New Generation Computing, 14 (1996) 169-193
OHMSHA, LTD. and Springer-Verlag

Dependency-Directed Control of Text Generation Using Functional Unification Grammar

Kentaro, INUI, Takenobu, TOKUNAGA, and Hozumi, TANAKA
Department of Computer Science,
Tokyo Institute of Technology,
2-12-1 Ôokayama, Meguro, Tokyo 152, Japan.

Received 31 March 1994
Revised manuscript received 20 July 1994

Abstract In text generation, various kinds of choices need to be decided. In conventional frameworks, which we call *one-path generation frameworks*, choices are made in an order carefully designed in advance. In general, however, since choices depend on one another, it is difficult to make optimal decisions in such frameworks. Our approach to this issue is to introduce the revision process into the overall generation process. In our framework, revision of output texts is realized as dependency-directed backtracking (DDB). As well as Justification-based Truth Maintenance System (JTMS), we maintain dependencies among choices in a dependency network.

In this paper,we propose an efficient implementation of DDB for text generation using functional unification grammar (FUG). We use bindings of logical variables in Prolog and destructive argument substitutions to decrease the overhead of handling a dependency network. This paper describes the algorithm in detail and shows the results of preliminary experiments to demonstrate the performance of our implementation.

Keywords: Text Generation, Surface Generation, Revision, Dependency-Directed Backtracking, Functional Unification Grammar Prolog

§1 Introduction

In text generation, various kinds of choices need to be decided. In conventional frameworks, which we call *one-path generation frameworks*, those decisions are made in an order carefully designed in advance. In general, however, since choices depend on one another, it is difficult to make optimal decisions in such frameworks.

This issue has been discussed and several solutions have been proposed.[1,7,12] Our approach to this issue is to introduce the revision process into the overall generation process.[13] Our genration model is illustrated in Fig. 1. In this model, the generation process consists of the initial generation process and the revision process. The revision process is realized by repeated revision cycles, each of which consists of evaluation of the output text, revision planning, and regeneration.

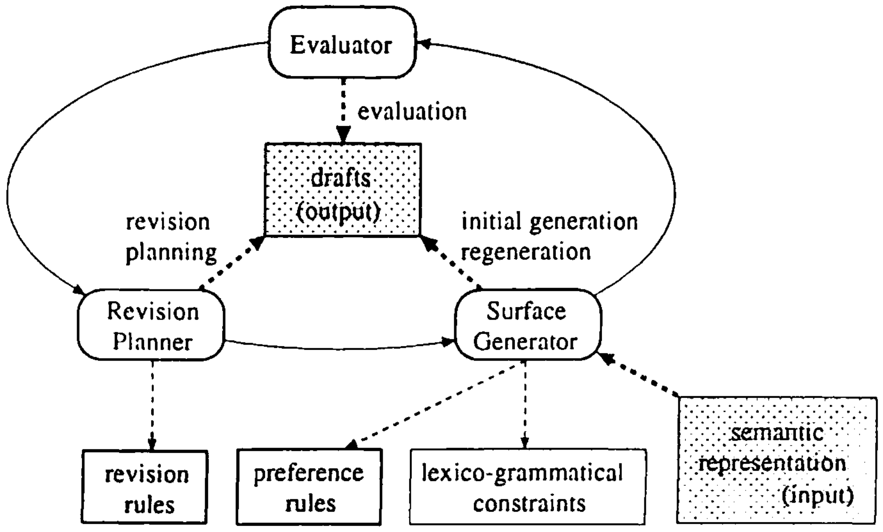

Fig. 1 Generation model with revision component.

At the present, we focus on surface generation*; therefore, we assume that inputs to the system be rhetorically organized semantic representations.[17] In the initial generation process and the regeneration process, the surface generator generates a text applying lexico-grammatical constraints and preference rules. The lexico-grammatical constraints provide a set of choice points and the alternatives for each of them. The preference rules are heuristics which assign preferences to those alternatives. The surface generator makes decisions according to the preference rules. In our model, however, decisions are tentative and may be changed in the revision process. Thus outputs from the surface generator are called *drafts* in this paper. A draft contains not only semantic information but also syntactic and lexical information. What should be noted here is that all the necessary decisions have been made at the end of the initial generation process. In the revision process, if the evaluator detects a problem in the current draft, the revision planner refers to revision rules to solve it. Revision rules are assumed to be heuristics that would suggest which choice should be changed to solve the problem. We call such a choice a *culprit choice*. In the regeneration process, the surface generator changes a culprit choice and generates another

* 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.[19].

draft.

A major difficulty in handling the interdependencies among decisions is that at some choice points the system could not make optimal decisions unless the system had the infomation about their effects on the following decisions and the final text. In one-path generation, therefore, the system would have to anticipate the future decisions to make the current decision; while, in our model, the evaluator can get access to all the information necessary for evaluating the decisions. This is because in the evaluation process all the necessary decisions have already been made.

Assuming that evaluation and revision planning be done successfully, the system would revise the current 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. In chronological backtracking the system would go back to the latest choice point, while our backtracking is *dependency-directed* in the following senses.

- The system directly goes back to a culprit choice point by referring to revision rules,
- 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 historty of decisions and their effects on the current draft. In the previous paper, we proposed a method to apply the framework of Justification-based Truth Maintenance System (JTMS)[3)] to 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 network handling in JTMS requires some computational overhead. We need an efficient implementation of network handling 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 in a dependency network are represented by bindings of logical variables, which facilitates traversing the network. In addition, we use destructive argument substitution to make efficient both updating networks and changing drafts.

So far we have fully implemented the surface generator, while implementation of the evaluator and the revision planner have not been completed yet. Concerning criteria for evaluation of drafts, we have so far considered structural complexity and structural ambiguity of each sentence.[13)] They are mainly matters of decisions on stylistic choices, which are free of constraints specified in inputs.[20)] Obviously we still need to discuss what kind of criteria we should consider and also how to develop effective revision rules (see Ref. 22)). But the discussion on these issues would not be so straightforward since it is very likely that they keenly depend on actual descriptions of lexico-grammatical knowledge. In this context we are now developing an extensive grammar of Japanese, which is expected to provide us with a lot of useful suggestions on these issues.

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 represnting linguistic knowledge. In Section 3, we propose an efficient implementation of the FUG unification with DDB. In Section 4, we show the results of preliminary experiments to demonstrate the performance of 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 (SFG).[11] SFG has desirable features for text generation and has been used by several text generation systems.[5,9,16,18,21] This is mainly because of the following respects.[18]

- SFG organizes the linguistic information based on the "paradigmatic" perspective, which makes the choices in generation explicit.
- Text generation is often a goal-oriented task motivated by some goals of the speaker/writer; therefore, considering functionality of language is indispensable in text generation. SFG describes the linguistic constraints in terms of functionality.

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

As a computational tool to implement SFG, we use functional unification grammar (FUG),[15] which has good properties for implementation of DDB in the following respects.

- SFG can be straightforwardly represented in the FUG notation.[14] For example, each choice in SFG corresponds to a disjunction in FUG, and conflation in SFG* corresponds to unification in FUG.
- Unification is a basic operation of FUG, and thus description of FUG do not constrain the order of decisions. This property is desirable for our model since the decision order may be changed during the revision process.**

In this section, we first briefly describe taxt generation using FUG, and then

* Conflation is an important device to realize multi-functionality of SFG. In SFG, a constituent can have more than one functions. 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, multi-functionality can be realized by conflating (i.e. unifying) constituents: a constituent labeled Agent and another one labeled Theme in this example.

** Note that DDB imposes changes of the decision order in general.

explain the control of unification by DDB.

2.1 Text Generation Using FUG

In the FUG formalism, a grammar is described in the form of functional description (FD), which we call *grammar functional description* (GFD) in this paper. Inputs to the system are also represented as FDs. To generate a rext, the system unifies an input FD with a GFD. As unification proceeds, features in the GFD are added to the input FD. We refer to such a FD, which is to be enriched, by a *working functional description* (WFD). Figure 2 shows an example of GFD and WFDs. In this paper, following the conventional notation, a disjunction is denoted by a pair of curly brackets { }. WFD_0 is an input FD, which is to be a sentence "*Jone loves Mary.*"

In our system, unification of a WFD and a GFD proceeds in the top-down and depth-first manner. In most cases of text generation, the top-down

$$\text{GFD} = \left\{ \begin{array}{l} \boxed{1} \begin{bmatrix} \text{rank: s} \\ \text{senser: [rank: np]} \\ \text{process: [rank: vp]} \\ \text{pattern: [senser, process, range]} \end{bmatrix} \\ \boxed{2} \begin{bmatrix} \text{rank: np} \\ \text{n: [rank: noun]} \\ \text{pattern: [n]} \end{bmatrix} \\ \boxed{3} \begin{bmatrix} \text{rank: vp} \\ \text{v: [rank: verb]} \\ \text{pattern: [v}\cdots\text{]} \end{bmatrix} \end{array} \right\}$$

$$\text{WFD}_0 = \begin{bmatrix} \text{rank: s} \\ \text{senser: [n: [lex: John]]} \\ \text{range: [n: [lex: Mary]]} \\ \text{process: [v: [lex: love]]} \end{bmatrix}$$

$$\text{WFD}_1 = \begin{bmatrix} \text{rank: s} \\ \text{senser: } \begin{bmatrix} \text{rank: np} \\ \text{n: [lex: John]} \end{bmatrix} \\ \text{range: } \begin{bmatrix} \text{rank: np} \\ \text{n: [lex: Mary]} \end{bmatrix} \\ \text{process: } \begin{bmatrix} \text{rank: vp} \\ \text{v: [lex: love]} \end{bmatrix} \\ \text{pattern: [senser, process, range]} \end{bmatrix}$$

$$\text{WFD}_2 = \begin{bmatrix} \text{rank: s} \\ \text{senser: } \begin{bmatrix} \text{rank: np} \\ \text{n: } \begin{bmatrix} \text{lex: John} \\ \text{rank : noun} \end{bmatrix} \\ \text{pattern: [n]} \end{bmatrix} \\ \text{range: } \begin{bmatrix} \text{rank: np} \\ \text{n: } \begin{bmatrix} \text{lex: Mary} \\ \text{rank : noun} \end{bmatrix} \\ \text{pattern: [n]} \end{bmatrix} \\ \text{process: } \begin{bmatrix} \text{rank: vp} \\ \text{v: } \begin{bmatrix} \text{lex: love} \\ \text{rank : verb} \end{bmatrix} \\ \text{pattern: [v]} \end{bmatrix} \\ \text{pattern: [senser, process, range]} \end{bmatrix}$$

Fig. 2 An example of GFD and WFD.

and depth-first search strategy will work efficiently. Since a text generation system does not have to give possible outputs all at once and in addition our framework allows the system to revise drafts, depth-first search is suitable for our purpose.

In the example in Fig. 2, the GFD has three alternatives at the top level. Here only the first alternative can unify with WFD_0, which produces WFD_1. Then the system tries to recursively unify each constituent that is listed in the value of feature pattern. In this case, senser is unified first with the GFD, the second alternative being chosen. After unifying senser with the GFD, the system moves to the next constituent process. The final result is WFD_2 in Fig. 2.* If unification fails during the process, the system goes back to the latest disjunction to try another alternative. Like this, the system tries alternatives one by one at each choice point, which means that the order of the alternatives represents a static preference order of them. Therefore, a GFD describes both lexico-grammatical constraints and preferences among the alternatives (i.e. the preference rules in Fig. 1).

2.2 DDB in FUG Unification

JTMS maintains the dependencies among assumptions using a dependency network in order to realize DDB. Similarly, we need to maintain dependencies among choices and features to realize DDB for FUG unification.

The system incrementally constructs a dependency network during the initial generation process and updates it during the regeneration process. For example, when the unification shown in Fig. 3 is performed, the system constructs the network shown in fig. 4. First, the system tries to unify WFD_0 with alternative $\boxed{1}$. Since this unification fails, the system tries the next alternative $\boxed{2}$.

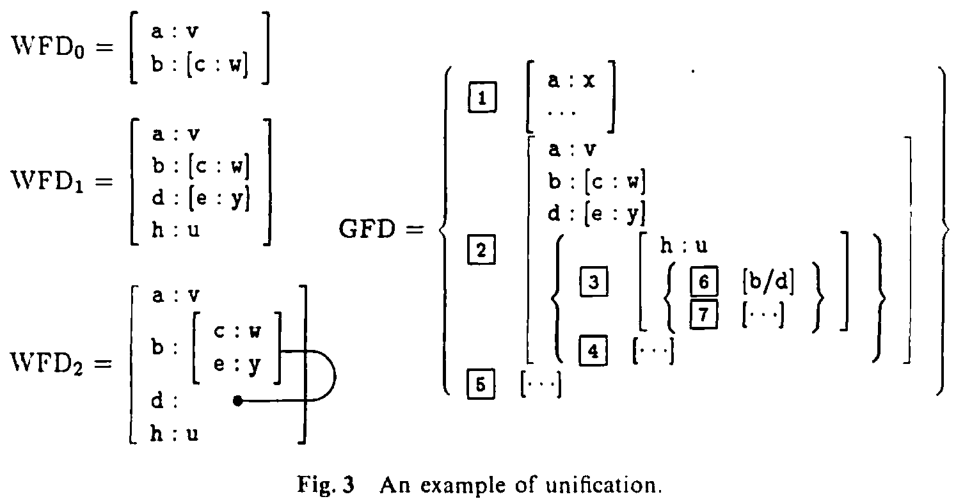
WFD0 = [a : v, b : [c : w]]
WFD1 = [a : v, b : [c : w], d : [e : y], h : u]
WFD2 = [a : v, b : [c : w, e : y], d :, h : u]
GFD =
1 [a : x ...]
2 [a : v, b : [c : w], d : [e : y]]
3 [h : u]
6 [b/d]
7 [· · ·]
4 [· · ·]
5 [· · ·]

Fig. 3 An example of unification.

* To generate a sentence from a final WFD, a further process, called *linearization*, is necessary. In linearization, the system traverses the WFD to retrieve all the lexical specifications and the pattern features. Those pieces of information is sufficient to generate the output sentence.

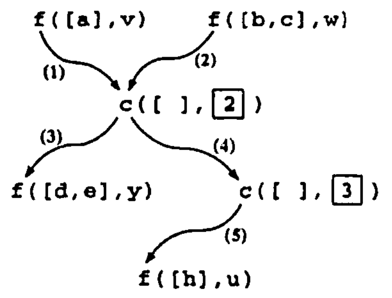

Fig. 4 An example of dependency network.

The first three features (a, b, and d) in $\boxed{2}$ are unifiable. WFD$_1$ shows a snapshot where features a, b and d in alternative $\boxed{2}$ and h in $\boxed{3}$ have been unified.

A dependency network contains two types 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 the information about the choice. A choice node consists of the path from the root of the WFD to the constituent on which the choice is made, and the identifier of the chosen alternative. At the same time, the system creates feature nodes each of which corresponds to a feature 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. Arcs represent justifications in terms of JTMS. Arcs are created in the following procedure.

(1) Crate 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, WFD$_0$ and alternative $\boxed{2}$ had already shared 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 are (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 WFD$_1$ is newly introduced by unifying WFD$_0$ and alternative $\boxed{2}$. So an arc is created between feature e and choice $\boxed{2}$, which is denoted by arc (3). Arc (5) is also created analogously.

(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 supported by choice $\boxed{2}$. Thus it is necessary to create an arc between these choices, which is denoted by arc (4).

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 WFD_1. The important point to note is that the system preserves all the features and the choices that are independent of the culprit choice.* In the regeneration process, the system tries another alternative at the culprit choice point, and resumes unification, skipping the choice points where the decisions have already been made 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.

§3 Implementation

We use Prolog to implement the system, since Prolog's depth-first search can be straightforwardly applied to our depth-first generation. In addition, logical variables can be used as a versatile device in constructing dependency networks.

Our unification algorithm is based on Eisele and Dörre's.[4] In their algorithm, an FD is represented as a Prolog list of feature-value pairs whose tail is an unbound variable. For example, WFD_0 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 can be defined as follows, which will be referred to by the *basic unification algorithm***:

```
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).
```

* This point is not shown clearly in the example.

** This is based on the program presented by Gazder and Mellish.[10]

Predicate pathval finds the value Value of a featue 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.

Although our unification is basically the same as the above, 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 the algorithms for network construction, backtracking, and regeneration.

3.1 Data Structure

Figure 5 illustrates the data structure of WFD_0 shown in Fig. 3. Here a pair of square brackets denotes a Prolog cons cell. A vertical bar separates the head and the tail of a cons cell. WFD_0 is enclosed by the dashed box, while the feature nodes of the dependency network are shown outside the box.

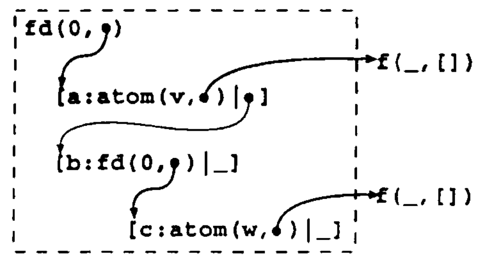

Fig. 5 Data structure of WFD_0 in Fig. 3.

An FD is represented as a structure:

fd(cycle_id, list_of_features).

Cycle_id is the identifier of a revision cycle, which is updated in every revision cycle. This is used to avoid unnecessary recomputation in backtracking (see Section 3.3). As in the basic unification algorithm, the tail of *list_of_featurers* 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 as a structure:

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

Value denotes an atomic feature value and structue *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 that feature.

Choice nodes are indexed by another data structure which we call a *choice history*. The data structure of a choice history is defined as follows.

choice_history::= *history*(*choices*, *constituent_history*).
choices::= [*choice_node* | *choices*] | [].
constituent_history::= [*label*: *choice_history* | *constituent_history*] | [].

A choice node is represented as a structure *c*:

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($\boxed{1}$,_,_)],
  [senser: history([c($\boxed{2}$,_,_)], []),
   process: history([c($\boxed{3}$,_,_)], []),
   range: history([c($\boxed{2}$,_,_)], [])]).

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

3.2 Network Construction

As described in Section 2.2, the dependency network is updated each time an alternative is chosen. Figure 6 shows a Prolog data structure corresponding to WFD$_1$ in Fig. 3 and the dependency network in Fig. 4. WFD$_1$ is enclosed by the dashed box, while the dependency network is shown outside the 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 also represented by variable bindings. Each of the variable bindings (1) through (5) in Fig. 6 corresponds to the dependency arc that has the same number in Fig. 4.

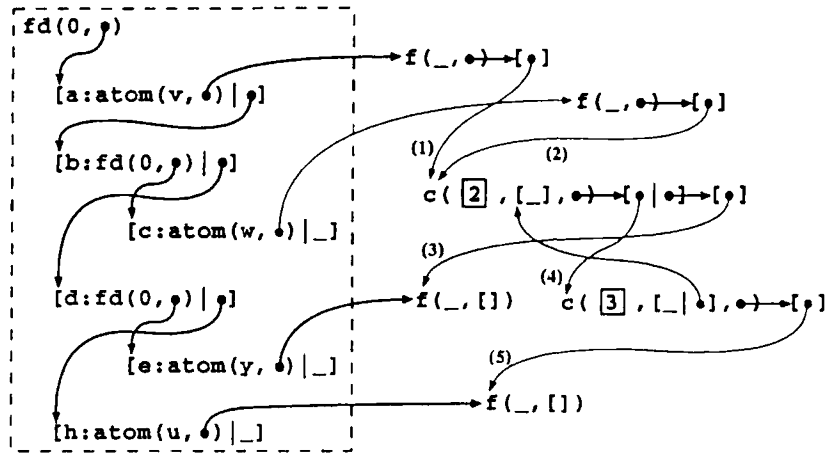

Fig. 6 Data structure of WFD$_1$ in Fig. 3.

An outline of the unification algorithm is shown in Fig. 7 in the form of a Prolog Program. Except for network handling, a significant difference of our algorithm from the basic unification algorithm is that our algorithm does not treat two FDs symmetrically. Features in a GFD may be added to a WFD, but not vice versa. GFDs are always kept intact during unification.

Predicate unify/3 assumes the first argument be a GFD, the second be a WFD, and the third be a dependency network to construct. unify/3 is defined in

```
 1: unify([],_,_) :-!.
 2: unify([Feature: Value|GFD], WFD, CNode) :-!,
 3:      feature Val(WFD, Feature, Value, CNode),
 4:      unify(GFD, WFD, CNode).
 5: unify([alt(Alternatives)|GFD), WFD , CNode) :-!,
 6:      unifyAlt(Alternatives, WFD, CNode),
 7:      unify(GFD, WFD, CNode).
 8: unify([Feature1/Feature2|GFD], WFD, CNode) :-
 9:      conflate(Feature1, Feature2, WFD, CNode),
10:      unify(GFD, WFD, CNode).

11: featureVal(WFD, Feature, Value, c(_, _, Descendants)) :-
12:      var(WFD),
13:      atom(Value), ! ,
14:      WFD=[Feature: atom(Value, FNode)|_],
15:      createArc(Descendants, FNode).
16: featureVal(WFD, Feature, Value, CNode) :-
17:      var(WFD), ! ,
18:      WFD=[Feature: fd(_, FD)|_],
19:      unify(Value, FD, CNode).
20: featureVal([Feature: atom(Value1, FNode)|_], Feature, Value, CNode) :-!,
21:      Value=Value1,
22:      creareArcFromFNode(FNode, CNode).
23: featureVal([Feature: fd(_, FD)|_], Feature, Value, CNode) :-!,
24:      unify(Value, FD, CNode).
25: featureVal([_|WFD], Feature, Value, CNode) :-
26:      featureVal(WFD, Feature, Value, CNode).

27: unifyAlt([Id: GFD|Alternatives], WFD, c(_, Antecedents, Descendants)):-
28:      Descendants=[State|_],
29:      CNode1=c(Id, [State|Antecedents], []),
30:      createArc(Descendants, CNode1),
31:      unify(GFD, WFD, CNode1),
32: unifyAlt([_|Alternatives], WFD, CNode):-
33:      unifyAlt(Alternatives, WFD, CNode).

34: conflate(Feature1, Feature2, WFD, c(_, _, Descendants)):-
35:      featureVal2(WFD, Feature1, Value1),
36:      featureVal2(WFD, Feature2, Value2),
37:      createLeavesList(Value1, L0-L1),
38:      createLeavesList(Value2, L1-[]),
39:      createArc(Descendants, conflation(L0)),
40:      unify(Value1, Value2).

41: createArc(Descendants, Node):-
42:      Descendants=[State|Descendants1],
43:      setarg(2, Descendants, [Node|Descendants1]).
```

Fig. 7 Outline of unification algorithm.

terms of four clauses. The first clause is the termination clause for the case there is no more GFD fragment to unify. The second clause handles unification of a feature-value pair. This is actually performed by the predicate featureVal/4. The third clause handles a disjunction in a GFD. A disjunction is represented as the following data structure:

$$alt([id_1\colon\ GFD_1,\ id_2\colon\ GFD_2,\ldots]).$$

The order of alternatives within a disjunction represents their perference. Predicate unify-alt/3 tries each alternative one by one. The last clause of unify/3 handles conflation of features.

Given a feature and its value in the GFD, predicate featureVal/4 searches the WFD for the feature and unifies these two values. Unike the basic unification algorithm, featureVal/4 keeps the GFD intact. featureVal/4 receives a WFD, a feature from a GFD and its value as the inputs in the first three arguments, and returns the 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 since this feature is added to the WFD due to the current choice. This is performed by the predicate createArc/2 in line 15, where the Prolog built-in predicate setarg* is employed to destructively modify the *descendants* list (see lines 41 to 43). Arcs (3) and (5) in Fig. 6 are created by this operation. If the WFD and the GFD share a unifiable feature, either the third or the fourth clause is used. If the feature value is atomic, this feature is considered as a justification of the current choice; therefore, a dependency arc is created between the feature and the current choice. This is realized in line 22. Arcs (1) and (2) in Fig. 6 illustrate this case.

unify-alt/3 deals with disjunctions. 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 30. Arc (4) in Fig. 6 is an example of such an arc. Lines 28 and 29, which create a list of the backward pointers to the antecedent choices, says that the *antecedents* list of a choice consists of both the state and the *antecedents* of its mother choice.

Predicate conflate/4 deals with conflation. First, the system extracts the values of the features to conflate in lines 35 and 36. Predicate featureVal2/3 finds the value of the given feature in the WFD. Then createLeavesList in lines 37 and 38 traverses the FD given in the first argument (i.e. the FD to conflate) to collect cons cells whose tail is an unbound variable. This information would be used to cancel the conflation in the revision process. Then the system creates a *conflation node* and creates an arc between the current choice node and the conflation node (line 39). The conflation node has the pointer to the list created

* setarg is available in SICStus Prolog.[2)]

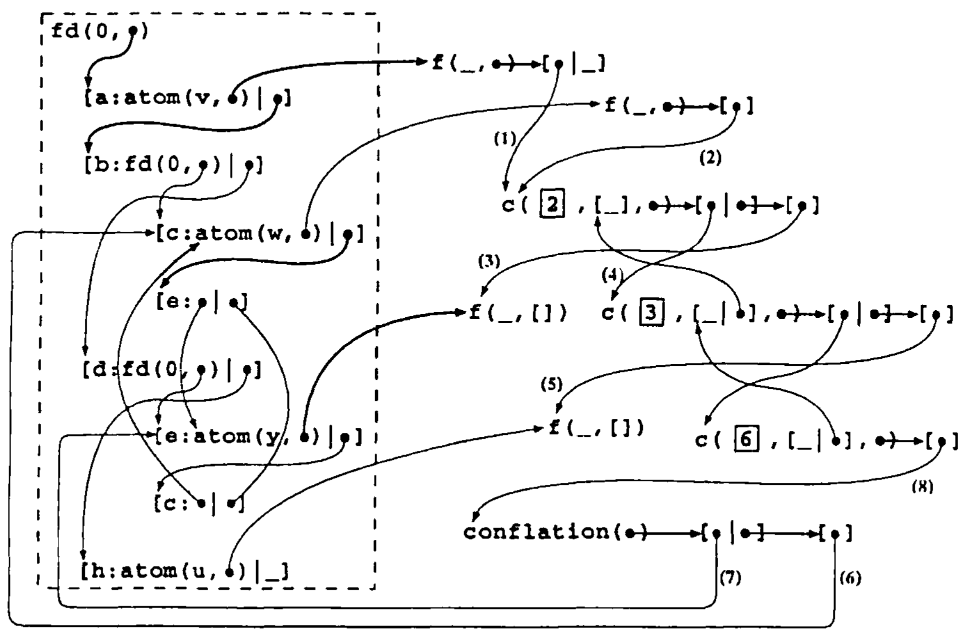

Fig. 8 Data structure of WFD$_2$ in Fig. 3.

by createLeavesList. Finally, the system performs the basic symmetric unfication unify/2 in line 40.

Suppose, for example, the system choose alternative $\boxed{6}$ in Fig. 3. b/d says that b and d are to be unified constituents. Thus the system conflates b and d. In this case, the leaf cons cells under b and d are [c: atom(w, f(_, ●))|_] and [e: atom(y, f(_, ●))|_] respectively. Then the system creates a conflation node conflation(●) with the pointers to these two nodes (arcs (6) and (7)) as illustrated in Fig. 8. Also, the choice node corresponding to choice $\boxed{6}$ points to the conflation node (arc (8)).

What is important to note about this algorithm is as follows. When choosing an alternative at a choice point, the system refers to the 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 still 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 dependency arcs (e.g. arc (3)) simultaneously with unifying these features.

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 proess, 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 the reserved value "*changed*".
(3) **[state propagation]**
Traverse the dependency network through dependency arcs starting from the culprit choice node, and perform the following procedures on each node.
  (a) If the node is a feature node, bind its state to the reserved value "*out*".
  (b) If the node is a choice node, bind its state to "*out*" and also bind each variable in its antecedents list to the reserved value "*unknown*".
  (c) If the node is a conflation node, substitute the tail of each cons cell pointed by the conflation node with an unbound variable. This substitution is performed destructively and cancels the conflation. This is realized by setarg.
(4) **[removal of invalid features]**
Remove all the features that are marked "*out*" from the 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 its antecedent, c($\boxed{2}$,...) in this case, is changed to "*unknown*", and the state of its descendant, c($\boxed{6}$,...), is changed to "*out*". State "*out*" means that the node is invalid. On the other hand, a node, say N, is "*unknown*" when all the nodes supporting N are valid but some of N's descendant choices are invalid (only choice nodes can be marked "*unknown*"). Let us see this difference using the current example. Figure 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. $\boxed{2}$ will become valid again if the system finds, in regeneration, another unifiable alternative instead of $\boxed{3}$ at that choice point. Since such cases occur frequently, it would be better not to invalidate choices such as $\boxed{2}$ for efficiency. Therefore, state "*out*" propagates via descendant links, while "*unknown*" does not.

Cancellation of conflation (step(3)(c)) is illustrated by the snapshots shown in Figs. 8 and 9. Here the system substitutes the tails of the nodes pointed by the conflation node ([c: atom(w, f(_,●))|●] and [e: atom(y, f(_,_))|●]) with unbound variables. As a result, the WFD in Fig. 8 comes to be that in Fig. 9. In Fig. 9., features b and d are not conflated any longer.

Finally, the system removes feature [h: atom(u, f(out,_))|_] by substituting the tail of node [d: fd(0, ●)|●] with an unbound variable. In this process, the

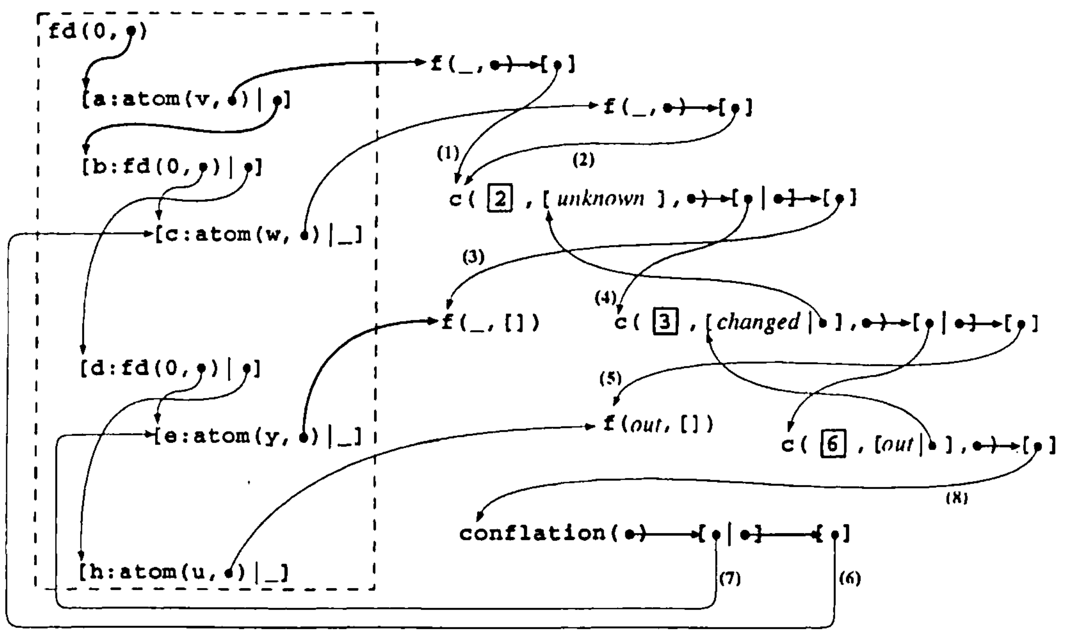

Fig. 9 The snapshot after step 3 in backtracking.

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_id* (Section 3.1). If the system comes across an FD whose *cycle_id* is not equal to the current *cycle_id*, the system updates it and traverses the FD; otherwise, the system skips the FD.

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 section, 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 operations.

(1) If the choice point has a choice whose state stays 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 the alternatives one by one from the first one.

For example, when the system starts regeneration after removing the invalid features from the WFD 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.

3.5 Extensions: Handling Special Features and Values

In this section, we describe the extensions of the algorithm in Fig. 7 to handle special reserved features and values: pattern, given, none, etc.

A pattern feature specifies a constraint on the linear order of constituents. For example, a constraint that goal has to follow process would be described by the pattern:

[..., process,..., goal,...],

where "..." represents an arbitrary sequence of constituents. Similarly a constraint that destination has to follow process would be described by

[..., process,..., destination,...].

If these two constraints are imposed simultaneously, the resultant pattern would be either

[..., process,..., goal,..., destination,...]

or

[..., process,..., destination,..., goal,...].

As shown in this example, unification of pattern features is not deterministic in general. Particularly it is the case with free word-order languages like Japanese. Given this fact, it would be reasonable not to immediately compute the resultant pattern, but just to check the compatibility of the pattern fragments. For example, unifying

$$\text{WFD}=\begin{bmatrix} \ldots \\ \text{pattern: [[..., goal,...]]} \\ \ldots \end{bmatrix}$$

and

$$\text{GFD}=\begin{bmatrix} \ldots \\ \text{pattern: [..., destination,...]} \\ \ldots \end{bmatrix}$$

would produce

$$\text{WFD}=\begin{bmatrix} \ldots \\ \text{pattern:} \begin{bmatrix} \text{[..., goal,...]} \\ \text{[..., destination,...]} \end{bmatrix} \\ \ldots \end{bmatrix}.$$

This is exactly the same algorithm as that of PFUF, which was developed by Fasciano et al.[8]

Since we just accumulate the compatible pattern fragments, we could deal with each fragment as if it were an atomic feature, and assign it a feature node. In addition, since we could regard a set of pattern fragments as a FD, we can implement unification of pattern features in a way similar to conflation of FDs.

In Elhadad's FUF system,[5] which is a generation system employing the FUG formalism, special atomic feature values such as given, none and any are implemented. Elhadad has demonstrated that those feature values are useful to describe various kinds of linguistic constraints in the FUG formalism. In our system, two of them, given and none, have been implemented.

A feature-value pair *feature*:given means that the value of *feature* must have been specified in the WFD at the time of unification. If our system finds the feature whose value is specified in the WFD, then it creates a dependency arc from that feature to the current choice. According to Elhadad, given is typically used to describe a constraint that a feature has to be specified in the input. As far as it is used in that way, there would be no point to create an dependency arc from the input feature and the choice because the input feature will never change all through the revision process.

A feature-value pair *feature*:none means that *feature* cannot have any value. Similarly to the FUF system, our system treats this reserved value as if it were a normal atomic value. That is, if the current alternative includes a pair *feature*:none, our system adds it to the WFD. Once the WFD has a pair *feature*: none, no longer can it unify with an alternative that has *feature* whose value is not none. When a pair *feature*:none is added to the WFD, the system creates a dependency arc from the current choice to that feature.

§4 Experiments

In this section we show the results of preliminary experiments to demonstrate the performance of our implementation.

4.1 Experiment (1)

We first examined some simplest cases with WFDs and GFDs shown in Fig. 10 to roughly estimate the cost of network handling. WFD_1 and GFD_1 are identical. WFD_1 (GFD_1) has 100 atomic features; five features $a_i (i = 1, \ldots, 5)$, and four features $b_j (j = 1, \ldots, 4)$ for each a_i, and finally five atomic features $c_{ik} (k = 1, \ldots, 5)$ for each pair of a_i and b_j. WFD_2 has no feature. GFD_2 imposes two conflation constraints.

The counterpart of our implementation was given by removing all the operations for network handling from the program outlined in Fig. 7. We call this the chronological backtracking (CB) algorithm to distinguish it from our DDB algorithm. Unlike DDB, the CB algorithm directly employs Prolog's chronological backtracking mechanism.

$$\mathrm{WFD}_1=\left[\begin{array}{ll} a_1: & \left[\begin{array}{ll} b_1: & \left[\begin{array}{l} c_{11}:\ d \\ c_{12}:\ d \\ c_{13}:\ d \\ c_{14}:\ d \\ c_{15}:\ d \end{array}\right] \\ b_2: & \left[\begin{array}{l} c_{11}:\ d \\ c_{12}:\ d \\ \cdots \end{array}\right] \\ b_3: & [\cdots] \\ b_4: & [\cdots] \end{array}\right] \\ a_2: & \left[\begin{array}{ll} b_1: & \left[\begin{array}{l} c_{21}:\ d \\ c_{22}:\ d \\ c_{23}:\ d \\ c_{24}:\ d \\ c_{25}:\ d \end{array}\right] \\ b_2: & \left[\begin{array}{l} c_{21}:\ d \\ \cdots \end{array}\right] \\ \cdots & \end{array}\right] \\ a_3: & \left[\begin{array}{ll} b_1: & \left[\begin{array}{l} c_{31}:\ d \\ \cdots \end{array}\right] \\ \cdots & \end{array}\right] \\ a_4: & [\cdots] \\ a_5: & [\cdots] \end{array}\right] \qquad \mathrm{GFD}_1=\left[\begin{array}{ll} a_1: & \left[\begin{array}{ll} b_1: & \left[\begin{array}{l} c_{11}:\ d \\ c_{12}:\ d \\ c_{13}:\ d \\ c_{14}:\ d \\ c_{15}:\ d \end{array}\right] \\ b_2: & \left[\begin{array}{l} c_{11}:\ d \\ c_{12}:\ d \\ \cdots \end{array}\right] \\ b_3: & [\cdots] \\ b_4: & [\cdots] \end{array}\right] \\ a_2: & \left[\begin{array}{ll} b_1: & \left[\begin{array}{l} c_{21}:\ d \\ c_{22}:\ d \\ c_{23}:\ d \\ c_{24}:\ d \\ c_{25}:\ d \end{array}\right] \\ b_2: & \left[\begin{array}{l} c_{21}:\ d \\ \cdots \end{array}\right] \\ \cdots & \end{array}\right] \\ a_3: & \left[\begin{array}{ll} b_1: & \left[\begin{array}{l} c_{31}:\ d \\ \cdots \end{array}\right] \\ \cdots & \end{array}\right] \\ a_4: & [\cdots] \\ a_5: & [\cdots] \end{array}\right]$$

$$\mathrm{WFD}_2=[\ \ ] \qquad \mathrm{GFD}_2=\left[\begin{array}{l} a_1/a_2 \\ a_3/a_4 \end{array}\right]$$

Fig. 10 WFDs and GFDs used in the experiment.

We measured the CPU time of the following cases.

(a) unification of WFD_1 and GFD_1
(b) unification of WFD_2 and GFD_1
(c) unification of WFD_1 and GFD_2

In case (a) no atomic feature is added to the WFD, while in (b) all the atomic features in the GFD are added to the WFD. In (c), the system conflates each pair of b_j's under a_1 and a_2, and similarly those under a_3 and a_4.

The results are summarized in Table 1. Let us first see the column of unification. The rows of CB and DDB denotes the CPU time for the unfication process in each of cases (a) to (c). The bottom row denotes the ratio of the overhead of network construction to the cost of unification in CB. In case (a), for example,

$$(25.8_{[\mathrm{msec}]}-24.1_{[\mathrm{msec}]})/24.1_{[\mathrm{msec}]}=7.0_{[\%]}.$$

According to the table, the overhead of network construction is at most 7 to 8 percent of the cost of initial generation. Note that in case of actual text generation the process would include not only the above operations but also various other kinds of operations; such as unification of pattern features and control of constituent-wise recursive unification. The relative cost of network construction involved in these operations should be much smaller than that of

Table 1 Overhead of network handling.

	unification			backtracking		
	(a)	(b)	(c)	(a)	(b)	(c)
CB [msec]	24.1	27.3	16.5	0.4	0.8	0.6
DDB [msec]	25.8	29.3	17.4	0.1	3.2	0.2
				5.6	6.1	5.6
overhead [%]	7.0	7.3	5.5	22.0	31.1	31.5

(SONY NEWS 3860)

network construction involved in unification operations as examined in this experiment. Therefore, the overhead of network construction is expected to be smaller than the results of this experience (see Section 4.2).

We also examined the cost of backtracking for canceling all the unification operations in each of cases (a) to (c). The results are shown in the column of backtracking in Table 1. In the row of DDB, the upper row denotes the cost of state propagation and the lower denotes that of removing invalid features. The bottom row denotes the ratio of the overhead of network handling in backtracking to the cost of unification in CB. In case (a), for example,

$$\{(0.1_{[\mathrm{msec}]}+5.6_{[\mathrm{msec}]})-0.4_{[\mathrm{msec}]}\}/24.1_{[\mathrm{msec}]}=22.0_{[\%]}.$$

One might say that network handling seems fairly expensive. In actual text generation, however, the system would spend relatively higher proportion of the total time on unification in each revision cycle. This is because actural unification would encounter a number of unification failures, and furthermore it would require additional operation that are not included in this experiment as mentioned above; on the other hand, the cost of backtracking depends not on the cost of unification but just on the sizes of WFDs. The former will be higher as the grammer becomes complicated, while the latter will not. Therefore, the relative cost of network handling for backtracking is expected to be significantly smaller than the results shown in the table.

4.2 Experiment (2)

Next, we used a small experimental Japanese grammar, which consists of 39 grammatical disjunctions and 38 lexical entries, to evaluate the performance of our algorithm in a more natural setting. Figure 11 shows an input WFD represented in terms of rhetorical structure.[17] In this figure, features n and s mean *nucleus* and *satellite* respectively. Those constituents labeled n and s are organized in terms of a rhetorical relation elaborate.

The draft initially generated from this input is draft (1) shown below, where both the propositions keep and located are realized by a single sentence. This draft, however, includes an unexpected long noun modifier "*tonari-no tatemono-no 4 kai-no itiban oku-ni aru* (which is located in the most inner part on the fourth floor of the next building)". The system detects this problem and solves it by changing a culprit choice. The second draft is draft (2). Note that the

```
┌ cat: segment                                              ┐
│ pred: elaborate#1                                         │
│    ┌ pred: elaborate#2                               ┐    │
│    │    ┌ pred: keep#1                          ┐    │    │
│    │ n: │ topic: document#1                     │    │    │
│ n: │    │ agt: [pred: library#1]                │    │    │
│    │    └ 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. 11 The input WFD.

first sentence in draft (1) is split into two sentences in draft (2).

Draft (1) *sono syorui-wa tonari-no tatemono-no 4 kai-no itiban oku-ni aru siryôsitu-ni hokansareteimasu. tadasi, siryôsitu-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. It 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. It is located in the most inner part on the fourth floor of the next building. It will be closed during the summer vacation from July 21 to August 31.)

Here we introduce another counterpart: the bk-class framework proposed by Elhadad.[6] A bk-class is a pair of the name of a feature and the choice points that may cause a failure of unification of that 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. Bk-class' would play a role similar to our revision rules. Significant differnces between the bk-class framework and ours 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 specifying the identifier of them. Since a revision rule describes candidates of a certain culprit choice more specifically 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, on the other hand, we can reuse the results of the computation that are independent of the change of the culprit choice. This difference is significant particularly in such cases that the draft has more than one proplems that can be solved independently of one another . In such cases, our method would be able to solve the problems one by one independently, while the bk-class method would have to deal with the combinations of the problems.
- In this paper, we assume that a revision rule identify a culprit choice by specifying 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 choices and 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 2 CPU time to generate drafts.

draft	1	2	3	···	17	Total
CB	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])

Table 2 shows the CPU time to generate draft (2) for each case. Our framework solves the current problem by only a single backtrack; while, the bk-class framework requires two, and furthermore chronological backtracking reguires sixteen. In general, a bk-class does not identify the candidates of a culprit choice as specifically as a revision rule; therefore, backtracks tend to occur more frequently in the bk-class framework than in ours. 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 in vain. 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, siryôsitu-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 fouth 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 3 Overhead of network handling.

	generation	backtracking
bk-class[mesc]	1.18	0.03
DDB [msec]	1.23	0.07
overhead [%]	4.2	3.4

Table 3 shows the CPU time for network handling in the initial generation and backtracking. Each column and row denotes analogous to that of Table 1. The table says that the CPU time for the network construction was 4.2 percent of that for the initial generation, and furthermore that the cost of network handling for backtracking was only 3.4 percent, which is much better than the results of experiment (1).

Summarizing this section, in our implementation the overhead of DDB is estimated to be at worst up to 35% of the initial generation process and it seems reasonably small in actual cases. Comparing the potential advantages of our method (e. g. avoidance of unnecessary recomputation) with its overhead, we would conclude that it is worth introducing.

§5 Conclusion

In text generation, various kinds of choices need to be decided. Since these choices depend on one another, it would be difficult in one-path generation frameworks to design a set of heuristic rules for optimal decisions. Introducing the revision process can be a solution to this problem. In our previous paper, we presented a framework in which revision is realized as DDB.

In this paper, we have proposed an efficient implementation method to realize DDB for text generation using FUG in Prolog. FUG is suitable for DDB because FUG allows the decision order to be flexible. 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 results 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.

We applied the framework of JTMS to the task of text generation with

DDB. Similarly to JTMS, the system constructs a dependency network to maintain dependencies among choices and features. However, we also need to consider the overhead of network handling. We proposed a method to realize 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 updating the network is realized by means of destructive substitutions. According to preliminary experiments, the cost of network handling seems to be reasonably small.

With the Prolog implementation of FUG, Fasciano et al. improved the efficiency by precompiling a grammar as a set of Prolog clauses, where each feature roughly corresponds to a Prolog clause. Since this implementaion does not require any interpreter of GFDs, it may be more efficient than our interpreter-based implementation (like unify/5 in Fig. 7). Therefore, it would be reasonable to integrate our method of network handling into Fasciano's implementation. This should not be difficult since in our algorithm network construction is done simultaneously with unification of each atomic feature.

In addition, in our framework the grammar writer can describe a revision rule such that it specifies a *culprit feature*, as it were, rather than a culprit choice; i.e. a feature that should be removed to solve a problem. This is possible because our method maintains the dependencies between choices and features. 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 to implement DDB may be applicable to other applications. 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 any disjunctive justifications in FUG.

In this paper, we mentioned neither evaluating drafts nor revision planning. To realize the overall generation system, we need further research on both evaluation criteria and revision rules. These issues depend on actual descriptions of lexico-grammatical knowledge. 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 for development of good evaluation criteria and revision rules.

References

1) Appelt, D. E., *Planning English Sentences*, Cambridge University Press, 1985.
2) Carlsson, M. and Widén, J., *SICStus Prolog User's Manual*, Swedish Institute of Computer Science.
3) de Kleer, J., Forbus, K., and McAllester, D., "Truth Maintenance Systems," *The Eleventh International Joint Conference on Artificial Intelligence, tutorial program*, 1989.
4) Eisele, A. and Dörre, J., "A Lexical Functional Grammar System in Prolog," in *Proceedings of the International Conference on Computational Linguistics*, pp.

551-553, 1986.

5) Elhadad, M., "FUF: The Universal Unifier—User Manual," version 5.0, *Technical Report, CUCS-038-91*, Columbia University, 1991.

6) Elhadad, M. and Robin, J., "Controlling Content Realization," in *Aspects of Automated Natural Language Generation* (R. Dale, E. Hovy, D. Rösner, and O. Stock, eds.), Springer-Verlag, pp. 89-105, 1992. *Lecture Notes in Artificial Intelligence, Vol. 587*.

7) Emele, M., Heid, U., and Zajac, R., "Interactions between Linguistic Constraints: Procedural vs. Declarative Approach," *Machine Translation, 6, 4*, 1991.

8) Fasciano, M. and Lapalme, G., "A Prolog Implementation of the Functional Unification Grammar Formalism," in *Proceedings of International Workshop on Natural Language Understanding and Logic Programming*, pp. 37-45, 1994.

9) Fawcett, R. P., Tucker, G. H., and Lin, Y. Q., "How a Systemic Functional Grammar Works: The Role of Realization in Realization," in *New Concepts in Natural Language Generation*, Pinter, pp. 114-186, 1993.

10) Gazder, G. and Mellish, C., *Natural Language Processing in Prolog*, Addison Wesley, 1989.

11) Halliday, M. A. K. and Hassan, R., *Language, Context, and Text: Aspects of Language in a Social-Semiotic Perspective*, Deakin University Press, 1985.

12) Hovy, E. H., *Generating Natural Language under Pragmatic Constraints*, Lawrence Erlbaum Associates, 1988.

13) Inui, K., Tokunaga, T., and Tanaka, H., "Text Revision: A Model and Its Implementation," in *Aspects of Automated Natural Language Ceneration* (R. Dale, E. Hovy, D. Rösner, and O. Stock, eds.), Springer-Verlag, pp. 215-230, 1992. *Lecture Notes in Artificial Intelligenc, Vol. 587.*

14) Kasper, R., "Systemic Grammar and Functional Unification Grammar," in *Systemic Functional Approaches to Discourse*, Ablex, chapter 9, pp. 176-199, 1988.

15) Kay, M., "Functional Unification Grammar: A Formalism for Machine Translation," in *Proceedings of the International Conference on Computational Linguistics*, pp. 75-78, 1984.

16) Mann, W. C., "An Overview of the Penman Text Generation System," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 261-265, 1983.

17) Mann, W. C. and Thompson, S. A., "Rhetorical Structure Theory: A Theory of Text Organization," *Technical Report, ISI/RR-87-190*, USC-ISI, 1987.

18) Matthiessen, C. and Bateman, J., *Text Generation and Systemic-Functional Linguistics: Experiences from English and Japanese*, Printer Publishers, 1991.

19) McKeown, K. R. and Swartout, W. R., "Language Generation and Explanation," in *Advances in Natural Language Generation* (M. Zock and G. Sabah, eds.), Ablex Publishing Corporation, chapter 1, pp. 1-51, 1988.

20) Meteer (Vaughan), M. M. and McDonald, D. D., "A Model of Revision in Natural Language Generation," in *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, pp. 90-96, 1986.

21) Patten, T., *Systemic Text Generation as Problem Solving*, Cambridge University Press, 1988.

22) Robin, J., "A Revision-Based Generation Architecture for Reporting Facts in Their Historical Context," in *New Concepts in Natural Language Generation*, Pinter, chapter 9, pp. 238-268, 1993.

Kentaro Inui, Dr.: He is a research associate of Department of Computer Science, Tokyo Institute of Technology. He received the B. S. degree in 1990, the M. S. degree in 1992, and the Dr. Eng. degree in 1995 from Tokyo Institute of Technology. His current interest is in natural language processing.

Takenobu Tokunaga, Ph.D: He is an associate professor of Department of Computer Science, Tokyo Institute of Technology. He received the B.S. degree in 1983 from Tokyo Institute of Technology, the M.S. and the Dr. Eng. degrees from Tokyo Institute of Technology in 1985 and 1991, respectively. His current interests are natural language processing, information retrieval.

Hozumi Tanaka, Ph.D: He is professor of Department of Computer Science, Tokyo Institute of Technology. He received the B.S. degree in 1964 and the M.S. degree in 1966 from Tokyo Institute of Technology. In 1966 he joined in the Electro Technical Laboratories, Tsukuba. He received the Dr. Eng. degree in 1980. He joined in Tokyo Institute of Technology in 1983. He has been engaged in artificial intelligence and natural language processing research.