Open Source Graph Transducer Interpreter and Grammar Development Environment

Bernd Bohnet[1], **Leo Wanner**[1,2]

[1]Department of Information and Communication Technologies, Pompeu Fabra University
[2]Institució Catalana de Recerca i Estudis Avançats (ICREA)
C/ Roc Boronat, 138
08018 Barcelona, Spain
{bernd.bohnet|leo.wanner}@upf.edu

Abstract

Graph and tree transducers have been applied in many NLP areas—among them, machine translation, summarization, parsing, and text generation. In particular, the successful use of tree rewriting transducers for the introduction of syntactic structures in statistical machine translation contributed to their popularity. However, the potential of such transducers is limited because they do not handle graphs and because they ”consume” the source structure in that they *rewrite* it instead of leaving it intact for intermediate consultations. In this paper, we describe an open source tree and graph transducer interpreter, which combines the advantages of graph transducers and two-tape *Finite State Transducers* and surpasses the limitations of state-of-the-art tree rewriting transducers. Along with the transducer, we present a graph grammar development environment that supports the compilation and maintenance of graph transducer grammatical and lexical resources. Such an environment is indispensable for any effort to create consistent large coverage NLP-resources by human experts.

1. Introduction

From an abstract formal viewpoint, nearly any NLP application can be interpreted as the transformation of a source representation into a target representation. The most general representation, we can think of is a graph. Graph transformation emerged in theoretical computer science as an extension of classical string rewriting known from Chomsky grammars (Chomsky, 1956). Over the last three decades, it has become a major research field of its own (Rozenberg, 1997). In NLP, graph transformation has been rarely used so far; most applications are restricted to tree transformation (or *tree rewriting*). However, the potential of tree rewriting is limited because it does not handle graphs (while, e.g., in text generation, the input structures tend to be (semantic or conceptual) graphs rather than trees), and because it ”consumes” the source structure in that it changes it step by step into the target structure (while it is often convenient to be able to access the source structure at any time during the transformation—which presupposes that it is kept intact).

Two-tape graph transducers (or, more precisely, *graph transducer interpreters*), which combine the advantages of graph transducers and two-tape finite state transducers, are in this sense more adequate in that they create the target representation without destroying the source representation. The disadvantage that remains is that, as graph transformation in general, two-tape graph transducers that are intended to provide a large coverage require extensive transformation rule sets. These rule sets are difficult to write and maintain manually without the support of an elaborate environment that provides editing, debugging and consistency control aids.

In what follows, we present a two-tape graph transducer approach and a corresponding graph transformation development environment. The next section contains a brief outline of the theoretical background of our transducer. In Section 3., we then present our graph transformation formalism and a set of examples. In Section 4, we sketch its implementation. Section 5. contains a short outline of the development environment that supports the development of resources for this formalism. Section 6. describes the application of the graph transformation formalism to text generation. Section 7. gives a brief overview of related work, before in Section 8. some conclusions are drawn and the outline of our future work in this area is given.

2. Theoretical Background

Graph and tree transducers use rules to describe the transformation of an input graph G to a target graph H, which might be the input to the next transformation step. A graph transformation rule $p : L \rightarrow R$ consists of a pair of graphs L and R. L denotes the left-hand side graph which defines the application conditions of p and R is the result of p. The transformation process usually consists of at least three steps (with $i = 1, \ldots, n$):

1. Search for rules p_i that are applicable to G;
2. Creation of the images of the right-hand sides of p_i (i.e., R_i) in H;
3. Embedding of the images of the right-hand sides R_i in H;

To embed the images of R_i, i.e., the created graph fragments, into the target graph H, two alternative techniques are known from Computer Science, both are to be specified within the rules. Thus, the rules can either (1) indicate via correspondence links nodes from the image of an R_i and nodes from the already partially built target graph H that should be glued together; or (2) grasp in H parts and connect these parts with the image of an R_i by a disjoint union. The two techniques are the main discriminatory feature of the two most common approaches to graph rewriting in Computer Science: the *algebraic* (or *double-pushout*,

DPO) approach (Ehrig et al., 1973) and the *node-label controlled* (NLC) approach (Janssens and Rozenberg, 1980). In the DPO-approach, a new graph is formed by gluing, while in the NLC-approach it is formed by connecting.
Gluing is most useful when the rules are applied simultaneously since the graph transducer can create the right-hand sides in parallel and connect the created subgraphs in the same stage. Connecting is useful when rules have to be applied in sequence. This procedure is comparable to Finite State Transducers (FSTs), which are applied left to right, and to tree transducers, which are applied top down. However, in the case of FSTs the sequence originates from the processing technique and is not required for many tree- and graph-based applications. As other modern graph transformation approaches, we provide both techniques to embed subgraphs, gluing and connecting.

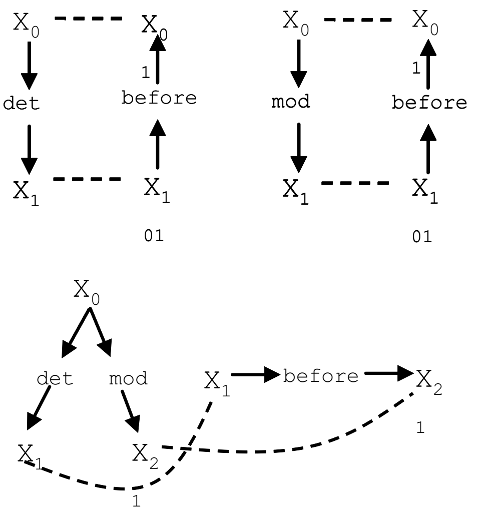


Figure 1: Three Tree Transducer Rules

Figure 1 shows three sample graph transducer rules. The nodes in the left-hand side of each rule are connected with their corresponding nodes in the right-hand side of the rule via *correspondence links* indicated in the figure by dashed lines. Figure 2 illustrates the application of these three rules. The input graph G is displayed on the left. The rule interpreter matches the left-hand sides (L_i) of the rules with G and creates for each of the matched rules an image of the corresponding right-hand side (R_i). The middle (framed) part of Figure 2 displays the result of the creation. Correspondence links connect nodes in the isomorphic images of L_i in G with nodes of the images of R_i All nodes that have a correspondence link to the same node in G are candidates for being glued together. In the example, the graph transducer glues together the nodes with same the label (*flower*, *nice*, and *the*).
On the left of Figure 2, the target graph H after gluing is shown. H is a topological graph that orders the nodes and thus the words of a sentence. The graph represents the correct order of the nodes: *the nice flower*.

3. Graph Formalism and Rule Language

As already mentioned above, our graph transformation formalism is based on a two-tape graph transducer. For the representation of graph structures, we use hierarchical graphs in which nodes can contain other nodes (Busatto, 2002). Hierarchical graphs are useful to superimpose additional information on graphs—for instance, the information structure, or to indicate that a set of words belongs to a distinct sentence, paragraph, constituent or topological field.
The syntax for representing graphs is simple:

Graph-Part	Definition
node:	node-name [:id] ['{' node-body '}']
node-body:	{ edge | attribute | node }*
edge:	edge-name-> node
hyper-edge:	edge-name-> { node$^+$ }
attribute:	attribute-name = value
graph:	node*
graph-def:	'structure' name type '{' graph '}'

Consider an example that shows a graph that represents a dependency tree and the order of the words (a graphical representation of the tree is shown in Figure 3); the attribute-value structures associated with each node are not displayed.

```
structure g1 dep {
  is {
    pos=VBZ
    SBJ-> quality {
      pos=NN
      NMOD-> The{ pos=DT }
      NMOD-> air{ pos=NN }
    }
    PRD-> good { pos=JJ }
  }
  The {b->air{b->quality {b->is {b->good}}}}
}
```

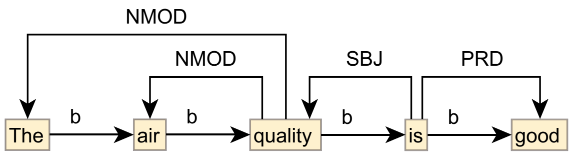


Figure 3: Graphical representation of a dependency tree

The left-hand L and the right-hand R sides of the rules are defined in terms of hierarchical graphs which can contain node, attribute, attribute value and edge label variables. The name of a variable starts with a question mark instead of a letter. For instance, L of the first rule below contains an edge variable '?r', which matches all edges of the input graph, except those that are labeled by 'b'.[1] Parts of both the left-hand side and right-hand side graphs can be marked as context. The correspondence links are defined using the '<=>' sign.
The rule syntax further foresees the following operators:

exists:	'exists' graph
not:	'!' graph
and:	graph '&' graph
scope:	'(' graph ')'
lexicon-access:	lex-name '::' { '(' var ')' '.' }* (var)

[1]This condition implies that the application of this rule results in an unordered dependency tree.

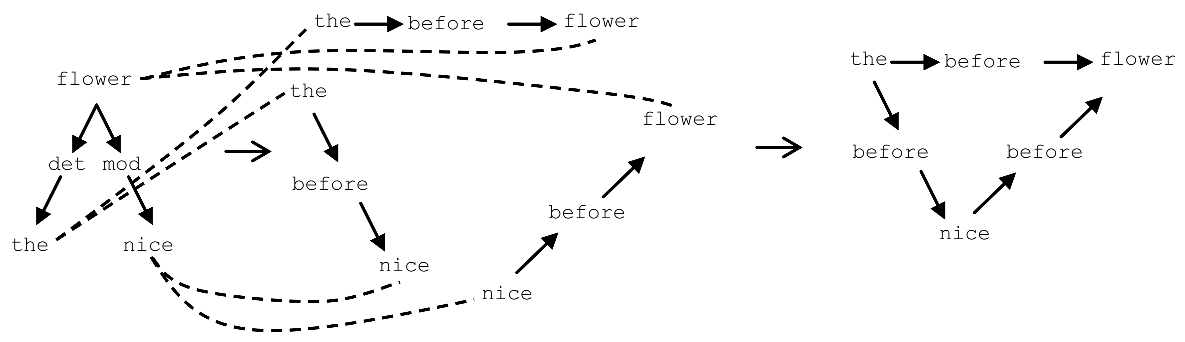


Figure 2: Graph Transducer Rules Using Correspondence Links

For illustration, consider the specification of two rules in our formalism. The first rule matches all nodes for which at least one outgoing edge that is not labeled by ‘b’ exists, such that the target node of the ‘b’ -edge has an outgoing edge labelled by ‘MATR’. The second rule maps a node to the target graph, if this node has an entry in the resource called ”semanticon”, and this entry has an attribute ‘type’ but no attribute ‘lex’. The node itself must not have an attribute ‘anaphora’ with the value ‘elision’ or ‘pronoun’.

```
Con<=>Sem example1
leftside= [
  ?Xl { exists (?r ->?Y {?l=?m})
        & ! ( b->?C { MATR->?Ml } )
        name=?name
  }
]
rightside= [
  ?Xr { <=>?Xl      // create nodes and correspondence links
     sem=?name // create the attribute sem with the value of ?name
  }
]

Sem<=>Synt lex_standard : rule
leftside = [
  ?Xs { sem=?s }
  & semanticon::(?s).(type)
  & ! semanticon::(?s).(lex)
  & ! ?Xs { anaphora=elision | anaphora=pronoun }
]
rightside = [
  ?Xds { <=>?Xs
     sem=?s
  }
]
```

The rules above use gluing to combine parts of the newly created graph. The following two rules illustrat the connecting approach. The first rule maps the first argument of a predicate to a edge labeled with subject. The second rule matches in the target graph an edge labeled with ’SBJ’ and adds an attribute ‘case = nominative’.

```
DSynt<=>Synt sbj : rule
leftside = [
  ?Xl { dlex=?s I->?Yl // match the first argument
     & semanticon::?s.gp.I.SBJ
  }
]
rightside = [
  ?Xr { <=>?Xl // create a node and a correspondence link
     sbj->?Yr{<=>?Yl} // create an edge label with SBJ
  }
]

DSynt<=>Synt sbj : rule
leftside = [
  ?Xl
]
rightside = [
  rc:?Xr { <=>?Xl // match a node with a correspondence link
     rc:sbj->rc:?Yr{ // match an edge labeled with SBJ
        case=nominative } // create an attribute
  }
]
```

4. Sketch of the Implementation

For the implementation of finite state transducers, usually finite state machines (FSMs) are used. We developed a similar technique for graph and tree transducers. In contrast to FSMs, we separate the left-hand side of the finite state machine from the right-hand side. Figure 4 shows a network which is built from the rules of Figure 1. As any rule, the network contains two sides—one for matching (the left-hand side) and one for creating of graphs (the right-hand side). The subnetworks of both sides consist of the same elements and can be applied bidirectionally.
In the top down direction, the network describes the steps that are to be taken to match the left-hand sides of a given rule set, while in the bottom up direction, the network defines the steps needed to generate the right-hand sides of the rule set. The numbers represent the states of the automaton. The edges are labelled by the actions that are to be taken in the next step. Depending on the application direction, the matching starts with state 1 in the network on the left or on right.
For illustration, let us assume that we have to match the rules in the source graph of Figure 2. In the first step, X_0 is matched. This causes the transition from state 1 to state 2. X_0 can match all nodes in the source graph because there are no further restrictions as yet. In the next step, the automaton goes either into state 3 or state 7. If the matching procedure finds an edge labeled by *det*, it continues with 3. From state 3, the automaton can go to state 4 and match a node with any label. State 4 is a finale state and thus

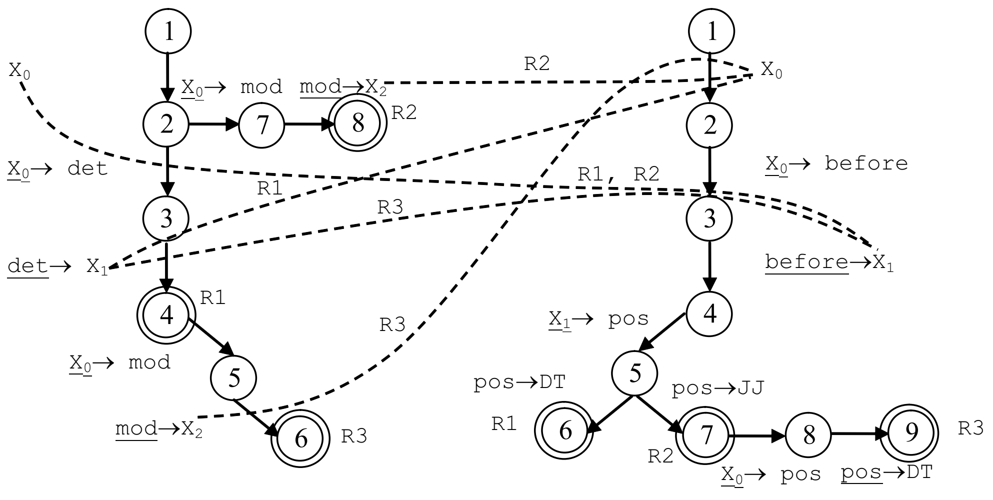
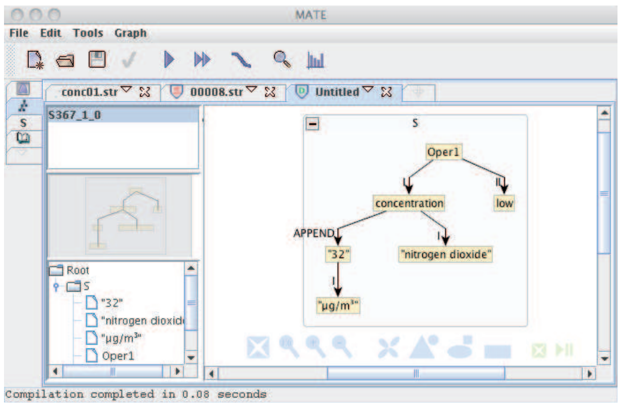


Figure 4: Rule Automaton

indicated by a double circle. It belongs to rule 1 (det_v) and means that rule 1 is applicable.
The creation of the right-hand side of rule 1 starts at state 7. In this case, the rule interpreter has to go up and perform the listed steps to build the right-hand side of the rule. The correspondence links are drawn as dashed lines and labelled by the name of the rule to which they belong.
This realization easily accommodates for the integration of the the Kleene Star (*), the 'plus' operator, the 'not' operator and the 'or' operator. As in FSTs, the operators are allowed only as context. The '*' operator and the 'plus' operator are just cycles; the 'not' operator leads away from an already found state; and the 'or' operator is realized by alternative paths in the network.

5. Development Environment

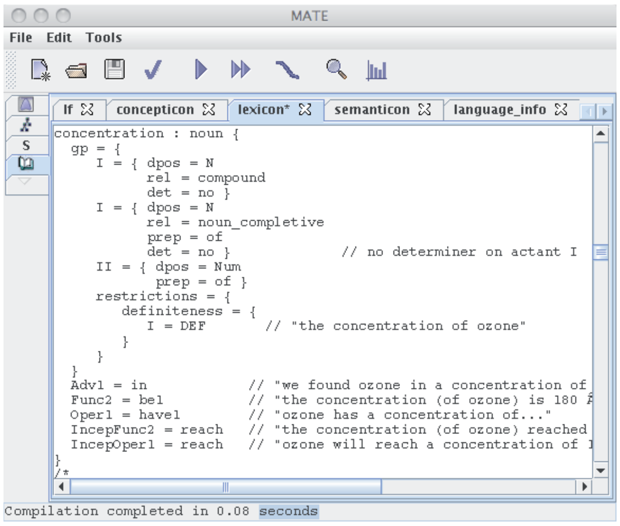
The manual development and maintenance of large grammars is feasible only if appropriate tools are available to support the developer. Therefore, we developed such an environment for the above formalism. The environment (called MATE: *Meaning-Text Development Environment*) contains editors for graph construction, rule and lexicon writing, a debugger, as well as a tool for regression tests.



Compilation completed in 0.08 seconds

Figure 5: Graph Editor

The graph editor (cf. Figure 5) allows for an interactive drawing of graphs and for a quick inspection of already

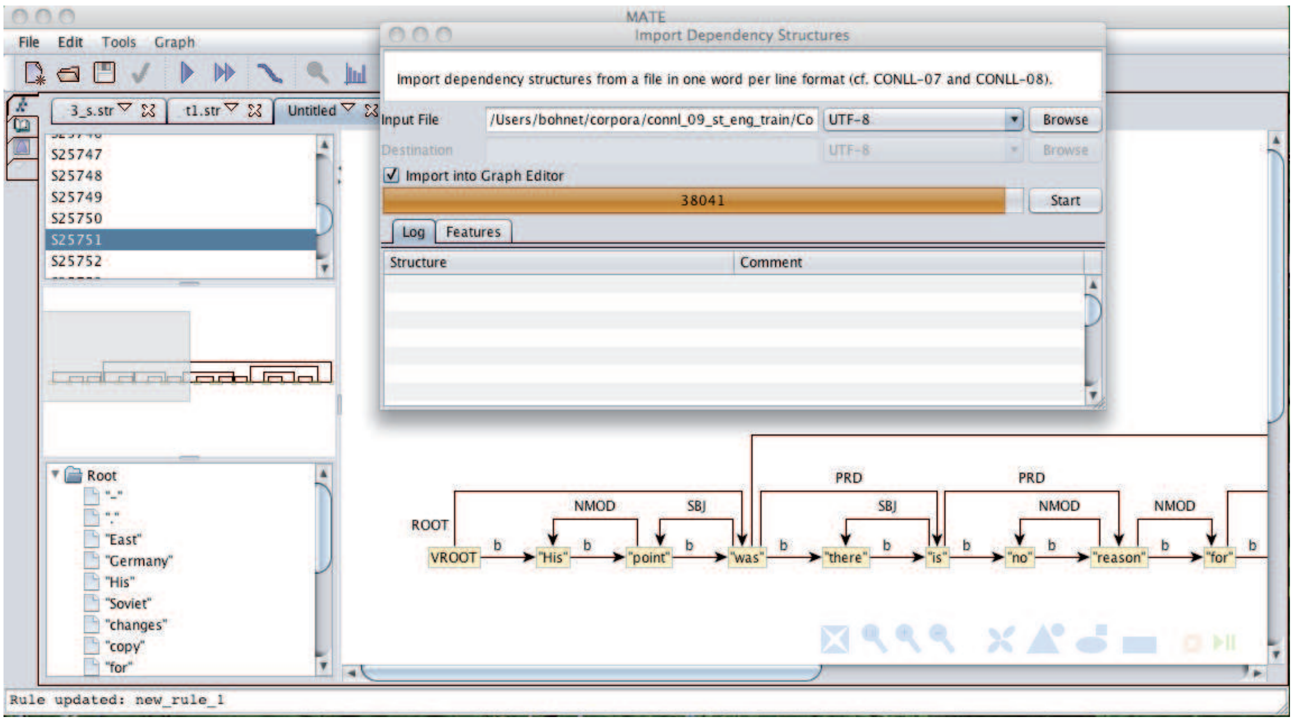


```
concentration : noun {
  gp = {
     I = { dpos = N
           rel = compound
           det = no }
     I = { dpos = N
           rel = noun_completive
           prep = of
           det = no }                // no determiner on actant I
     II = { dpos = Num
            prep = of }
     restrictions = {
        definiteness = {
           I = DEF       // "the concentration of ozone"
        }
     }
  }
  Adv1 = in             //  "we found ozone in a concentration of
  Func2 = bel           //  "the concentration (of ozone) is 180 Â
  Oper1 = have1         //  "ozone has a concentration of..."
  IncepFunc2 = reach    //  "the concentration (of ozone) reached
  IncepOper1 = reach    //  "ozone will reach a concentration of I
}
/*
```

Compilation completed in 0.08 seconds

Figure 6: Lexicon Editor

created graphs. More specifically, the graph editor provides drawing support and layout algorithms for semantic graphs, syntactic dependency trees, phrase structures, and topologic graphs. Graphs can be saved (as attributed labeled hierarchical graphs), loaded into new editor windows, and exported in several formats—among them the graph markup language (graphml) format, CoNLL (2009) format, scalable vector graphics (SVG) format, GIF and JPG. Furthermore, the graph editor provides facilities to import corpora in specific formats such as the CoNLL-format. Figure 7 shows a screenshot of the import tool.
The lexicon editor provides the functionality for entry search, control of entry duplication, syntax check, etc.
The most important functionality of the rule editor includes the syntax check, the possibility to group rules, and to interactively apply a subset or all rules to a selected graph.
The debugger gives answers to questions such as *Which rules have been applied and to which parts of the input graph*, *Which part of the result graph was created by which rule*, etc. Consider Figure 8 for a snapshot of the debugger window. The snapshot shows the state after the second round of the application of a rule. Therefore, the structure in the middle is incomplete. The window on the left shows


Import dependency structures from a file in one word per line format (cf. CONLL-07 and CONLL-08).

Input File: /Users/bohnet/corpora/connl_09_st_eng_train/Co — UTF-8

Import into Graph Editor: ✓

Rule updated: new_rule_1

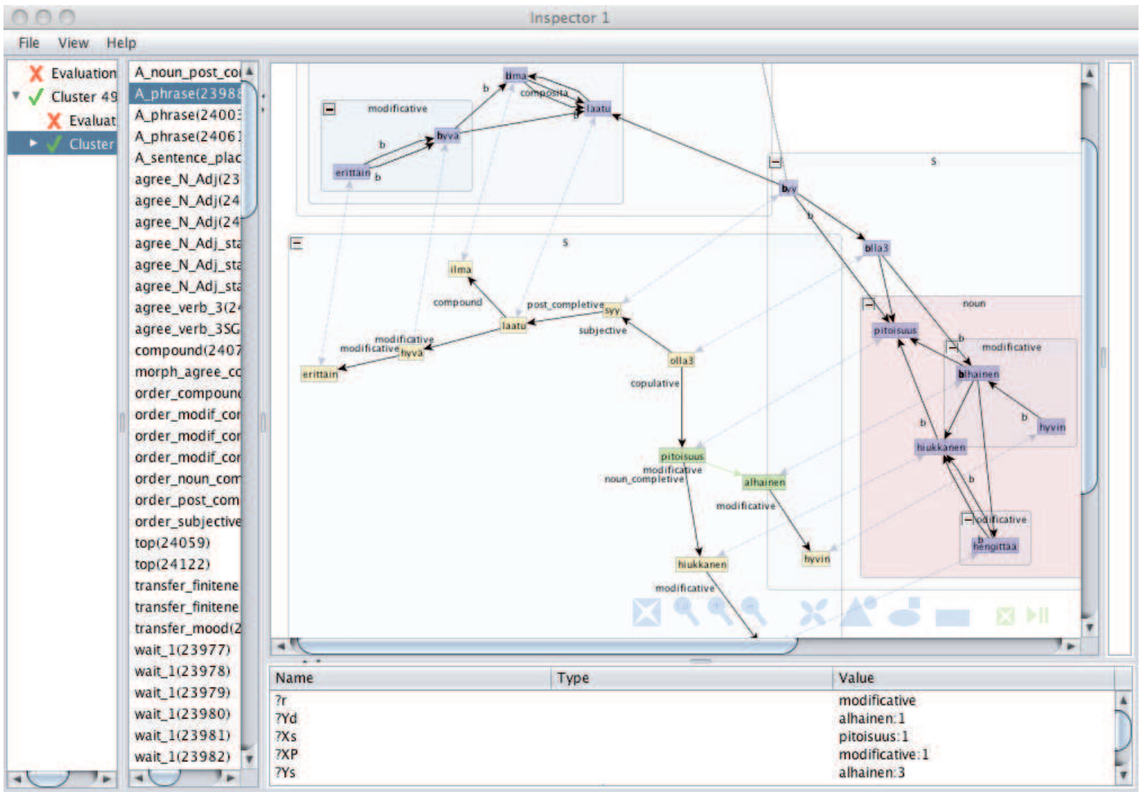
Figure 7: Import of Corpora in CoNLL-Format

the rule execution phase. The next window shows the set of rules that have been applied in the current execution phase in parallel. The window in the middle shows two graphs, the source graph and the target (or result) graph. The source graph is a dependency tree; the target graph is a topological graph that represents the linear order within a sentence. Different colors mark the source graph, the target graph and the parts to which the selected rules apply.
The regression test tool applies selected grammars to sets of predefined graphs and compares the result with reference graphs. If the regression tool finds any differences, then they are reported.

6. Application in Text Generation

The graph transducer formalism and the development environment presented above have already been used in several large scale projects for the development of text generation and summarization resources—among them, the European-scale projects MARQUIS (EDC-11258), PATExpert (FP6-ICT-028116), and PESCaDO (FP7-ICT-248594). The global objective of the MARQUIS was to develop an advanced European information service for generation of multilingual user tailored air quality information; cf. (Wanner et al., 2007b). One of the objectives of PATExpert was the multilingual summarization of patent claims (Wanner et al., 2007a). The just started PESCaDO targets the discovery and configuration of web-based environmental services and delivery of user-tailored multilingual environmental information.
For illustration of the use of the formalism, we focus on the generation process as implemented in MARQUIS and PESCaDO. Figure 10 shows an overview of this process.
The input to the text generator is a document plan which contains the content that has to be rendered into a text. Let us briefly discuss each of the major steps until linearization.

Conceptualization. In the first step, we map the document plan to a conceptual graph configuration that serves as input to the ”linguistic generator”. The nodes in a conceptual graph are concepts and the arcs between them conceptual relations in the sense of (Sowa, 2000).
Semanticisation. In the next step, the graph transducer maps the conceptual graph configurations to semantic graph configurations. A semantic graph is a hierarchical graph which consists of a predicate-argument structure on which the information structure (focus, background, giveness, theme/rheme, etc.) is superimposed.
Deep syntaxicisation. The deep syntactic representation (cf. Figure 5) is a tree which contains ”deep” lexical units connected by universal syntactic relations: the actantial relations (I, II, III, . . .), attributive relation (ATTR), appositive relation (APOS), and the coordination relation (COORD). The set of deep LUs of a language L contains all LUs of L—with some specific additions and exclusions. Added are two types of artificial LUs: (i) symbols of lexical functions (LFs), which are used to encode lexico-semantic derivation and lexical co-occurrence (Mel’cuk, 1996); (ii) fictitious lexemes, which represent idiosyncratic syntactic constructions of L. Excluded are: (i) structural words, (ii) substitute pronouns and values of LFs.
The graph transducer maps the semantic graph configuration to a deep-syntactic tree configuration using a relatively small set of about 120 rules. The rules use additional information from a lexicon represented as a graph. The key words of the lexicon entries are stored in a hash table and point to the nodes in the lexicon graph to provide a fast access to the entries. The lexicon contains information of the words and their combination. It defines the details of the mapping for building the syntax tree.
Surface syntaxicisation. The surface-syntactic structure is a tree which contains all words of a sentence. The edges



Evaluation
Cluster 49
Evaluat
Cluster

A_noun_post_co
A_phrase(2398
A_phrase(2400
A_phrase(2406
A_sentence_plac
agree_N_Adj(23
agree_N_Adj(24
agree_N_Adj(24
agree_N_Adj_sta
agree_N_Adj_sta
agree_N_Adj_sta
agree_verb_3(2
agree_verb_3SG
compound(2407
morph_agree_co
order_compound
order_modif_co
order_modif_co
order_modif_co
order_noun_com
order_post_com
order_subjective
top(24059)
top(24122)
transfer_finitene
transfer_finitene
transfer_mood(2
wait_1(23977)
wait_1(23978)
wait_1(23979)
wait_1(23980)
wait_1(23981)
wait_1(23982)

Name	Type	Value
?r		modificative
?Yd		alhainen:1
?Xs		pitoisuus:1
?XP		modificative:1
?Ys		alhainen:3

Figure 8: Debugger

are labelled with grammatical functions such as *subject*, *direct object*, *determiner*, etc. In this step, the grammar thus has to add structural words and values of LFs and label the edges with grammatical functions. Again, information for the mapping is retrieved from the lexicon.

Linearization. During the linearization step, the surface-syntactic tree is mapped onto a topological graph which defines the word order. To represent word order, we use hierarchical graphs that consist of word order domains and precedence relations. Each word order domain is a bag of words or domains that are grouped together in a sentence as a constituent. The precedence relation is realized as a directed edge between words and/or domains. The actual word order is derived by a topological-sort algorithm.

The result of the generation are mid-size texts. For instance, each of the generated bulletins in MARQUIS contains up to fifteen sentences, depending on the current air quality situation and the user profile. The bulletins are generated on demand. For most of the input content, the system is able to generate several alternative sentences, such that the texts look not the same and do not become boring. Consider a sample bulletin:

> *The air quality index is 4, which means that the air quality is poor. This is due to the high nitrogen dioxide concentration. The PM10 and ozone concentration do not have influence on the index. The nitrogen dioxide concentration (156 ug/m^3) is high. The high concentration is due to inversion. Therefore, an increase of reversible short term effects to human health (e.g. beginning irritation of the respiratory tract) is likely with sensitive people.*

7. Related Work

In NLP, so far also mainly tree rewriting approaches (called *tree transducers*) have been used. Cf., for instance, Knight and Al-Onaizan (1998), Alshawi et al. (2000), Kumar and Byrne (2003), Gildea (2003), Eisner (2003), and Echihabi and Marcu (2003) in machine translation, Wu (1997) in parsing and Lavoie and Rambow (1997), Bangalore and Rambow (2000), Bohnet and Wanner (2001), and Corston-Oliver et al. (2002) in text generation. Top down tree transducers have been independently introduced by Rounds (1970) and Thatcher (1970) as extensions of finite state transducers. Tree transducers traverse the input trees from the root to the leaves. Rules are applied in parallel to the branches such that they rewrite the tree in a top down manner. There are many extensions and types of tree transducers, among them R-transducers with finite look ahead (context) or regular-look ahead, Frontier-to-root transducers, which process a tree bottom up, etc. For a good overview of probabilistic tree transducers, see Knight and Graehl (2005). Levy and Andrew (2006) provide a combined engine for tree querying (Tregex) and manipulation (Tsurgeon) that can operate on arbitrary tree data structures.

Relevant to our work are also Finite State Transducers (FSTs). Aho and Ullman (1972) distinquishes between *generative schemas* and *transducers*: while a generative schema uses one tape, a transducer users two. However, this distinction got blurred in the course of the years. As a consequence, tree transducers are called "transducers" although they do not share the properties of FSTs that we describe in what follows. FSTs are standard tools in com-