# On Using a Parallel Graph Rewriting Formalism in Generation

**Bernd Bohnet** and **Leo Wanner**
CS Department, Intelligent Systems Group
University of Stuttgart
Breitwiesenstr. 20 - 22, 70565 Stuttgart, Germany
{bohnet|wanner@informatik.uni-stuttgart.de}

**Abstract**

In this paper, we present a parallel context sensitive graph rewriting formalism for a dependency-oriented generation grammar. The parallel processing of the input structure makes an explicit presentation of all alternative options for its mapping onto the output structure possible. This allows for the selection of the linguistic realization that suits best the communicative and contextual criteria available.

## 1 Introduction

Graph-rewriting formalisms received a considerable attention in generation grammar implementations and in the area of transfer in machine translation. A graph-rewriting formalism is either sequential or parallel (Rozenberg, 1997). A sequential graph-rewriting formalism gradually transforms an input structure specified in the formal language $\mathcal{L}_1$ into an output structure, which is specified in the formal language $\mathcal{L}_2$, by using explicitly or implicitly defined rewriting rules.[1] Explicit rewriting rules may have the format of classic rewriting rules, as, e.g., in (Frank, 1999) or of bidirectional rules that establish a correspondence relation between minimal structures of $\mathcal{L}_1$ and $\mathcal{L}_2$, as, e.g., in (Iordanskaja et al., 1988; Lavoie and Rambow, 1997). Implicit rewriting rules are encoded in terms of $\mathcal{L}_1$-constraints that are associated with structure chunks and lexical items of $\mathcal{L}_2$; see, e.g., (Nicolov et al., 1996; Beale et al., 1998; Stede, 1999).

A parallel graph-rewriting formalism maps a given input structure to an output structure instead of transforming the former into the latter. Although parallel graph-rewriting shows several advantages when compared to sequential graph rewriting (see Section 3), sequential graph-rewriting formalisms are much more common.

In this paper, we present the implementation of a parallel graph rewriting formalism for the grammar of the *Meaning-Text Theory* (MTT) (Mel'čuk, 1981; Mel'čuk, 1988). The focus of the presentation is on one of the major stages of the algorithm: the spelling out which rules are to be applied to which fragments of the input structure in order to achieve its most optimal coverage. This is a search problem.

In the next section, a brief introduction to MTT and its formal basics is given. In Section 3 we present the stages of processing in parallel graph rewriting. Section 4 explains the search algorithm in detail and presents an example of how the search algorithm works in practice. Section 5 discusses some of the related work in this area. In Section 6, finally, a summary and some conclusions are given.

## 2 The Meaning-Text Theory

### 2.1 Linguistic Foundations

The Meaning-Text Theory is a multistratal dependency theory. Five of its strata are immediately relevant for generation: (1) the semantic stratum, (2) the deep-syntactic stratum, (3) the surface-syntactic-stratum, (4)

[1]Note that $\mathcal{L}_1$ and $\mathcal{L}_2$ may be identical, but do not need to be so.

the deep-morphological stratum, and (5) the surface-morphological stratum, which is the linearized surface structure. Linguistic structures at the semantic stratum are predicate-argument structures, i.e., directed acyclic graphs in which nodes stand for predicates and objects, and edges establish relations between predicates and their arguments (with each edge being labelled by the number of the respective argument). Linguistic structures at both syntactic strata are dependency trees with lexemes being represented as nodes and syntactic relations as edges. At the deep-syntactic stratum, the encoded syntactic relations are *actant* or *participant* relations. The actant relations are not named (as, e.g., in the systemic grammar), but simply numbered by I, II, III, ...). As (grammatical) functions in the *f*-structure in LFG, actant relations are assumed to be universal. At the surface-syntactic stratum, the encoded syntactic relations are language-specific grammatical functions (such as subject, direct object, etc.). Linguistic structures at the morphological strata are (ordered) sequences of word forms. Figure 1 shows the deep-syntactic structure and the surface-syntactic structure for the sentence *The assembly forced Socrates to drink the cup of hemlock in the dawn.* In the deep-syntactic structure, the dashed line represents the referential link between the two 'Socrates' nodes.

At all levels of representation, the nodes of linguistic structures are, in fact, feature structures. They are defined in terms of attribute-value pairs (such as, e.g., '`lex = Socrates`', '`cat = verb`', '`voice = passive`', etc.). For instance, the node `force` carries the attribute-value pairs `cat = verb`, `form = finite`, `tense = past`, `voice = active`. In the graphic representation, the attribute-value pairs of a node are shown only upon request. Both the nodes and the attributes are typed.[2]

The grammar in MTT is *a priori* an equative

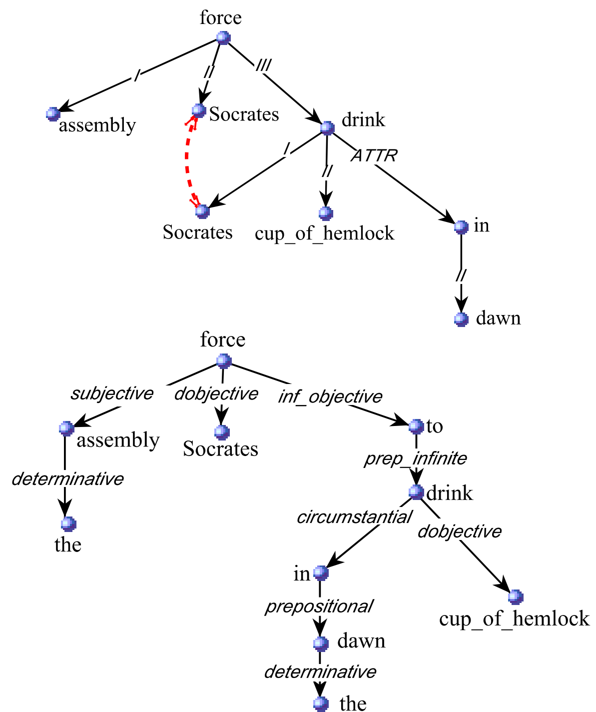


Figure 1: Deep-syntactic and surface-syntactic structures of the sentence *The assembly forced Socrates to drink the cup of hemlock in the dawn.*

device (Kahane, forthcoming). It consists of a set of rules that establish the correspondence between minimal structures at two adjacent strata—with a minimal structure being a feature of a node, a node, a relation between two nodes, or a configuration of relations. Figure 2 shows a sample grammar rule as implemented in MATE (Bohnet et al., 2000). This rule maps the deep-syntactic relation **II** (the second actant) onto the surface-syntactic relation **dobjective** (i.e., direct object). The rule applies if there is the first actant available (i.e.., the relation **I** is specified in the input structure), the verbal head of the structure contains the attribute-feature pair '`cat = verb`', and no attribute-value pair '`voice = passive`'.[3] The relation **I** is specified as being in the context. That is, it is not "consumed" by the rule; it rather serves as a constraint for

[2]As a matter of fact, the linguistic structures and the rules in MTT can be represented in terms of typed feature structures; see (Mel'čuk and Wanner, forthcoming).

[3]Note that if there would be no actant **I** available, in order to get a grammatical sentence, the second actant would have to be realized as **subjective**. In other words, in such a case, passivization would take place.

```
context:          ?Xds -I-> ?Zds
left-h.s.:        ?Xds -II-> ?Yds
conditions:       ?Xds.cat = verb
                  NOT ?Xds.voice = passive
right-h.s.:       ?Xss -dobjective-> ?Yss
correspondences:?Xss <=>?Xds
                  ?Yss <=>?Yds
```

Figure 2: A sample grammar rule.

the application of the rule.

In the process of generation, a compiler applies grammar rules to an input structure. If a fragment of the structure matches with the left-hand side of a rule, and the constraints specified in context and in the condition slots of this rule are met, the fragment is mapped onto the substructure specified at the right-hand side of the rule.

Our grammar formalism is intended to be bidirectional,[4] i.e., to be applicable for generation and for parsing. Therefore (and because of the general policy adopted in MTT), in grammar rules, only purely linguistic criteria are specified as conditions. No criteria, e.g., from the situational context are considered. As a result, the grammar might well produce several output structures. It is left to specific submodules of sentence planning (such as lexicalization, syntacticticization, etc.) to further restrict and monitor the realization of an input structure by the grammar. However, we do not discuss the interaction of these modules with the grammar in what follows. Rather, we restrict ourselves to the presentation of the grammar formalism.

### 2.2 Formal Description

From the formal viewpoint, linguistic structures can be considered as attributed graphs with a variant degree of freedom.

**Definition 1 (Attributed Graph)** *Let $\Sigma^e$, $\Sigma^a$ and $\Sigma^{val_{1,..,n}}$ be sets with $\Sigma^e$ being the set of edge labels, $\Sigma^a$ the set of attributes used in node descriptions, and $\Sigma^{val_{1,..,n}}$ the set of possible attribute values.*

[4]When a generation rule as shown in Figure 2 is reversed, the conditions in the conditions-slot become right-hand side statements. The context information does not need to be modified since the formalism allows for 'right-hand side contexts'.

*Then a directed attributed graph is a triple $G_i = \langle N, E, A\rangle$ where $N$ is a finite set of nodes, $E$ is a subset of $\Sigma^e \times N \times N$, $A$ is a subset of $N \times ((\sigma^1 \times \Sigma^{val_1}) \cup \ldots \cup (\sigma^n \times \Sigma^{val_n}))$ and $\sigma^j \in \Sigma^a$, and $i \in \{Sem, DSynt, SSynt, DMorph, SMorph\}$.*

In $(e, n_1, n_2) \in E$, $n_1$ is the source node and $n_2$ is the target node of the edge $e$.

In this context, an MTT- grammar rule is a graph rule of the following kind:

**Definition 2 (Graph Rule)** *A graph rule is a quintuple $GR = \langle G_l, G_r, C, R, c\rangle$. $G_l$ is the left-hand side (connected) graph and $G_r$ is the right-hand side graph as defined in Definition 1. $C$ is the set of conditions which must hold in order for the rule to be applicable. $R$ is the relation between parts of $G_l$ and $G_r$. $c$ is a function that is defined for each node, and edge: $c(x) = \{y|(x, y) \in C_o\}$ with $C_o = N \times \{context\ consume\} \cup E \times \{context\ consume\}$.*

$R$ is a subset of $N \times N$; it is what in graph grammar literature is called " embedding". In the most simple case, $R$ holds between nodes of $G_l$ and $G_r$.

Due to space restrictions, we don't introduce the definition of the conditions here. The interested reader can consult the MATE-Manual (Bohnet et al., 2001a).

A graph grammar $GG$ consists thus of a set of static rules of the above kind. A graph system compiles then a given "source" graph using $GG$ into a "destination" graph—in our scenario a structure at a given stratum into a structure at the stratum adjacent to the former. It can thus be defined as outlined in the next section.

## 3 Graph Systems

### 3.1 Basic Approaches

**Definition 3 (Graph System)** *A graph system is a triple $G = \langle G_{G_l}, GG, G_{G_r}\rangle$. $G_{G_l}$ is a set of graphs at a given stratum; $GG$ is the graph grammar applicable to $g \in G_{G_l}$, and $G_{G_r}$ is the set of graphs resulting from the application of $GG$ to $g \in G_{G_l}$.*

The problem one faces when using a Graph System is to find the optimal strategy for matching the $G_l$s (see the Definition 2 above)

of the rules with fragments of $G_{G_l}$. As mentioned above, there are two different basic approaches for how to proceed: (i) sequential graph rewriting and (ii) parallel graph rewriting (Rozenberg, 1997).

Sequential graph rewriting systems identify fragments of the source graph that match with the left-hand side of one of the given graph rules and replace these fragments with the right-hand side of the rule in question. By a successive application of the rules to the source graph, the latter is rewritten. In the course of the process, we have thus an intermediate graph that consists partly of the vocabulary of the source side language and partly of the vocabulary of the target side language. The process terminates if there are no more rules applicable to the intermediate graph. The main problem one faces when following the sequential graph rewriting approach is thus to figure out how to embed a new chunk gained from the application of a rule into the intermediate graph produced so far.

The sequential graph rewriting approach bears some disadvantages when applied to generation. For instance, in order to achieve a predictable resulting structure, the rules must be ordered before hand. However, a predefined ordering of rules is linguistically not justified. Furthermore, in generators that separate the task of grammar processing from the tasks of sentence planning (as is the case in our generator), it must be possible to examine which alternative structures are possible in the given situation context so as to invoke the generation of the most appropriate one. In a sequential graph rewriting approach, this requires a non-trivial book keeping overhead for backtracking or alternative structure processing.

Parallel graph rewriting systems identify parts of the source structure that correspond to the left-hand side of one of the available rules in the same way sequential graph rewriting systems do. However, unlike in a sequential system where the rules are applied in sequence to intermediate graph structure, in a parallel system, first a "rule binding map" (or "lock map"; see below) of the source graph is created. In this map, it is indicated which rules are applicable to which fragments of the graph. This allows for the determination of an optimal "coverage" of the source graph by the available rules before the rules are actually executed "in one shot". That is, no intermediate graph structure is produced and no unmodified parts of the source graph appear in the resulting graph. The main problem one faces when following the parallel graph rewriting approach is thus to find optimal strategies for binding rules to the source graph and for unifying the resulting fragments.

We chose the parallel approach because of four reasons. First MTT defines the correspondence of meaning and text in terms of *equative* rules. This view is supported by the parallel approach.[5] Second, the parallel approach allows for a more powerful concept of a context sensitive graph rule. That is, a parallel graph rewriting rule can contain a declaration of a context—a chunk of the source graph which must be available for the rule to be applicable, but which is not "consumed", i.e. mapped onto the target side, when the rule is executed. Contexts provide an indispensable means for making rules as specific as necessary and as elementary as desired. This is possible only because the source graph does not change in the course of the process. Third, in the parallel approach, the grammarian does not need to take care of the order in which the rules should be applied. Furthermore, if the same grammatical resources are to be made available for generation and parsing, no hard wired order of rule execution can be accepted. Fourth, the parallel graph rewriting approach allows for (but does not enforce) the generation of alternative result structures. This is useful, e.g., for grammar maintenance, and for advanced sentence planning strategies.

[5] Although, to our knowledge, all so far existing MTT-based generators use the sequential approach.

## 3.2 Implementing a Parallel Graph System for Generation

**A parallel graph system cycle.** In the realization of a parallel graph system for MTT-grammars, the mapping between the graphs of two adjacent strata $S_i$ and $S_{i+1}$ is performed in cycles. A cycle consists of five stages: (1) binding, (2) evaluation, (3) clustering, (4) application and (5) unification.

In what follows, we briefly introduce (1) to (5). Since the binding stage (in combination with the evaluation of simple conditions) is the most difficult (and the most interesting) part, we discuss it in Section 4 in more detail.

**Binding**. In the source graph $g_x \in G_i$, all parts that match the left-hand side $d_x \in D_l$ of one or several rules that are available for $S_i \Leftrightarrow S_{i+1}$ are identified and bound. Obviously, a rule may match more than one part in the source graph. To increase efficiency one-node (= simple) conditions such as

```
Lexicon::(?Xdsyn.lex).cat = noun
?Ydsynt.theme = yes
NOT ?Ydsynt.perspective = background
```

are evaluated already during the binding stage.

**Evaluation of complex conditions**. After the binding stage, the evaluation of complex conditions takes place. Unlike simple conditions, complex conditions draw on several nodes in the input structure; cf., e.g.:

```
?Xds.form = finite AND ?Yds.form =
infinite
```

The result of the evaluation stage are sets of instances of applicable rules.

**Clustering**. During the clustering stage, rules that are applicable together to the input structure in question without contradicting each other are grouped or "clustered". Two rules contradict if they apply to the same fragment of the input structure.

The clusters are retrieved from the "lock map", which contains the association of rules to fragments of the input structure.

Figure 3 shows a screen shot of the lock map as presented in the inspector of MATE while processing the deep-syntactic structure in Figure 1. The first column contains the names of instantiated nodes and relations of the input structure (in parentheses, the numbers of instances are given that are used by the compiler for book keeping). The first row contains the names of rules that apply to the given input structure. The application of a rule to a fragment of the input structure is marked by an 'xl' in the respective slot of the lock map matrix. 'xl' stands for "exclusive lock". That is, only one rule is allowed to apply to a fragment. If two locks occur in one row, there is a contradiction and two clusters are built. The numbers in the slots of the lock map matrix are numbers of rule instances that lock non-exclusively the respective parts of the input structure as context.

| nodes and relations / instances | 0 circum... | 1 dobje... | 2 subje... | 4 infiniti... | 5 subje... | 3 stand... |
|---|---|---|---|---|---|---|
| drink:1 (m288e0) | | | | xl | | |
| in:1 (m289e1) | | | | | | |
| dawn:1 (m290e2) | | | | | | |
| Socrates:1 (m291e3) | | | | | | |
| cup_of_hemlock:1 (m292e4) | | | | | | |
| force:1 (m293e5) | | | | | | |
| assembly:1 (m294e6) | | | | | | |
| force:1 (m293e5) -II-> drink:1 (... | | | | [129] | | |
| drink:1 (m288e0) -ATTR-> in:1... | xl | | | | | |
| drink:1 (m288e0) -I-> Socrate... | | [124] | xl | | | |
| drink:1 (m288e0) -II-> cup_of_... | | xl | | | | |
| in:1 (m289e1) -II-> dawn:1 (m... | | | | | | xl |
| force:1 (m293e5) -I-> assemb... | | | | | xl | |

Figure 3: The lock map.

**Application**. During the application stage, the rule clusters specified in the lock map are applied to the respective parts of the input structure and thus fragments of the output structure (as specified in the right-hand sides of the rules) are generated. As Figure 4 shows, the result of this stage are isolated elementary structures that are similar to elementary trees of a TAG and segments of the Segment Grammar.

**Unification**. During the last stage, the stage of unification, the elementary structures are "glued" together. That is, nodes, which correspond to the same source node are unified. For each cluster a result structure is generated. Operations at this stage are equivalent to *substitution* in a TAG.

**Cycle repetition.** A repetition of the processing cycle as sketched above becomes necessary if one or several rules contain in the

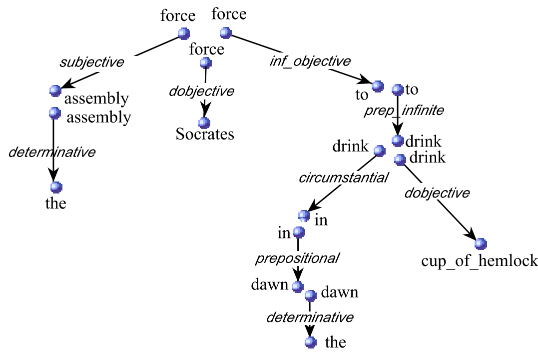


Figure 4: Elementary structures as produced at the stage oft rule application.

context slot a target substructure (i.e., a structure produced by the preceding rules). In this case, the stages (1) to (5) are repeated with rules which access the target structure and rules that compete with target structure accessing rules (i.e., rules that apply to the same parts of the source structure).

The termination of cycle repetition is ensured since no correct grammar rule accesses only the target structure. In other words, each rule consumes some source structure information. The algorithm terminates when the entire source structure has been "consumed".

## 4 Binding

Above, we introduced as the first stage of graph processing the binding of the left-hand side in a source graph and the binding of the context of a rule that contains a fragment of the target structure. In this section, we present the internals of the binding algorithm. In order to keep the presentation as simple as possible, we dispense with the discussion of some advanced features of our approach. This is for example the use of a rule hierachy, the reuse of rule instances, and an optimized strategy for rule evaluation; cf (Bohnet et al., 2000).

In what follows, the algorithm is presented and illustrated by an example.

### 4.1 Basic Algorithm

The binding procedure consists of two stages: (1) binding the source structure with the left-hand sides of $r_i \in R$ (with $R$ being the set of rules available for the the mapping between the Strata $S_i$ and $S_{i+1}$) and (2) binding (after the first cycle) the target structure generated so far with the contexts of $r_i \in R$. The output of the algorithm are instances of applicable rules. Each instance contains a copy of the rule in question and a copy of the fragment of the input structure this rule applies to.

Figure 5 shows the binding of the rule

```
left-h.s.:        ?Xds -I-> ?Yds
conditions:       NOT ?Xds.voice = passive
                  ?Xds.form = finite
right-h.s.:       ?Xss -subjective-> ?Yss
correspondences:?Xss <=>?Xds
                  ?Yss <=>?Yds
```

to the input substructure `drink -I-> Socrates`.

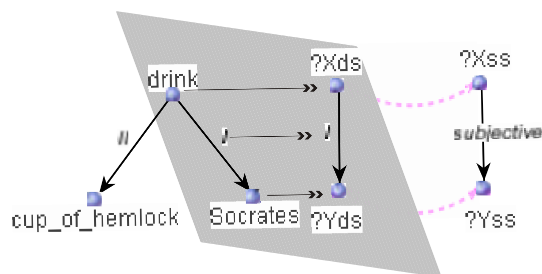


Figure 5: Example of a rule binding.

(1) and (2) can be divided into (i) searching for the initial node within the input structure from which the matching procedure starts; (ii) identify relations that match the rule structure and the source/target structure (starting with the initial node found before); (iii) the actual binding of the nodes and relations.

#### 4.1.1 Searching for an Entry Node

The entry node search function loops over all nodes of the input graph $G$ and over all candidate rules. If $G$ is a predicate-argument structure (i.e., a semantic net), the search starts from any arbitrary node of $G$; if $G$ is a tree structure, from the root node.

In pseudocode, the search function looks as follows:

searchNodes $(G, R)$
  $I_{result} \leftarrow \{\}$
  **for** $n_i \in G$ **do**
    **for** $r_j \in R$ **do**

```
      n^l_{r_j} ← getRuleNode(n_i, r_j)
      flag ← evalSimpleConditions(n_i,n^l_{r_j},r_j)
      if flag == T
         i ← createInstance(n_i, n^l_{r_j}, r_j)
         I ← {i}
         active-edges ← getEdges(n^l_{r_j})
         for e_j ∈ active-edges do
              I ← searchEdges(e_j, I, r_j, G)
         I_result ← I_result ∪ I
   return I_result
```

In the inner `for`-loop, we first pick a node $n^l_{r_j}$ in $r_j$. For efficiency, this is always the node with the highest number of simple conditions: compared to graph traversal, the evaluation of conditions is a "cheap" operation. Then, we evaluate whether the graph node under consideration $n_i$ matches the conditions of the node picked, i.e. of $n^l_{r_j}$. If the conditions match, $n^l_{r_j}$ is associated with ("bound" to) $n_i$, and the triple $\langle n^l_{r_j}, n_i, r_j \rangle$ is kept in the set of bound node instances $I$. Otherwise, we loop over the nodes in $r_j$ until either a bounding node is found or one of the conditions of all nodes in $r_j$ has been evaluated to F(alse). In the first case, the incoming and the outgoing edges of $n^l_{r_j}$ must be further matched against the incoming and the outgoing edges of $n_i$. This is done in the function `searchEdges` after the function `getEdges` retrieved all edges of which $n^l_{r_j}$ is either the tail or the head node. In the second case, $r_j$ is rejected.

The bound nodes and edges are kept in the global variable $I_{result}$.

#### 4.1.2 Searching for Edges to Match

Once an edge $e^l_{r_j}$ in a rule $r_j$ has been selected for matching, the task is to identify edges in the graph to which $e^l_{r_j}$ can be bound. A rule edge is defined as bound if both its tail node and its head node are bound. Above, the node bounding information has been kept in $I$ (in terms of triples $\langle n^l_{r_j}, n_i, r_j \rangle$). Therefore, the function `searchEdges` checks first if $I$ contains instances of both nodes of $e^l_{r_j}$. If yes, $e^l_{r_j}$ is immediately added to the set of bound edges. Otherwise, we proceed with its bound node $n^l_{r_j}$ (recall that `searchEdges` is invoked after the entry node binding procedure has been performed). If $n^l_{r_j}$ is the tail node of $e^l_{r_j}$, all edges for which the graph node $n_i$ is the tail node are plausible binding candidates. If $n^l_{r_j}$ is the head node of $e^l_{r_j}$, all edges for which $n_i$ is the head node are plausible binding candidates. `searchEdges` retrieves the graph edges accordingly and invokes the function **bindEdges**.

```
searchEdges (e, I, r_j, G)
  I_result ← {}
  for i ∈ I do
    if boundP(tail(e),i) & boundP(head(e),i)
      I_result ← I_result ∪ {i}
    else
      n^l = getBoundNode(e,i)
      if tailP(n^l,e)
        E ← getOut(getCorrespondence(n^l, i, G))
      else
        E ← getIn(getCorrespondence(n^l, i, G))
      I_result ← bindEdges(E, I_result, r_j, G)
  return I_result
```

#### 4.1.3 Bind Edges

The function `searchEdges` thus identifies a set of edges $E$ in $G$ that potentially match with an edge $e^r$ of the rule $r_j$ and calls `bindEdges`. The function `bindEdges` does the actual evaluation and binding. If the name of an $e_i \in E$ matches with the name of $e^r$ and the not yet bound node of $e_i$ $n'_{e_i}$ fulfills the conditions of the not yet bound node of $e^r$ $n'_{e^r}$, $n'_{e^r}$ is bound to $n'_{e_i}$. The instance of which $e_i$ is part is copied, and the triple $\langle n'_{e^r}, n'_{e_i}, r_j \rangle$ is added to the copy $i_c$ before the set of instances dealt with $I$ is initialized with $i_c$.

If this was the last unbound edge of the rule $r_j$, the complex conditions of the rule are checked. If they are fulfilled, the rule and the fragment of $G$ it applies to are introduced into the lock map. If $e^r$ was not the last unbound edge, the function `searchEdges` is invoked with each incoming and each outgoing edge of $n'_{e^r}$.

In pseudocode, the function **bindEdges** reads as follows:

```
bindEdges (E, i, r_j, G)
  for e_i ∈ E do
      if (name(e_i) ≠ name(e^r))
         next e_i
      endif
      n'_{e^r} = getUnbound(e^r)
      n'_{e_i} = getUnbound(e_i)
      if ∃ condition(n'_{e^r}, n'_{e_i}, r_j) ≠ t
        next e_i
      endif
```

```
    i_c ← copy(i)
    i_c ← i_c ∪ ⟨n'_{e^r}, n'_{e_i}, r_j⟩
    I ← {i_c}
    active-edges ← getEdges(n'_{e^r})
    for e_a ∈ active-edges
        I ← searchEdges(e_a, I, r_j, G)
    I_result ← I_result ∪ I
return I_result
```

### 4.2 Example

This section illustrates how the algorithm that has been presented above functions in practice. It shows the application of the rule introduced in Figure 2, which maps the second syntactic actant onto the surface-syntactic relation **dobjective**.

**a1. Searching for an entry node.**

As pointed out above, in tree structures, the seach of an entry node starts with the root. In our sample structure, this is the node `force`. In the rule, the node with highest number of simple conditions is `?Xds`. `force` meets the conditions specified for `?Xds`: its `cat` feature is set to `verb` and its `voice` feature is set to `active` (i.e., not `passive`). Therefore, `force` is bound to `?Xds`.

`?Xds` has two outgoing edges: `?Xds-I->?Zds` and `?Xds-II->?Yds` (with `?Xds-I->?Zds` being in the context). For both the function `searchEdges` is invoked.

**b1. Searching for edges to match.**

In `?Xds-I->?Zds`, the `?Xds` node is bound, while `?Zds` is not. Therefore, we get all edges in $G$ in which the node to which `?Xds` is bound (= `force`) is the tail. These are the edges `force-I->assembly`, `force-II->Socrates`, and `force-III->drink`.

**c1. Binding edges.**

In the function `bindEdges`, the relation **I** in `force-I->assembly` matches with the relation **I** in `?Xds-I->?Zds`, and `assembly` fulfills the conditions specified for `?Zds`. Therefore, `?YZds` is bound to `assembly` and, subsequently, the edge `?Xds-I->?Zds` is bound to the edge `force-I->assembly`.

The node `assembly` has no other incoming and outgoing edges. The recursion stops thus at this point.

**c2./c3. Binding edges.**

The relation **II** in `force-II->Socrates` and the relation **III** in `force-III->drink` do not match with the relation **I** in `?Xds-I->?Zds` and are thus both rejected.

**b2. Searching for edges to match.**

As in `?Xds-I->?Zds`, in `?Xds-II->?Yds`, the tail node is bound while the head node is not. The edges we get at this point for processing are the same as above for `?Xds-I->?Zds`.

**c4./c5./c6 Binding edges.**

From the three edges evaluated, one, namely `force-II->Socrates`, is found to match the rule edge `?Xds-II->?Yds`. `?Xds-II->?Yds` is thus bound to it. This is the last edge of the rule under examination to be bound. That is, the rule can be applied. The bounding information is introduced into the lock map (an exclusive lock for the left-hand side edge and a rule instance reference for the context edge).

The other nodes of the input structure are examined along these lines and another fragment to which the rule in question can be applied is identified: `Socrates <-I- drink -II-> cup_of_hemlock`.

### 4.3 Some Complexity Considerations

To estimate the complexity of the binding algorithm, we count the binding attempts for both nodes and edges.

For nodal rules, the cost is $|G| \times |R|$, where $|G|$ stands for the number of the nodes in the graph $|G|$ and $|R|$ is the number of rules. The number of rules can be considered as constant. That is, we get the complexity of $O(n)$ (with $n = |G|$).

For one-edge rules, we get in the worst case a cost of $|G| \times (|G| - 1) \times |R|$. Since, again, the number of rules can be considered as being constant, we arrive at $O(n^2)$. Given that the number of types of outgoing and incoming edges is very restricted. Thus, at the deep-syntactic stratum there are only nine[6] (I-VI, ATTR, COORD, and APPEND), and most of the rules contain nodal conditions, which are evaluated first, in practice, the complexity

[6] We did not introduce all of these relations because they were not important for the understanding of the approach.

is near $O(n)$. However, obviously, the complexity rises with the number of edges in the rules. Especially in cases where more than five edges of the same type appear in rules we run into a combinatorial explosion. The nature of the binding problem, remains, after all, NP complete. But such rules appear—if at all—very seldom; the overwhelming majority of the rules contains no more than four edges, rather less.

The complexity of clustering depends on the number of alternative rules for the same chunk of the input structure. In the (hypothetical) worst case, where all rules in the grammar are alternative, it is thus again NP complete. However, this case never occurs: the number of alternative rules is strictly constrained.

In applications, the run time of the algorithm is acceptable: *AutoText-UIS* (Bohnet et al., 2001b)–a text generator which uses a $\beta$-release of our formalism implementation generates 60 air pollution reports with five complex sentences each, in about three minutes on a Pentium III PC with 800 MHz.

## 5 Related Work

Tree rewriting which is the more constrained version of graph rewriting has been central in transfer-oriented MT for a long time. In the last few years, there has been increasing interest in MT in graph rewriting (Emele and Dorna, 1998; Frank, 1999; Dymetman and Tendeau, 2000).

Tree rewriting is also used in generation—for instance, by the MTT-based generator *RealPro* (Lavoie and Rambow, 1997; Lavoie et al., 2000). Other well-known MTT-based generators such as GOSSIP (Iordanskaja et al., 1988) use a sequential graph rewriting formalism.

Apart from MTT-oriented approaches, there are several other approaches in generation that are related to our work. In what follows, we would like to mention two of them. The first is (Nicolov et al., 1996)'s work. The difference between Nicolov *et al.*'s approach and ours is threefold. First, Nicolov *et al.* use a graph rewriting grammar formalism, while we use a parallel graph rewriting formalism. Second, we strictly separate between grammatical processing and tasks of sentence planning. Our grammar rules thus do not contain any but linguistic conditions, while Nicolov *et al.*'s rules may also contain pragmatic and situational conditions. And third, finally, Nicolov *et al.* use complex rules which cover whole fragments of the input structure. As a result, it may well occur that with the rules chosen not all of the input structure is rendered into wording. This makes it necessary to evaluate the rules at disposal with respect to their potential to (i) cover best the remaining parts of the input structure and (ii) to be compatible with the rules already applied. In our approach, with most of the rules covering only one edge or node (or even a feature of a node), the probability of this problem is reduced to nearly zero. Furthermore, the lock map provides a full picture of how the different rules cover the input structure. This allows for an optimal mapping of the input structure onto the output structure.

Our approach also resembles Beale's constraint-satisfaction based *Hunter and Gatherer*-strategy (Beale, 1997; Beale et al., 1998) in that *Hunter and Gatherer* is a parallel graph rewriting formalism. However, Beale's approach is an integrated approach. It contains all information necessary for generation in the lexicon (including discourse information) in terms of structures that can be interpreted as complex mapping rules. Also, the binding strategy in *Hunter and Gatherer* is different: before the actual binding stage takes place, chunks of the input structure that possess a minimal number of relations to other chunks are recursively identified. The transformation of chunks with a minimal number of connections to other chunks reduces the number of conflicting cases during the stage of gluing together the resulting substructures.

## 6 Conclusions

In this paper, we presented a parallel graph rewriting formalism and illustrated how this formalism can be used to implement a gram-

mar for generation. Although the implementation is for MTT, the algorithm is *per se* theory independent.

An optimization of the rewriting procedure can be achieved by making use of a rule generalization hierarchy. Then, the statements that are located higher in the hierarchy are evaluated first, which means that whole classes of rules can be excluded from evaluation very early. See (Wanner and Bohnet, forthcoming) for details.

Unlike in many generators, we consider the grammar to be a resource (in the same vein as a knowledge base and a lexicon are resources) rather than a generation module. This resource is used by different sentence planning modules to render a semantic structure into a wording according to communicative and contextual criteria. However, it is beyond the scope of this paper to describe how the sentence planning mechanisms make use of the grammatical resource.

### Acknowledgements

Many thanks to the two anonymous reviewers for helpful comments and suggestions.

## References

S. Beale, S. Nirenburg, E. Viegas, and L. Wanner. 1998. De-Constraining Text Generation. In *Proceedings of the International Workshop on Natural Language Generation*, Niagara-on-the-Lake, ON, Canada.

S. Beale. 1997. *HUNTER-GATHERER: Applying Constraint Satisfaction, Branch-and-Bound and Solution Synthesis to Computational Semantics.* Ph.D. thesis, School of Computer Science, Carnegie Mellon University.

B. Bohnet, A. Langjahr, and L. Wanner. 2000. A Development Environment for an MTT-Based Sentence Generator. In *Proceedings of the First International Natural Language Generation Conference.*

B. Bohnet, A. Langjahr, and L. Wanner. 2001a. MATE*–Manual.* University Stuttgart.

B. Bohnet, L. Wanner, and *et al.* 2001b. Autotext-UIS – Automatische Erstellung von Ozonkurzberichten im Rahmen des Umweltinformationssystems Baden-Württemberg. In *Workshop Hypermedia im Umweltschutz.*

M. Dymetman and F. Tendeau. 2000. Context-Free Grammar Rewriting and the Transfer of Packed Linguistic Representations. In *COLING 2000.*

M. Emele and M. Dorna. 1998. Ambiguity Preserving Machine Translation Using Packed Representations. In *COLING 1998*, pages 365–371.

A. Frank. 1999. From Parallel Grammar Development towards Machine Translation – A Project Overview –. In *MT-Summit VII. MT in the Great Translation Era*, pages 134–142.

L. N. Iordanskaja, R. Kittredge, and A. Polguère. 1988. Implementing a Meaning-Text Model for Language Generation. In *COLING 1988.*

S. Kahane. forthcoming. Transductive Generative and Equative Grammars. In Polguère A. & Wanner L., editor, *Selected Topics in Dependency Grammar.* Benjamins, Amsterdam.

B. Lavoie and O. Rambow. 1997. A Fast and Portable Realizer for Text Generation Systems. In *Proceedings of the ANLP Conference.*

B. Lavoie, R. Kittredge, T. Korelsky, and O. Rambow. 2000. A Framework for MT and Multilingual NLG Systems Based on Uniform Lexico-Structural Processing. In *Proceedings of the ANLP/NAACL Conference.*

I.A. Mel'čuk and L. Wanner. forthcoming. Towards a Lexicographic Approach to Lexical Transfer in Machine Translation (Illustrated by the German-Russian Language Pair). *Machine Translation Journal.*

I.A. Mel'čuk. 1981. "Meaning-Text Models: A Recent Trend in Soviet Linguistics". *Annual Review of Anthropology*, 10:27–62.

I.A. Mel'čuk. 1988. *Dependency Syntax: Theory and Practice.* State University of New York Press, Albany.

N. Nicolov, C. Mellish, and G. Richie. 1996. Approximate Generation from Non-hierarchical Representations. In *Proceedings of the 8th International Workshop on Natural Language Generation*, Herstmonceux.

G. Rozenberg, editor. 1997. *Handbook of Graph Grammars and Computing by Graph Transformation.* World Scientific, Singapore, New Jersey, London, Hong Kong.

M. Stede. 1999. *Lexical Semantics and Knowledge Representation in Multilingual Text Generation.* Kluwer Academic Publishers Group.

L. Wanner and B. Bohnet. forthcoming. Inheritance in a MTT Grammar. In Polgu'ere A. & Wanner L., editor, *Selected Topics in Dependency Grammar.* Benjamins, Amsterdam.