural Language under Pragmatic Constraints.* Lawrence Erlbaum Associates, 1988.

[9] K. Inui, T. Tokunaga, and H. Tanaka. Text revision: A model and its implementation. In R. Dale, E. Hovy, D. Rösner, and O. Stock, editors, *Aspects of Automated Natural Language Generation*, pp. 215–230. Springer-Verlag, 1992. Lecture Notes in Artificial Intelligence Vol. 587.

[10] R. Kasper. Systemic Grammar and Functional Unification Grammar. In *Systemic Perspective on Discourse*, chapter 9, pp. 176–199. Ablex, 1987.

[11] M. Kay. Functional Unification Grammar: A formalism for machine translation. In *Proceedings of the International Conference on Computational Linguistics*, pp. 75–78, 1984.

[12] W. C. Mann. An overview of the Penman text generation system. In *Proceedings of the National Conference on Artificial Intelligence*, pp. 261–265, 1983.

[13] W. C. Mann and S. A. Thompson. Rhetorical Structure Theory: A theory of text organization. Technical report, USC-ISI, 1987.

[14] C. Matthiessen and J. Bateman. *Text Generation and Systemic-functional Linguistics: Experiences from English and Japanese.* Printer Publishers, 1991.

[15] K. R. McKeown and W. R. Swartout. Language generation and explanation. In M. Zock and G. Sabah, editors, *Advances in Natural Language Generation*, chapter 1, pp. 1–51. Ablex Publishing Corporation, 1988.