**Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH**

**Research
Report**

TM-92-08

Realization of Tree Adjoining Grammars with Unification

Anne Kilger

September 1992

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

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

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

Deutsches Forschungszentrum für Künstliche Intelligenz

The German Research Center for Artificial Intelligence (Deutsches Forschungszentrum für Künstliche Intelligenz, DFKI) with sites in Kaiserslautern and Saarbrücken is a non-profit organization which was founded in 1988. The shareholder companies are Atlas Elektronik, Daimler-Benz, Fraunhofer Gesellschaft, GMD, IBM, Insiders, Mannesmann-Kienzle, Sema Group, and Siemens. Research projects conducted at the DFKI are funded by the German Ministry for Research and Technology, by the shareholder companies, or by other industrial contracts.

The DFKI conducts application-oriented basic research in the field of artificial intelligence and other related subfields of computer science. The overall goal is to construct systems with technical knowledge and common sense which - by using AI methods - implement a problem solution for a selected application area. Currently, there are the following research areas at the DFKI:

Intelligent Engineering Systems
Intelligent User Interfaces
Computer Linguistics
Programming Systems
Deduction and Multiagent Systems
Document Analysis and Office Automation.

The DFKI strives at making its research results available to the scientific community. There exist many contacts to domestic and foreign research institutions, both in academy and industry. The DFKI hosts technology transfer workshops for shareholders and other interested groups in order to inform about the current state of research.

From its beginning, the DFKI has provided an attractive working environment for AI researchers from Germany and from all over the world. The goal is to have a staff of about 100 researchers at the end of the building-up phase.

Dr. Dr. D. Ruland

Director

Realization of Tree Adjoining Grammars with Unification

Anne Kilger

DFKI-TM-92-08

This work has been supported ba a grant from The Federal Ministry for Research and Technology (ITWM-8901 8).

Realization of Tree Adjoining Grammars with Unification

Anne Kilger

Abstract

The syntactic generator of the WIP system is based on the representation formalism 'Tree Adjoining Grammars' (TAGs). We have extended the formalism by associating elementary rules of the grammar (trees) with feature structures, leading to 'Tree Adjoining Grammars with Unification' (UTAGs). The extended formalism facilitates a compact and adequate representation of complex syntactic features.

The contradiction between the monotonic operation of unification and the combination operation for trees – adjunction – that is nonmonotonic in a way can be solved by several different approaches to realization. Two of them are presented in this report and compared with respect to the restrictions that are given by the system, i.e., the adequacy of the realization for incremental and parallel generation.

It can be shown that UTAGs are subsumed by FTAGs (Feature Structure based TAGs) that have been defined by Vijay-Shanker and Joshi. That is why the results for realization can be applied to both UTAGs and a restricted version of FTAGs.

Contents

1 Introduction

The WIP*system (see [Wahlster et al. 91], [Wahlster et al. 92]) automatically and dynamically creates a multimodal presentation on the basis of given information and generation parameters. Up to now there are two modes integrated, i.e., text and graphics. A presentation planner decides which piece of information should be realized in form of text or graphics. It hands over the respective data to the two cascades of the mode specific generators. Each cascade consists of a 'Design'-module developing the rough structure of the presentation and a 'Realization'-module for the final construction of a text or a picture. The Text Design component designs the text structure and computes word choice. The Text Realization component is a syntactic sentence generator, part of which is examined in this work.

The Text Realization component is based on a multi-level, description-directed approach (see [McDonald 87]): The input is transformed across two levels into resulting sentences. At the first level, the underlying hierarchical structure of the sentence is constructed. At the second level, word order is computed and the result is uttered. At both levels, the existence of structures triggers their computation. They are represented with the help of a descriptive representation formalism: 'Tree Adjoining Grammars with Unification' (UTAGs, see [Buschauer et al. 91]).

In the next section, the formalism of UTAGs is defined. Section 3 contains the description of two possible methods for a realization of UTAGs. In Section 3, they are compared with each other thereby evaluating the specific demands and design principles underlying the syntactic generator. In Sectionsec-comp-incr UTAGs are compared with Feature Structure based TAGs (FTAGs), another approach to the combination of TAGs and unification by [Vijay-Shanker & Joshi 88]. [Vijay-Shanker 92] already described UTAGs as a special case of FTAGs. This section will lead to another view of UTAG that eases understanding this relationship. Because of the close relation of UTAGs and FTAGs, the two forms of realization can be used for both formalisms.

2 TAGs with Unification

2.1 Tree Adjoining Grammars

The formalism 'Tree Adjoining Grammar' has been introduced by ([Joshi et al. 75]). It is a tree generating system. Its elementary rules consist of a set of trees which is divided into two classes. The *initial trees* correspond to context-free derived trees as their root is labelled with the grammar's start symbol, their internal nodes with nonterminals, and their leaves with terminals (see the left tree in Figure 1). *Auxiliary trees* have a specific form that enables them to substitute internal nodes of initial (or already modified initial)

*WIP is the acronym for "Wissensbasierte Informationspräsentation" which means knowledge-based presentation of information. The WIP project is supported by the German Ministry of Research and Technology under grant ITW 8901 8.

trees. They look like initial trees except for one leaf called *foot node*. The foot node and the root node of an auxiliary tree are associated with the same label. The combination operation which combines two trees by substituting an internal node of the first by the whole second tree is called *adjoining* or *adjunction*. The foot node of the second tree becomes the new father of the subtree of the substituted node. An auxiliary tree and the result of an adjunction are illustrated in the middle and in the right of Figure 1. Each auxiliary tree must have at least one terminal leaf in order to prevent an indefinite repetition of adjunctions without expansion of the terminal string.

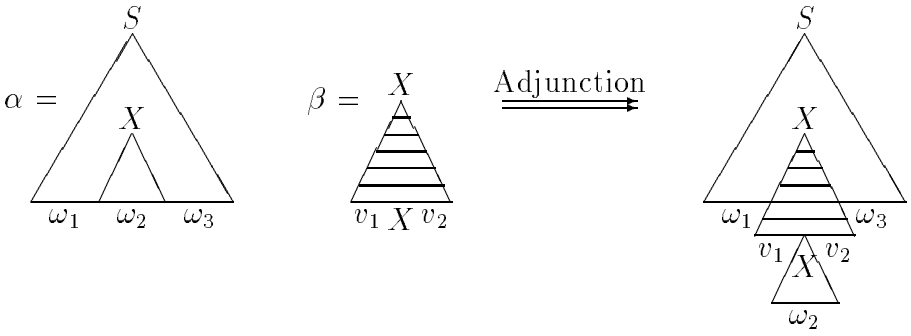


Figure 1: Elementary Trees and the Adjunction Operation in a TAG

In our generator, we use a version of TAGs that is extended by another combination operation for trees. TAGs with Substitution allow elementary trees to have leaves labelled with nonterminals. In order to distinguish them from foot nodes, these so–called *substitution nodes* are marked with a downward arrow as can be seen in Figure 2. Substitution is defined as replacing a substitution node by an initial tree whose root is labelled with the same nonterminal (that doesn't have to be the start symbol of the grammar). An example for a substitution is shown in Figure 2.

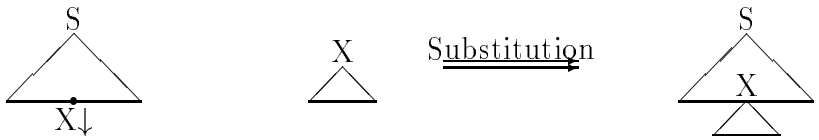


Figure 2: The Substitution Operation in a TAG

An important advantage of TAGs – in comparison with Context-Free Grammars – is their extended domain of locality that makes it possible to represent syntactic properties such as cooccurrence restrictions locally within single trees. This extension makes TAGs more powerful than CFGs: They are mildly context-sensitive (cf. [Joshi 85], [Weir 88]) and probably well-suited for the representation of natural language (see [Joshi 85]). But there exists – for CFGs as well as for TAGs – a well known disadvantage: The encoding of complex syntactic features into the labels of grammar rules (e.g., "V.1.sg" for a verb in first person singular) leads to a combinatory explosion of the grammar. The problem has

been solved for CFGs by combining them with unification rules, leading to Unification Grammar. The same can be done for Tree Adjoining Grammars.

2.2 Unification

Unification is introduced here according to the PATR–II formalism (see [Shieber et al. 83]) on the basis of Context-Free Grammars. It allows to specify complex data within feature structures (here abbreviated as FSs). The connection of context-free rules and FSs is specified in PATR–II by two lists. The *constituent list* defines the constituents of a context-free rule in their order from left to right (e.g., (S NP VP) for S $\rightarrow$ NP VP). The *specification list* describes the associated FS using pairs of the form (path path) or (path value). A path consists of an attribute list. The attributes are roots of substructures within the FS. The path starts with a number uniquely referring to a constituent of the associated rule – 0 for the left-hand side, 1 for the first son, 2 for the second and so on. Either it is unified with another path – which means that the two substructures which are reached via traversing the paths are combined – or it leads to an atomic value. FSs are often represented as DAGs (directed acyclic graphs). Common prefixes of substructures are represented only once, different sons become adjacent edges in the graph. The specification list in Figure 3 defines the value 'sg' (for singular) behind the path (2 syntax num), that means

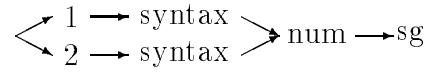
constituent list	specification list	as DAG
(NP DET N)	(((2 syntax num) sg) ((1 syntax) (2 syntax)))	

Figure 3: Constituent List, Specification List and DAG

as number–value for the noun. The specification rule ((1 syntax) (2 syntax)) claims that noun and determiner must reach the same value or subdag via the attribute 'syntax'. This rule either means a test – if values or subdags were previously defined at both places – or the inheritance of a value, which is the case in this example. In the resulting DAG the num-subdag can be reached via the paths (1 syntax) and (2 syntax) as was specified in the rule. This joint of the values of several subdags is further called 'unification point'.

Finally we present the formal definition of unification. The result of unifying of two DAGs d_1 and d_2 is a DAG d, with

1. $d = d_1$, if $d_1 = d_2$,
2. $d = d_1$, if d_1 consists of a value and d_2 is empty,
3. $d = d_2$, if d_1 is empty and d_2 consists of a value,
4. if neither d_1 nor d_2 consist of a value, then:
   $\forall$ attributes l, with: $l \rightarrow n_1 \in d_1, l \rightarrow n_2 \in d_2$ (common path prefixes), there is $l \rightarrow$

Unification$(n_1, n_2) \in$ d and
$\forall$ attributes l, with: l $\rightarrow n \in (d_1 \cup d_2) \setminus (d_1 \cap d_2)$ (i.e., path starting in exactly one DAG), there is l $\rightarrow n \in d$,

5. otherwise the unification fails.

There are at least two ways to associate TAG trees with feature structures. First, we can use the unique reference numbers to identify single nodes of the trees. Then each tree can be associated with one specification list that describes the FSs of all nodes and relations between them. These relations need not be limited to father-son relations but they can link FSs of nodes, that are not immediately related. Obviously, this kind of definition results in rather small FSs because there is no need for inheritance of values over a whole path of direct ancestors and descendants, as is in unification grammars (e.g., the number–value of the noun must be inherited from N via NP and VP to V, because there is no direct connection between N and V). The dependencies among the nodes can be stated directly. This is the way, FTAGs are defined (see Section 5.1).

The second way to associate feature structures with TAG trees is to use the close relation between trees and context-free rules. The father-son relations inside TAG trees can be interpreted as context-free rules with the father node on the left and the sons on the right side. This approach associates each node with a single feature structure. Obviously, this leads to more redundant structures but it allows to adapt directly a PATR-style unification to TAG trees as has been done for UTAGs (see Section 2.3).

A short discussion about the different properties that result from the definitions is given in Section 5.2.

2.3 TAGs with Unification (UTAGs)

The basic idea of UTAGs is to break the trees into CFG-rules and to associate these rules with specification lists that can be compiled into (local) feature structures. The example in Figure 4 shows how the agreement between a verb and a noun can be defined by use of FSs within one single elementary rule (an initial TAG tree). This is not possible for Context-Free Grammars. The left part of the figure shows that the nodes of the elementary tree are associated with specification lists which describe FSs according to the PATR-II–notation. In the right part the same tree is shown with local FSs that result from compiling the given specifications. The arrows behind the syn-attributes point to common subdags that are locally empty, i.e. that don't have a value yet. The inheritance of the syn-subdag of the noun from N via NP, S and VP to V is guaranteed because of the meaning of the different FS-parts. Each i–subdag (each subdag that starts with the reference number i) of a node refers to its i–th son. At a point in time when no adjunction takes place any more the local FSs are unified. This unification corresponds to the combination operation for PATR–II–rules: The i–subdag of a node is unified with the 0–subdag of its i–th son.

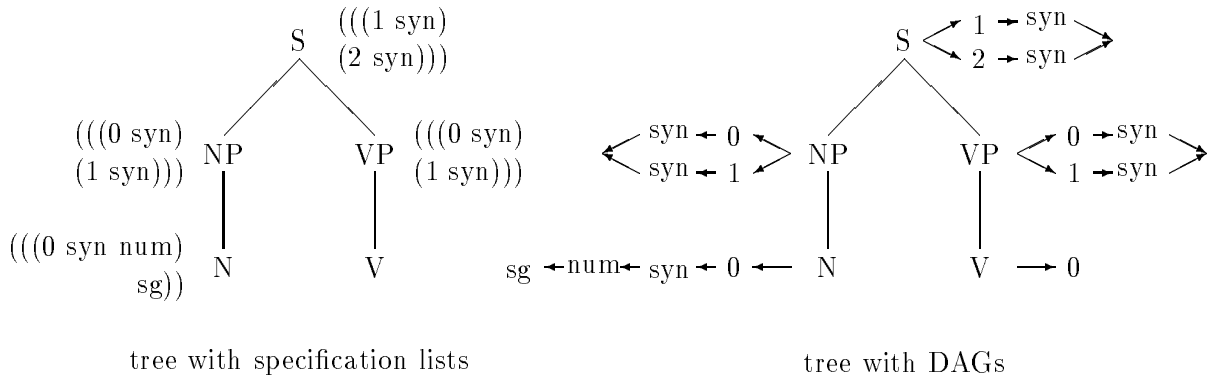


Figure 4: TAGs with Unification

While it is easy to combine the structures of the TAG formalism with feature structures, it is hard to redefine the adjunction operation. There is a problem with adjunction that forbids to directly unify the local FSs of the single nodes. Such a destructive unification would lead to one global FS where the original local parts cannot be identified any more. But during adjunction, the local FS of the node of adjunction must be identified, removed from the global FS and substituted by the FS of the auxiliary tree. The need for identifying local knowledge conflicts with building a global FS by destructive unification. That is why the locality of single FSs must be preserved in our approach in order to define adjunction with unification.

It can be stated in an additional way what is problematic about this definition: Unification is a monotonic operation because it always enlarges structures instead of really modifying them. In contrast to this, adjunction can be viewed as nonmonotonic in the following sense: By substituting a whole tree for an internal node the former relations between father and sons of the node of adjunction are changed and so are the relations between the associated feature structures. E.g., a value that was defined in a son of the node of adjunction X and inherited through the feature structure of X to its father, might not be inherited any more after an adjunction in X. This depends on the feature structures and connections inside the inserted auxiliary tree.

It is clear that adjunction with unification does not only mean to substitute a tree for an internal node, but also to transfer the neighborhood relations of this node to the auxiliary tree. These relations include the edges in the tree to father and sons of the node, which must be linked to the root and foot of the auxiliary tree respectively. Furthermore, the feature structure of the node of adjunction is related with the feature structures of the surrounding nodes. These relations must also be retained. In order to transfer them to the auxiliary tree they first have to be identified. This task is eased by the PATR–style organization of feature structures that we use in our definition of UTAGs. Each node that has a father in the tree, is represented in the FS of the father node as i–subdag (if it is the i–th son). Therefore, the FS relation between a node and its father (which we call $\uparrow$X for a node with label X) is specified locally within the FS of the father. The FS of the

node itself represents the relation to the sons (called ↓X) by referring to them via unique reference numbers. An example for this identification is shown in Figure 5. Some of the rules at the nodes S and X do not only belong to ↑X and ↓X but also to the respective neighbored nodes (e.g., to ↓S), because they describe *relations* between a node and its sons.

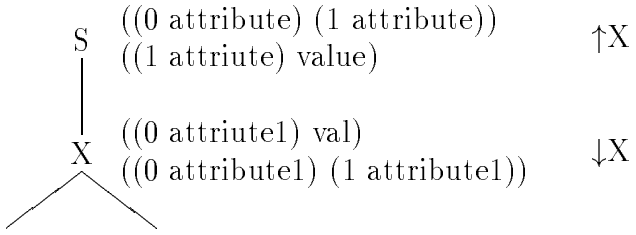


Figure 5: The Separation of Specification Lists at the Node of Adjunction

What happens with the feature structures during adjunction? ↑X and ↓X must be brought into relationship with the FSs of the root and the foot node of the auxiliary tree, respectively. According to our definition of TAGs with unification this means that nothing has to be done for ↑X because it is part of the father of X which becomes the father of the root node now. A 'real' unification with structure-sharing of the FSs of the foot node and the node of adjunction directly transfers the relation between the node of adjunction and its sons to a relation between the foot node and those sons.

An example for adjunction with unification is shown in Figure 6. The auxiliary tree in the middle of the figure is to be adjoined in the node with label X in the left tree. All subdags that have to be transferred from the node of adjunction to the auxiliary tree are marked by a bold face.

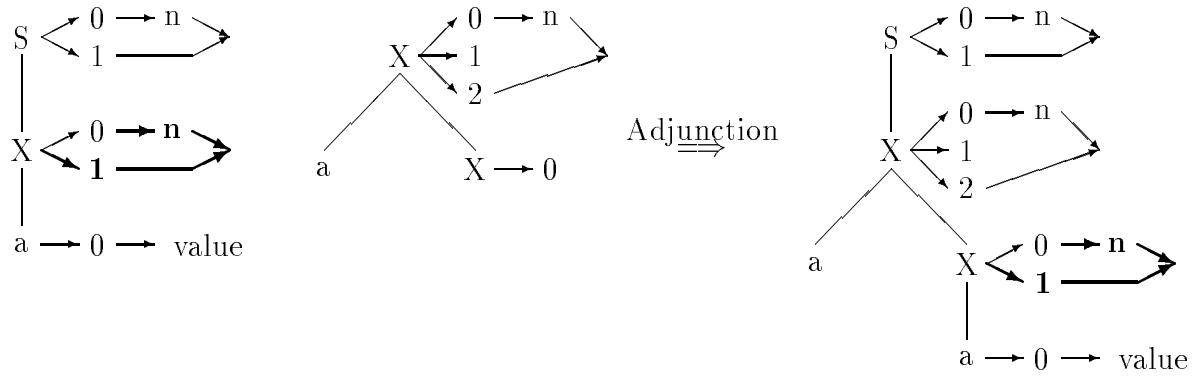


Figure 6: Adjunction with Unification

At this point, it is necessary to refer to our former definition of UTAGs (see [Buschauer et al. 91]). We examined the FS associated locally with a node of adjunction. While our definition of ↑X was the same as described above, ↓X was further

split into 'the old ↓X' and ∘X. Thereby, ∘X meant all value definitions (e.g., ((0 attribute) value)) that are made for X, 'the old ↓X' meant all rules describing the relation of X to its sons. Since we meant ∘X to consist of local value definitions that neither refer to the supertree nor to the subtree we claimed that they had to be transferred to the inserted auxiliary tree in a way, which would allow this tree to play the part of X with respect to the surrounding nodes. A complicated inheritance examination was defined to compute the node(s) of the auxiliary tree where the value definitions should be defined.

A detailed comparison with the motivation and the definition of FTAG (see below) made us see the value definitions in X from another perspective. With respect to FTAG and the original PATR–II definition it is more senseful to say that all unification rules that are associated with X describe the relation of X to its subtree. All value definitions that are defined for X therefore have to be transferred to the foot of the auxiliary tree. This does not mean, that there can never be value definitions associated with the root of the auxiliary tree, because these kinds of definitions may be made with the father of X (e.g., ((1 attribute) value), X be the first son of its father). They always stay associated with the father node of X, even if X is substituted by an auxiliary tree during adjunction. Thereby, they become valid for the root of the inserted tree. This new definition is more direct and easier than the old and therefore has to be preferred.

3 Two Approaches for the Realization of TAGs with Unification

The following basic realization technique is underlying both approaches for the implementation of UTAGs: The nodes are associated with feature structures which are specified in form of specification lists. In order to realize a more compact representation, they are compiled into directed acyclic graphs (DAGs, cf. Section 2.2). In the next two sections, the term 'DAG' is used whenever we refer to the compiled form of feature structures.

3.1 Structure-Sharing Unification

We first tried to implement our former definition of UTAGs during a graduate course (see [Buschauer et al. 91]). We didn't intend to produce software for a new-style unification, but we wanted to use the PATR implementation that was available (a COMMON LISP version of D–PATR, cf. [Karttunen 86]). Therefore, we had no means to realize FSs which at the same time could be used globally and were separated from all other FSs in the tree. Instead we had to use structure-sharing of feature-structures and recompilation to simulate the separation.

Obviously, structure-sharing violates the locality of the feature structures associated with single nodes. Following [Vijay-Shanker 92] the initial unification of top- and bottom-structures reflects the default assumption that no adjunction will take place in the respective nodes (here all nodes of all elementary trees). Therefore, unification must be

withdrawn if adjunction is initiated nevertheless. Since structure-sharing leads to one connected feature structure for the whole tree, the parts of which cannot be assigned to single nodes any more, the underlying (local) specification lists must be stored in addition to the DAGs and are used during adjunction as the basis for rebuilding the global feature structure.

Adjunction with recompilation is shown in Figure 7. The feature structures of the left tree with the node of adjunction X must be thrown away: They cannot be transformed

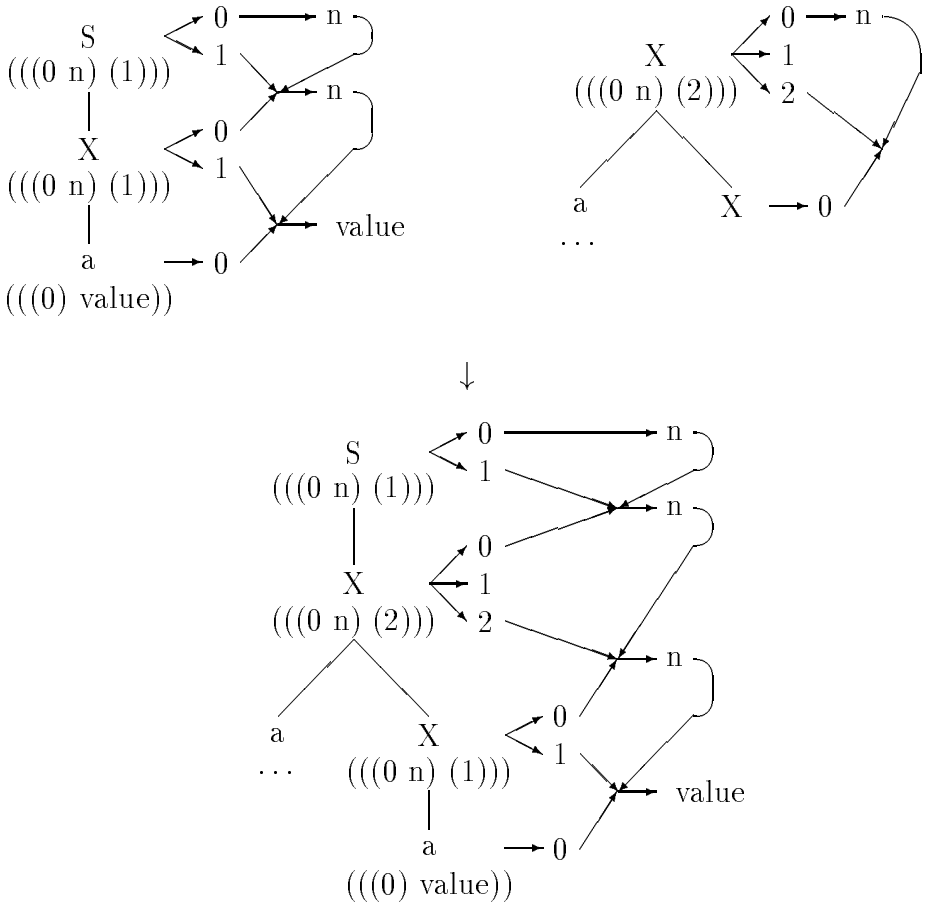


Figure 7: Adjunction with Recompilation of FSs

in a way, that allows to fit in the feature structures of the auxiliary tree correctly because structure–sharing makes it impossible to identify the local parts ↑X and ↓X of the node of adjunction. The auxiliary tree is inserted into the left tree, thereby the specification lists of the foot node and the node of adjunction are combined. The specification of the relation between the node of adjunction and its father needs not to be explicitly transferred to the root of the auxiliary tree, because it is defined locally with the father which automatically becomes the father of the root node (see Figure 7). Then the local DAGs are compiled using a function from the PATR package. Finally, their i-subdags (i be a reference number

as introduced in Section 2.2) and 0-subdags are unified leading to a new global DAG with structure-sharing. The resulting tree in Figure 7 shows that the insertion of an auxiliary tree can lengthen a path of attributes. In the initial tree, the value 'value' could be read from the root node with category S via the path (0 n n). In the resulting tree the path is (0 n n n).

The realization of TAGs using unification with structure-sharing seems to be very expensive since during adjunction all DAGs are thrown away and rebuilt. The next section will show whether a new implementation using an explicit link between local FSs is more efficient.

3.2 Unification with Bidirectional References

We made a second approach to the realization of TAGs with unification[†]since we had the impression that the direct implementation of separated DAGs conncected with so–called *bidirectional references* could be suitable for using the formalism within an incremental syntactic generator.

Bidirectional references need not be realized directly. They only encode our knowledge about the points inside the FSs where unification has to take place at the end of a derivation. Either this knowledge is used implicitly and procedurally during computation or links are explicitly defined between the respective subdags. We preferred the latter approach because it eases the design and the explanation of the algorithms.

Bidirectional references are realized by associating each node of a DAG with a list of pointers to the respective nodes of other DAGs. Figure 8 shows two trees that illustrate this approach. The nodes are associated with local DAGs. The subparts of the DAGs that would have been unified in the first approach are the i–subdag of a node and the 0–subdag of its i–th son, respectivly. They are connected by bidirectional references, represented as links in the figure.

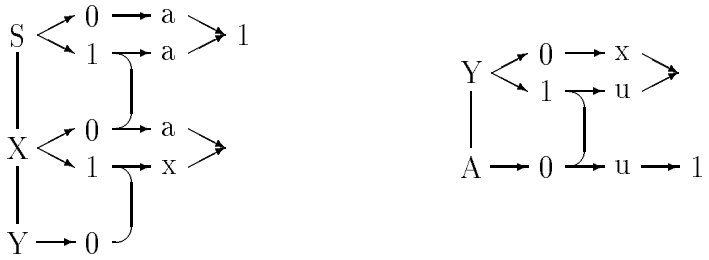


Figure 8: UTAGs with Bidirectional References

For unification with bidirectional references, adjunction seems to be cheaper in principle: Only the references must be cut off and rebuilt, not the whole DAG structure. But we must not forget that a test for compatibility must take place before different DAGs

[†]Special thanks to Peter Poller, who had a lot of work with the implementation.

are combined because this combination operation substitutes real unification, whereby the compatibility of the respective FSs is the precondition for success. If we postpone this test until all adjunctions have been made and the unification of local FSs starts, we eventually find out that some unifications fail. When this happens at the end of a long derivation, it is very hard to detect the reason for the fail. This approach is not suitable for *incremental* processing where intermediate results are produced as soon as possible. In such an application it is no good idea to proceed with building a syntactic structure on the basis of trees the combination of which is forbidden.

In the previously described approach to realization, the test for compatibility is done implicitly during structure-sharing. For unification with bidirectional references, this test must be made explicitly in addition to the computation of references. Therefore, the costs for such compatibility tests must not be neglected for a comparison of the two approaches.

The function 'compatible' tests two DAGs for contradictions for reasons of different values. The rough algorithm illuminates how costly the test is:

compatible (*dag1 dag2*)
∀ *hdag1* that are connected with *dag1* via references and/or unification points
(except those which have already been visited)
  ∀ *hdag2* that are connected with *dag2* via references and/or unification points
  (except those which have already been visited)
    compare the subdags of *hdag1* and *hdag2* by recursive calls of 'compatible' or
    compare the defined values[‡]

It becomes clear that often the whole tree must be traversed during this process, not only because of the recursion but also as a consequence of the two universal quantifications. The two trees of Figure 8 can be used as an example for input structures to the compatibility test. In order to test the two nodes with label Y, both trees must be fully traversed before it is found out that the same value can be read via references in both DAGs. So the result of this test is that both DAGs are compatible. The realization of this kind of traversing of trees made it necessary not only to store references with the respective DAGs but also to realize bidirectional pointers between the nodes of the DAGs (e.g., to come from (1 x) to (0 a) in the DAG associated with the X-node).

On the basis of this discussion it comes to no surprise that adjunction is at least as costly for unification with bidirectional references as for unification with structure-sharing if we assume that the adjunction operation must not be destructive. The DAG structures must be rebuilt for both approaches. If we assume similar costs of this computation, the higher costs of the compatibility test for unification with bidirectional references are the reason for longer running times.

In the destructive version, unification with references should be cheaper because the only things that have to be done are the compatibility test and a transfer of some bidirec-

[‡]The values may be 'NIL' for 'not specified' or a disjunctive list of atomic values, e.g. '(1 2 3)'. A list with one atomic value is often abbreviated in the figures by leaving out the parenthesis.

tional references to new FSs in the auxiliary tree. In contrast with this, unification with structure-sharing makes it necessary to build up the whole DAG structure again.

The costs for unification with bidirectional references do not only come up with adjunction but also at the times when the value is to be computed which can be read in a DAG via a specified path. The algorithm for 'get-path' is briefly described here because it also shows what happens during the computation of the two universal quantifications in 'compatible'.

get-path (*dag path*)

1. mark *dag* as 'visited'
2. $\forall$ *dag'* that are connected with *dag* via references and that are not yet marked as visited: (get-path *dag' path*)
3. $\forall$ *dag'* that point to the same subdag as *dag* (sisters) and that are not yet marked as visited: (get-path *dag' path*)
4. $\forall$ *dag'* that point to *dag* as their subdag (ancestors) and that are not yet marked as visited: (get-path *dag' path'*), with *path'* := *path* enlarged by the attribute of the root of *dag*
5. search for value in *dag*
   path $\neq$ NIL $\rightarrow$ if $\exists$ *dag'* that is subdag of *dag* (descendant) behind the first attribute specified in *path*, and that is not yet marked as visited: (get-path *dag' path'*), with *path'* := *path* shortened by the first attribute
   path = NIL $\rightarrow$ *dag* is one of the results

The result of 'get-path' consists of a list of subdags (with values realized as a special kind of DAG) that can be reached behind *path* in *dag*, thereby examining all references and unification points. Figure 9 shows how the different parts of the algorithm are used for the

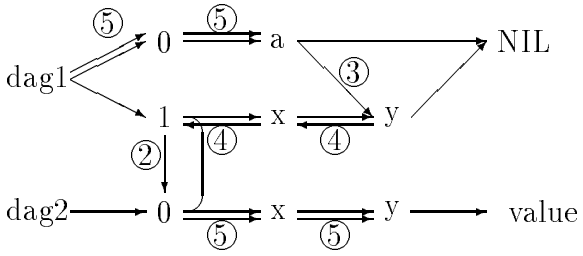


Figure 9: The Function 'get-path'

total traversing of connected DAGs. The function is called for *dag1* and the path *(0 a)*. First, the DAG is traversed according to the given path (Step 5 of the algorithm). When the node with attribute *a* is reached, the first result is found in form of the NIL-subdag (representing that *any* value is possible because no explicit value has been specified yet).

Following Step 3 of the algorithm and searching for another ancestor of the NIL-subdag, the *y*-node is reached. As pointed in Step 4 the DAG is traversed in backward direction. Since the *1*-node is connected with the *0*-subdag of *dag2* via a bidirectional reference, another call of the algorithm is made for the *0*-subdag (Step 2). During the backward traversal in *dag1*, a new path *(x y)* has been computed that is now used in *dag2* to find another result. Indeed, a subdag is found representing the value 'value'. The final result of the algorithm is a list of the two found value DAGs.

Another approach to non-destructive unification has been suggested by [De Smedt 91]. He deals with the problem of backtracking for 'flat' unification (unification without reentrancy) with disjunctive values. Destructive unification makes it impossible to recompute the original values for two feature structures which have been combined during unification. During non-destructive unification, there is no new feature structure created but the access to the original structures is filtered. Each feature structure is associated with a set of *companions* serving as filters. During each unification both feature structures become companion at their respective partner. The final result is then computed by an intersection over the sets of disjunctive values. The costs are moderate because unification is flat and not recursive.

Our approach to non-destructive unification is similar to that of De Smedt. We use references instead of companions to connect the feature structures which would have been combined during destructive unification. But the process of virtual combination (remember the function 'compatible') is much more expensive for recursive (reentrant) than for flat unification. If we could show that flat unification in combination with Tree Adjoining Grammars is sufficient for the description and incremental processing of natural language, then the realization of UTAG with bidirectional references would become cheaper. Furthermore, this could change the valuation of the two approaches to the realization of TAGs with unification that is presented in the next section.

The result of this section is that a direct implementation of our definition of TAGs with unification is possible but seems to be inefficient in comparison with structure-sharing. Since the two approaches to realization have different advantages and disadvantages, their final evaluation depends on their use.

4 Using UTAGs for Incremental and Parallel Generation

4.1 Incremental Generation

We want to use TAGs with unification within the WIP project for an incremental natural language generator. Incremental generation means for the syntactic level of processing that the construction of a complete sentence tree has to be done by combining (adjunction and substitution) partial trees on the basis of each part of the stepwise given input. Since we want to generate incremental output (cf. [Finkler & Schauder 92]) and we use feature

structures to realize inheritance and tests between syntactic features of distinct nodes it is necessary to have *connected* FSs in the derived tree at every time when a part of the output is to be produced. For example, a verb cannot be inflected before it inherits values for number and person from its subject. Immediate inheritance of information is possible for both approaches for the realization of TAGs with unification. For TAGs with bidirectional references, values can be read dynamically via the references. But this reading operation is much more expensive than for unification with structure-sharing.

On the other hand, the problem with the realization of adjunction by means of structure-sharing is that adjunctions at internal nodes trigger the rebuilding of the global feature structure. As long as adjunction is realized as a destructive operation, unification with bidirectional references is cheaper, if it is non-destructive it is more costly (see Section 3.2).

The answer to the question which approach to use depends on the design of the surrounding system. We will discuss next what parallelism means for TAGs with unification.

4.2 Parallel Generation

We use a distributed parallel model of active communicating objects to support incremental generation. The task of the objects is to build the syntactic structure of a sentence from their local knowledge that is specified in the input. Thereby, they exchange copies of their local information in order not to loose their independence which would be the case if they would combine their structures. Are there differences in the suitability of the two approaches to realization with respect to parallel generation?

Parallel Substitution

Figure 10 shows schematically how 'parallel' substitution is realized using unification with bidirectional references. The large oval boxes represent objects of the distributed parallel system, each managing one TAG tree. Again, the small boxes associated with the trees represent subdags, the links represent bidirectional references. The pair of objects in the left of Figure 10 shows the initiation of a substitution. Message passing is controled within our generator by the principle that dependent objects are in most cases active and initiate communication with their regents. The object with the substitution tree sends a copy of relevant parts of its DAGs to the goal object. Relevant are all subdags that can be read directly by structure-sharing or via bidirectional references from the 0-subdag of the root. They form exactly that piece of knowledge that NP↓ should know about its subtree. The function which computes these structures is called 'global-rule'. The parts of the DAG are sent in their uncompiled form as specification lists because this form is both well suited for message passing and useful for the test for compatibility with the respective DAG[§] Again, it is possible that the whole subtree must be traversed during the

[§] A test for compatibility is rather cheap if one of the DAGs is given in uncompiled form as specification list. The specified pairs can be used for direct tests in the DAG, e.g., ((0 attribute) value) triggers the

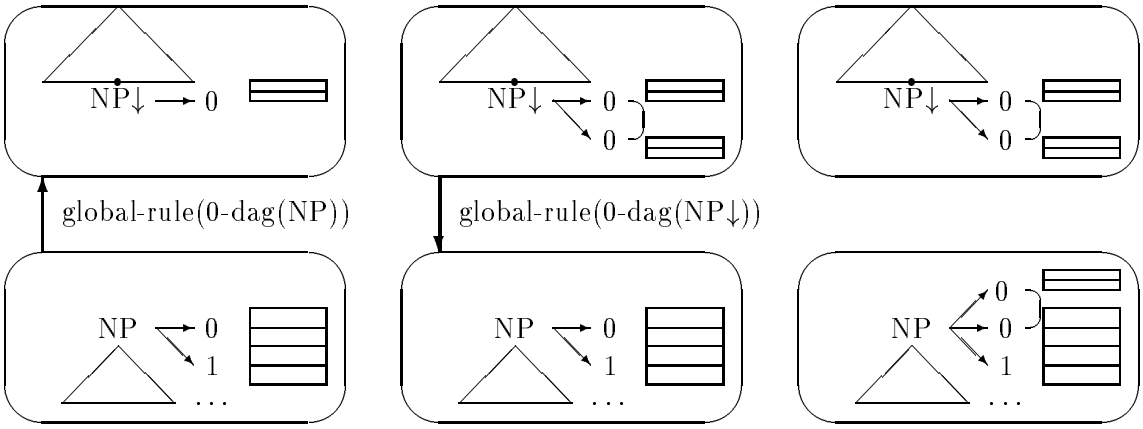


Figure 10: Parallel Substitution using Unification with Bidirectional References

computation of these lists (compare the function 'get-path', Page 13). In the next step of 'parallel' substitution, the goal object receives the message, compiles the sent rule into a DAG and tests for the compatibility of this DAG with the 0-subdag of the substitution node. In case of success, it sends a copy of relevant parts of its DAGs (all that can be read via the 0-subdag of the substitution node) back to the initiator of the substitution. It connects the new DAG (the so-called 'partner-dag') with the 0-subdag of NP↓ by a bidirectional reference. Each further flow of information from the subtree just leads to changes in the partner-dag, never in an original DAG of the tree. The same – except for the test for compatibility – is done within the object with the substitution tree.

'Parallel' substitution using unification with structure-sharing is shown in Figure 11. It is very similar to substitution using unification with bidirectional references. The computation of the rule that represents the DAG at the root node of the substitution tree is cheaper because the basis is a compact DAG with structure-sharing within the whole tree. But lateron it is more expensive to keep the original rules of an object separated from the rules that are sent by other objects. Each mixing of rules is problematic because it is important that rules can be associated with the individual nodes in an unambiguous way if adjunction changes the global DAG structure.

Since the specification lists (or 'rules') are stored at the nodes as the basis for rebuilding the global FS the same is done for the sent rules: They are stored as 'partner-rules', are compiled into DAG-form and are unified with the 0-subdag of the substitution node (in the middle of the figure) or the root node (in the right of Figure 11).

The costs for parallel substitution using unification with bidirectional references and using unification with structure–sharing are summarized in the following table:

test, whether 'value' can be found in the DAG behind 'attribute'.

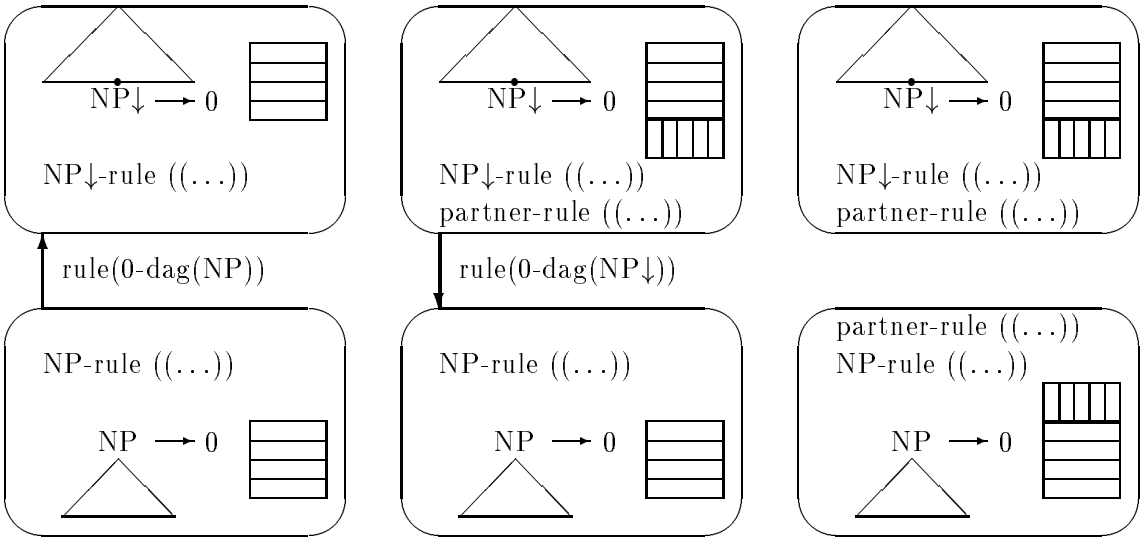


Figure 11: Parallel Substitution using Unification with Structure-Sharing

	Costs for parallel substitution using	
	unification with bidirectional references	unification with structure–sharing
computation of rules via references	2	-
computation of rules via structure–sharing	-	2
compatibility test with structure–sharing	1	1
setting new references	2	-
unification of subdags	-	2

While the computation of rules is cheaper in the realization of unification with structure–sharing, the unification of subdags is more expensive than setting references. It seems that it depends on the internal structure of the DAGs and their relations which of the two possible realizations is the better one. The larger the DAGs are and the more they are distributed over the tree, the stronger are the advantages of unification with structure-sharing.

As long as unification is used during the monontonic operation substitution, unification with bidirectional references seems to be slightly worse than the other approach. But for adjunction – especially for a quasi-destructive realization of adjunction, i.e. without copying the trees – unification with bidirectional references is better suited, because adjunction was one of the motivations for this realization.

Parallel Adjunction

Interestingly, we found it necessary to use a concept similar to 'quasi-nodes' when we defined 'parallel' adjunction (adjunction in a parallel system), even before we knew

about [Vijay-Shanker 92]. Remember the idea of parallel substitution: The substitution node and the root node of the substitution tree are locally associated with DAGs that represent the sub- or supertree from the partner, respectively (called partner-dags or partner-rules). This can easily be realized because all nodes that are involved in substitution are either leaves or root nodes. In contrast with this, nodes of adjunction are often internal nodes. In order to represent the inserted auxiliary tree – without really combining the two trees, i.e. the two objects – the node of adjunction must be associated with *two* partner-dags: The first represents the view to the auxiliary tree from the top and must be connected with the FSs in the supertree of the node of adjunction, the second represents the view from the bottom and must be connected with the FSs in the subtree.

Figure 12 shows how parallel adjunction using unification with bidirectional references can be realized. During the first step (in the left of the figure) the two FSs that are associated with the node of adjunction X are transformed into specification lists and sent

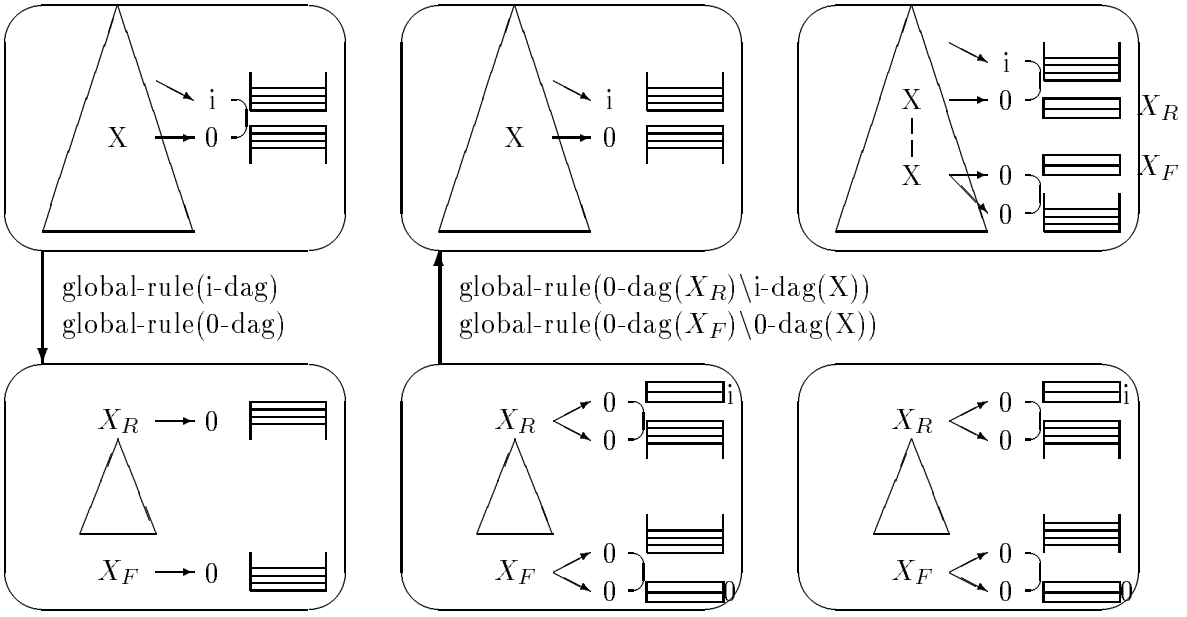


Figure 12: Parallel Adjunction using Unification with Bidirectional References

(as two different rules) to the object that selected the auxiliary tree. They represent the future relation of the root of the auxiliary tree X_R to the supertree of X and the relation of X_F to the subtree of X. The second step can be seen as a sequence of two substitutions: The DAG of X_R is connected with the DAG, that represents the top-FS of X, then the DAG of X_F with the bottom-FS of X. If both pairs of DAGs are compatible then the insertion of the FS of the auxiliary tree between the top- and bottom-FS of X has succeeded.

The resulting new structures must also be represented in the upper object at the node X. Therefore, two specification lists are computed: The first represents the view to the auxiliary tree from the top (from X_R), including all FSs that can be reached

via unification points and bidirectional references within the auxiliary tree but without the top-FS of X because this information came from the upper object. If all DAGs are connected information from all FSs of the auxiliary tree can be read including the FSs of the former subtree of X that is now connected with the foot node. The second specification list is computed for the foot node in the same way. They are sent to the upper object where in the third step the node of adjunction is split and each part is associated with the respective DAG. The top X-node is associated with the DAG that is compiled out of the specification list of X_R. In this way, the supertree of X can have access to the knowledge in the inserted auxiliary tree and the former subtree of X. The bottom X-node is associated with the DAG that is compiled out of the specification list of X_F. So the other way round, the subtree can have access to the knowledge in the inserted auxiliary tree and the supertree. The FSs of the inserted auxiliary tree are represented in two parts because then each further flow of information through the auxiliary tree into the X-node or the other way round can be handled in exactly the same way as if two substitution nodes were defined there.

Any further auxiliary tree that is to be adjoined in X in the upper object is now forwarded to the object with the auxiliary tree and there can be adjoined in the root node or the foot node, respectively.

For unification with structure-sharing it is not possible to insert feature structures just by cutting and setting bidirectional references. As motivated in Section 3.1, adjunction using unification with structure-sharing can only be realized on the basis of locally associated specification lists. All DAGs of the resulting tree must be rebuilt, using the specification lists that are associated with the nodes. Figure 13 shows what parallel adjunction means for unification with structure-sharing. Similar as for unification with bidirectional references, the first step of parallel adjunction consists in computing the specification lists that represent $\uparrow$X and $\downarrow$X of the node of adjunction X in the upper object. They are sent to the object with the auxiliary tree where they are associated as partner-rules with the root and the foot node. They are compiled into feature structures and unified with the global DAG of the auxiliary tree. Then the new view from the top and from the bottom to the auxiliary tree is computed and sent to the upper object. There the node of adjunction is doubled, all feature structures of the tree are thrown away and rebuilt for two separated parts of the tree. The first part consists of FSs of the supertree of X unified with the DAG representing the view to the inserted auxiliary tree from the top including the former subtree of X. The second part consists of the FSs of the subtree of X unified with the DAG that represents the view to the inserted auxiliary tree from the bottom including the former supertree of X.

A rough comparison of the two approaches shows that parallel adjunction tends to be cheaper for unification with bidirectional references than for unification with structure-sharing. While it is more expensive to compute the global copies of FSs for unification with bidirectional references, the (virtual) insertion of the auxiliary tree in the goal tree is much more cheaper because it just consists of setting two references. When realizing TAGs using unification with structure-sharing, all FSs of the tree with the node of adjunction

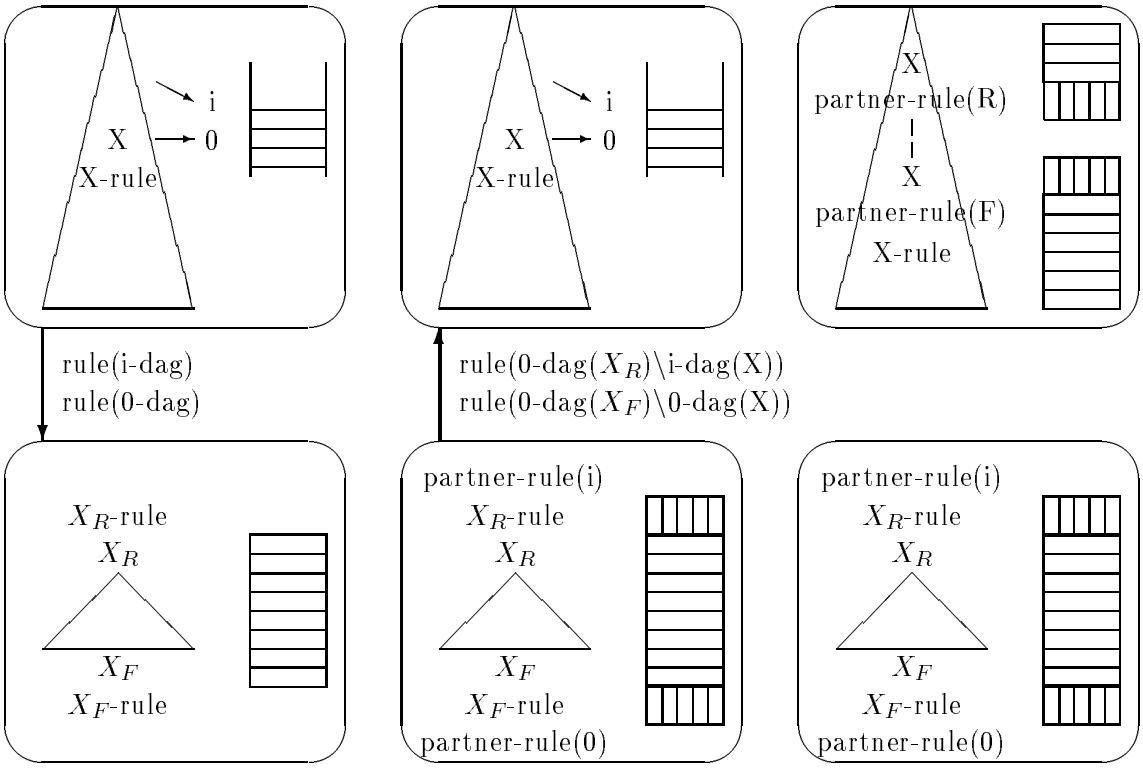


Figure 13: Parallel Adjunction using Unification with Structure-Sharing

have to be rebuilt. Even worse is the fact that these kinds of nonmonotonic changes in local trees can trigger a 'flow of nonmonotonically changed information' through a set of connected objects. If an object receives a new partner-rule from its partner object that has been changed nonmonotonically, then it can only include the new information into its global feature structure in the same way as if an adjunction would take place. It has to throw away all FSs and rebuild them using the new partner-rule. The reason for this is the same as for adjunction: The parts of the DAG that have been nonmonotonically changed and that must be replaced cannot be identified because of structure-sharing.

This observation brings the two approaches to realization into balance again.

4.3 Conclusion

The consequence of comparing the two approaches to realization is that TAGs using unification with structure-sharing seem to be better suited for substitution and TAGs using unification with bidirectional references seem to be better suited for adjunction. So it depends on the design of the grammar and on the statistical occurrence of the respective forms in natural language, which of the two approaches is to be preferred. Much more work in the linguistic part of our project must be done and more detailed benchmarks measuring the run time have to be made in order to answer this question.

While the two described approaches to realization are rather contrastive it is possible to imagine some solution in between. One alternative would be to associate parts of feature structures with information about their origins. In this way, it would be possible to identify and delete specific parts of FSs during adjunction. Another idea is to use more links at deeper parts of FSs in order to ease the access to connected FSs and/or to associate links with information about the connected FSs. This could help to guide the access to parts of FSs in a more specific way. Perhaps it would be possible to adapt some algorithms for unification that allow to recompute the original FSs for TAGs with unification (e.g., see [Tomabechi 91]). Searching for a better realization for non-destructive unification for UTAGs is an interesting problem for which no satisfying solutions exist.

5 Comparison of UTAGs and FTAGs

[Vijay-Shanker & Joshi 88] defined another kind of combination of TAGs and unification within the FTAG formalism that will be described and compared with our definition of UTAGs in the next two sections.

5.1 Feature Structure based TAGs (FTAGs)

Vijay-Shanker and Joshi motivate their approach with the following observation: Each (internal) node of a TAG tree serves at the same time as a father and as a son for the surrounding nodes. These distinct relations to the subtree and the supertree are reflected in a basical separation of the feature structures associated with each node. Feature structures at TAG nodes consist of a t- (for top) and a b- (for bottom) part representing the relation to the supertree and the subtree, respectively.

Figure 14 shows how feature structures are handled during adjunction. The FS parts of the node of adjunction are called t and b, the equivalent parts of the root node of the auxiliary tree t_r and b_r, the parts of the foot node t_f and b_f. When the auxiliary tree substitutes the node of adjunction during adjunction, its t- and b-parts are transferred to nodes of the auxiliary tree. Those nodes can directly be found: Since t reflects the relation of the node of adjunction to its supertree it is transferred to the root node and unified with t_r, b is transferred to the foot node and unified with b_f.

A prerequisite for this kind of adjunction is that the t- and the b-part of a FS are separated (and not unified), because they have to be transferred to two different nodes. Since it is not known which nodes of the elementary trees will serve as node of adjunction, the t- and b-part of the FSs of all nodes must not be collapsed before all adjunctions have been made; and after the unification has taken place no further adjunction is possible. We had to integrate the same constraint into the definition of UTAGs.

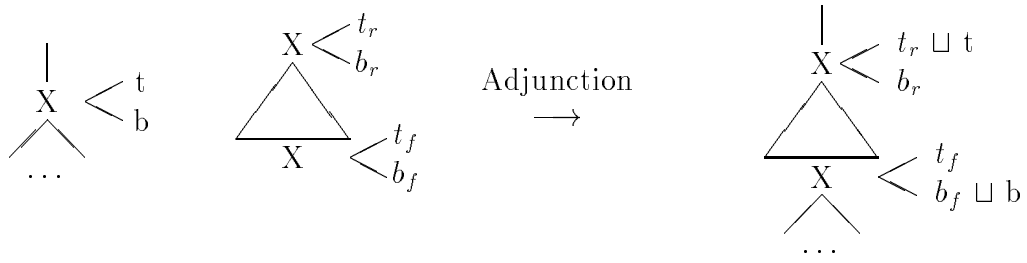


Figure 14: Adjunction with FTAGs

5.2 Comparison of UTAGs and FTAGs

As mentioned at the end of Section 2.2, FTAGs associate elementary trees with feature structures whereas UTAGs divide the trees into a CFG-like rule based system. UTAGs necessitate more elements in the specification lists compared to FTAGs to obtain inheritance of values in a tree. For FTAGs it is possible to refer directly to structures at different nodes. Therefore, UTAGs are more redundant and inefficient than FTAGs. In spite of these disadvantages, we use this limited formalism because it gives a better intuition about the flow of information in an elementary tree and about what happens during adjunction just because of its close relation to CFGs. Since we use the formalism in an incremental natural language generator – where the number of fails and revisions is expected to be rather high – it is important for us that the formalism facilitates the localization of reasons for fails. It is easier to follow a flow of information along the arcs of a tree than to check feature structures that may point to arbitrary nodes within a tree. Furthermore, we do not really loose the property of specifiying dependencies which use a whole elementary tree as domain of locality when we only allow to specify father-son relations. We can use the trees to precompile the DAGs that are associated with the nodes, thereby realizing direct connections between FSs that would not have been possible with a CFG-based system such as PATR–II.

In order to have a common basis for the further comparison we presuppose that FTAGs are restricted to father-son or brother-brother relations in the definition of unification constraints.

This restricted definition of FTAGs can best be compared with the definition of UTAGs on the basis of a generalized definition of FTAGs that has been introduced by [Vijay-Shanker 92]. He states that it is useful not to assume that the building blocks of an FTAG are trees but that they are some tree-like structures, i.e., partial descriptions of trees. This idea can directly be used for the definition of a new and more general formalism, using so-called 'quasi-nodes' that are pairs of nodes. Thereby, a top quasi-node dominates a bottom quasi-node. The domination relation is defined to be reflexive, so the two quasi-nodes can either be identified (if no adjunction takes place) or their dominance relation is further specified by the insertion of an auxiliary tree (adjunction). Both nodes are associated with one feature structure: the top-node with the t-part and the bottom-

node with the b-part. In this way, the idea of the first definition of FTAGs is just set into a more general context.

Interestingly, the definition of UTAGs – that has been strongly influenced by the PATR–formalism and not by the idea of explicitly doubling nodes – fits into the same generalized formalism. This has already been mentioned by [Vijay-Shanker 92]. The notion of ↑X and ↓X is very similar to the t- and the b-part of a FS at a node in FTAG. Figure 15 illustrates this similarity. The left tree shows nodes with feature structures

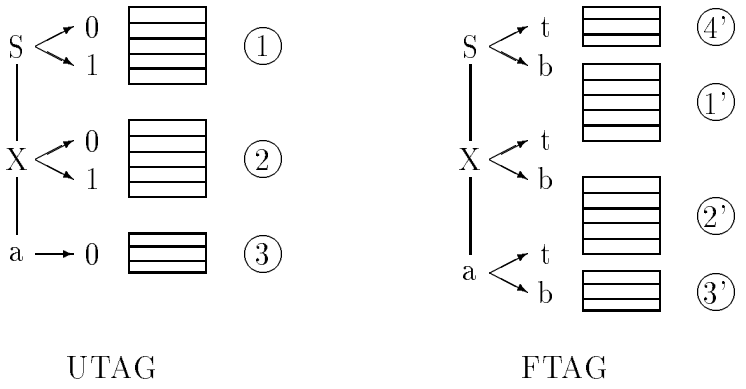


Figure 15: Comparison of UTAG and FTAG trees

according to UTAG. Each feature structure contains n+1 top-level branches (n be the number of sons of the node) whose attributes are reference numbers. The subdags reached via the reference numbers are combined by structure-sharing which is illustrated by boxes in the figure. For the i-th son of a node, the i-subdag of the node is related with the 0-subdag of the son. Depending on the kind of realization they are connected by a bidirectional reference or destructively unified.

The right tree corresponds to the (original) FTAG definition. Each node is associated with two FSs, called t- and b-part. While the t- and b-part of the FS at each single node must be kept separated in order to allow the insertion of FSs by adjunction, each b-FS of a node may be unified with the t-FSs of its sons, leading to the boxes shown on the right side of the tree.

The boxes are associated with numbers (i for UTAGs and i' for FTAGs) which illustrate corresponding FSs. This correspondence will become more clear in the next figure.

Transferring the example to the generalized definition of TAGs with quasi nodes leads to Figure 16. The differences between the two trees diminish. First, there is a rule – which is either realized procedurally or declaratively by using the bidirectional references – that describes the fact that the each i–subdag of a node and the 0–subdag of its i–th son *should* be unified. This unification is forbidden because of the well-known reasons, at least as long as adjunctions are allowed. For FTAGs, there exists a similar statement: when no more adjunction is to be done, the t- and b-FSs of all nodes may be unified. Again, the structures are identified, that are to be combined (this relation is illustrated in Figure 16 by double lines).

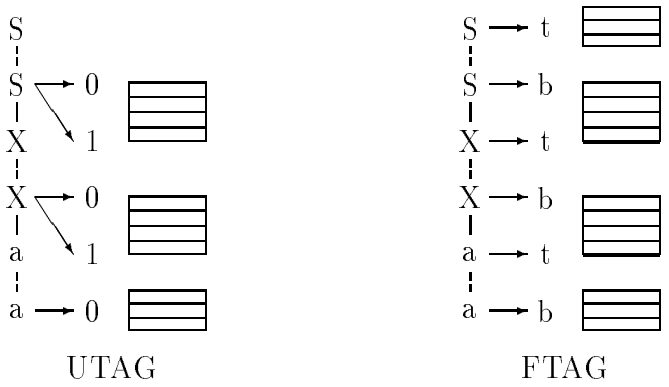


Figure 16: UTAG and FTAG with Quasi Nodes

Second, regarding the distribution of the boxes in both trees, the main difference seems to be the place (the node) where feature structures are defined. For UTAGs, each node is associated with its own bottom-FS and additionally with all top-FSs of its sons. For FTAGs, each node is associated with its own top- and bottom-FSs. For both trees, bottom-FSs are unified with the top-FSs of the sons. But for UTAGs this leads to compact feature structures associated with each node (which was the aim of our definition), for FTAGs compact feature structures are shared by different nodes (father and sons) of the tree, which corresponds to the principle of unification grammar (father and sons represent a contextfree rule).

There remains one difference that becomes visible in Figure 16. Since – for UTAGs – the top-FS of a node is defined with its father, there is no top-FS of the root node of a tree. That is why, an adjunction in a root node may lead to qualitatively different results for UTAGs and FTAGs (see Figure 17). In FTAG-trees the root node can be seen as quasi-node with a top- and a bottom-FS. This holds for auxiliary trees as well as for

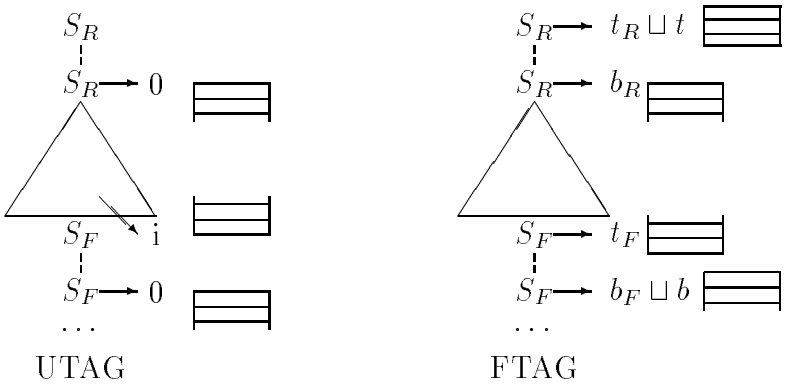


Figure 17: Adjunction in a Root Node for UTAG and FTAG

initial trees. Therefore, adjoining an auxiliary tree into the root node of an initial tree leads to a resulting tree with a root node FSs of which are computed from the FSs of

the two original root nodes. Following the definition of adjunction, the top-FS of the root node of the auxiliary tree (t_R in Figure 17) is unified with the top-FS of the node of adjunction (t in the figure). In this way, parts of the FS that is specified at the node of adjunction (which is a root node) can move to the root of the resulting tree.

This doesn't hold for UTAG-trees as neither for auxiliary nor for initial trees there can be t-FSs defined in root nodes because there are no ancestors where the i-subdag could be defined. It is not clear whether this property facilitates the representation and computation of structures during the processing of natural language.

Does the same hold for substitution? While in the FTAG formalism the root node of a substitution tree can be associated with a top-FS for its future supertree, this cannot be done for UTAGs. But the other way round, in UTAGs a substitution node can be associated with t- *and* b-FSs, since it has a father node, in FTAG this separation is not realized, because no adjunction may take place in a substitution node. So the only difference seems to be the place, where the pair of feature structures can be specified.

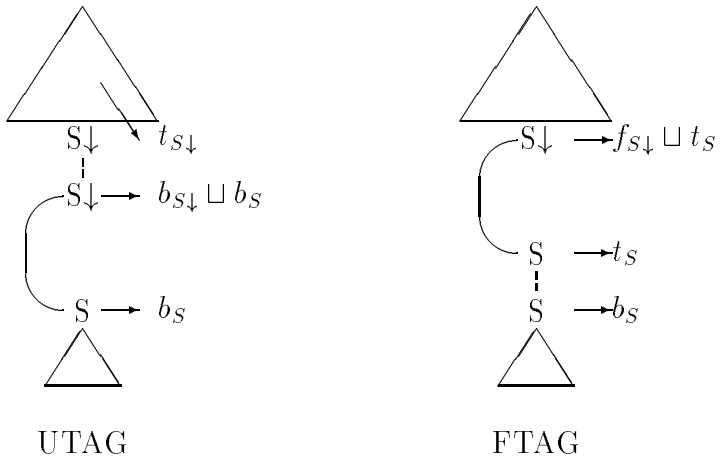


Figure 18: Substitution in UTAG and FTAG

The comparison of UTAGs and FTAGs does not lead to a clear result for the question which formalism should be preferred. It seems to strongly depend on the taste of the grammar designer if he likes PATR-style notation more than top/down separation of feature structures and if he wants to define constraints directly between arbitrary nodes of elementary trees.

Because of the close relation of the two formalisms the possible realizations for TAGs with unification that have been described in the last chapter can be applied to both of them.

References

[Buschauer et al. 91] B. **Buschauer**, P. **Poller**, A. **Schauder**, and K. **Harbusch**.

Tree Adjoining Grammars mit Unifikation. Technical Memo TM-91-10, DFKI, Saarbrücken, FRG, 1991.

[De Smedt 91] K. **De Smedt**. *Revisions during Generation using Non-destructive Unification.* In: Proceedings of the Third European Workshop on Natural Language Generation, 1991.

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

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

[Joshi 85] A. **Joshi**. *How much context-sensitivity is required to provide reasonable structural descriptions: tree adjoining grammars.* In: D. Dowty, L. Karttunen, and A. Zwicky (eds.), Natural Language Processing: Psycholinguistic, Computational and Theoretical Perspectives. Cambridge: Cambridge University Press, 1985.

[Karttunen 86] L. **Karttunen**. *D-PATR: A Development Environment for Unification-Based Grammars.* Technical report, SRI International and Center for the Study of Language and Information, Stanford, 1986.

[McDonald 87] D. **McDonald**. *Natural Language Generation: Complexities and Techniques.* In: S. Nirenburg (ed.), Machine Translation. Cambridge: Cambridge University Press, 1987.

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

[Tomabechi 91] H. **Tomabechi**. *Quasi-Destructive Graph Unification.* In: 29th Annual Meeting of the ACL, University of California, Berkeley, California, 1991.

[Vijay-Shanker & Joshi 88] K. **Vijay-Shanker** and A. **Joshi**. *Feature Structure based Tree Adjoining Grammars.* In: Proceedings of the 12th International Conference on Computational Linguistics (COLING'88), Budapest, 1988.

[Vijay-Shanker 92] K. **Vijay-Shanker**. *Using Descriptions of Trees in a Tree Adjoining Grammar.* Computational Linguistics, 1992. to appear.

[Wahlster et al. 91] W. **Wahlster**, E. **André**, W. **Graf**, and T. **Rist**. *Designing Illustrated Texts: How Language Production is influenced by Graphics Generation.* In: Proceedings of the EACL'91, Berlin, FRG, 1991.

[Wahlster et al. 92] W. **Wahlster**, E. **André**, W. **Finkler**, H.-J. **Profitlich**, and T. **Rist**. *Plan-based Integration of Natural Language and Graphics Generation.* Artificial Intelligence Journal, 1992. to appear.

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