FR 4.7 Allgemeine Linguistik
Universität des Saarlandes

Generation With TAG – A Semantics Interface and Syntactic Realizer

Diplomarbeit

Angefertigt unter der Leitung von
Dr. Tilman Becker
und
Prof. Dr. Wolfgang Wahlster

Tatjana Scheffler
`tasc@coli.uni-sb.de`

25. August 2003

Hiermit erkläre ich, dass ich die Diplomarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Saarbrücken, den 25. August 2003

Tatjana Scheffler.

Abstract

Tree Adjoining Grammars have long been claimed to be particularly useful for natural language generation, and they have been widely used in natural language processing systems for generation. However, the solutions provided so far are tailored specifically for their tasks or for one natural language processing system. This approach involves at least one of the following idiosyncrasies:

- A huge set of hand-written rules that capture the generation decisions. (e.g., in Verb*mobil*)
- System-specific and domain-specific input that was traversed and used in a hand-tailored fashion not easily transferrable to other generation tasks. (Verb*mobil*, SPUD)
- A TAG grammar specifically designed to fit the tasks at hand. (SPUD)

These facts make the solutions to specific generation problems basically non-transferrable to other systems.

This thesis proposes a more modular, reusable approach to natural language generation with FB-LTAG. I consider ongoing work on the syntax-semantics interface for TAG to define a compositional semantics interface for standard TAG grammars.

One major subtask of natural language generation, *syntactic realization*, deals with the construction of a syntactic structure that realizes a given semantic representation. I will make use of the semantics-syntax interface to provide a canonical algorithm for syntactic realization from flat semantics. Special care is taken in order not to pose arbitrary restrictions on the kind of semantic input that the algorithm can handle. In particular, decisions that are in fact *syntactic*, such as which parts of the semantics will end up as the head of the sentence, which one of a set of synonymous words is chosen, and as what part of speech a semantic literal will be realized, will not be expected to be already represented in the semantic input.

The approach presented in this thesis will therefore provide a syntactic realizer that is modular in several senses:

Independence from sentence planning The syntactic realizer module can be used in different systems that include natural language generation tasks.

Independence from algorithm Different realizing algorithms can be implemented using the explicitly defined TAG semantics-syntax interface. I show the implementation of one algorithm in the second part of this thesis.

Generation as constraint solving The representation of the generation task as a constraint problem makes it tractable for further improvements.

Deutsche Zusammenfassung

Baumadjunktionsgrammatiken (TAG) eignen sich besonders für die Generierung natürlicher Sprache. Daher wurden sie in der Vergangenheit mehrfach in Generatorsystemen eingesetzt. Die früheren Ansätze zur Generierung mit TAG waren aber spezifisch auf ihre Aufgabe eingestellt, oder sie nutzten die speziellen Eigenschaften des jeweiligen Gesamtsystems, in den der Generator eingebettet war. Solche Ansätze haben daher zumindest einen der folgenden Nachteile:

- Eine sehr große Regelmenge, die zum Treffen der Generierungsentscheidungen herangezogen wird. (z.B. in Verb*mobil*)
- Systemspezifische oder aufgabenspezifische Eingaben, die in einer maßgeschneiderten Weise verarbeitet werden; so daß dies nicht leicht auf andere Generierungsaufgaben übertragbar ist. (Verb*mobil*, SPUD)
- Eine TAG-Grammatik, die speziell auf die Aufgaben abgestimmt ist. (SPUD)

Durch diese Fakten sind die früheren Lösungen für das Generierungsproblem im Grunde nicht übertragbar auf neue Systeme.

Diese Diplomarbeit stellt einen neuen, modulareren und wiederverwendbaren Ansatz zur Generierung mit FB-LTAG vor. Ich betrachte zunächst aktuelle Forschung an der Semantik-Schnittstelle für TAG, um dann ein Semantik-Syntax-Interface für übliche Baumadjunktionsgrammatiken zu definieren.

Ein wichtiges Unterproblem der Generierung, die *syntaktische Realisierung*, behandelt die Erzeugung einer grammatischen Struktur, die eine gegebene semantische Struktur ausdrückt. Ich benutze die Semantik-Syntax-Schnittstelle, um einen Algorithmus für die syntaktische Realisierung von flacher Semantik anzugeben. Es wird besonders darauf geachtet, daß die semantischen Eingaben, die der Algorithmus bearbeiten kann, nicht zu stark eingeschränkt sind. Insbesondere wird von Entscheidungen, die von Natur aus *syntaktisch* sind, wie welcher semantische Teil der syntaktische Kopf wird, welches Wort aus einer Menge von Synonymen gewählt wird, oder als welche Wortart ein Semantikteil realisiert wird, nicht erwartet, daß ihre Lösung schon in der Eingabe angegeben ist.

Der in dieser Arbeit präsentierte Ansatz zeigt daher einen *modularen* syntaktischen Realisierer:

Unabhängigkeit von der Satzplanung Der Realisierer kann in verschiedenen Systemen, die Generierungsaufgaben enthalten, verwendet werden.

Unabhängigkeit vom Algorithmus Verschiedene Realisierungsalgorithmen können angegeben werden, die die explizit definierte Semantik-Syntax-Schnittstelle benutzen. Ich habe einen solchen Algorithmus implementiert und stelle ihn in Kapitel 6 vor.

Generierung als Constraint-Problem Die Repräsentierung der Generierungsaufgabe als Constraint-Problem macht weitere Verbesserungen leicht möglich.

Acknowledgements

I would like to thank Prof. Dr. Wolfgang Wahlster for providing a great research environment in Saarbrücken and for reviewing this thesis. The most gratitude I owe to Dr. Tilman Becker, who got me interested in Tree Adjoining Grammar and natural language generation, and who supervised this thesis. I am grateful for the many discussions we had during the almost three years I worked as his research assistant at DFKI. Christian Pietsch helped me greatly and improved the quality of this thesis by reading drafts of it.

Many thanks go to my friends in Saarbrücken who have motivated and helped me, especially during the last year.

Finally, all this would have been impossible without the constant encouragement and support by my parents, Petra and Uwe Scheffler. I want to thank them for making me enthusiastic about science.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

> Natural language generation is the process of deliberately constructing a natural language text in order to meet specified communicative goals. [McDonald, 1992]

1.1.1 Natural Language Generation

This thesis deals with natural language generation, i.e. the task of producing a grammatical sentence, given some specification of *what to say*. Generation is a central part of many applications using natural language. It is for example needed in machine translation systems, question-answering systems, spoken dialogue systems, in systems that produce automatic text summaries etc.

Depending on the specific application the generator is situated in, input specifications (also called *text plans*) can be quite different. They range from intentional goals (such as in a dialogue system, the intention for the dialogue partner to perform some act or come to some belief) to very specific content descriptions of the sentences that are to be realized. Output of generators can be just text strings, but some applications require more structured output. Structured output ranges from formatted text (e.g., HTML) to syntactic structures expressed in a grammar formalism, for example for the use in Concept-to-Speech synthesis (see section 1.1.4 for more on this topic).

Generators typically use descriptive *grammar formalisms* as a resource to determine the grammaticality of outputs. Thus, a specific generation module depends on the kind of input it gets, the output it is required to produce, and the grammar (formalism) it can use for this purpose. These factors make it highly specialized for the given task, drawing on domain knowledge as well as specific linguistic knowledge, as it is represented in the given grammar formalism.

It is worthwile to explore the possibility of using *modular* generators or at least developing modules within such generation systems that can be reused across applications, and across languages.

1.1.2 Grammar formalisms

Grammar formalisms are formal devices that allow the description of (at least) the syntax of a natural language. As such, a grammar formalism can be used to judge the grammaticality or ungrammaticality of a given sentence of the natural language. Well-known grammar formalisms used in computational linguistics are Head-Driven Phrase Structure Grammar (HPSG), Lexical-Functional Grammar (LFG), Combinatory Categorial Grammar (CCG), Dependency Grammars (DG), Systemic-Functional Grammars (SFG) and Tree Adjoining Grammar (TAG).

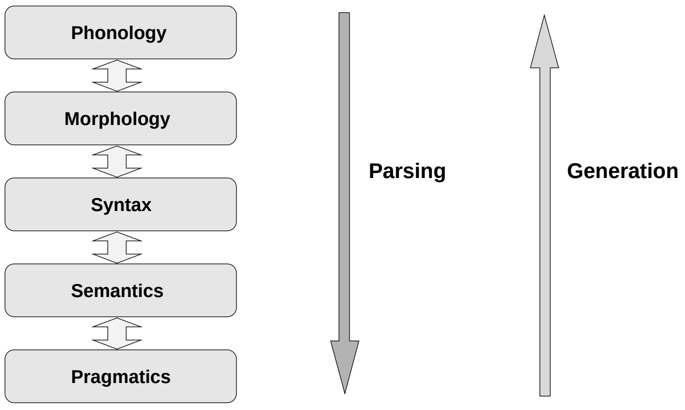


Figure 1.1: Grammar formalisms.

In theoretical linguistics, a natural language utterance has representations on several levels, reaching from phonology (the sound structure of the utterance) to semantics (the structure of meaning) and pragmatics (the structure of actions, intentions and inferences of language users associated with the utterance). The levels of linguistic representation are depicted in figure 1.1, along with two major language processing tasks performed on them.

A grammar formalism covers minimally the syntactic level of representation, but quite often, formalisms also integrate morphology and semantics, or at least well-defined interfaces to some special formalism dealing with morphology or semantics. The grammar formalism is therefore the natural connection between semantics and morphology. If the interfaces are descriptively defined within the grammar formalism, then the mapping of semantics to morphology happens entirely within the grammar. All that is left for the NLP system is traversing the search space and finding morphological representations that are linked to the given semantic representations via the grammar formalism.

However, the term *generation* describes a larger task than just the "transformation" of semantic descriptions into morphological descriptions: it includes also the *choice* of what is said. In this thesis, I will consider the subpart of natural language generation that is sometimes called *syntactic realization*; and I will understand by this the mapping of flat semantic content representations to syntactic structures. The following sections give a short motivation why this subproblem is especially interesting.

1.1.3 Previous Work on Generation with TAG

Tree Adjoining Grammars have long been claimed to be particularly useful for natural language generation, and they have been widely used in natural language processing (NLP) systems for generation. However, the solutions provided so far are tailored specifically for their tasks or for one natural language processing system. This approach involves at least one of the following idiosyncrasies:

- A huge set of hand-written rules that capture the generation decisions. (e.g., in Verb*mobil*)
- System-specific and domain-specific input that was traversed and used in a hand-tailored fashion not easily transferrable to other generation tasks. (Verb*mobil*, SPUD)
- A TAG grammar specifically designed to fit the tasks at hand. (SPUD)

These facts make the solutions to specific generation problems basically non-transferrable to other systems. Especially the last problem yields unnecessary extra work, as large-scale standard Feature-based, Lexicalized Tree Adjoining (FB-LTAG) Grammars have become available at least for English (XTAG, see XTAG Research Group, 2001), French (Abeillé and Candito, 2000), and German (a grammar developed in Verb*mobil*, that is based on the CDL-TAG variant).

1.1.4 The SmartKom project

This thesis was developed in the context of the SmartKom system, a multi-modal human-machine interaction system. SmartKom is a typical state-of-the-art NLP system with some interesting new features.

SmartKom has three basic scenarios: a *public* scenario corresponding to an information kiosk, a *home* scenario much like a computer terminal that is an interface to other electronic devices at home, and a *mobile* scenario where the system is integrated into a PDA handheld or into a car. For the respective scenarios, interaction with several different applications from different domains is possible, for example:

public: cinema programme information, reservations, document transfer

home: tv programme information, tv/vcr programming, user authentification

mobile: car/pedestrian navigation, sightseeing information

Setting for the generator In the SmartKom system, speech is only one output modality, others are graphical presentations and gestures by an animated agent on the screen. The generator input is determined by a modality fission module that specifies which information should be presented by which modality. The input is an abstract presentation goal encoded in an XML file conforming to the M3L (multi-modal markup language) standard developed in SmartKom. The beginning of an example input is shown in figure 1.2. It includes information on the type of speech act (`inform`), on the style of the output (`comment`

```
<speechGenerationTask goalKey="6">
  <speechPresentationGoal id="mf480">
    <inform id="mf481">
      <comment id="mf482">
        <informFocus id="mf483">
          <graphicalRealizationType id="mf484">
            list
          </graphicalRealizationType>
          <deepFocus id="mf485" idReference="mf478"/>
          <content id="mf486" idReference="mf479"/>
        </informFocus>
      </comment>
    </inform>
    <abstractPresentationContent id="mf487">
...
```

"Hier sehen Sie eine Übersicht über das Programm der Heidelberger Kinos."
(Here is an overview of the movie theater programme in Heidelberg.)

Figure 1.2: SmartKom generator abstract presentation goal.

instructs the generator to comment on a graphical presentation rather than realize all given information linguistically), on focus location (`deepFocus`, referring to some domain object through its identifier), on what the user requested from the system (`taskSpecification`), and on what the actual result of the user request in the system was (in this case, the system retrieved a list of movie performances for the cinemas in Heidelberg).

The output of the SmartKom generator is used in a Concept-to-Speech synthesis module (see Schweitzer et al., 2002). Such synthesis uses structural information from the syntax and information structure of an utterance in order to determine for example prosody. Thus, the generator is required to produce complex syntactic representations, and not just strings. This motivates the use of developed grammar formalisms in the generator instead of a shallow generation module.

Modularization Going from the abstract input goals in just one step to complex syntactic structures has some obvious disadvantages. Most prominently, it makes the generator heavily dependent not only on the grammar formalism, but also on the domain, especially on the type of ontology objects.[1] Furthermore, the task is very complex, and basically requires a large rule base to be solved.

The idea underlying this thesis is, therefore, to modularize the architecture and to assume a first process that computes flat semantic representations of what should be realized from the abstract input goals. In contrast to the input

[1]Note that in SmartKom, an ontology defines what items can occur in the generator's input specifications.

data, these semantic specifications should not be domain dependent, but rather use general semantic predicates. In this thesis, I will not consider this first step in more detail, but assume that it exists and produces the correct outputs.

Instead, I will be interested in the second step during generation: finding appropriate syntactic structures (utterances) for a given flat semantic representation, given a grammar with a semantics interface. This step is called *syntactic realization*. It is no longer dependent on the specific domain or the kind of input representation of the generator as a whole.

The direct advantage of modularization is the potential reusability of the syntactic realizer. Once a system develops a first processing module that produces flat semantic representations of the intended utterances, the same syntactic realizer can be used independently of the domain.

Descriptive semantics interface Modularization can be taken a step further by specifying a descriptive semantics interface for the grammar formalism that is to be used in the realizer. This well-defined interface will make it possible to develop and compare several algorithms for syntactic realization that use the same grammar.

To achieve this goal, I will not only define a semantics-syntax interface for the chosen grammar formalism (TAG) in chapter 3, but also restate the realization problem as a constraint satisfaction task to make it accessible for a range of standard constraint solving algorithms. I have developed and implemented a relatively simple depth-first search strategy that solves the realization problem as a feasibility study (see chapters 5 and 6). Efficiency (in terms of run-time) could be increased in the realizer by adapting other constraint solvers for the task, but this is left for further work.

1.2 Goals of this thesis

This thesis proposes a more modular, reusable approach to natural language generation with FB-LTAG. I consider ongoing work on the syntax-semantics interface for TAG to define a compositional semantics interface for standard TAG grammars.

One major subtask of natural language generation, *syntactic realization*, deals with the construction of a syntactic structure that realizes a given semantic representation. I will make use of the semantics-syntax interface to provide a canonical algorithm for syntactic realization from flat semantics. Special care is taken in order not to pose arbitrary restrictions on the kind of semantic input that the algorithm can handle. In particular, decisions that are in fact *syntactic*, such as which parts of the semantics will end up as the head of the sentence, which one of a set of synonymous words is chosen, and as what part of speech a semantic literal will be realized, will not be expected to be already represented in the semantic input.

The approach presented in this thesis will therefore provide a syntactic realizer that is modular in several senses:

Independence from sentence planning The syntactic realizer module can be used in different systems that include natural language generation tasks. As long as the tactic generator (sentence planner) provides a flat semantics output, our module can be employed as the appropriate realizer. This makes it independent of transfers to new domains or tasks.

Independence from algorithm Different realizing algorithms can be implemented using the explicitly defined TAG semantics-syntax interface. Thus, generation performance can actually be improved (e.g. in terms of runtime) without the need to change anything in other parts of the generator. This makes also the direct comparison of different realization algorithms possible. I have implemented one such algorithm, and present it in the second part of this thesis.

Generation as constraint solving The representation of the generation task as a constraint problem makes it tractable for further improvements. One important point that my approach aims at is the usability of extra-semantic information in the generation decisions. I am confident that pragmatic constraints can be naturally integrated in the constraint problem which syntactic realization is conceived of as.

1.3 Structuring of the following chapters

The remainder of this thesis is organized as follows:

Chapter 2 provides an overview of the Tree Adjoining Grammar (TAG) formalism. It describes the concept of lexicalization and the usage of feature structures in TAG. Ongoing work in grammar development in the TAG framework is mentioned. Finally, the chapter considers the special properties that make TAG interesting for the use in natural language generation systems. TAG's *extended domain of locality* and its *factoring of recursion* will be noted as important advantages of the formalism.

Chapter 3 dedicates a large section to previous attempts on finding a suitable semantics-syntax interface for TAG. It goes on to define requirements for such an interface, finding that Minimal Recursion Semantics (MRS) is an adequate semantic representation. The last part of the chapter presents the semantics interface for TAG to be used in natural language generation that is one of the contributions of this thesis. It provides definitions of lexical entries of such semantics-enabled TAG and the composition operations.

Chapter 4 discusses related work on syntactic realization with TAG.

Chapter 5 contains the second main contribution of this thesis. After a motivation of the task, it shows how the syntactic realization task is formalized as a constraint satisfaction problem. The chapter also presents the generic depth-first search algorithm proposed for its solution.

Chapter 6 describes the implemenation of the TAG semantics interface and the syntactic realizer in Java. It additionally includes a short presentation of the TAG grammar used in the realizer. The appendix contains the specification schema for grammar files; as well as a condensed version of the Java API for the semantics interface and syntactic realizer that were implemented in the practical part of this thesis.

Finally, chapter 7 concludes with a discussion of what has been contributed by this thesis. It also discusses possible extensions which remain for further work.

Chapter 2

Generation with Tree Adjoining Grammars

It has been noted quite early in the development of Tree Adjoining Grammars (see (Joshi, 1987) for a discussion), that they are especially well-suited for sentence generation. In the meantime, several systems that use a natural language generation module have employed TAG as their grammar formalism (and I will discuss some of them in chapter 4). This thesis explores a new attempt at using TAG for generation (see chapters 5 and 6).

In this chapter, I will therefore shortly recapitulate the foundations of Tree Adjoining Grammars. Section 2.1 describes the version of TAGs used in this thesis. The following section 2.2 reconsiders the particular suitability of TAG for natural language generation.

2.1 Feature-based Lexicalized Tree Adjoining Grammar

An in-depth definition of Tree Adjoining Grammars (TAG) can be found in (Joshi and Schabes, 1997). An introduction to the application of the formalism to linguistic analyses is presented in (Kroch and Joshi, 1985). Here, I will only concentrate on those aspects of the TAG framework that are directly relevant for this thesis.

A TAG consists of a finite set of elementary structures, called the *elementary trees*, and two operations on elementary trees: substitution and adjunction.[1] Intuitively, substitution replaces a leaf node in a tree with another tree; while adjunction inserts a tree into another by replacing an internal node. The operations are depicted schematically in figure 2.1.

The tree *in which* the substitution or adjunction is carried out is called the *outer tree* of the operation, while the substituting or adjoining tree is called the *inner tree*.

Elementary trees that can be adjoined into other trees (such as β_1 in figure

[1]Technically, *substitution* is just a convenient extension, and only adjunction (or *adjoining*) is the central operation for this grammar formalism, after which it was named.

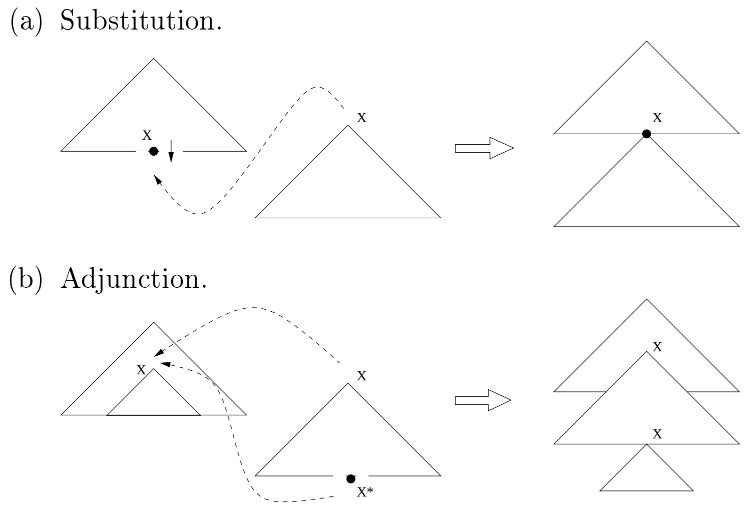


Figure 2.1: Schematic representation of TAG operations

2.2) are called *auxiliary trees*. They contain exactly one *foot node*, which is marked with a "*". All other elementary trees are called *initial trees*, and can only take part in substitution (as an inner tree). Nodes reserved for substitution are marked with a "↓".

A TAG analysis of a sentence, then, is usually not only represented by the sentence's *derived tree*, i.e. the phrase structure tree that results by composing all relevant elementary trees. In addition, the *derivation tree* is used to show *how* the elementary trees are composed. The derivation tree consists of one node for each used elementary tree. An edge connects each pair of elementary trees that take part in a substitution or adjunction operation. The edges are labelled with the address[2] in the outer tree, at which the operation is carried out.

Figure 2.2 shows some examples of lexical entries in a Tree Adjoining Grammar for German.[3] The derivation of the example sentence (2.1) can be seen in figure 2.3 below.

(2.1)	*Peter*	*liebt*	*Maria*	*immer.*
	Peter	loves	Mary	always.

Peter always loves Mary.

Figure 2.4 shows the derivation[4] and derived trees for the sentence, using the

[2]The Gorn address, which is in this thesis assumed to be the identifier of nodes in elementary trees (Gorn, 1967).

[3]Note that I do not assume a specific S category for sentences, as sentences are seen as the maximal projections of verbs.

[4]In the derivation tree, the nodes name elementary TAG trees. A full edge stands for an adjunction, a dashed edge for a substitution. The goal node of the operation is given as an edge label by its Gorn path.

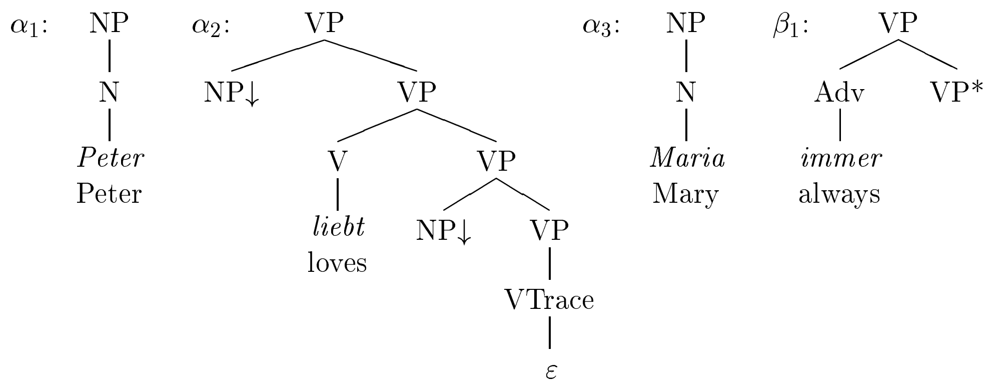


Figure 2.2: Example TAG trees.

lexicon from figure 2.2.

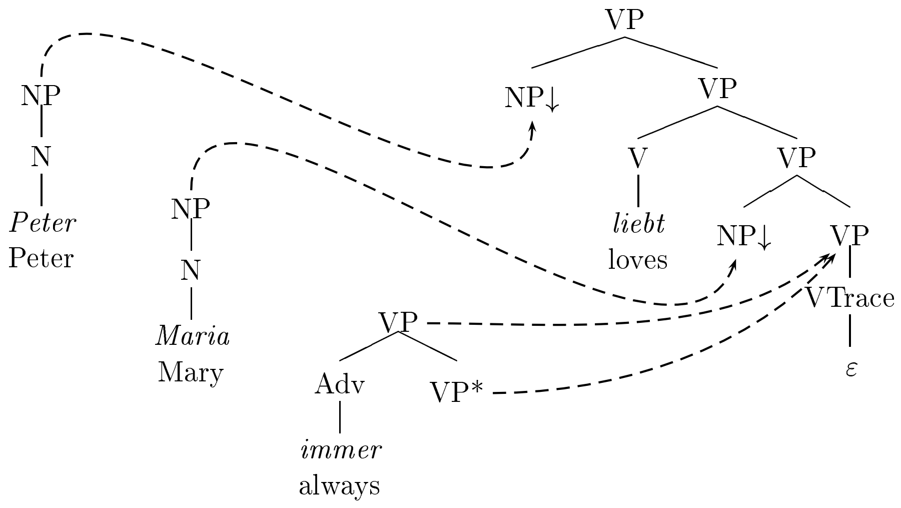


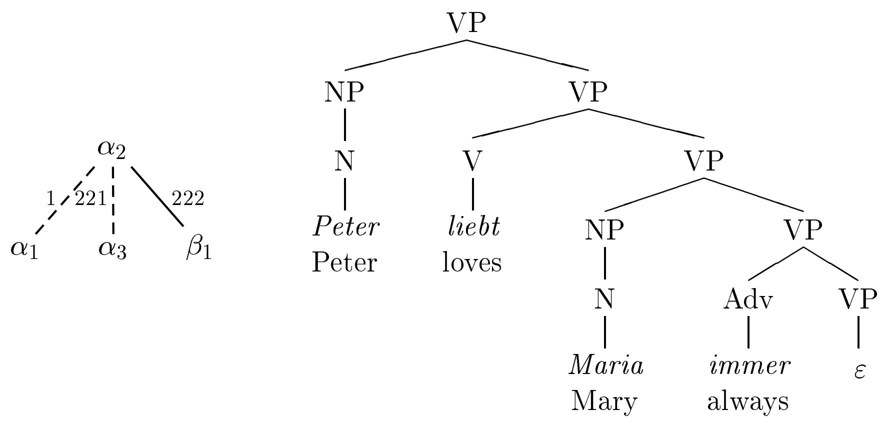
Figure 2.3: TAG derivation.

2.1.1 Lexicalization

Here, I presuppose a fully lexicalized TAG. In a fully lexicalized TAG, each elementary tree is anchored by at least one word. Therefore, elementary trees represent maximal projections of their anchor. Typically, all syntactic arguments of the lexical item are encapsulated in the elementary tree as substitution nodes. Adjunction is then used for syntactic modification.[5]

In fact, an elementary tree can be anchored by more than one word. This is especially obvious for the German verbs with separable prefixes (as in examples (2.2)-(2.3)).

[5]However, the footnotes of auxiliary trees may sometimes stand for semantic arguments of the anchor. See the right part of figure 3.6 for an example.



Derivation tree. Derived tree.

Figure 2.4: Analysis of sentence (2.1).

(2.2) …,	*daß*	*solche*	*Fehler*	*oft*	***auftreten***.
…,	that	such	errors	often	occur.

(2.3) *Solche*	*Fehler*	***treten***	*oft*	***auf***.
Such	errors	occur	often	PART.

Figure 2.5 shows an elementary TAG tree for the separate prefix verb "treten auf" (*occur*).

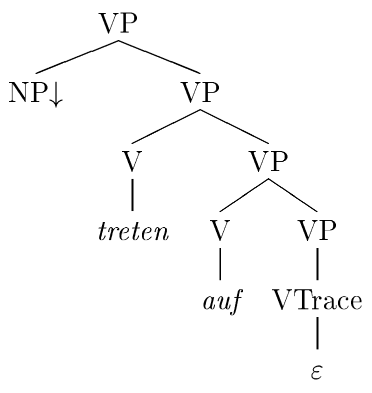


Figure 2.5: Multi-anchored elementary tree.

Multi-word expressions and idiomatic phrases are also candidates for the multiple anchoring of one and the same elementary tree.[6] The formalism does not pose restrictions on the size of the elementary trees. In fact, it is a special property of TAG that elementary trees can be *minimal* in the sense that initial

[6]The more so, as they can even be seen as one *lexical item*.

trees do not contain recursion; and all recursion is factored into minimal recursive units, the auxiliary trees. Frank (1992) defines a *Condition on Elementary Tree Minimality* (CETM). See his dissertation for more on this topic.

I will later argue that for a better integration of lexical semantics into the grammar, even other multi-anchored lexical entries (*templates*) might be needed (see chapter 3).

2.1.2 Adding features

Furthermore, following Vijayashanker (1987) and Vijay-Shanker and Joshi (1988), I suppose that each node is enriched with top and bottom feature structures carrying additional linguistic information. Note, though, that these feature structures are not recursive (as for example in HPSG) and only carry *local* information. However, conindexation between feature structures in an elementary tree uses TAG's extended domain of locality to express feature interdependencies. Such feature equations can become non-local during a derivation by the adjunction operation.

Typically, e.g. agreement is controlled by appropriate features. Figure 2.6 shows an example elementary tree with feature structures from the lexicon produced for this thesis.

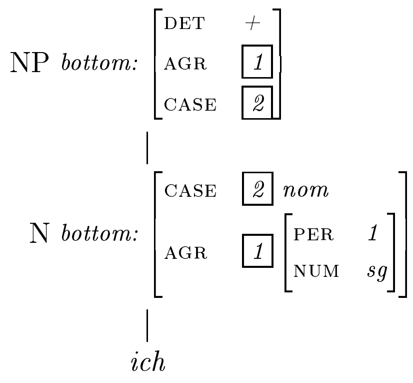


Figure 2.6: Example elementary tree with feature structures.

Upon substitution and adjunction, the feature structures of replacing and replaced nodes must be unified (a unification failure means that the operation can not be carried out). Figure 2.7 shows the necessary unifications for substitution and adjunction, respectively.

Finally, in the complete derived tree the top and bottom feature structure must be unified at each node.

Adjunction constraints Feature structures do also, in a large part, take over the tasks that adjunction constraints had in previous TAG versions. Nodes in the elementary tree can specify whether they demand or allow adjunction and which trees can adjoin into them. (E.g., adjunction into substitution nodes is commonly disallowed, while for German, verbal lexical entries like the one

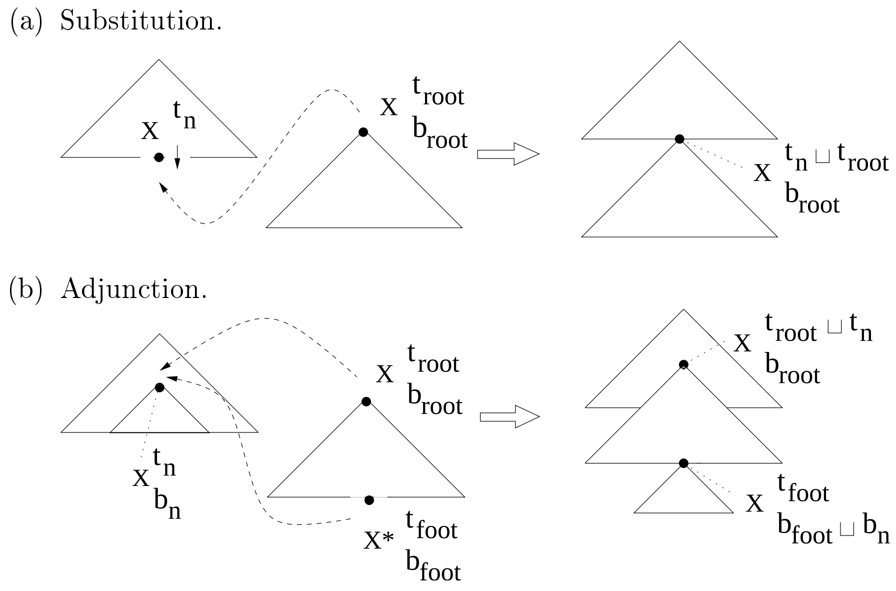


Figure 2.7: Feature structure unifications during TAG operations.

in figure 2.8 require the adjunction of a sentential modifier at the top-most VP node (VP_0) to yield a correct declarative sentence with verb-second word order.)

Specifying a list of possible trees for adjunction is quite complex, though, and does not allow for easy extendability of the grammar. Thus, features guide adjunctions in a more elegant way: In the lexical entry of figure 2.8, the top feature structure at VP_0 specifies a feature DECL(arative) as [DECL +], while the

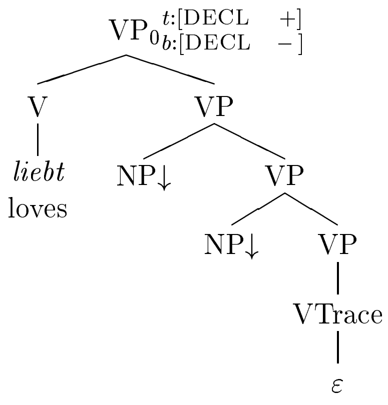


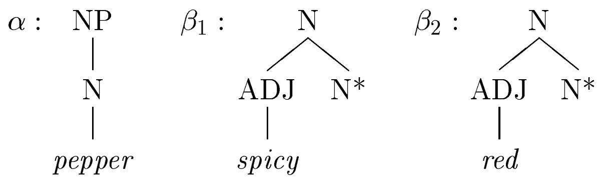
Figure 2.8: Lexical entry with *obligatory adjoining* node.

bottom feature structure at VP_0 includes [DECL −]. This inhibits unification of the top and bottom feature structures during finalization, unless a modifier had been adjoined that does not specify this feature (or whose specification unifies) — because in the case of adjunction the top and bottom feature structures in question would land in different nodes of the derived tree.

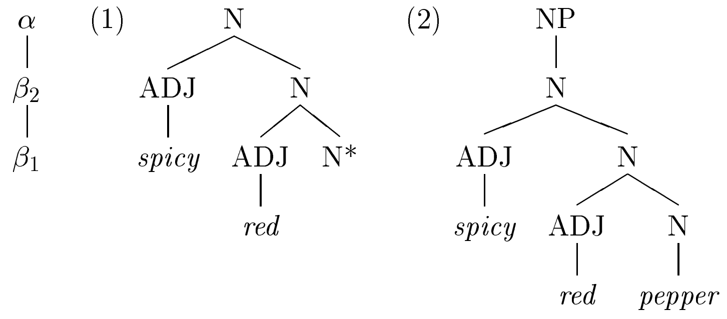
2.1.3 Multiple adjunctions

In the standard definition of derivation, multiple adjunctions into the same node are not allowed, as this would lead to ambiguities in the derivation tree. Technically, this restriction does not constrain the derivable trees. This can easily be seen from the example in figure 2.9: if two auxiliary trees β_1 and β_2 should be adjoined into the same node in elementary tree α, the same effect can be achieved if β_1 is adjoined into the root of β_2 first, and then the adjunction of the resulting tree into α is carried out.

TAG elementary trees.



Embedded adjoining.



Multiple adjunctions.

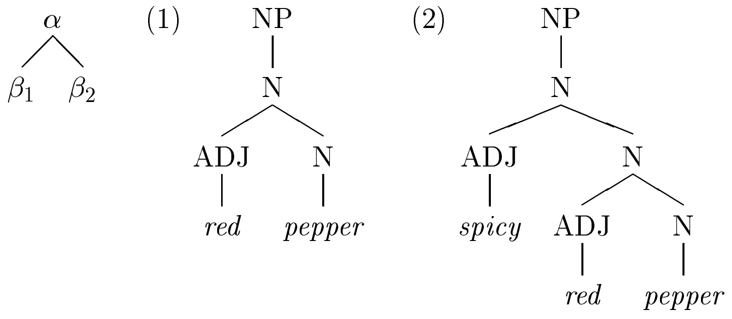


Figure 2.9: Multiple adjunctions vs. embedded adjunctions.

However, allowing multiple adjunctions into the same node makes more sense linguistically, as modifiers can usually be adjoined independently at the node in question, and should continue to do so if other modifiers are also adjoined. This also reflects the (semantically) monotonic addition of modifiers during generation much better.

Schabes and Shieber (1994) showed that, equivalently to the "standard definition of derivation" above, multiple adjunctions can be allowed, if on the other hand the derivation tree contains a partial *order*, such that for each pair of adjunctions into the same outer node, one of them is specified to be carried out before the other. We will adopt their analysis here, and think of multiple adjunctions on the same node as allowed and *ordered*.

2.1.4 Large-scale grammars in the TAG formalism

Building wide-coverage grammars has also been an issue in the TAG community. Large, centrally maintained grammars are particularly useful for the development of natural language processing systems, such as generators.

For TAG, the main project concerned with grammar development is the XTAG project at the University of Pennsylvania. XTAG started in 1987 with an English grammar that is still maintained and constantly extended. The XTAG grammar for English now has a quite large coverage including the most interesting phenomena in English, such as auxiliaries, control and raising, ergatives, inversion, particle movement, punctuation, etc. A detailed technical report shows one "frozen" status of the grammar: (XTAG Research Group, 2001).

Grammar organization The XTAG English grammar currently contains almost 900 trees. Trees are grouped into tree families. These tree families represent subcategorization frames, and contain all the trees realizing the same frame. A lexical entry in a TAG lexicon selects a number of tree families or trees.

TAG grammar maintenance and extension Maintaining such a large grammar is not an easy task, as the grammar developer needs to ensure that the addition of new analyses does not produce any new errors. The hand-crafted XTAG grammar is furthermore prone to inconsistencies, as it was developed distributedly over time, and by several grammar writers.

Therefore, tools have been introduced[7] that enable automatic changes in the grammar. One is the *metarules* approach: each tree family designates a *base* tree, from which all the other trees in the family can be automatically generated using metarules. This decreases the size of the grammar, and makes changes to existing tree structures easier, as they need to be introduced in much fewer places in the grammar (each affected tree had to be touched before).

Another approach is to define smaller *blocks* from which the elementary trees are composed.

These automatic tools are currently also used to re-construct the XTAG grammar or induce new grammars from a few base trees or blocks. See (Prolo, 2002) for a discussion. This helps detecting inconsistencies in the linguistic analyses and missing structures. It also helps the grammar developers to explicitly state and examine the linguistic assumptions behind the analyses.

[7]See (Doran et al., 2000) for more on this topic and on the evolution of the XTAG project in general.

Other grammars The XTAG project also constructed smaller grammars for Chinese, Hindi and Korean. Furthermore, a large-coverage French grammar has been developed at the University of Paris 7. This called is the FTAG grammar under maintenance of Anne Abeillé and others (see Abeillé and Candito, 2000).

A large TAG grammar for German (as well as a Japanese fragment) has been compiled from existing HPSG grammars for use in the Verb*mobil* speech translation project. Furthermore, quite large hand-crafted grammars for English and German were developed in the Context-Dependent Linearization TAG (CDL-TAG) (see Kilger and Finkler, 1995) variant.

A smaller German grammar is currently being hand-crafted (partly with the use of previously built larger templates) in the SmartKom project at DFKI Saarbrücken (see section 4.5).

2.2 Formal advantages of TAG for generation

Joshi (1987) and McDonald and Pustejovsky (1985) discuss the special properties of TAG that make it interesting for generation, and I will present some of them here as a motivation.

Extended domain of locality In contrast to CFG and similar formalisms, semantic predicate-argument relationships are *local* in TAG. This comes from the fact that TAG has a special kind of locality: the domains of locality are the elementary trees. In CFG, we cannot locally state that a transitive verb is a function which takes two NPs as arguments, because this fact comes from the interaciton of two rules, i.e. two local domains: S $\rightarrow$ NP VP and VP $\rightarrow$ V NP. In TAG, however, both NPs are locally realized in the elementary tree of the verb, which makes them directly available for such statements (see figure 2.8).

The domain of locality also makes the feature checking (in principle) local in TAG. For example, agreement features between the subject and the verb, which have to be percolated up and down the syntax tree in other formalisms, can be stated locally in the verb's elementary tree in TAG, as the subject is present there.

Factoring of Recursion Initial trees in a TAG grammar are non-recursive. All the recursion is neatly factored into the auxiliary trees, and introduced into syntactic analyses by the operation of adjunction.

For generation, this implies that complete sentences or phrases can first be built by filling all argument positions of an element. Then, recursion comes into play to successively extend the structure by adjunctions and substitutions.

Monotonicity The operation of adjunction preserves syntactic dependencies (which may become long-distance dependencies), as well as predicate-argument relationships. A strictly monotonic derivation of structures results, i.e. no transformations or deletions change a tree once it is selected during generation. This directly reflects the monotonic "adding" of semantic content to an utterance.

Incrementality The incremental construction of syntactic structures through adjunction corresponds to the incremental construction of semantic structures in a sentence planner. These two properties of TAG, the extended domain of locality and strict monotonicity of derivations, therefore induce the possibility of incremental generation, interleaved with the planning process.

Correspondence of syntax and semantics (McDonald and Pustejovsky, 1985) even put forward two hypotheses on the correspondence between realization specifications (i.e. sentence plans, or some conceptual representations) and TAG grammar:

1. Each semantic unit corresponds to (exactly) one of a set of possible elementary trees of a suitable TAG grammar. I.e., linguistic considerations should drive the modularization of planner decisions. This also means that a one-to-one mapping of conceptual (or semantic) units to syntactic units is possible.

2. For all possible combinations of two conceptual elements, there is a corresponding TAG operation that combines the appropriate TAG elementary trees. This (together with the first hypthesis) implies that all possible realization specifications can in fact be realized.

Although these hypotheses make in fact very strong claims about both the input specification for a syntactic realizer, as well as the TAG grammar that is suitable for such a task, the implications are quite appealing. Therefore, it is worthwile exploring these conjectures further by exploring the usability of TAGs in different natural language generation systems.

Chapter 3

A Semantics Interface for TAG

This chapter develops a semantics interface for Tree Adjoining Grammar to be used in a syntactic realizer.

A semantic interface for TAG then consists in defining the semantic representations that each of the lexical entries are assigned, the mapping of syntactic elements to semantic arguments (or vice versa) and the redefinition of the operations substitution and adjunction, to receive parallel semantics construction during the construction of syntactic structures.

For the semantic representations, generation poses special requirements:

- A **flat** semantic representation is highly desirable. Semantic embeddings do not always correspond to syntactic embeddings. E.g., the treatment of control predicates in TAG as auxiliary trees *turns* the direction of the syntactic embedding with respect to the semantic embedding.
- The semantic representation must be able to distinguish **scope**.
- Semantic composition should be **monotonic**. This follows directly from the fact that syntactic composition in TAG also consists of only *adding* elements to the structure.
- **Underspecifiability** of the semantics is desirable. Underspecification would allow generation from partial semantic representations.

First, I will discuss some previous attempts on developing a semantics interface for TAG in section 3.1. Section 3.2 then proceeds to present Minimal Recursion Semantics (MRS), a formalism for the representation of flat, underspecified semantics.

Sections 3.3 and 3.4 contain one contribution of this thesis: they show how MRS can be used as the semantic representation in a semantics interface appropriate for generation with TAG. Chapters 5 and 6 will then present a realization algorithm and its implementation based on this definition.

3.1 Previous Work

Finding an appropriate semantics interface for TAG has been in the center of research on Tree Adjoining Grammars for the last few years. In this section, some existing approaches will be presented.

As in TAG arguments are localized in the elementary trees that constitute the lexical entries, argument composition and also modification happens through the composition operations substitution and adjunction. These operations are directly represented in the derivation tree of a TAG analysis. Thus, the locus of semantics construction is commonly seen to be the derivation tree (as opposed to the derived phrase-structure tree).[1]

3.1.1 Kallmeyer & Joshi 1999

Kallmeyer and Joshi (1999) propose a flat semantic representation associated with elementary trees. They make use of Multi-Component Lexicalized Tree Adjoining Grammar (MC-TAG) in order to capture quantifier scope ambiguities and to allow underspecified representations. In their analysis, the semantic contribution of a quantifier is split into a predicate-argument part and a scopal part. The syntactic analysis of quantifiers they assume is one where the quantifiers first adjoin into the verbal tree and then take the NP-tree as a (substitution) argument — in contrast to standard TAG analyses of quantifiers as adjoining into the NP node (see XTAG Research Group, 2001; Abeillé and Candito, 2000, for English and French, resp.). They also give analyses of adjunct scope and island restrictions.

One central point of the approach is the use of multi-component lexical entries. However, the use of multi-component lexical entries in combination with the unrestricted possibility of multiple adjunctions at the same node yields a grammar that would be much more powerful than LTAG. As such an increase in the generative capacity of the grammar should be avoided, multiple adjunctions have to be restricted in Kallmeyer & Joshi's approach (they are still necessary for the multiple adjunctions of the scopal parts of quantifiers at the "S" node). This necessary restriction constitutes a possible flaw for grammars like the one assumed in this thesis where multiple adjunctions are considered otherwise linguistically motivated.

3.1.2 Joshi & Vijay-Shanker 1999

Joshi and Vijay-Shanker (1999) argue that using underspecified semantic representations contradicts the essential spirit of TAG: intermediate trees without open substitution nodes are seen not as *partial* trees, but as complete analyses that can be further extended by subsequent adjunctions. For example, the intermediate tree received by adjunction of the auxiliary tree for "said" into a tree for "likes" is shown with its associated semantics in figure 3.1. In the standard TAG interpretation, it should be understood that this tree represents the "act of saying that x likes y", rather than "an act of saying something that will somehow involve x *likes* y"[2] – which would be the underspecified interpretation, allowing for subsequent adjunction of, e.g., "seems" to yield $say(z, seem(like(x, y)))$.

[1]But see section 3.1.4 below and (Frank and van Genabith, 2001) for attempts to construct the semantics on the derived tree instead.

[2](Joshi and Vijay-Shanker, 1999, p. 137)

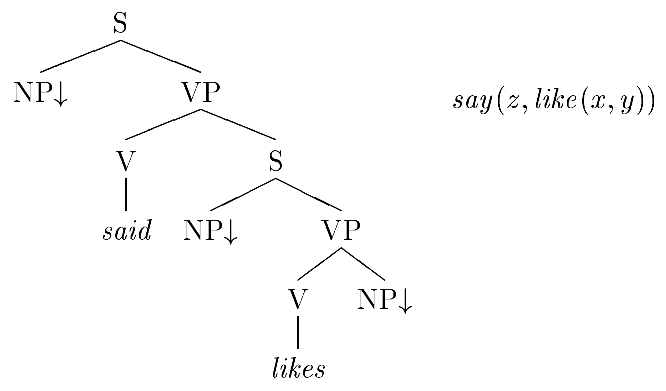


Figure 3.1: Intermediate tree and associated semantics.

Instead of underspecification, they suggest to order the adjunctions into the trunk of an elementary tree in order to monotonically build up the semantics "inside-out." As an ordering in the derivation tree is independently needed in the case of multiple adjunctions at the same node, this suggestion does not impose too strong changes on the formalism. In the semantics interface used in this thesis, I will therefore incorporate this proposal.

In a second part of the paper, Joshi and Vijay-Shanker hint at the possibility for using of MC-TAG for the modelling of scope ambiguities in TAG, without the need for underspecification.

3.1.3 Kallmeyer 2002

Kallmeyer (2002a,b) notes that although the derivation tree seems to be the right place for semantics construction in TAG, it sometimes does not provide the (semantic) links that are needed in order to identify semantic arguments. As noted above, the analysis of quantifier scope depends on the analysis of quantifiers as multi-component TAG trees that take the noun they determine as a (substitution) argument. However, in standard TAG grammars determiners are rather analysed as auxiliary trees adjoining into NP trees.

Thus, the derivation tree does not exhibit the semantic link between the quantifier and the verb, which is needed to determine that the verbal semantics lies in the body of the quantifier. An example derivation tree for the sentence "Every dog barks." can be seen in the left part of figure 3.2. Its derived tree is shown in figure 3.3.

Kallmeyer therefore suggests to enrich the derivation tree in a regular way by adding "semantic" edges between nodes that are also "semantically linked." The resulting *enriched derivation structure* (or e-derivation structure) contains all the edges of the conventional derivation tree, plus additional edges between a mother node γ and any lower nodes β that are adjoined into the root of children of γ or any such β_i. The e-derivation structure for the sentence "Every dog

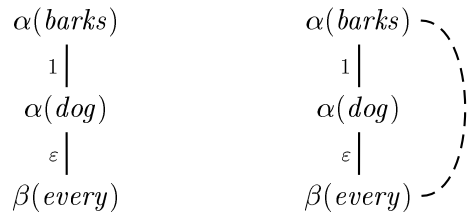


Figure 3.2: Conventional and e-derivation structure for "Every dog barks."

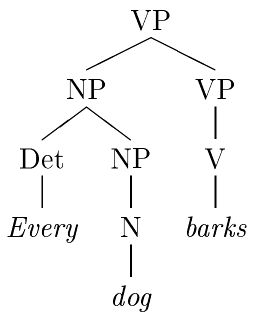


Figure 3.3: Derived tree for "Every dog barks."

barks." is shown in the right part of figure 3.2.[3]

The e-derivation structure is claimed to also capture multiple modifications and unbounded dependencies in embedded interrogatives correctly.

In this thesis, I will not adopt Kallmeyer's proposal to enrich the derivation structure, because I will not explicitly deal with quantifier scope. However, I feel that enriching the derivation structure should not be necessary, if the composition operations for the semantics construction are adequately defined. This can be seen intuitively, as the e-derivation structure makes links between elementary (or derived) trees explicit, that are already there: for example, unification of the feature structures of the involved nodes is carried out.

3.1.4 Gardent & Kallmeyer 2003

In contrast to the approaches mentioned above, (Gardent and Kallmeyer, 2003) abandon the idea of constructing the semantics based on the *derivation* tree, and claim to process semantics on the *derived* tree.

Approach. In their model, each node is associated with a semantic index and a semantic label variable in the top and/or bottom feature structures. When two TAG trees are combined, the flat semantic representations associated with them

[3]In that figure, I follow the notation in (Kallmeyer, 2002a), where substitution and adjunction edges are not distinguished in the derivation tree. Instead, full lines are edges that were taken over from the conventional derivation tree, while dashed lines signify edges that have been introduced in the e-derivation structure.

are conjoined, and an appropriately extended feature unification machinery ensures the unification of semantic variables. A simple example of a derivation in their formalism is given in figure 3.4.[4]

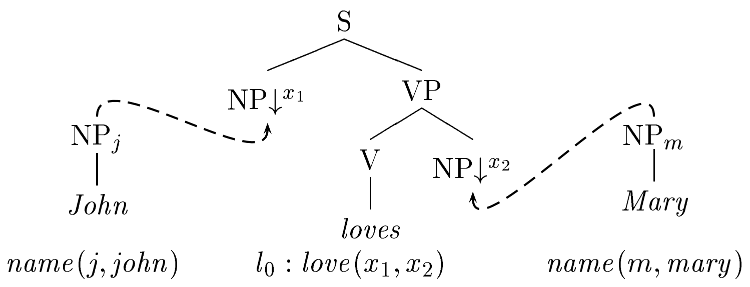


Figure 3.4: Derivation of "John loves Mary."[5]

Gardent and Kallmeyer sketch how, using their approach, they can correctly model the behavior of intersective adjectives, adverbials, control verbs, and wh-questions.

Derived tree-based analysis? The method of semantics construction is not very different from the derivation tree-based approaches, despite the claim that it is constructed on the derived tree. In fact, this is as much a derivation tree-based method as any of the others:

1. Each elementary tree is assigned some semantic structure, in a flat semantic representation.

2. The composition of semantic structures consists basically of monotonically conjoining the semantic structures of the two trees connected in the derivation tree.

3. Adjunction and substitution are appropriately extended in order to simultaneously build up the derived trees and manage the "unification" of semantic argument variables.

Implementation. The authors hint at two possibilities for an implementation of their formalism (for parsing): to also construct the semantics in the feature structures — here the problem is that the complexity will increase — or to keep the syntax and semantics construction completely separate and do the semantics construction outside of the derived tree.

In the first case, the semantic structure would be constructed literally *in the derived tree*, which would probably lead to an unwanted mixing of syntax and semantics: This is because really, the tree would only serve as a container for the semantics, Thus, nothing would be gained, while the explicit definition of the semantics construction would be obscured. Furthermore, TAG feature structures are currently non-recursive. A list-valued semantics feature would

[4]Only the semantic index variables associated with nodes are shown.
[5]After (Gardent and Kallmeyer, 2003)

necessarily be recursive, and thus the formalism would have to be extended. This would yield serious complexity problems.

The second case is more appealing, and I will follow a similar trail in this thesis.

3.1.5 GlueTag

For the sake of completeness, I will briefly mention another, quite different approach to a semantics interface for TAG: Frank and van Genabith (2001) explore the application of glue semantics (a semantic formalism commonly used in LFG[6]) to LTAG. They use relatively complex node labelling rules to associate nodes in the elementary trees with semantic variables.

This approach actually works on the derived tree: During substitutions and adjunctions, the semantic variables are just collected, and unifications at the end of a derivation trigger the correct variable instantiations that allow the glue semantics construction. The semantics construction itself can only be carried out on the complete derived tree (because only then the variables have been correctly instantiated). This has two consequences:

1. Incremental semantics construction is not possible, the semantic contribution of an utterance can only be computed on the complete derived tree.

2. Intermediate syntactic structures during the TAG derivation do not have a significant semantic structure associated with it.

Consequently, the approach does not seem appealing from the perspective of generation, as it would be desirable to incrementally check the semantic contribution of partial utterances.

In the GlueTag approach, scopal modifiers are analysed in an alternative way to taking the order of adjunctions into account: A single derived tree is constructed, for which the glue semantics model can construct two different goal semantic representations by applying functors in a different order. The ordering of adjunctions on the trunk of an elementary tree should have the same effect, though.

3.2 Minimal Recursion Semantics

In this thesis, I use a flat semantic representation very much in the spirit of Minimal Recursion Semantics (Copestake et al., 1999, 2001). MRS was chosen because it is especially well-suited to the requirements noted above: It specifies a *flat* semantic representation allowing for the distinction of different *scopings*; the semantic composition is *monotonic*; and MRS was designed specifically with underspecification in mind.

MRS allows flat semantics in two senses: First, the conventional binary operator $\wedge$ is understood as an n-ary, non-hierarchical logical “and.” Second, it

[6](see Dalrymple, 1999, 2001)

assumes semantic predicates in the neo-Davidsonian style where scopal embeddings are not expressed as actual embeddings of the semantic predicates, but rather by the introduction of event variables and scoping over these variables (see Barwise and Perry, 1981, 1983). Compare, for example, a predicate calculus representation of the semantics of sentence (3.1) in (3.2)(a) with the flat semantic representation in (3.2)(b).

(3.1) Every big white horse sleeps.

(3.2) (a) $every(x, ((big(x) \wedge white(x)) \wedge horse(x)), sleeps(x))$

(b) $h_0 : every(x, h_1, h_2) \wedge h_1 : big(x) \wedge h_1 : white(x) \wedge h_1 : horse(x) \wedge h_2 : sleeps(e, x)$

Furthermore, MRS enables the underspecified representation of scope ambiguities, by using handles and holes and scope constraints on these variables (see definition 2). In MRS, each lexical entry is assumed to be assigned a conjunction (or *bag*) of *elementary predications*, of which one is picked out as the *key elementary predication* of the lexical item.

3.2.1 MRS structure

Definition 1 (Elementary Predication) *An elementary predication (EP) in MRS*[7] *is a tuple* $\langle h, r, V \rangle$ *with:*

h *a handle labelling the EP*

r *a relation*

V *a list of zero or more ordinary variable arguments of the relation followed by an arbitrary number of handles (scopal arguments)*

Commonly, predications will only contain zero, one, or two scopal argument variables: one for usual scopal modification, and two for quantifiers, where the first corresponds to the restriction, and the second to the nuclear scope of the quantifier. An example elementary predication (corresponding to the lexicon entry for "every") is shown in (3.3).

(3.3) $h_0 : every(x, h_1, h_2)$

An MRS structure, then, is defined as follows:

Definition 2 (MRS structure) *An MRS structure is a triple* $\langle t, L, C \rangle$ *with*

t *a* top handle*, which is above or equal all other handles in the MRS structure,*

L *a bag of EPs,*

C *a set of outscoping constraints on the handles contained in the structure.*

[7]The original definition can be found in (Copestake et al., 1999, p. 7f)

The handle constraints are needed for underspecified semantic representations: When a handle is said to *outscope* a label, it can eventually (in a scope-resolved structure) be equal to that label, or (a chain of) quantifiers can introduce other handles in between. Thus, the underspecified MRS structure in (3.4) describes both possible scopings (of the general quantifier over the existential one, and vice versa).[8]

(3.4) $\langle h_0; \{h_1 : \mathit{every}(x, h_2, h_3) \wedge h_2 : \mathit{cat}(x) \wedge l_1 : \mathit{chase}(e, x, y) \wedge$
$h_4 : \mathit{some}(x, h_5, h_6) \wedge h_5 : \mathit{mouse}(y)\}; \{h_3 \geq l_1, h_6 \geq l_1\}\rangle$

3.2.2 Compositional semantics

The definitions of compositional semantics with MRS given in (Copestake et al., 1999) can be applied to TAGs almost directly with an appropriate semantics interface. I will briefly sketch their original definition here,[9] and show how to implement them as a TAG semantics-syntax interface in the following sections of this chapter.

The MRS definitions assume a Context-Free Grammar (CFG)-like syntactic composition: I.e., words or phrases are composed to bigger phrases. During such composition, the EP bags of the involved daughters are just appended, and all handle constraints are collected. According to the introduction of new handle constraints, three cases are distinguished:

Intersective combination. Both key elementary predications do not contain any scopal arguments. Then, the top handles of the daughters are equated and become the new top handle. For example, the combination of "white" with the MRS $\langle h_1; \{h_1 : \mathit{white}(x)\}; \{\}\rangle$ and "cat" with the MRS $\langle h_2; \{h_2 : \mathit{cat}(y)\}; \{\}\rangle$ yields:[10] $\langle h_1; \{h_1 : \mathit{white}(x) \wedge h_1 : \mathit{cat}(x)\}; \{\}\rangle$

Fixed scopal combination. This is the straightforward case of fixed scopal adjunction, where a scopal argument of one daughter has scope over the other daugther. I do not consider more complex cases here, because in the following sections, I will assume binary grammars and only fixed scopal predicates with one scopal argument.

In this case, one additional handle constraint is introduced: The handle-taking argument of the scopal daughter is said to outscope the label of the other EP. The label of the scopal EP becomes the top handle of the phrase. For example, the composition of "probably" with the MRS $\langle h_1; \{h_1 : \mathit{probably}(h_2)\}; \{\}\rangle$ with the MRS $\langle h_3; \{h_3 : \mathit{sleeps}(e, x)\}; \{\}\rangle$ of "sleeps" results in the structure $\langle h_1; \{h_1 : \mathit{probably}(h_2) \wedge h_3 : \mathit{sleeps}(e, x)\}; \{h_2 \geq h_3\}\rangle$.

Floating scopal combination. Floating scopal combination deals with quantifiers. The quantifier's restriction is considered the scopal argument (in

[8] For more details on the definitions, I refer the reader to (Copestake et al., 1999). Familiarity with MRS will be assumed in the remainder of this thesis, but an intuitive understanding will suffice for our purposes.

[9] modulo slight simplifications

[10] The details of the identification of semantic argument variables are omitted in the paper.

the sense of the fixed scopal combination), and the nuclear scope argument is always left unconstrained. Thus, the combination of "cat" and "every", with the MRS $\langle h_3; \{h_3 : \mathit{every}(x, h_4, h_5)\}; \{\}\rangle$, yields $\langle h_3; \{h_3 : \mathit{every}(x, h_4, h_5) \wedge h_2 : \mathit{cat}(x)\}; \{h_4 \geq h_2\}\rangle$.

3.3 A lexical semantics interface for TAG

In this section, I will define the (lexical) semantics interface for TAG, that will be assumed in the remainder of this thesis. Section 3.4 will then specify the re-definitions of the substitution and adjunction operations necessary to compositionally build up the semantics of more complex phrases.

3.3.1 Premises

In this thesis, semantics-enabled Lexicalized Tree Adjoining Grammar with feature structures is seen as a monotonic *extension* of conventional LTAG. That is, semantics construction should happen on the *derivation tree*, in parallel with the construction of syntax by the two TAG operations. Furthermore, lexical semantics need to specify all the information necessary for monotonic, compositional semantics construction.

On the other hand, the interface needed for the purpose of this thesis is intended for the use in a natural language generation system. Most importantly this means that the treatment of quantifier scope is not essentially needed, because in the output sentence quantification can be underspecified.

3.3.2 Lexical Entry

Definition 3 (Semantics-enabled TAG lexical entry) *A lexical entry t of the semantics-enabled LTAG is a triple $t = \langle \tau, \sigma, \phi \rangle$ with:*

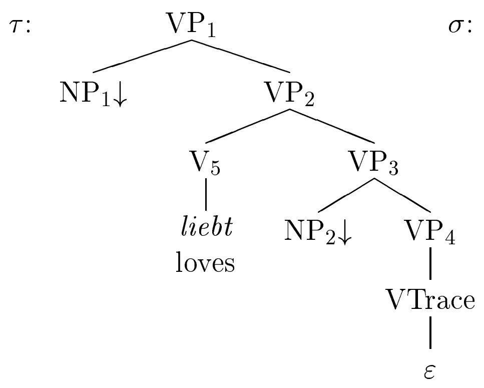
τ *an LTAG elementary tree,*

σ *a flat semantic expression, and*

ϕ *a function mapping the nodes of the tree to arguments in the semantic representation.*

An example lexical entry for the verb *liebt* (loves) is shown in figure 3.5. For abbreviation, the values of ϕ will in the following be shown directly in the tree as indices on the nodes. This should not obscure the fact that ϕ is a function separate from the syntactic structure associated with a lexical item; and that therefore the semantics is "added upon" (or wrapped around) the syntax in this interface. I assume that at least all substitution nodes, all nodes admitting adjunction as well as the root and foot nodes of elementary trees are assigned semantic indices by ϕ.

The semantic structure σ is modelled after the definitions in (Copestake et al., 1999), but it differs from MRS in two important respects. First, the predications consistently follow the neo-Davidsonian approach and introduce an event variable for all verbal predicates, and also for fixed scopal predicates.



σ:

$h_0 : loves(e, x, y)$

ϕ:

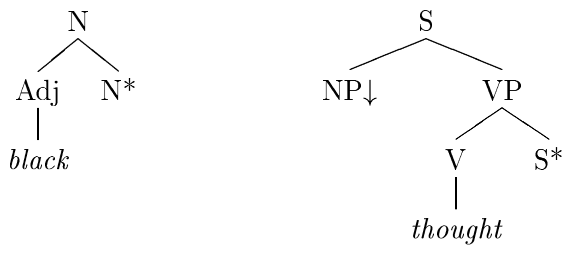
n	$\phi(n)$
VP_1	e
VP_2	e
VP_3	e
VP_4	e
V_5	e
NP_1	x
NP_2	y

Figure 3.5: A lexical entry for "liebt".

And second, I do not consider handle constraints in the current implementation. It will be shown in the following, how scoping of fixed scopal predicates can be treated without underspecification. Quantifier scope is omitted in the biggest part of this thesis. In section 3.4.3, though, I will discuss a possible extension of the semantics interface to deal with quantifier scope ambiguities.

3.4 Compositionality

The three different kinds of combination defined in MRS do not map isomorphically to TAG operations. In TAG, the substitution operation corresponds to the provision of syntactically and semantically obligatory arguments. (Copestake et al., 1999) does not explicitely define how predicate-argument structures are composed. Adjunction, on the other hand, must be differentiated: The typical case of adjunction is a (syntactically and semantically) optional modification. But in addition to these modifier auxiliary trees, there also exist predicative auxiliary trees, where the foot note corresponds to an *argument node* which is selected by the lexical anchor. Compare the elementary TAG trees for "black" and for "thought" in figure 3.6.



Modifier auxiliary tree. Predicative auxiliary tree.

Figure 3.6: Different types of auxiliary trees in TAG.

Furthermore, according to their semantic properties as specified in MRS composition, the modification cases must be distinguished into non-scopal (such as – arguably – intersective adjectives like “black”), fixed scopal (such as adverbs like “probably”) and floating scopal modifiers (the quantifiers).

In the remainder of this section, I will define the composition operations for semantic construction in LTAG assumed in this thesis. While substitution works rather straightforward, adjunction has to be differentiated into the above-mentioned cases.

3.4.1 Substitution

Substitution corresponds to the filling of an argument slot in the outer tree of the operation. The argument variable has to be provided by the inner tree. The substitution operation is defined on two TAG elementary or derived trees in which the argument variables occuring in the semantic expressions are distinct.

Definition 4 (Semantics-enabled TAG substitution) *If a tree $t_2 = \langle \tau_2, \sigma_2, \phi_2 \rangle$ with root node r (where τ_2 is the TAG syntax tree, σ_2 is the semantics it expresses and ϕ_2 is the mapping of the nodes in τ_2 onto arguments in σ_2) is substituted at node n of tree $t_1 = \langle \tau_1, \sigma_1, \phi_1 \rangle$, the result of the substitution is defined as $t_3 = \langle \tau_3, \sigma_3, \phi_3 \rangle$, with:*

τ_3 *the result of conventional TAG substitution of τ_2 into τ_1 at node n;*

σ_3 *the conjunction of (the EP bags of) σ_1 with a version of σ_2, where the argument variable $\phi_2(r) = x$ is substituted with $\phi_1(n) = y$ (the argument variable y that the substitution node n was mapped to by ϕ_1) in all places (the handles remain distinct); and*

ϕ_3 *the combination of ϕ_1 with ϕ_2, where all occurences of x have been substituted by y.*

As an example, the lexical entry for “Peter” is depicted in figure 3.7 along with the result of its substitution into the lexical entry for “liebt”, which was shown in figure 3.5.

3.4.2 Adjunction

Adjunction is also defined on trees whose semantic arguments have been substituted if necessary, so that the ranges of ϕ_1 and ϕ_2 are distinct.

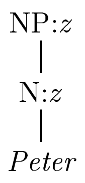
Definition 5 (Semantics-enabled TAG adjunction) *Let $t_1 = \langle \tau_1, \sigma_1, \phi_1 \rangle$, $t_2 = \langle \tau_2, \sigma_2, \phi_2 \rangle$. The root node of τ_2 be labelled r, the foot node f, and the adjunction site in tree τ_1 n.*
The result of adjoining t_2 into t_1 at n is a tree $t_3 = \langle \tau_3, \sigma_3, \phi_3 \rangle$ with:

τ_3 *the result of conventional adjunction of τ_2 into τ_1 at node n;*

σ_3 *the conjunction of σ_1 and σ_2's EP bags, where for non-scopal adjunction, all occurences of $\phi_2(r) = x$ have been substituted by $\phi_1(n) = y$ and the top labels of σ_1 and σ_2 are equated; while for scopal adjunction, all x (x is*

Lexical entry for “Peter” :

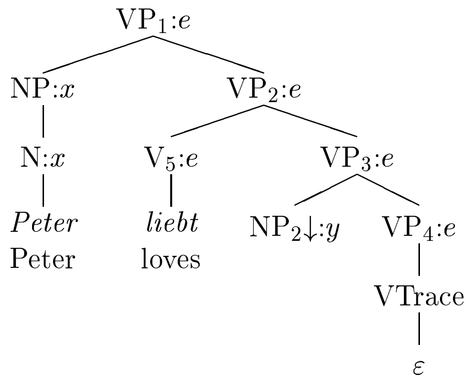
$\tau_p\,, \phi_p$:



$\sigma_p : \langle h_1, \{h_1 : \mathit{peter}(z)\}, \{\}\rangle$

Substitution result:

τ, ϕ :



$\sigma : \langle h_0, \{h_0 : \mathit{loves}(e,x,y) \wedge h_1 : \mathit{peter}(x)\}, \{\}\rangle$

Figure 3.7: Substitution of “Peter” into “liebt.”

a scopal handle variable) have been substituted by the current top handle h_{t_1} *of* σ_1*, and the new top handle of the phrase will be* σ_2*’s previous top handle; and*

ϕ_3 *the combination of* ϕ_1 *and* ϕ_2*, where* y *resp.* h_{t_1} *is substituted for* x *in all places.*

The second handle argument of quantifiers (the nuclear scope) always stays unconstrained.[11]

Note that “conventional adjunction” (or “conventional substitution”) also includes the unification of the appropriate feature structures, as it is defined for feature-based LTAG.

Figures 3.8 and 3.9 exemplify non-scopal and fixed scopal adjunction, respectively, by showing the TAG structures involved in adjunction of “schwarze” (*black*) into “Katze” (*cat*) and “immer” (*always*) into “liebt” (*loves*).

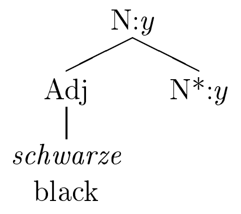
3.4.3 Underspecification and scope

In principle, the treatment of underspecification in MRS can be directly incorporated into the semantics interface given here. This would involve the use of hole variables and the introduction of handle constraints during adjunction, which are left empty in all the example lexical entries given above.

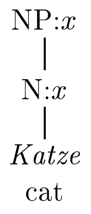
[11]as is also assumed in (Copestake et al., 1999)

Lexical entries for “schwarze” (*black*) and “Katze” (*cat*) :

$\tau_s\,,\phi_s$:



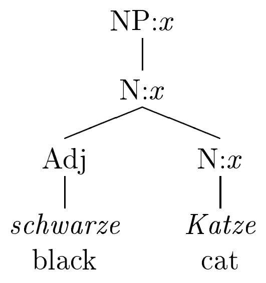
$\tau_k\,,\phi_k$:



$\sigma_s : \langle h_1, \{h_1 : \mathit{black}(y)\}, \{\}\rangle$ $\quad$ $\sigma_k : \langle h_0, \{h_0 : \mathit{cat}(x)\}, \{\}\rangle$

Adjunction result:

τ, ϕ :

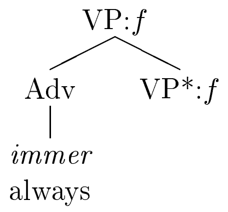


$$\sigma : \langle h_0, \{h_0 : \mathit{black}(x) \wedge h_0 : \mathit{cat}(x)\}, \{\}\rangle$$

Figure 3.8: Adjunction of “schwarze” into “Katze”.

Lexical entry for “immer” (*always*) :

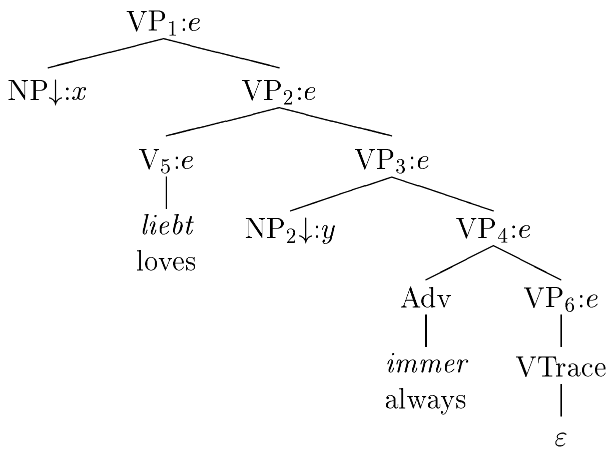
$\tau_i\,,\phi_i$:



$\sigma_i : \langle h_1, \{h_1 : \mathit{always}(f, h_2)\}, \{\}\rangle$

Adjunction result:

τ, ϕ :



$$\sigma : \langle h_1, \{h_0 : \mathit{loves}(e, x, y) \wedge h_1 : \mathit{always}(h_0)\}, \{\}\rangle$$

Figure 3.9: Adjunction of “immer” into “liebt.”

However, I do not assume generation from underspecified semantics here. Thus, the use of scope constraints is not necessary. As shown in (Joshi and Vijay-Shanker, 1999), different scopings can be achieved without the resort to underspecification, by an ordering of adjunctions. As I also assume monotonicity in the compositional semantics, I adopt their proposal here.

For quantifier scope, I follow the definitions in MRS, in that it is currently not resolved. In the same way as in (Copestake et al., 1999, p. 13), the nuclear scope of quantifiers always stays unrestricted in this semantics interface. See (3.5) for an example sentence and the corresponding semantics.

(3.5) a. Every man loves a woman.

b. $\langle h_0, \{h_3 : every(x, h_1, h_4) \wedge h_1 : man(x) \wedge h_0 : loves(e, x, y) \wedge$
$h_5 : some(y, h_2, h_6) \wedge h_2 : woman(y)\}, \{\}\rangle$

Thus, scope resolution could be left to a post-processing step computing possible domination patterns for the quantifiers. In that way, the representation chosen here can even be considered *underspecified* with respect to quantifier scope, but not concerning ordinary fixed scopal predicates.

Chapter 4

Related Work in TAG Generation

In this chapter I will review several previous attempts on generation from flat semantics with Tree Adjoining Grammars. I will point out certain restrictions that these approaches impose on either their input, the grammar they use, or certain formal aspects, for example of the semantic representation. In this way, these attempts are not easily transferrable to other domains or other systems, or simply unsatisfactory for the general task of surface realization from flat semantics.

As TAG is considered especially well-suited for natural language generation (see the discussion in section 2.2), there have been quite a few different attempts on generation with Tree Adjoining Grammars. Here, I will only discuss some of those that are more directly related to the topic of this thesis in more detail.

First, I will comment on an attempt by Koller and Striegnitz to represent TAG surface generation as a parsing task for free word order languages. Their approach, although interesting in the theoretical implications, has been left in an experimental state.

Much more evolved is the work of Matthew Stone, Christine Doran and others (Stone et al., 2001) in the natural language generation system SPUD (sentence planning using descriptions). Their work differs from the attempt presented here in the general setting. It is presented in section 4.2.

Following the lead of SPUD, the German InDiGen project (see Striegnitz, 2000) also deals with generation of referring expressions with TAG, but for German instead of English. I will briefly discuss their work in section 4.3.

Section 4.4 presents the TAG generator used in the Verb*mobil* project for machine translation of spontaneous speech. A short section 4.5 gives an overview of the approach to generation taken in the SmartKom project which is dominated by the need for *rapid prototyping*.

This chapter concludes with the mentioning of other approaches to TAG generation, that I will not consider in greater detail here. These are in particular systemic functional grammar approaches like Mumble (see McDonald and Pustejovsky, 1985; Meteer et al., 1987) and related work.

The generator algorithm and its implementation that is the major contribution of this thesis is presented in the subsequent chapters 5 and 6.

4.1 Koller & Striegnitz

In an ACL paper Koller and Striegnitz (2002) suggest an algorithm for surface realization from flat semantics for TAG. They conceive of the realization problem as a parsing problem in a free word oder language (in this case, the flat semantic representation). With Topological Dependency Grammar (TDG) (Debusmann, 2001; Duchier and Debusmann, 2001), quite fast parsers for the typical cases exist.

4.1.1 Description

Koller and Striegnitz show how the realization problem can be represented as a constraint satisfaction problem, the constraints being incoming and outgoing edges of nodes in the derivation tree.

For an encoding of the problem, they use Topological Dependency Grammar (TDG), a grammar formalism that separates linear precedence from dominance constraints. A TDG parse tree consists of nodes corresponding to the words of a sentence and edges labelled with syntactic relations between the words. Figure 4.1 shows a simple example. A TDG grammar specifies for each lexi-

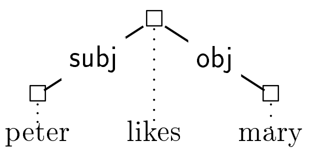


Figure 4.1: TDG parse tree for "Peter likes Mary."[1]

cal entry which outgoing edges it requires (corresponding to its *valency*), and which incoming edges it allows (its *labels*). For TDG, parsers employing this constraint-based formalization of the grammar have been shown to be quite fast on typical ("well-behaved") inputs. A toy grammar is given in table 4.1.

word	*labels*	*valency*
likes	$\emptyset$	{subj,obj,adv*}
Peter	{subj,obj}	$\emptyset$
Mary	{subj,obj}	$\emptyset$

Table 4.1: TDG toy grammar.[2]

Sentence generation from a bag of ground atoms of predicate logic is then formalized as a TDG parsing problem in the following way: TDG is used to "parse" the input semantics and produce a TAG derivation tree as the parse tree. This is possible because TDG can efficiently deal with completely free word order languages (as in this case predicate logic bags).

[1]see (Koller and Striegnitz, 2002, Figure 3)
[2]see (Koller and Striegnitz, 2002, p. 3)

Therefore, for each elementary TAG tree γ (represented by the semantic atom it is associated with), a TDG lexical item is introduced whose *valency* consists of exactly one outgoing substitution edge for each open substitution node in γ, and an arbitrary number of outgoing adjunction edges for each node in γ that admits adjunctions.[3] The *labels* for the lexical item admit one incoming adjunction edge with the semantic index associated with the root node of γ if γ is an auxiliary tree. If it is an initial tree, it allows an incoming substitution edge with the appropriate semantic index.

A special start symbol is also introduced into the grammar, which picks out the head of the derivation tree. Its lexical entry specifies only one outgoing substitution edge with the semantic index of the head of the sentence. This index thus needs to be given in the input semantics.

Koller and Striegnitz automatically compiled their grammar from the XTAG grammar for English. They indicate promising runtime results on some test inputs.

4.1.2 Discussion

However, Koller and Striegnitz impose serious restrictions on several parts of the problem, thus leaving their results unsatisfactory for the general realization task.

Simplified semantic representation. Their semantic representation is *flat* in a very strong sense, i.e. scope relations can not be expressed in it.

Specification of head necessary. In addition to the semantic representation to be realized, the generator requires the head index as input. Although some ambiguities in the realization process can be circumvented by this requirement, one would not in general want to restrict the input in this way. The following examples, e.g., would have the same semantic representation, and differ only in the fact which of the predicates is realized as the head of the sentence:

(4.1) *Hier*	*sehen*	*Sie*	*die*	*Filme,*	*die*	*heute*	*laufen.*
Here	see	you	the	movies,	that	today	run.

Here you see the movies that are running (on TV) today.

(4.2) *Heute*	*laufen*	*die*	*Filme,*	*die*	*Sie*	*hier*	*sehen.*
Today	run	the	movies,	that	you	here	see.

These movies are running today, which you see here.

Simplified grammar. While Koller and Striegnitz use the XTAG grammar for English, it is first simplified: features are removed and the only accepted adjunction constraints are "null adjoining" constraints (thus implying that wherever adjunction is possible, it is non-selectively and arbitrarily often possible). It is obvious that such a grammar without the help of features can not prevent ungrammatical output for more than trivial input specifications.[4] This

[3]Note that adjunction constraints, as well as features, are ignored in this approach.
[4]The authors do in fact give examples with quite complicated input semantics.

definitely does not cover the linguistic facts well enough.

Introducing features and feature checks into the proposed algorithm would not be a straightforward extension. At the least, it would make the performance much slower, eliminating the biggest advantage of the approach.

Syntax-oriented semantics interface. Finally, the semantics interface employed by Koller and Striegnitz is automatically extracted from the XTAG grammar, using word forms as names of semantic predicates. Obviously, this yields a very syntax-oriented semantics interface, that presupposes a semantic representation with very few ambiguities. For example, ambiguities that would occur on a much smaller scale are those relating to word category, i.e. when the same predicate can be expressed by (different) words from different part of speech categories.[5] Furthermore, the issue of synonymous words should definitely be one of syntax, not of semantics.

Thus, conceptionally, Koller and Striegnitz's approach is quite similar to the one presented in this thesis, as we both see the realization task as a constraint satisfaction problem. However, many design choices restrict the applicability of their approach. Additionally, the authors note that "the computation that takes place in our system is very different from that in a chart generator."[6] While the algorithm proposed in this thesis is not a chart generation algorithm, the computation that takes place in it is much more comparable to that of chart generators.

4.2 Sentence Planning Using Descriptions (SPUD)

The SPUD system ((Stone and Doran, 1997), or see (Stone et al., 2001) for a detailed project description) implements a microplanner realizing utterances from a communicative goal specification. In fact, the tasks of microplanning and surface realization are not clearly separable, and SPUD deals with both of them. In that approach, each lexical entry is associated not only with semantic, but also with pragmatic information, i.e. presuppositions and such information as definiteness or the text style. During generation, the pragmatics constrains lexical choice and guides realization decisions.

4.2.1 Description

The input to the SPUD generator is an intention such as "describe e," together with some semantic atoms that are also intended to be realized. Lexical entries that are used in a sentence contribute *assertions* (the direct semantic content), *presuppositions* (basically representing usage conditions for the words) and *pragmatics* (other constraints such as definiteness or text style).

[5]The only exception are words like "book" that occur in the same word form for different syntactic categories (it can either be "a book" or "to book a flight"). If these words also have the same semantic arity, they will be both considered by Koller & Striegnitz' algorithm.

[6](Koller and Striegnitz, 2002), p. 1

Lexical choice, which SPUD mostly deals with, then follows some heuristics that depend on the presuppositions and pragmatics associated with a word. Thus, for example, always the most specific possible word is chosen, preferring “slide” over “move” in contexts where this is appropriate. The system uses an inference procedure to subsequently chose syntactic structures that either themselves contribute to the realization goal or that provide the necessary presuppositions for other syntactic elements that were already chosen.

The SPUD system concentrates on the modelling of pragmatic contributions to realization choices. Its grammar is specifically designed for this purpose, simplifying other parts of the representation. E.g., adjunction is restricted in a special way, and determiners are factored into the lexical entries of nouns.

4.2.2 Discussion

Obviously, the task for the SPUD system is in some respects quite different from the one considered in this thesis. Its input is not a declaratively specified flat semantics, but a communicative goal (such as “describe e”), which must be elaborated by drawing on the system’s factual knowledge (such as what kind of event e is, and what participants it has). But the system also produces derived TAG trees as output, and captures therefore many of the same tasks as the algorithm I present in this thesis. In the following, I will concentrate on the factors of the SPUD system related to my work, and ignore other (admittedly very important) features that are not directly relevant.[7]

Non-scopal semantics. In SPUD, each lexical item is assigned some semantic specification (either assertions or presuppositions), using flat semantic representations. SPUD does not use handles, though, so the representation is flat in a strong sense similar to (Koller and Striegnitz, 2002) above. This makes it impossible to express scopal relations for example among different modifiers of a verb. Consider the following examples with their MRS-like semantic representations:

(4.3) Peter intentionally knocked twice.
$h_3 : \mathit{peter}(p) \wedge h_0 : \mathit{intentionally}(h_1) \wedge h_2 : \mathit{knocked}(e,p) \wedge h_1 : \mathit{twice}(h_2)$

(4.4) Peter twice knocked intentionally.
$h_3 : \mathit{peter}(p) \wedge h_0 : \mathit{intentionally}(h_2) \wedge h_2 : \mathit{knocked}(e,p) \wedge h_1 : \mathit{twice}(h_0)$ [8]

The semantic representation for these two sentences without handles would be the same, ignoring the difference in scoping: $\mathit{peter}(x) \wedge \mathit{intentionally}(e) \wedge \mathit{knocked}(e,x) \wedge \mathit{twice}(e)$.

Hand-tailored grammar. The SPUD system uses a specifically designed grammar, which is not equivalent to standard TAG grammars like XTAG (XTAG Research Group, 2001) or FTAG (Abeillé and Candito, 2000). The treatment of deteminers is non-standard, but only mildly so. Determiners do

[7] In particular, I will not discuss the inference mechanisms and parts of the system concerned with *planning* as such.

[8] And maybe, both examples are even amgiguous between the two readings.

not get separate lexical entries in SPUD. Instead, they are always already part of the nouns they determine. Thus, the authors avoid problems of definiteness, and also certain modification issues for which special features would be needed. Still, this is only a minor divergence from the standard XTAG grammar for English.

More importantly, the treatment of modification in general diverts from the standard TAG analyses in a more serious way. For each elementary tree, all possible modifiers that can adjoin into the tree have to be determined, and separate nodes are introduced that allow adjunction of only one kind of modifier. An example elementary tree and one appropriate modifier can be seen in figure 4.2. These lexical entries are necessary because of the heavy use of semantic

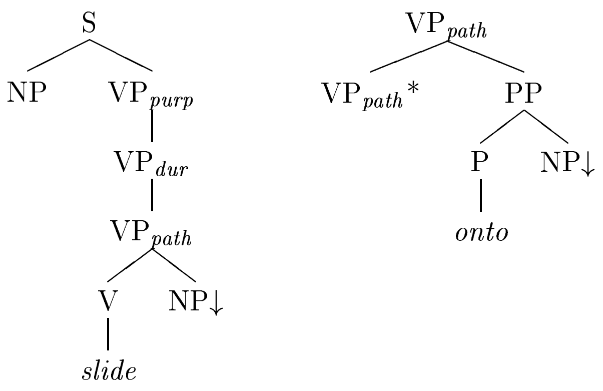


Figure 4.2: SPUD elementary trees.

indices and in order to achieve the particular kind of semantic and pragmatic updates that the SPUD system attempts. Thus it is impossible to use the existing large TAG grammars for English (XTAG) or French (FTAG) in the SPUD system, and new grammars have to be compiled instead.

Linguistically, this treatment of modification directly contradicts the notion of localization in TAG, which says that all and only the *required* arguments of a lexical anchor are introduced in the elementary tree of that lexical element. Furthermore, the grammar gets hardly extendable. The introduction of a new (type of) modifier would require changes in all the elementary trees that the modifier can possibly adjoin into.

Finally, lexical entries like the ones shown above may be defendable for English, where word order is mostly fixed, and modifiers also tend to occur in a specific order. For free word order languages (like German, which I deal with in this thesis), such an approach would certainly yield serious problems. In the German middle field, modifiers can usually appear in any order, and intermixed with other arguments. It is not clear how this can be captured using lexical entries like the ones in figure 4.2.

Head specification. As mentioned before, the SPUD system requires an entity to describe as input. Its syntactic realization will then obligatorily be regarded as the head of the realized utterance. Thus, similarly to (Koller and Striegnitz, 2002), certain variations in the realization of a generation input cannot be achieved.

As I have shown, the task whose solution SPUD attempts differs in important aspects from the general task of realization from flat semantic representations I am considering in this thesis.

SPUD is a system that can only deal with specific input, that can not be easily extended to other domains, and whose grammars have to be hand-tailored to its purpose (and therefore can not be replaced by existing standard TAG grammars).

4.3 Integrated Discourse Generation (InDiGen)

The goal of the German InDiGen project[9] at the University of the Saarland is an integrated approach to discourse and sentence planning. Generation is done with an LTAG for German, and thus realization tasks are also part of the project.

4.3.1 Description

The generation in InDiGen differs from that in the SPUD system in that it employs a chart-based generation algorithm (Kay, 1996).[10] Furthermore, all possible realizations of an intended utterance are generated, so that in a post-processing step they can be evaluated according to their appropriateness.

However, InDiGen follows SPUD in associating pragmatic content in addition to semantic contributions with lexical entries. The main focus of the system are also the inferences that yield to the generation of contextually appropriate referring expressions.

4.3.2 Discussion

The LTAG grammar used in InDiGen supposedly follows the design decisions in SPUD, though the example lexical entries do not show the idiosyncratic treatment of modification I commented on above.

Two of the flaws (for our purpose) of the SPUD system still carry over to InDiGen:

Non-scopal semantics. In contrast to SPUD's modal logic, first order logic is used as the semantic representation language in InDiGen. As above, this makes it impossible to express scopal relations in the semantics, and to distinguish between the examples (4.3) and (4.4) considered above.

Head specification necessary. The input specifications of InDiGen quite resemble those in SPUD, and consist (minimally) of the intention to "describe *e*," where *e* is some entity in the system's world knowledge. This puts realization

[9]see `http://www.coli.uni-sb.de/cl/projects/indigen.html`

[10](Striegnitz, 2001) shows the integration of a chart-based generation algorithm into the architecture used in SPUD

decisions into the responsibility of the user (or the planning system that specifies the generator's input).

4.4 Verbmobil

Verb*mobil* (Wahlster, 2000) was a huge joint project for the translation of spontaneous speech, whose generation module (Becker et al., 1998, 2000) also used Tree Adjoining Grammar. Because of the special properties of on-line spoken dialogue translation, there were specific requirements for the generator:

- The module had to robustly deal with the special grammar of spontaneous speech, speech errors, corrections, etc., as well as errors or inconsistencies the preceding modules produced.
- On-line translation required very efficient next-to real-time translation.
- The generator input was not produced by a macroplanning module, but came from a semantic transfer module. Therefore, some decisions usually left for the microplanner were already specified in the input, e.g. sentence mode, syntactic categories etc.

4.4.1 Description

The generation task was split into two phases in Verb*mobil*, the microplanning and the syntactic realization phase. Microplanning included the tasks of lexical choice, aggregation, theme and focus control, and reference specification. It was seen, much like in this thesis, as a constraint satisfaction problem. However, the solution in Verb*mobil* consisted of a huge database of thousands of rules, mapping the input specifications to parts of a sentence plan. These rules were of course closely adapted to the specific type of input specifications (VITs, Verbmobil Interface Terms) on which they had to operate, and are therefore not easily transferrable to other systems.

The sentence plans that served as input to the syntactic realization module were trees consisting of a node for each lexical item (remember that word choice was almost complete at this stage) annotated with syntactic features, and edges labelled with semantic/syntactic roles (like *agent* or *modifier*) connecting these nodes.

The tasks remaining for syntactic realization were the choice of an elementary TAG tree for each lexical item, and the determination of the syntactic operations combining them. This was accomplished using a first tree-selection phase which depended on the output (in particular the syntactic annotations) of the microplanner. A combination phase followed, employing a guided best-first search strategy. A post-processing step dealt with morphology.

4.4.2 Discussion

Modularity The syntactic realizer of Verb*mobil* is very domain independent and modular, it uses descriptive knowledge sources (for example a standard, reusable TAG grammar). However, the task it performs is very narrow. The realization problem considered in this thesis also subsumes subtasks that were accomplished by the microplanner in Verb*mobil*, most prominently lexical choice.

The Verb*mobil* micro planner, on the other hand, receives strongly domain dependent input that is not easy to mimick in other generation systems. Furthermore, the many planning rules can not easily be reproduced in such a new system. In effect, of the two Verb*mobil* generation components, only the syntactic realizer might be reusable.

Interleaved micro planning and realization It is worth pointing out that the construction of a flat semantic representation should in principle be possible in a dialog translation system, and make it possible to use a syntactic realizer like the one presented in this thesis. This gets even more appealing as in some cases, realization is interleaved with micro planning decisions, particularly word choice. For example in the two German examples below, reference specification (i.e. the choice of a pronoun vs. a complete NP) interacts with the choice of an actual TAG elementary tree for the verb lexical item:

(4.5)	*Ich*	*zeige*	*Ihnen*	*die*	*Karte.*
	I	show	you$_{dat}$	[the	map]$_{acc}$.

I show you the map.

(4.6)	*Ich*	*zeige*	*sie*	*Ihnen.*
	I	show	it$_{acc}$	you$_{dat}$.

I show it to you.

The verb lexical entry (with either the NP$_{dat}$ before the NP$_{acc}$, or the other way around) would have to be chosen with respect to the fact whether those NPs are pronominalized; but likewise, information structure might determine the word order (and thus the choice of the elementary tree for the verb) – which would in turn constrain the reference specification for the objects.

4.5 SmartKom

The multi-modal human-machine interaction system SMARTKOM includes a linguistic presentation module that uses TAG for generation. Sentences belonging to various domains and tasks need to be expressed, such as:

- question answering in the movie theater domain

(4.7)	*Hier*	*sehen*	*Sie*	*eine*	*Übersicht*	*über*	*das*	*Programm*	*der*
	Here	see	you	a	overview	about	the	program	of-the
	Heidelberger	*Kinos.*							
	Heidelberg	cinemas.							

Here is an overview of the movie theater program in Heidelberg.

- pedestrian navigation

(4.8)	*Welchen*	*Weg*	*möchten*	*Sie*	*nehmen?*	
	Which	route	want	you	take?	

Which route do you want to take?

- user authentification

(4.9)	*Bitte*	*sprechen*	*Sie*	*nach*	*dem*	*Ton!*
	Please	speak	you	after	the	sound.

Please speak after the sound.

- meta dialogue

(4.10)	*Dazu*	*habe*	*ich*	*leider*	*keine*	*Informationen.*
	For-that	have	I	unfortunately	no	information.

Unfortunately, I do not have any information regarding this request.

A fix requirement for the generator was to produce an early prototype that could already generate a range of outputs like the ones above. In addition, simple sentence (string) generation was not possible, as the output of the generator was to be used in a Concept-to-Speech (CTS) synthesis module (see Schweitzer et al., 2002). For this reason, rich syntactic output had to be produced.

Thus, the SmartKom generator employed the fact that TAG lexical entries need not be minimal. Instead, *templates* were used as the elementary syntactic structures that comprised phrases or even whole sentences. The planning module could then choose these larger templates without the need to have a complete syntactic realizer that builds them from words. The templates could also contain variable parts that the planner could instantiate with parts of the generator input.

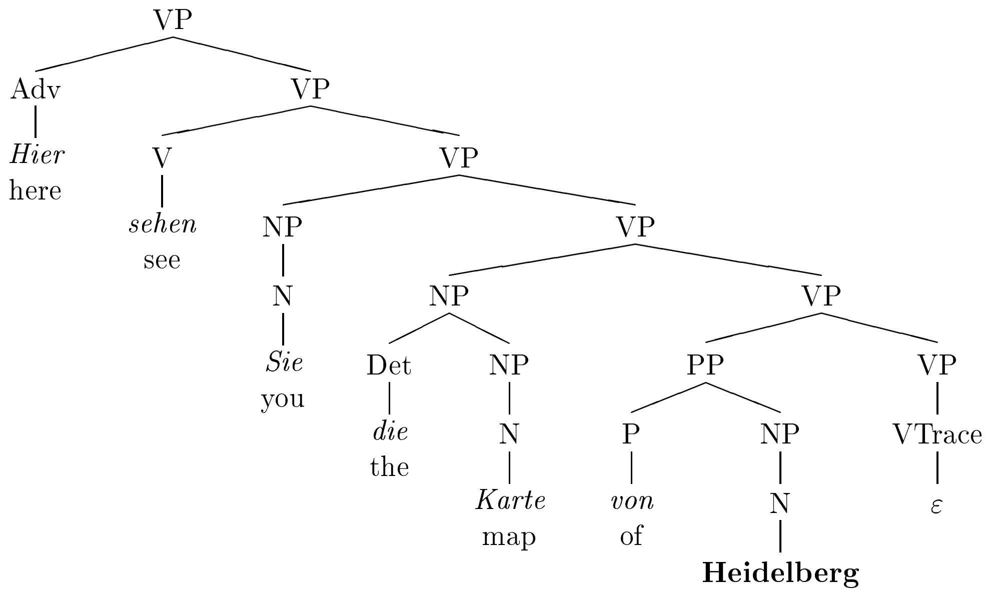
Figure 4.3 shows a template like those that were used in the SmartKom generator. The variable part is printed in bold face. It could be replaced by other words from the domain knowledge base.

For planning, SmartKom used a large rule base that traversed the abstract input specifications (described in the introduction of this thesis) and chose the appropriate templates. This method makes the generator hardly manageable as the rule base gets larger.

However, the templates are in fact TAG derived (or intermediate) trees that could be seen as the result of a sequence of substitutions and adjunctions on "real" elementary trees. Thus, they can be split up into their elementary parts (the "words") in later development stages, which makes the rule base and therefore the whole generator much more modular. This has in fact been explored in the SmartKom system.

In summary, the method of using templates can be judged very effective where rapid prototypes are needed. The templates can later easily be split up to yield a "real", lexicalized TAG grammar. This has been proven feasible in the SmartKom project.

The use of a large hand-written rule base, though, is not an advantage of the system. The rule base depends directly on the type of input specifications and the domain ontology employed in the project, and can thus not be used in other systems.



(Here you see the map of **Heidelberg**.*)*

Figure 4.3: Example SmartKom template.

4.6 Other Approaches

There have been several other attempts at using Tree Adjoining Grammars for generation. As they are not directly relevant to this thesis, I will only mention some of them here.

One of the very early TAG generators is Mumble. Mumble-86 makes use of a systemic-functional approach for its generation decisions. There, realization starts at the head of the derivation tree and proceeds to successively find new parts of the semantic input that can be adjoined or substituted into the already derived syntactic structure. This yields a strict top-bottom generation algorithm.

Followers of Mumble in the fact that they employ systemic-functional strategies are McCoy, Vijay-Shanker, and Yang (1992). The DSG/TAG generation architecture of (Kozlowski, 2002), in turn, follows (McCoy et al., 1992) by also specifying active *regions* during the realization process, which is split into a *descent* and an *ascent* phase.

Another approach is G-TAG, a generation system for (multi-sentential) texts by Laurence Danlos of the University at Paris 7. See (Danlos, 2000) for a presentation.

Chapter 5

A Syntactic Realizer

In this chapter, I describe the syntactic realization problem and the general algorithm I propose for its solution. I give a motivation and the broader context of the problem in section 5.1. Then, I will go on to characterize the syntactic realization task as a constraint satisfaction problem. I will sketch the algorithm in section 5.3.

Chapter 6 describes the implementation of this algorithm in Java.

5.1 Motivation

Syntactic realization is the problem of, given a representation of some semantic content, finding an adequate syntactic structure expressing that semantics, in accordance to some given grammar and semantics interface.

In a natural language generation (NLG) system, syntactic realization is just one of the tasks that have to be performed. Reiter and Dale (1997) distinguish six tasks in NLG:

Content determination. Specification of what should be said.

Discourse planning. The structuring of the content into subgoals, and the determination of discourse relations that hold between the subparts (e.g., ELABORATION).

Sentence aggregation. Aggregation of content into sentences, this includes coordination, subordination, the construction of ellipses and so on.

Lexicalization. The choice of lexical items for the parts of the content. This includes choice not only between synonyms, but also across syntactic categories.

Referring expression generation. The choice of appropriate referring expressions for the entities that are referenced in the content specification. Referring expression generation is strongly intertwined with lexical choice in general.

Linguistic realization. The actual building of a syntactic structure by means of the rules of a given grammar. This step includes verb cluster aggregation, inflection, etc.

In real natural language generation systems, these tasks often can not be that clearly distinguished, though. For this thesis, we assume a slightly broader understanding of the term *syntactic* (or *linguistic*) *realization.* It is meant to also include the largest part of the lexicalization task.

5.1.1 Context of the realization task

The context for the syntactic realizer developed in this thesis is the SmartKom[1] system, a multi-modal human-technology interaction system (Wahlster et al., 2001). The generator module controls only one of several output modalities, the others being graphical and textual presentations, as well as gestures (of an animated humanoid figure on a screen). Figure 5.1 shows a screen shot of the system.

Figure 5.1: SmartKom screen shot.

The generator input is an abstract presentation schema containing a (domain dependent) speech act classification and a presentation task that contains references to entities in the ontology or to other objects, such as items in the graphical presentation.

The current generator architecture is shown in figure 5.2. The generation is currently done in one huge planning step, using the `PrePlan` planning system, going directly from the abstract input representations to TAG derivation trees. For this, a TAG grammar consisting of *templates*, i.e. elementary trees that are anchored by phrases instead of single lexical items, is used. Obviously, as the input specification depends strongly on the dialogue domain, the rules of

[1]see `http://www.smartkom.org` for more information

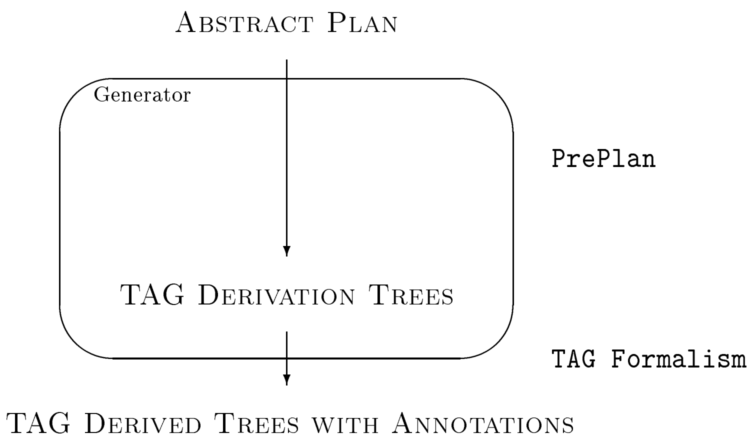


Figure 5.2: Old generator architecture.

the microplanner are also domain-dependent and can not be easily extended to cover different topics.

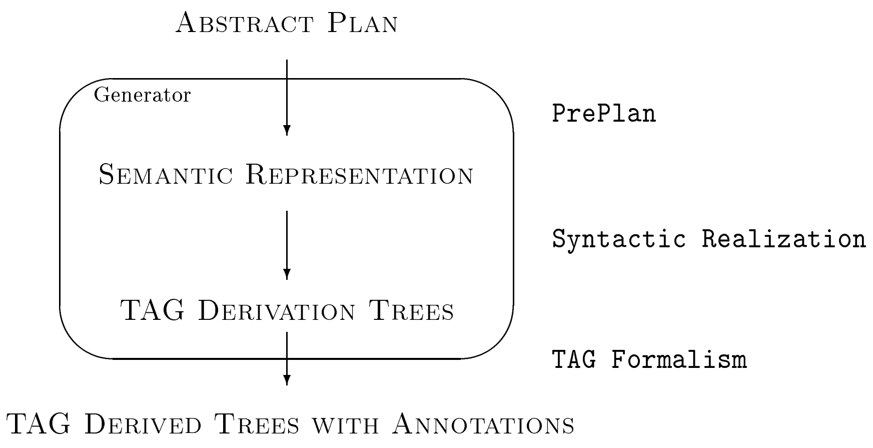


Figure 5.3: Intended generator architecture.

Therefore, I will introduce an intermediate level of representation into the generator architecture. This is the flat semantic representation presented in chapter 3. I will not deal with the domain-dependent initial planning step in this thesis. The semantic representation, though, should be domain independent. This allows a syntactic realization module that is truly reusable in other generation systems. Figure 5.3 depicts the intended new generator architecture.

5.2 Realization as a constraint satisfaction problem

The syntactic realization problem can be approached in many different ways. One particularly well-suited way to see it is as a constraint satisfaction problem. The problem is to find a syntactic structure that correctly expresses all and only the given semantics. The three basic global constraints are therefore:

(1) that the result should be a well-formed sentence according to some input grammar,

(2) that the input semantics should be completely expressed by it, and

(3) that nothing additional should be expressed.

5.2.1 Constraint systems

A thorough introduction into constraint satisfaction problems (CSP), constraint solving algorithms and the representation of natural language generation as a CSP can be found in (Löckelt, 2000), in the context of the Verb*mobil* project. Here, I will just give a very brief note on what is understood as a constraint system in this thesis.

A constraint problem consists of a set of *variables* with fixed domains, and a set of *constraints* over possible *assignments* of values to these variables. A solution to the constraint problem is an instruction assigning values to each of the variables in such a way, that all the constraints hold at the same time. The constraints are relations between certain combinations of the assigned values.[2]

The central point in viewing a computational problem as a constraint satisfaction problem is to separate the problem specification from the methods for the production of solutions. In that way, when the variables, their domains, and the constraints constituting a problem have been specified, different algorithms can be applied to find its solution(s). These algorithms include simple trial-and-error up to sophisticated methods using propagation, etc. Chapter 3 in (Löckelt, 2000) describes many algorithms used in constraint-solvers.

Advantages Representing a task as a constraint problem has many advantages. Once a problem is specified in that way, different algorithms can be used for its solution. This ensures that optimizing the efficiency of a program is possible with minimal effort, as the task specification does not have to change. Furthermore, many efficient constraint solving algorithms exist, and can be reused for new constraint problems.

For many applications, natural representations as constraint problems exist, and such representations are usually very descriptive in nature (and can for example be more easily understood by domain experts). In addition, the splitting of the problem solving process reduces its complexity: later optimizations will usually only affect one of the two parts. Thus, the introduction of additional constraints is easily possible and will not have consequences for other parts of the system.

[2]Definitions of the basic terms *assignment*, *constraint*, *constraint problem* etc. can be found in (Löckelt, 2000, chapter 3).

Constraint systems in LT Consequently, the use of constraint solvers has already been explored for some language technology applications, mostly for parsing (e.g. parsing of dependency grammar, see (Duchier and Debusmann, 2001)) or underspecified semantic representations.

For natural language generation, a constraint-based approach has been explored in the Verb*mobil* system (see Becker et al., 2000) and more thoroughly, and building upon the work in Verb*mobil*, in (Löckelt, 2000).

5.2.2 Task

Characterizing the syntactic realization task more specifically, a set of lexical items has to be chosen from the grammar, each of which expresses some (non-overlapping) part of the semantics. For each of the lexical items, the grammar usually contains several TAG elementary trees anchored by the word. The realizer has to chose one tree for each item, and consistently use it during the derivation. Then, these trees need to be combined binarily by syntactically and semantically appropriate substitutions and adjunctions, in order to produce a well-formed derivation tree.

The two global *syntactic* constraints posed by the TAG formalism that remain at the end of the realization process are:

(4) that all operations may be carried out in combination (i.e. the feature unifications do not yield unification clashes), and

(5) that the computed derived tree may be *finalized*, i.e. the top and bottom feature structures at each node unify.

These two constraints can only be checked on a complete potential solution, as feature equations may inherit certain values up or down the tree and thus cause complex interactions in the feature unifications.

5.2.3 Requirements

As discussed above, the global constraints can only be verified after the completion of the realization process. However, producing large TAG trees is computationally expensive; and particularly the bigger feature structures and operations thereon cost much. Thus, these constraints should be checked at least partially on partial results of the realization.

Furthermore, the algorithm should provide measures that avoid that intermediate structures are computed multiple times if they occur in different stages of the realization process.

5.3 Algorithm

In this section, I will point out how the constraint variables and their domains for a constraint-based representation of the realization problem can be constructed from an MRS-like structure. This descriptive presentation of the task naturally lends itself to a range of constraint solvers.

Subsection 5.3.3 sketches the algorithm proposed in this thesis, which has the advantage of conceptual simplicity, followed by the mentioning of some other possible algorithms that might improve the efficiency of the realizer.

The implementation of the realizer in Java is presented in chapter 6.

5.3.1 Semantic graphs

In contrast to the syntactic realization task in Verbmobil (see section 4.4) the input to the realizer assumed in this thesis is not a (rather syntactic) sentence plan, but a semantic representation σ. It does therefore not suffice to introduce a constraint variable for each "link" between lexical items in the input tree.

I will therefore develop a slightly more complex specification of the task as a set of variables. Firstly, I will introduce the notion of a *semantic graph*, that will make certain semantic links in the input more explicit. This will in the second step allow the determination of the set of variables that constitute the realization problem, and their appropriate domains.

The bag of elementary predications of the semantic input σ can be seen as a semantic graph G:

Definition 6 (Semantic Graph) *The semantic graph G for an MRS structure $\langle t, L, C\rangle$ is constructed as follows: For each semantic literal in L, introduce a node in G. For all elementary predications (literals) l_1 and l_2 in L containing the same (ordinary) semantic argument, G shall contain an undirected edge linking the nodes corresponding to l_1 and l_2. Furthermore, for each l_i containing a scopal handle variable h_0 as its argument, let H_i be the set of all literals "dominated" by h_0, i.e., H_i contains all literals whose label is h_0, and recursively all those which are labelled with any handle argument h_i that any of the literals in H_i takes. Then, introduce an undirected edge between the node corresponding to l_i with the nodes corresponding to each element in H_i.*

See the semantic graph for the MRS structure

$$\begin{aligned}(5.1)\quad \langle h_2, \{&h_5 : \mathit{every}(x, h_3, h_4) \wedge h_3 : \mathit{rich}(x) \wedge h_3 : \mathit{man}(x) \wedge h_2 : \mathit{allegedly}(h_1)\\ &\wedge h_1 : \mathit{usually}(h_0) \wedge h_0 : \mathit{drives}(e, x, y) \wedge h_8 : a(y, h_6, h_7) \wedge\\ &h_6 : \mathit{cadillac}(y)\},\\ \{\}&\rangle\end{aligned}$$

(corresponding to one of the possible scopings for the sentence "Every rich man allegedly usually drives a cadillac.") in figure 5.4 as an example.

Some properties of the graph can instantly be seen: It contains complete subgraphs, one for each semantic argument variable. These subgraphs roughly contain all the literals "about x," "about y," and so on.

Moreover, two complete subgraphs typically have just one node in common. Thus, the derivation tree we are looking for just constitutes one of the spanning trees of the semantic graph (figure 5.5 shows an appropriate derivation tree for the semantic graph above[3]). This is obvious because semantic linkings (the "unifications" of semantic argument variables or handle variables) can only be

[3]The arrows point from the inner to the outer tree.

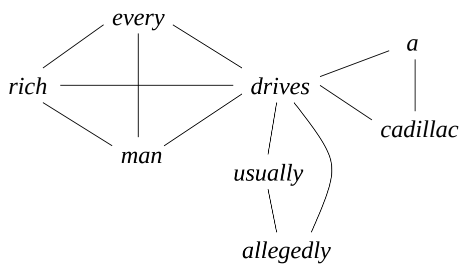


Figure 5.4: An example of a semantic graph.

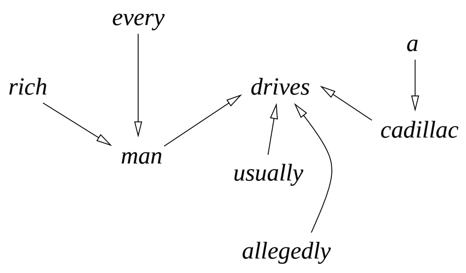


Figure 5.5: Derivation tree as spanning tree of a semantic graph.

obtained between two literals by carrying out an operation between the elementary trees that express these semantic literals. For example, if in a semantic input both $h_1 : \mathit{peter}(x)$ and $h_0 : \mathit{sleeps}(e, x)$ contain the semantic argument variable x, the tree chosen for *peter* has to take part in a syntactic operation with *sleeps*. However, it is not determined in the semantic graph, in which direction the substitution or adjunction will take place (i.e., which tree is the outer, and which the inner one). Sometimes even both directions (with appropriate TAG trees for the literals) might be possible.

5.3.2 Variables and Constraints

As shown above, it is not possible to just take each semantic "link" as a variable for the realization problem here (as is done in Verb*mobil*). This comes from several factors that may occur:

- the semantic input is not a tree;
- there may be several lexical items expressing a semantic literal, in particular also from different syntactic categories, yielding completely different syntactic structures;
- a lexical entry may be associated with more than one literal of the semantic input.

Consequently, I define the variables for the realization problem as follows: For each semantic literal in the input, one variable is introduced. Conceptually, the assignment of such a variable represents the syntactic operation which integrates the elementary predication into the sentence. The structure of such a variable v is a tuple $\langle \sigma_o, t_o, t_i, OP, n \rangle$. A well-formed assignment of a variable specifies the outer semantics σ_o, i.e. with which of the neighbors in the semantic graph the tree chosen for this literal will be combined; the TAG elementary tree t_o chosen for the outer semantic literal; the inner TAG elementary tree t_i which expresses the literal this variable is associated with; the operation $OP \in \{Subst, Adj\}$; and the node address n of the substitution node or adjunction site in the outer tree t_o.

Trivial assignment A variable can also be trivially assigned, which means that no operation will be carried out to include the semantic literal associated with it into the semantics of the sentence. Most importantly, the variable constructed for the syntactic head of the sentence always gets the trivial assignment (as the TAG tree expressing this semantics will only be an outer tree for operations, never an inner tree). Moreover, as certain TAG elementary trees may express more than one semantic literal, the semantics associated with a variable might have been already expressed by some tree chosen elsewhere in the semantic graph. Such a variable is also trivially assigned, as each semantic literal is expected to be realized exactly once in the sentence.

N:m-mapping from semantics to syntax On the other hand, there may in general be an n:m-mapping of semantics to syntax, i.e. one (or several) semantic literals may also be expressed by a combination of several syntactic elements. In my implementation, I do not deal with this very general case. However, as was explored in the SmartKom project, a TAG generator may make heavy use of *templates*. Thus, the syntactic elements can be put together into one TAG *template* which is then integrated as one elementary unit into the grammar. I obtain an n:1-mapping from semantics to syntax, which can be dealt with as discussed in the previous paragraph.

Constraints on the variable assignments There remain very few global constraints. In addition to the ones mentioned above, it is required that:

(6) wherever a semantic literal is chosen as the outer semantics, one and the same TAG tree must be chosen for it, and if it ever is an inner tree (if its own variable is not assigned trivially), this inner tree must also be the same

Furthermore, the local constraint that

(7) the feature structures in one operation are unifiable

is subsumed in the global constraint (4). But as constraint (4) holds between *all* variables in the specification, it can only be checked finally, when all variables have been set. The unary constraint (7), though, can be checked at each assignment, and will therefore be much more efficient. Because many feature phenomena are local, this will improve the overall performance of the generator, as it reduces the search space.

5.3.3 Depth-first search

Plain “bottom-up” or “top-down” generation is not well-defined for the semantic graph as input, as it per se does not have a determined head (see figure 5.6 for the ambiguous example given above in (4.1)-(4.2), or the even more notorious example in figure 5.7, presented here after (Kozlowski, 2002)). For a first demonstration of feasibility, I chose a straightforward depth-first traversal of the graph. However, one could imagine using a pre-processing step in order to find possible semantic heads, and then try a top-down (or left-edge) traversal instead.

(5.2) $\{h_0 : \mathit{see}(e, x, y) \wedge h_2 : \mathit{you}(x) \wedge h_3 : \mathit{movies}(y) \wedge h_4 : \mathit{def}(y, h_3, h_5)$
$\wedge h_1 : \mathit{run}(f, y) \wedge h_6 : \mathit{here}(e) \wedge h_7 : \mathit{today}(f)\}$

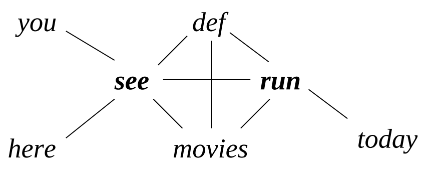


Figure 5.6: Semantic graph with undetermined head.

(5.3) (a) $\{h_0 : \mathit{teach}(e, x, y) \wedge h_1 : \mathit{good}(e) \wedge h_2 : \mathit{mary}(x)\}$
(b) Mary excels at teaching.
(c) Mary teaches well.
(d) Mary is a good teacher.

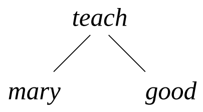


Figure 5.7: Ambiguous semantic graph and realizations.

The basic algorithm is presented in figure 5.8. It is a relatively straightforward depth-first traversal of the semantic graph, during which at each new node, the associated variable is assigned a value. The traversal is executed by simple backtracking in the case of a failure to assign a variable its value.

A depth-first traversal with backtracking guarantees that all solutions to the realization problem will be found. With the current algorithm, a post-processing module could chose among the grammatical realizations the one which fits best for the current system purpose.

Finding assignments for variables The search space for the assignments for variables is the set of all possible combinations of outer semantics, appropriate outer and inner trees, operation and goal node. This necessitates a complexly cascaded loop to test all possibilities. The algorithm is presented in figure 5.9.

Algorithm 1. (Depth-first realization)

```
Called on: a start vertex $v$ in the semantic graph $G$.
  1. If there is not yet a variable for $v$, create a new variable $var$.
  2. Find a new assignment for $var$.
     (a) If there is a new assignment:
         i. Find a new, unvisited vertex $w$ in $G$. (depth-first-search)
         ii. Start recursively with step 1 on $w$.
     (b) Else (no assignment can be found any more) backtrack to the last visited variable.
  3. When the whole input semantics has been realized (an assignment is found, but no other unvisited vertices exist), execute all operations that were assigned to the variables.
     (a) If a global feature clash occurs, start backtracking from the last set variable.
     (b) Else, store the realization result. If all possible realizations should be found (instead of just one), start backtracking from the last set variable.
  4. Return the stored realization result(s).
```

Figure 5.8: The realization algorithm.

```
Algorithm 2. (Variable assignment)
Called on: variable $var$.
  1. If no variable has been chosen as the head, and $var$ has not
     been the head in an earlier step during backtracking, try the
     trivial assignment. Return true.
  2. Else, for all possible outer semantics (neighbors of the vertex
     $n$ in the semantic graph $var$ is associated with):
     (a) For all trees that express the chosen outer semantics:
         i. For all trees that express the (inner) semantics $var$
            was created for:
            A. Choose an operation; and
            B. Choose a new goal node in the outer tree for
               this operation.
  3. If no such assignment can be found, return false.
  4. Else, return true.
```

Figure 5.9: Finding an assignment for a variable.

Furthermore, the variable assignment interacts with the traversal. In the case of backtracking, the assignment loop has to be restarted in just the position that it was left in, when the last valid assignment had been found. Therefore, the status of the assignments has to be stored, to enable the realizer to continue with only those assignments that have not been tested before.

Execution During the search process, operations are collected as the assignments of variables, and only the local constraint (7) can be checked. Thus, in a final step, all other global constraints need to be checked for the solution candidate. If a constraint fails at this stage, backtracking needs to be triggered to find a new candidate.

Then, the collected substitutions and adjunctions need to be actually carried out in order to produce the derived TAG tree that is the solution to the realization problem.

5.3.4 Ordering of adjunctions

Actually, another parameter plays a role in the variable assignment, that has been ignored until now. This is the order in which adjunctions are carried out. As mentioned above (see subsection 2.1.3), I follow (Schabes and Shieber, 1994) in allowing multiple adjunctions into the same node in the supposed TAG grammar.

Furthermore, (Joshi and Vijay-Shanker, 1999) have shown that the ordering of adjunctions is not only necessary for adjunctions into one and the same node, but also if trees adjoin into different nodes of the trunk of an elementary tree, in order to avoid semantic underspecification (see subsection 3.1.2).

Of course, the order in which substitutions are carried out, or adjunctions into different elementary trees, does not play a role (and does not yield different derivations or derived trees). Schabes and Shieber (1994) show how derivations need to be seen as equivalent if they only differ with respect to the orderings of such operations.

Redefinition of variables This means that in fact the variables assumed as the representation of the constraint problem should be a tuple $\langle \sigma_o, t_o, t_i, OP, n, \mathcal{A} \rangle$, with σ_o, t_o, t_i, OP, and n as defined above; and where $\mathcal{A}$ is a set of adjunctions (e.g. represented by the variables that are assigned those adjunctions) that need to be carried out *before* this one.

Discussion An upper bound for the combinatorics introduced by this additional parameter is the powerset of the set of variables (equivalent to the set of semantic literals by definition). That is, for a variable $var \in V$, $\mathcal{A}_{var} \in \wp(V)$. It can be seen that the introduction of this parameter in the general case increases the search space significantly.

However, $\mathcal{A}_{var\,i}$ that differ only in elements that refer to variables which are assigned *substitutions* or adjunctions into different outer trees are equivalent. Moreover, the variation of the parameter $\mathcal{A}$ only makes sense if $OP = Adj$. These facts ensure that in all practical cases, the search space will not explode exponentially.

For the adjunctions, the specified scoping in the input semantics actually determines the ordering of operations completely. As in (Joshi and Vijay-Shanker, 1999), inner trees will be adjoined "inside-out," i.e. scoped-over elements before the out-scoping elements. This fact will be exploited in the implementation (see chapter 6).

Chapter 6

Implementation

In this chapter I will describe the implementation of the semantics interface and the syntactic realizer in Java. Section 6.1 gives some preliminary remarks about the context of the implementation in the larger SmartKom system, section 6.2 presents the implementation of the semantics interface, and the realization algorithm is shown in section 6.3.

6.1 Preliminaries

As mentioned briefly in the previous chapters, the realizer presented in this thesis is part of a larger system, the SmartKom system. Generation in SmartKom is currently already done with Tree Adjoining Grammars, using a large rule base for the generation decisions.

6.1.1 Grammar organization

For the testing of the realizer, a small toy grammar for German was produced including a set of lexicalized TAG trees with features and an appropriate semantics interface. For the development of the grammar, existing templates from the SmartKom grammar were partly used and split up (as described in section 4.5).

I introduced only those features into the grammar, that were directly needed in order to prevent ungrammatical outputs. A TAG grammar with larger coverage would certainly need a much larger set of features, possibly modelled after the English XTAG grammar.

The grammar is written in an XML format. The syntax schema is given in appendix B. In SmartKom, tools have been developed that allow the construction and maintenance of a TAG grammar using graphical interfaces. But still, a lot of fine-tuning has to be done by hand by editing the XML text files.

Figures 6.1 and 6.2[1] show example entries from the toy grammar. A list of

[1] constraint or cstr designates the *adjunction constraint*, NA means *null adjoining*. argSt stands for *argument status*, i.e. subject, object, or adjunct. This feature is required by the concept-to-speech synthesis.

NP↓:x labels a substitution node of category NP, whose associated semantic argument variable is x. Feature structures are only shown where they contain values.

trees in the toy grammar is given in appendix A.

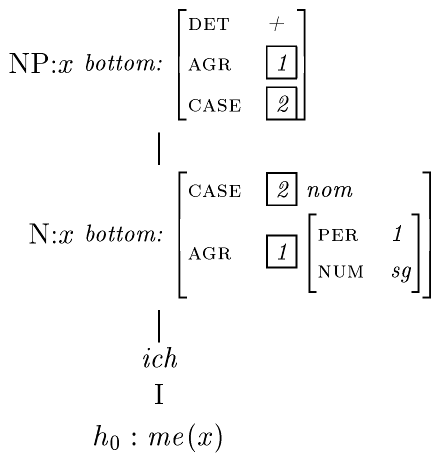


Figure 6.1: Example grammar entry for "ich" (*I*).

6.1.2 Technicalities

The SmartKom system is implemented in Java, and so is this realizer. Apart from making an integration into the current system easier, the Java language has several advantages:

- it is operating system-independent; and
- there exist many predefined packages in the large Java community that can be used and are easily accessible through their API specifications.

An abridged version of the Java API documention for the two packages implemented for this thesis is included in the appendix (appendices D-F). It was automatically generated using the Doxygen tool (see van Heesch, 1997) which is freely available under the terms of the GNU General Public License.[2]

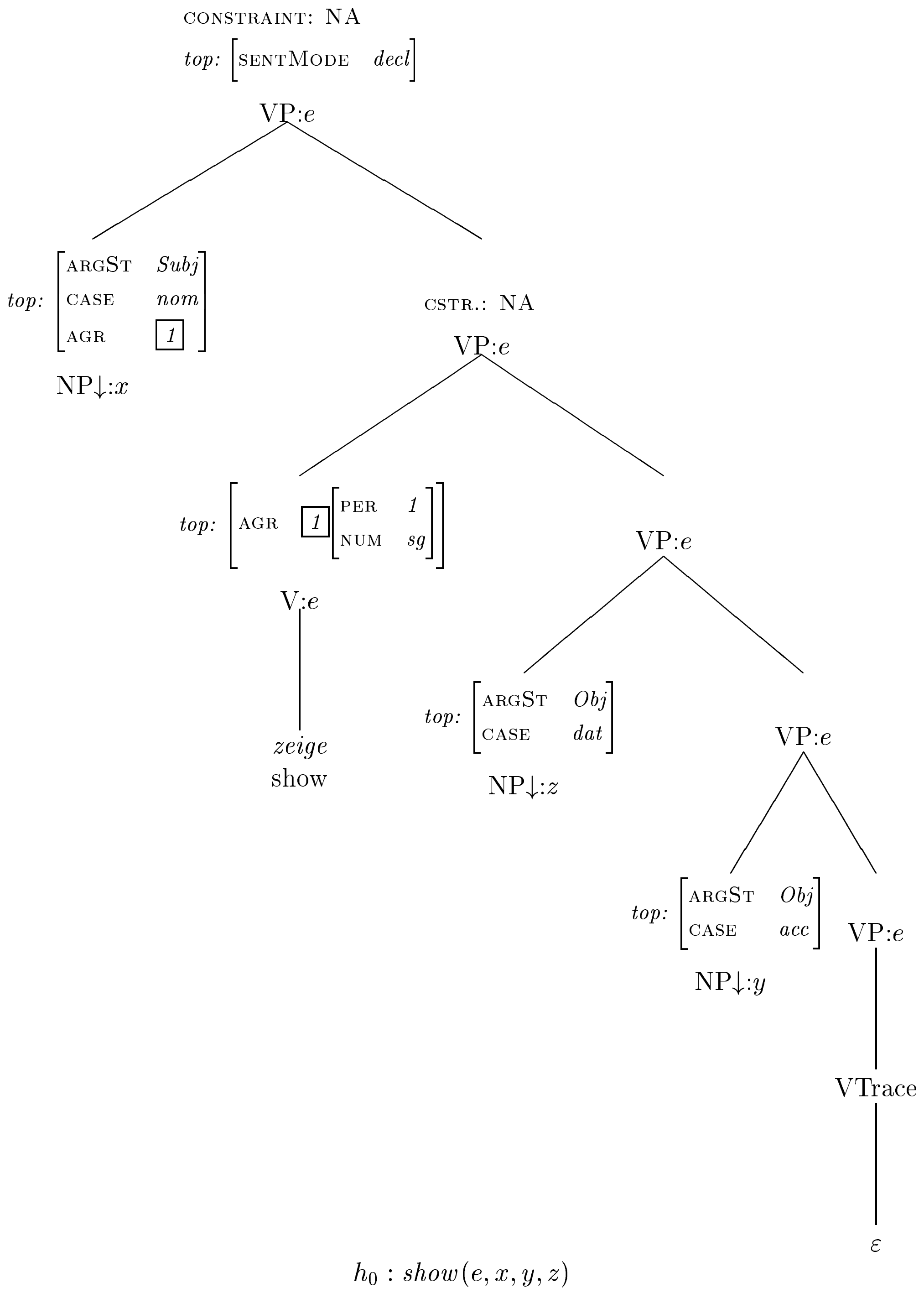
The complete javadoc documentation is contained in the source files. These are available in a CVS directory at the German Research Center for Artificial Intelligence (DFKI Saarbrücken). The basic path to the source directory is `/project/imedia3/imediabackbone/framework/SKGen/src/`.

6.1.3 Existing packages

The implementation of the realizer builds on existing packages developed for the SmartKom project. In particular, I use the following three packages:

`de.dfki.smartkom.generator.tag` The `tag` package contains data structures for Tree Adjoining Grammars. The classes and their most important functionality are shown in table 6.1.

[2]see `http://www.gnu.org/copyleft/gpl.html`



constraint: NA
top: [sentMode decl]
VP:e
top: [argSt Subj, case nom, agr 1]
NP↓:x
cstr.: NA
VP:e
top: [agr 1 [per 1, num sg]]
V:e
zeige
show
VP:e
top: [argSt Obj, case dat]
NP↓:z
VP:e
top: [argSt Obj, case acc]
NP↓:y
VP:e
VTrace
ε

$h_0 : show(e, x, y, z)$

Figure 6.2: Example grammar entry for “zeige” (*show*).

class	description
`TagTree`	TAG tree data structure; input/output through XML files; substitution, adjunction
`TagNode`	TAG tree node; recursive structure
`TagTreeFamily`	container for a family of TAG trees
`TagGrammar`	container class for a list of tree families; methods for finding trees
`TTools`	miscellaneous printing tools for debugging

Table 6.1: Classes of the `de.dfki.smartkom.generator.tag` package.

TAG grammars are saved in XML files and can be read and printed using the `templates` package. A TAG grammar (`TagGrammar`) is organized in tree families, as is assumed in standard descriptive TAG grammars, and represented here as `TagTreeFamily` data structures. These families group conceptually related trees together, such as all the trees for a certain verb class, etc. A `TagTreeFamily` consists of `TagTree`s, which can be found using the method `findTagTree(String name)` by their name.

`TagTree`s have a `[String identifier]` (the name), and a `[TagNode rootnode]`. They also specify any open substitution nodes, their derivation history (if they are not elementary trees) and the foot node (if they are auxiliary trees). As these are fully *lexicalized* trees, they keep track of their `[Vector anchors]` and their `[TagNode head]`, which is the prominent anchor (= the anchor whose maximal projection is the root of this tree). Nodes in the tree are stored as references.

The `TagTree` class provides functionality for reading trees from XML files and printing them, according to a specified XML schema. In addition to various predicates (such as `isAuxiliary()`, `isFinal()`), the implementation provides the TAG operations substitution (method `substitute(String nodeID, TagTree tree2)`) and adjunction (`adjoin(String nodeID, TagTree tree2)`). These methods also trigger feature unification, and store the results destructively in the trees that participate in the operation.

`de.dfki.smartkom.generator.templates` This package provides a specialized interface to the Xerces DOM Parser[3] that parses XML documents and constructs hierarchical Java document objects representing their content.

XML documents are used as implementation-independent data files in SmartKom.

`de.dfki.smartkom.generator.unifier` The `unifier` package contains a data structure for recursive feature structures with coreferences. This class also implements a unifier for feature structures dealing correctly with coreferences

[3]see `http://xml.apache.org/xerces2-j/index.html`

that reach over the whole TAG tree.

class	description
`FSPath`	paths to feature structures; with comparison functionality
`FSType`	feature structure data type, i.e. atomic value or recursive feature structure
`FeatureStructure`	class for recursive attribute value matrices; with unification functionality, input/output through XML files
`UnificationException`	unification clash

Table 6.2: Classes of the `de.dfki.smartkom.generator.unifier` package.

The `tag` and `unifier` packages were reimplemented in Java after a previous Common Lisp implementation in the Verb*mobil* project.

6.2 TAG semantics interface

The semantics interface for TAG was implemented as a wrapper around the existing TAG data structures. It is provided in the package `de.dfki.smartkom.generator.semantics`, whose classes are listed in table 6.3. Consequently, all

class	description
`Proposition`	an MRS elementary predication
`PropositionSet`	an MRS structure, as used in the semantics interface
`Function`	the function ϕ mapping from TAG tree nodes to semantic arguments
`SemTagTree`	a TAG tree enriched by the semantics interface
`SemTagNode`	a TAG node enriched by the semantics interface
`SemTagTreeFamily`	a family of `SemTagTrees`
`SemTagGrammar`	a semantics-enabled TAG grammar

Table 6.3: Classes of the `de.dfki.smartkom.generator.semantics` package.

`SemTag`-classes extend their respective basic classes in the `tag` package. The extensions ensure the correct function of the read-in and printing methods. Furthermore, the methods for substitution and adjunction are overridden in order to also compute the correct semantics of a phrase.

The UML class diagram showing the most important links between classes in the `de.dfki.smartkom.generator.semantics` and `de.dfki.smartkom.generator.realizer` packages can be seen in figure 6.3.

The `SemTagTree` class is extended from `TagTree` to include a `PropositionSet` representing the MRS structure that the leaves of the tree express. Each `SemTagNode` contains a reference to an ordinary semantic argument. The function ϕ is also represented explicitly, and contained in the TAG

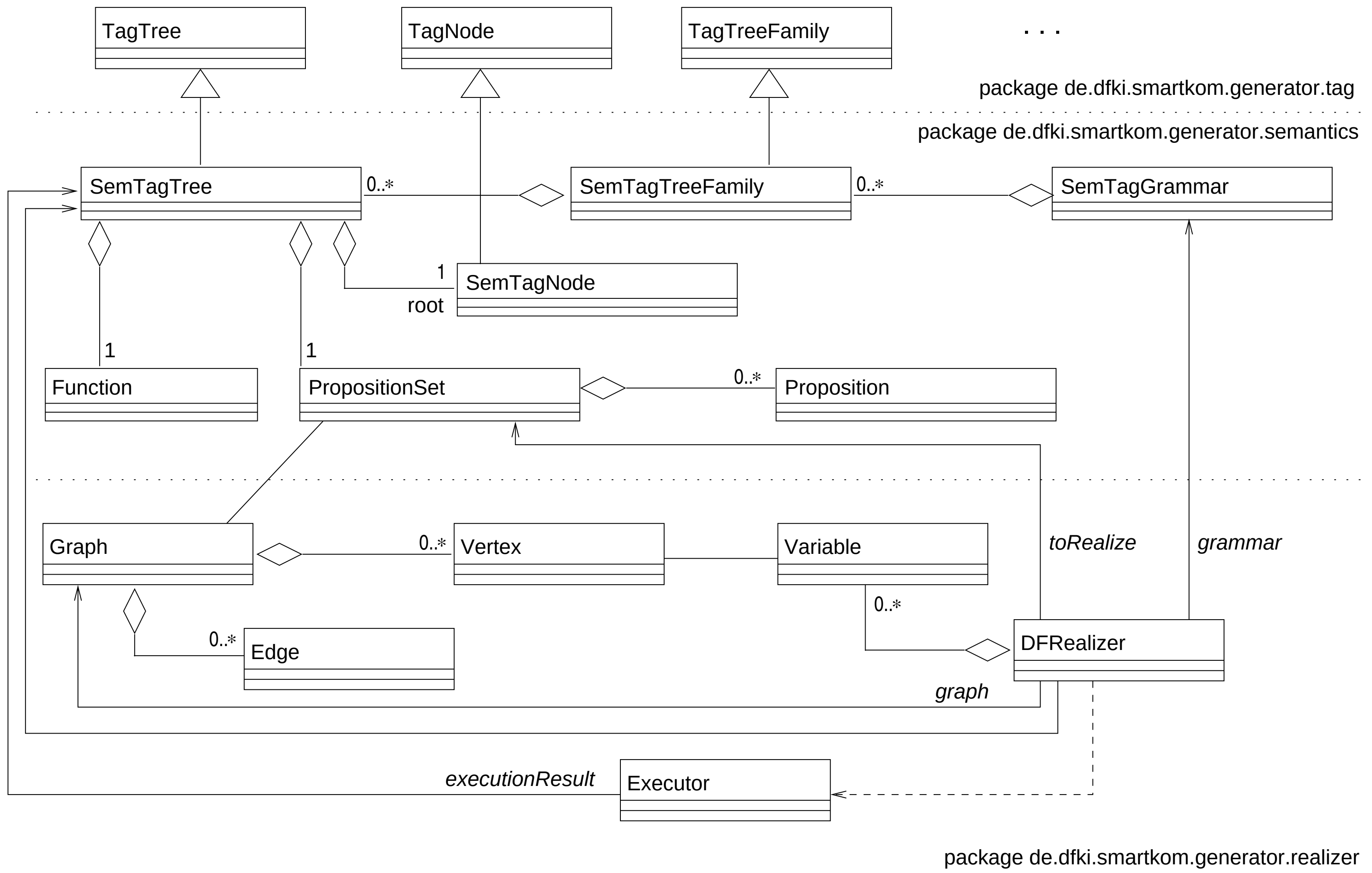


Figure 6.3: UML class diagram of the most important classes.

tree.

6.2.1 Computation of the semantic content of phrases

The definitions given in chapter 3 are very straightforwardly implemented. First, the conventional substitution or adjunction is carried out between the two trees. If this succceeds, the semantic content of the phrase needs to be computed.

Substitution For substitution of a tree t_2 into a tree t_1 at node n, the semantic variables occuring in t_2 are first consistently renamed in order to let the two functions ϕ_1 and ϕ_2 have different ranges, as is required in the definition.

Then, the semantic arguments of the root node of t_2 and the substitution site are equated: $\phi_2(\mathit{root}(\tau_2)) := \phi_1(n)$. This equation has to be stored in the semantics σ_2, the function ϕ_2 and of course on the tree nodes itself. As the handles are not affected by substitution, these are the only variable unifications that need to be carried out.

Finally, the set union of the two semantic representations is constructed and set as the semantic content of the resulting tree. The functions are also merged.

Non-scopal adjunction Adjunction of a tree t_2 into a tree t_1 at the node n also requires the renaming of all semantic variables (and handles) in t_2. If σ_2 does not have a scopal argument, the variable associated with the root and foot of τ_2 is unified with the variable associated with n: $\phi_2(\mathit{root}(\tau_2)) := \phi_1(n)$. Furthermore, the two top handles of σ_1 and σ_2 need to be equated.

In analogy to the substitution case, the two `PropositionSet`s representing the MRS structures are unified, and the functions merged.

Scopal adjunction In scopal adjunction, the semantics σ_2 associated with the adjoining tree has exactly one prominent scopal (handle) argument that needs to be bound. (Note that as explained in chapter 3 I do not yet treat quantifier scope, and the second handle argument of quantifiers is always left unconstrained.)

This handle argument is the current top handle of the outer tree. Such an approach ensures that subsequent adjunctions into the same tree will scope over one another according to their adjunction order. This treatment of scope without underspecification was put forward in (Joshi and Vijay-Shanker, 1999) and is thus relatively straightforwardly implemented.

6.3 Realizer architecture

The syntactic realizer is implemented in the `de.dfki.smartkom.generator.realizer` package, whose classes are listed in table 6.4.

6.3.1 Data structures

The `Graph` class for semantic graphs, along with its dependents `Vertex` and `Edge`, is straightforwardly implemented after the generic graph data structure in

class	description
`Graph`	a generic graph class
`Vertex`	vertices of a generic graph, containing some arbitrary piece of data
`Edge`	edges of a generic graph, containing arbitrary data
`DFRealizer`	the depth-first realizer
`Variable`	a variable of the realization problem associated with a semantic literal
`Executor`	executor for the realization script returned by the `DFRealizer`
`TreeHandle`	a handle associated with a semantic literal, returning trees expressing this literal one by one

Table 6.4: Classes of the `de.dfki.smartkom.generator.realizer` package.

(Goodrich and Tamassia, 1997). It provides the infrastructure for the semantic graphs with undirected edges on which the realizer operates. The vertices have a data field containing the semantic `Proposition` they are constructed for.

The variables of the realization problem are represented as instances of class `Variable`. `Variable` objects have a relatively complex inner structure to keep track of their current status during assignment and re-assignment (triggered by backtracking). Thus, they do not only have fields with references to the inner and outer semantics, as well as inner and outer TAG trees they are assigned, but also references to all available outer propositions, and all inner and outer trees that are still possible for subsequent assignments. The possible trees are stored in `TreeHandle`s.

The `TreeHandle` class implements an iterator for TAG trees that encapsules the mechanism for choosing the next best elementary expressing a certain semantics. In the current implementation, the trees are just returned in an arbitrary order. It is easily imaginable, however, that this class' functionality is extended to provide trees in an order specified by context, by pragmatic features, or other heuristics.

6.3.2 Depth-first algorithm

The depth-first realization algorithm is implemented in the `DFRealizer` class. It is started by a call to `realize(PropositionSet p)`, the only `public` method in this class.

First, the method `createGraph()` in the class `semantics.PropositionSet` is called to compute the appropriate semantic graph for the input MRS structure. This method explicitely constructs a new `realizer.Graph` object with the nodes and edges as specified in definition 6.

Then, the method `setVariables(Graph g)` is called, that starts the depth-first traversal of the semantic graph and the setting of the variables. A method `getStartNode()` in the `Graph` class determines at which vertex the traversal should be started. It currently just arbitrarily picks the first vertex in the vertices list, but this could be improved by some appropriate heuristics.[4]

The search process is then started in `DFRealizer` by calling the recursive method `setVariable(Vertex v)` with the computed start node. `setVariable` first creates a new `Variable` for the `Vertex` it was called on, unless the method was called in a subsequent (backtracking) step and there already exists a variable for this vertex. Then, on the variable *var* the method `findValidAssignment()` in the `Variable` class is called.

Backtracking In case of a clash, backtracking has to be started at the last visited variable. The variables are therefore kept in a list, in which backtracking just retracts by the index.

When a variable is successfully assigned a new value, this is stored in a `HashMap` containing all variables. Then, the search proceeds to the next variable on the backtracking list. If there is no such variable, the method `findNewVertex()` is called to find a new vertex in the semantic graph on which `setVariable()` can be called. The vertices are returned in depth-first order.

The backtracking also keeps track of which propositions have already been realized by the partial syntactic structures built at each stage. For this, all proposition sets belonging to TAG trees that are chosen as inner trees during the realization process are collected.

Variable assignment If the semantic content associated with this variable is already realized, the variable gets a trivial assignment and is skipped. Otherwise, the algorithm first attempts to make this variable the head of the sentence, in case there has not been chosen a head before, and this variable has not been tested as the head in previous steps.

If a trivial assignment is not possible, the complex loop sketched in algorithm 2 is started. `TreeHandle`s stored in the variable keep track of which trees have already been tried in previous attempts for a semantic proposition.

In order to choose the operation, once the participating trees have been determined, a complete search is not necessary: the type of the operation is uniquely determined by the type of the inner tree, i.e. whether it is an auxiliary tree (adjunction) or an initial tree (substitution). The possible goal nodes are also quite restricted: For substitution, only the substitution node which is associated with the appropriate semantic argument is considered. For adjunction, all nodes on the trunk need to be tested (because all of them yield the correct semantics); wrong adjunctions will be filtered out by failing feature unifications.

Execution Once all semantic literals are realized (and no further vertex can be found in the semantic graph), the execution of the collected operations (stored

[4]For example, a heuristic could determine which of the semantic vertices is likely to be expressed as the head, in order to enable top-down realization. A possible criterion would be the number of semantic argument variables (as opposed to handle variables) the literal takes, or generally the degree of the vertex.

in the assigned variables) is attempted. As the adjunction order is not directly implemented as a parameter of the variables, the operations have to be ordered in a post-processing step. The ordering happens with the following heuristics:

- A strict bottom-up derivation is produced by ordering all operations that specify a TAG tree t as the outer tree *before* the operation that uses this tree as the inner tree.
- Adjunctions are ordered according to their scoping: scoped-over adjoining semantic literals are ordered before all those literals that scope over them.

Finally, an `Executor` is created that takes the ordered derivation script and actually carries out all operations. The resulting TAG tree is then finalized and returned to the `DFRealizer` as a result. In case a clash occurs during execution (because all operations have up to this point only been tested *locally*, i.e. one at a time), or in case all possible realizations are requested to be found, not only the first one, backtracking is triggered in the `DFRealizer`, starting from the last visited variable.

Memoization The implementation uses memoization (see Norvig, 1992) in order to avoid spurious computation steps. Memoization is currently used for the construction of `TreeHandle`s, which requires a search through the complete `TagGrammar` to find all the trees that express a certain semantics. These trees are then stored in a `HashMap` to make them available for further steps.

Memoization of the results of TAG operation computations is currently not implemented. This is a possible addition that could improve the efficiency of the syntactic realizer.

Another important memoization effect would be possible for the checking of the consistency constraint (3), of using one and the same tree for a semantic literal during the derivation. This can be done by storing the trees that were chosen for a literal during the assignments, and deterministically choosing the same tree (without the construction of a TreeHandle for all possible trees) if the same literal takes part in another operation. This treatment interacts quite heavily with the backtracking, though, as chosen trees have to be unset in case of reassignment. It is therefore not implemented in the current version.

Parameters Memoization can be turned on and off by setting the field `memoize` to `true` and `false`, respectively.

Furthermore, the realizer can be tuned to either deliver only one possible realization of a given input, or return all realizations that are possible. This is done by setting the field `allRealizations` to `false` or `true`.

6.4 Examples

This section presents a detailed example from the running system. The grammar that was used to produce the examples had to be written by hand, as no TAG grammar with semantics interface previously existed. All the trees of the toy grammar are listed in appendix A. The grammar can also be found in the CVS repository at DFKI, at `SKGEN/etc/templates/semantics-test.xml`. Features were added "as needed," only in order to prevent ungrammatical results. This task was facilitated by the realizer, because the backtracking capability guaranteed the production of additional unwanted realizations if features were missing.

6.4.1 Usage

The realizer must be used through its Java API (see appendices). It can be tested using a test class like the one presented in appendix C. It shows how a grammar file must be parsed and a `SemTagGrammar` constructed, which can then be used to realize the `PropositionSet` in the input file by a `DFRealizer`.

The realization results are returned in a HashSet, and can be printed one by one into files.

6.4.2 Peter loves Mary.

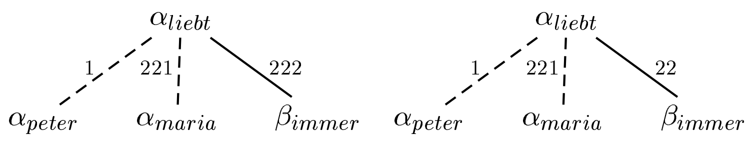
A relatively simple example is:

(6.1)	*Peter*	*liebt*	*Maria*	*immer.*
	Peter	always	Mary	loves.

Peter always loves Mary.

Its input semantics is: $\langle h_0, \{h_0 : \mathit{always}(e, h_1) \wedge h_1 : \mathit{loves}(e, x, y) \wedge h_2 : \mathit{peter}(x) \wedge h_3 : \mathit{maria}(y)\}\rangle$[5]

The generator finds two realizations according to the grammar. Their derivations are shown in figure 6.4. Note that the grammar contains both subject (nominative case) and object (accusative case) trees for "Peter" and "Maria". It



Peter liebt Maria immer. *(Peter loves Mary always.)* Peter liebt immer Maria. *(Peter loves always Mary.)*

Figure 6.4: Derivation trees for "Peter liebt Maria immer."

is the variable assignment algorithm that prevents the wrong substitutions right away, because they would lead to wrong semantics.

However, null adjoining constraints at the root node and second-to-top VP node of the tree for "liebt" (*loves*) prevent the adverb adjoining into them. The realization algorithm tests all these possibilities, and is forced to backtrack.

[5]Semantic variables as well as handle variables are represented by `Integer` objects in the code. They are written as Latin characters resp. h_i here for readability.

6.4.3 SmartKom system example

An example output the SmartKom system is required to produce in the *home* scenario is the information:

(6.2)	*Nachrichten*	*kommen*	*gerade*	*im*	*Ersten.*
	News	run	now	on	first-channel.

News are currently shown on the first (TV) channel.

This would be the appropriate answer for a user request like "Where can I watch the news right now?"

Its semantic representation given as input to the realizer is the following: $\langle h_3, h_1 : run(4,5) \wedge h_2 : news(5) \wedge h_3 : now(4,h_1) \wedge h_6 : loc(4,8) \wedge h_7 : ard(8)\rangle$[6]

Semantic graph creation As a first step, the semantic graph for this input `PropositionSet` is explicitely built using the algorithm defined in definition 6. It is depicted in figure 6.5.

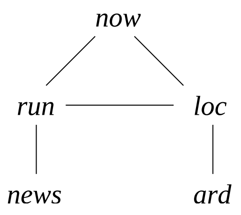


Figure 6.5: Semantic graph for "Nachrichten kommen gerade im Ersten."

Traversal An arbitrary vertex in the semantic graph is chosen as a start vertex. In one test, the system chose the vertex for *now*. The other vertices are found one by one after successful assignments of all previous variables. In the course of the depth-first search, 32 scripts are produced. These scripts are complete assignments that were ordered by the heuristics described on page 66.

Three such scripts (retrieved from a log file of a test run on August 21) are shown in figure 6.6. The variables are printed after the linear ordering of the derivation, one in one line. The first item (e.g., `ard`) is the semantic literal (or vertex in the semantic graph) the variable is associated with. The second item (`i: Nachrichten`) names the chosen inner TAG tree; the third item (`o: anxOKOMMEN`) names the chosen outer TAG tree. Fourth comes the type of operation (`SUBST`itution). The last item (`anxOKOMMEN1`) is the node identifier in the outer tree, at which the operation should be carried out.

Trivially assigned variables do not have any trees or operations assigned. The ordering heuristics always sorts them to the end of the script.

Execution The executor is called on realization scripts. There are three possible results of the execution:

[6]*ARD* is the name of the first public TV channel in Germany.

```
1. < ard ; i: Ersten ; o: auf ; SUBST ; auf12 >
   < loc ; i: auf ; o: gerade ; ADJ ; gerade >
   < now ; i: gerade ; o: anx0KOMMEN ; ADJ ; anx0KOMMEN22 >
   < run ; i: laufenden ; o: Nachrichten ; ADJ ; Nachrichten1 >
   < news -- trivial >

2. < news ; i: Nachrichten ; o: anx0KOMMEN ; SUBST ; anx0KOMMEN1 >
   < ard ; i: Ersten ; o: auf ; SUBST ; auf12 >
   < loc ; i: auf ; o: gerade ; ADJ ; gerade >
   < now ; i: gerade ; o: anx0KOMMEN ; ADJ ; anx0KOMMEN22 >
   < run -- trivial >

3. < news ; i: Nachrichten ; o: anx0KOMMEN ; SUBST ; anx0KOMMEN1 >
   < now ; i: gerade ; o: im ; ADJ ; im >
   < ard ; i: Ersten ; o: im ; SUBST ; im12 >
   < loc ; i: im ; o: anx0KOMMEN ; ADJ ; anx0KOMMEN22 >
   < run -- trivial >
```

Figure 6.6: Example derivation scripts.

1. The consistency constraint (3) is violated. An example is the first script above, where in the variable for `gerade`, the tree `anx0KOMMEN` is chosen for the semantics `run`; but in the variable for `run` itself, the chosen tree is `laufenden`. This is of course impossible, and such a script is rejected by the executor.

2. All operations can be carried out, but the finalization, which is called on the complete derived tree, fails. This reflects feature clashes that can usually only be detected during the unifications of top and bottom structures at each node.

   An example is given with the second script above. The realizer chose the tree `auf` for the semantics `loc`. However, the preposition "auf" (*on*) in German usually requires that its NP argument is determined (i.e., normally it is modified by a determiner "the" or "a"). As the NP argument in the example ("Ersten" = *first programme*) carries a feature $\left[\text{DET } -\right]$, a clash occurs when the top and bottom feature structures of the NP node are attempted to be unified.

3. The execution is successful. This is exemplified by script 3, which produces the sentence from example (6.2).

Result In the example run, there were 26 cases (of the 32 complete scripts) of a violation of the consistency constraint. Three times the finalization failed.[7]

[7]This number is quite small here, because of the few features and the very small size of the grammar.

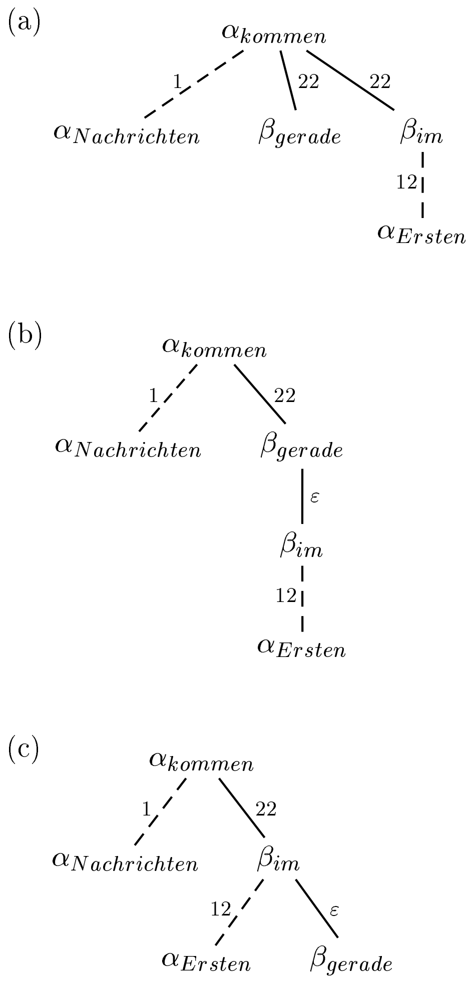
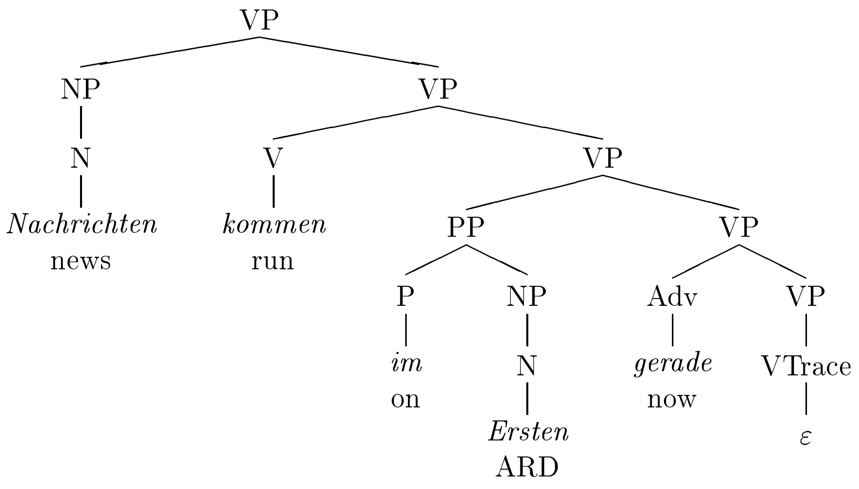


Figure 6.7: Derivation trees for "Nachrichten kommen gerade im Ersten."

Three correct realizations were found. Figure 6.7 shows their derivation trees,[8] and figure 6.8 the derived trees. There are two derivation trees for the sentence "Nachrichten kommen im Ersten gerade," because the grammar does not prohibit embedded adjoinings at the moment. Thus, both the embedded adjoining and the multiple adjoining versions exist.

Derivation trees (a) and (b) yield:



Derivation tree (c) produces the better version:

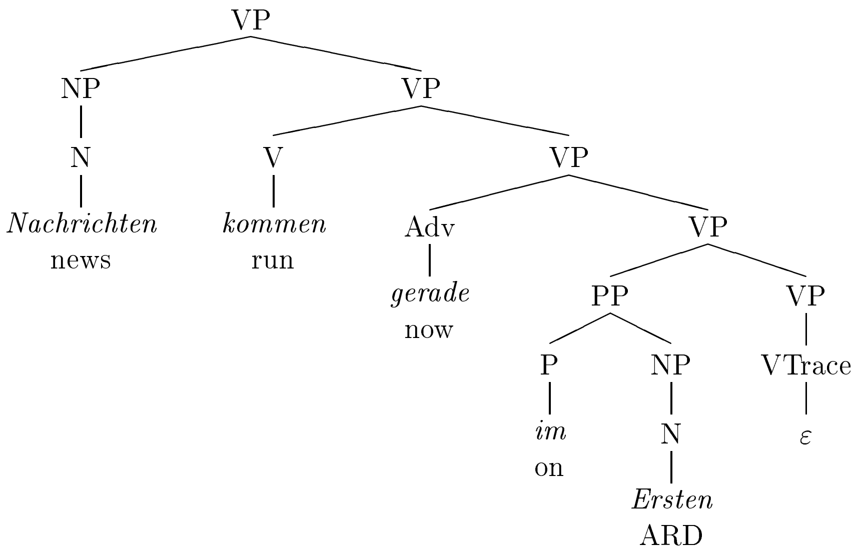


Figure 6.8: Derived trees for "Nachrichten kommen gerade im Ersten."

[8]Adjunctions, if necessary, are ordered left-to-right.

Chapter 7

Conclusion

This thesis has defined a semantics interface for Tree Adjoining Grammars to be used in generation; and presented a syntactic realizer which uses this interface to generate sentences from flat semantic input.

However, several extensions are possible, and remain for further work. I will discuss them at the end of this chapter.

7.1 Contribution of this thesis

In this thesis, I gave criteria for a semantic representation to be used as the input in natural language generation. I showed that Minimal Recursion Semantics fulfills the requirements, being a *flat*, *underspecified* semantic structure. I then defined a semantics interface for Tree Adjoining Grammar that uses a version of MRS as its semantic representation. This interface can correctly deal with (fixed) scope and scope ambiguities through the use of ordered multiple adjunctions. At the moment, it can not handle quantifier scope and underspecification, as dominance constraints on handles and handle argument variables are not implemented yet.

In the second part of the thesis, I reviewed previous approaches to generation with TAG, and found that most of them are not well modularized and/or not easily transferrable to other domains or other languages. I proposed a syntactic realizer that generates TAG structures from flat semantics using a generic depth-first search strategy. To accomplish this, I redefined the realization problem as a constraint satisfaction task, making it accessible for other constraint solving mechanisms.

This solution lacks many of the noted defects: the introduction of a level of semantic representation after the (often domain dependent) abstract generator input modularizes the architecture. The realizer presented here can deal with standard TAG grammars like the English XTAG grammar (of course enhanced with an appropriate semantics interface). It can handle input that is notorously ambiguous not only with respect to word choice in the case of synonyms, but also with respect to different immediate dominance structures (i.e., which of the semantic literals becomes the head) and to syntactic categories. The exhaustive depth-first algorithm guarantees that all realizations of a given input are produced.

7.2 Extensions

The algorithm proposed in this thesis for syntactic realization has the status of a feasibility study. However, the formalization of the problem in the way presented in this chapter as a constraint satisfaction problem makes it open to extensions. I will discuss some of the extensions that seem directly available and how they can be brought forward in this section.

7.2.1 More efficient algorithms

Constraint solving is a broad field, and many proposals exist for efficient constraint solving algorithms. It is one of the big advantages of formalizing a problem as a constraint system, that several solving mechanisms can be used once the formalization is complete. Therefore, the employment of another, more efficient constraint solving algorithm is easily possible in the proposed architecture, and could lead to an increase in system performance in terms of efficiency.

A path worth exploring is the use of constraint propagators instead of just plain search.

7.2.2 Quantifier Scope

The semantics interface in this thesis treats only fixed scope, not the floating scope of quantifiers. However, there is a possibility how quantifier scope could be directly integrated into the chosen architecture of the semantics interface. The only necessary extension would be to allow dominance constraints on handle variables and ensure the correct introduction of such a constraint between the second scopal variable of a quantifier and the label of its appropriate nuclear scope.

This could be done by following the proposal in (Gardent and Kallmeyer, 2003). They associate not only one semantic argument variable with each node in a TAG tree, but also a label (handle). The correct nuclear scope for quantifiers is then provided as the label of the verbal predicate is associated with NP substitution nodes. Thus, this label will eventually be dominated by any quantifiers that these NPs have.

Another possibility is to put the computation into the substitution operation: let an open (floating) scopal argument of a substituting tree dominate the current top handle of the tree into which it substitutes. This treatment, however, seems much more idiosyncratic.

7.2.3 More complex expressions

The introduction of a label index on nodes after (Gardent and Kallmeyer, 2003) would also improve the coverage of the grammar by allowing some operations that would otherwise be hard to control. They give examples of analyses of quite interesting phenomena in English, like control verbs.

However, I believe that the implementation of the mechanism into the feature structures could yield complexity problems, while a realization much like the definition of the semantics interface in this thesis (see chapter 3) seems to be straightforward.

7.2.4 Non-semantic information

One of the original goals of this thesis was to provide a modular description of the realization task as a constraint satisfaction problem, in order to make it possible to take other than strictly semantic information into account during the generation decisions.

In the current architecture, a module choosing contextually appropriate realizations could be easily included as a post-processor, because all the possible realizations are generated. This is not an efficient solution, though.

One point were contextual information can already influence the generation is the TreeHandles. The order in which trees are chosen is encapsuled into a TreeHandle for each semantic literal. The TreeHandle could therefore use information such as sentence mode (e.g., $\left[\textsc{sentMode} \quad \textit{interrogative}\right]$) to prefer certain trees over others (e.g., prefer verb-first trees in German).

For the inclusion of other pragmatic information, a declarative constraint base has to be built that specifies extra-semantic constraints on syntactic realizations, based on some input. For example, the SmartKom generator abstract input representations contain pragmatic information such as the type of the utterance, the user request, topic and focus, and so on. Once the constraints have been formalized, a generic constraint solving algorithm can be employed to make use of them.

I therefore believe that such extensions are relatively easily feasible in the proposed architecture.

Appendix A

List of trees in the toy grammar

Nominal trees

```
<elementaryTree type="initial" name="Peter"> (Subj)
<elementaryTree type="initial" name="Maria"> (Subj)
<elementaryTree type="initial" name="Peter-2"> (Obj)
<elementaryTree type="initial" name="Maria-2"> (Obj)
<elementaryTree type="initial" name="ich">
<elementaryTree type="initial" name="Sie">
<elementaryTree type="initial" name="Ihnen">
<elementaryTree type="initial" name="Uebersicht">
<elementaryTree type="initial" name="Filme">
<elementaryTree type="initial" name="Sendungen">
<elementaryTree type="initial" name="Nachrichten">
<elementaryTree type="initial" name="Ersten">
<elementaryTree type="initial" name="Pro7">
<elementaryTree type="initial" name="Programm">
<elementaryTree type="initial" name="Kinos">
```

Determiner trees

```
<elementaryTree type="auxiliary" name="eine">
<elementaryTree type="auxiliary" name="die">
<elementaryTree type="auxiliary" name="das">
<elementaryTree type="auxiliary" name="der"> (Genitive)
```

Preposition trees

```
<elementaryTree type="auxiliary" name="ueber">
<elementaryTree type="auxiliary" name="im">
<elementaryTree type="auxiliary" name="auf">
```

Adjectival trees

```
<elementaryTree type="auxiliary" name="laufenden">
<elementaryTree type="auxiliary" name="gewuenschten">
```

```
<elementaryTree type="auxiliary" name="Heidelberger">
```

Verbal trees

```
<elementaryTree type="initial" name="anx0LIEBTnx1">
<elementaryTree type="initial" name="anx0ZEIGEnx2nx1">
<elementaryTree type="initial" name="aSEHENnx0nx1">
<elementaryTree type="initial" name="anx0KOMMEN">
```

Adverbal trees

```
<elementaryTree type="auxiliary" name="immer">
<elementaryTree type="auxiliary" name="heute">
<elementaryTree type="auxiliary" name="abend">
<elementaryTree type="auxiliary" name="hier">
<elementaryTree type="auxiliary" name="gerade">
```

Relative pronoun tree

```
<elementaryTree type="auxiliary" name="die">
```

Appendix B

Grammar format

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE schema SYSTEM "http://www.w3.org/1999/XMLSchema.dtd">

<schema
    xmlns="http://www.w3.org/1999/XMLSchema"
    xmlns:xsd="http://www.w3.org/1999/XMLSchema">

<!-- <annotation/> -->

<element
    name="elementaryTrees"
    type="ElementaryTrees">
</element>

<complexType name="ElementaryTrees">
  <element
      name="treeFamily"
      type="TreeFamily"
      minOccurs="0"
      maxOccurs="unbounded"/>
</complexType>

<complexType name="TreeFamily">
  <element
      name="elementaryTree"
      type="ElementaryTree"
      minOccurs="0"
      maxOccurs="unbounded"> <!-- minOccurs="1" ?? -->
  </element>
</complexType>

<complexType name="ElementaryTree">
  <element name="semantics" type="Semantics" maxOccurs="1"/>
  <group ref="NodeAlternatives"/>
  <attribute
      name="comment"
      type="string">
  </attribute>
  <attribute
      name="name"
      type="string">
  </attribute>
  <attribute
      name="type"
      type="string">
  </attribute>
</complexType>

<complexType name="Semantics">
  <attribute
      name="globalHandle"
      type="string"/>
  <element name="predicate" type="Predicate" maxOccurs="unbounded"/>
</complexType>

<complexType name="Predicate">
  <attribute
      name="name"
      type="string"/>
  <attribute
      name="handle"
      type="string"/>
 <!--
  <attribute
      name="arity"
      type="integer"/>
 -->
  <element name="semArg" type="string" minOccurs="0" maxOccurs="unbounded"/>
  <element name="scopeArg" type="string" minOccurs="0" maxOccurs="unbounded"/>
</complexType>

<group name="NodeAlternatives">
  <choice>
    <element
        name="node"
type="RecursiveNode"/>
    <element
        name="node"
type="BaseNode"/>
  </choice>
</group>

<group name="OneFootNode">
  <group
      ref="NodeAlternatives"
      minOccurs="0"
      maxOccurs="unbounded"/>
  <element
      ref="node"
      type="FootNode"
      minOccurs="0"
      maxOccurs="1"/>
  <group
      ref="NodeAlternatives"
      minOccurs="0"
      maxOccurs="unbounded"/>
</group>

<group name="ConstraintAndFeatures">
  <element
      name="constraint"
      type="Constraint"
      minOccurs="0"
      maxOccurs="1">  <!-- minOccurs="0" implies <constraint type="none"/> -->
  </element>
  <element
       name="top"
       type="Top"
       minOccurs="0"
       maxOccurs="1">
  </element>
  <element
      name="bottom"
      type="Bottom"
      minOccurs="0"
      maxOccurs="1">
  </element>
</group>

<complexType name="RecursiveNode">
  <group ref="ConstraintAndFeatures"/>
  <group ref="OneFootNode"/>
  <attributeGroup ref="NodeAttributes"/>
  <attribute
      name="type"
      type="RecursiveNodeType"/>
</complexType>

<complexType name="FootNode">
  <group ref="ConstraintAndFeatures"/>
  <attributeGroup ref="NodeAttributes"/>
  <attribute
      name="type"
      type="FootNodeType"/>
</complexType>

<!-- leaf <element> node == BaseNode -->
<!-- if anchor then a text() node exists -->
<!-- if subst then BaseNode is "empty" -->
<complexType name="BaseNode" base="string" derivedBy="extension">
  <attributeGroup ref="NodeAttributes"/>
  <attribute
      name="type"
      type="BaseNodeType"/>
```

```
</complexType>

<attributeGroup name="NodeAttributes">
  <!-- ID -->
  <attribute
      name="cat"
      sem-arg="string"
      type="string"/>
  <!-- type is different in BaseNode, FootNode, and RecursiveNode -->
</attributeGroup>

<simpleType
    name="RecursiveNodeType"
    base="string">
  <enumeration value="int"/>
</simpleType>

<simpleType
    name="BaseNodeType"
    base="string">
  <enumeration value="anchor"/>
  <enumeration value="head"/> <!-- note that "head" implies "anchor" -->
  <enumeration value="subst"/>
</simpleType>

<simpleType
    name="FootNodeType"
    base="string">
  <enumeration value="foot"/>
</simpleType>

<complexType name="Constraint">
  <element
      name="tree"
      type="TreeReference"
      minOccurs="0"
      maxOccurs="unbounded"/>
  <attribute
      name="type"
      type="ConstraintType"/>
</complexType>

<simpleType
    name="ConstraintType"
    base="string">
  <enumeration value="SA"/> <!-- Selective Adjoining -->
  <enumeration value="NA"/> <!-- Null Adjoining -->
  <enumeration value="OA"/> <!-- Obligatory Adjoining -->
  <enumeration value="none"/> <!-- no constraint -->
</simpleType>

<complexType name="TreeReference">
  <attribute name="name" type="string"/>
```

```
</complexType>

<complexType name="Top">
  <group ref="FVPair" minOccurs="0" maxOccurs="unbounded"/>
</complexType>

<complexType name="Bottom">
  <group ref="FVPair" minOccurs="0" maxOccurs="unbounded"/>
</complexType>

<group name="FVPair">
  <choice>
    <element
        name="fvpair"
        type="RecursiveFVPair"/>
    <element
        name="fvpair"
        type="BaseFVPair"/>
  </choice>
</group>

<complexType name="RecursiveFVPair">
  <attribute name="feature" type="string"/>
  <!-- <attribute name="type" type="string"/> -->
  <!-- NO VALUE -->
  <attribute name="coreference" type="string"/> <!-- integer/ID? -->
  <group ref="FVPair" minOccurs="0" maxOccurs="unbounded"/>
</complexType>

<complexType name="BaseFVPair">
  <attribute name="feature" type="string"/>
  <!-- <attribute name="type" type="string"/> -->
  <attribute name="value" type="string"/>
  <attribute name="coreference" type="string"/> <!-- integer/ID? -->
  <!-- NO RECURSION -->
</complexType>

</schema>
```

Appendix C

RealizeTest class

```
package de.dfki.smartkom.generator.realizer;

import de.dfki.smartkom.generator.semantics.*;
import de.dfki.smartkom.generator.tag.TTools;
import de.dfki.smartkom.generator.templates.*;
import java.io.*;
import java.util.*;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;

public class RealizeTest {

  public static void main(String[] args) {
    // show debug messages.
    TTools.showDebugMessages = true;

    // parse the grammar document
    TemplateParser parser = new TemplateParser();
    try {
      parser.parse("../etc/templates/semantics-test.xml");
    }
    catch (SAXException se) {
      se.printStackTrace();
    }
    catch (IOException ioe) {
      ioe.printStackTrace();
    }
    // construct the grammar
    SemTagGrammar grammar = new SemTagGrammar(parser.getDocument());

    // argument is a filename containing some semantics.
    String filename = args[0];
    try {
      // construct the semantics contained in the file
      parser.parse(filename);
      Element el = (Element) parser.getDocument().
                     getElementsByTagName("semantics").item(0);
      PropositionSet pset = new PropositionSet(el);

      DFRealizer realizer = new DFRealizer(grammar);
      HashSet results = realizer.realize(pset);

      // print the resulting SemTagTrees into files
      Iterator resit = results.iterator();
      int i = 0;
      while ( resit.hasNext() ) {
        i++;
        ((SemTagTree)resit.next()).printToFile("tmp/" + i + "out.xml");
      }
    }
    catch ( Exception e ) { e.printStackTrace(); }
  }
}
```

Appendix D

Java API Class Index

Appendix E

Semantics Class Documentation

E.1 Package de.dfki.smartkom.generator.semantics

Classes

- class **Function**

  This class provides a function implementation.

- class **Function::FuncIterator**

  This class implements an iterator getting keys for values of a **Function** (p. 90).

- class **Proposition**

  This class implements an MRS elementary predication.

- class **PropositionSet**

  This class implements an MRS structure.

- class **PropositionSet::PropositionHandleIterator**
- class **PropositionSet::PropositionTypeIterator**
- class **SemTagGrammar**
- class **SemTagNode**

  A Tag Node implementation that also provides a semantics interface.

- class **SemTagTree**

  A Tag Tree implementation that also provides a semantics interface.

- class **SemTagTreeFamily**

  A tag tree family is a set of trees.

E.2 Function Class Reference

This class provides a function implementation.

Public Methods

- **Function** ()

  Constructor.

- Iterator **getKeysDirect** (Object value)

  Gets all keys mapping to the given value.

- boolean **disjointValues** (Function func)

  Tests whether the values of the two functions are disjoint.

- int **numberOfValues** ()

  Gets the number of disjoint values that this Function maps to.

E.2.1 Detailed Description

This class provides a function implementation.

Definition at line 8 of file Function.java.

E.2.2 Member Function Documentation

boolean Function::disjointValues (Function *func*)

Tests whether the values of the two functions are disjoint.

Parameters:

func The second Function whose values to test.

Returns:

true, if the value sets are disjoint; false otherwise.

Definition at line 28 of file Function.java.

Iterator Function::getKeysDirect (Object *value*)

Gets all keys mapping to the given value.

This implements the Function^-1, returning keys for values.

Parameters:

value the value to which the keys need to be found.

Returns:

an Iterator containing all keys mapping to the specified value.

Definition at line 19 of file Function.java.

int Function::numberOfValues ()

Gets the number of disjoint values that this Function maps to.

Returns:
the size of the set of values.

Definition at line 42 of file Function.java.

E.3 Proposition Class Reference

This class implements an MRS elementary predication.

Public Methods

- String **getName** ()

  Gets the name of this elementary predication.

- Integer **getHandle** ()

  Gets the label of this elementary predication.

- ArrayList **getSemArgs** ()

  Gets all (ordinary) semantic arguments of this elementary predication.

- ArrayList **getScopeArgs** ()

  Gets all scopal arguments of this elementary predication.

- boolean **hasScopeArgs** ()

  Tests whether this elementary predication has a scopal argument.

- **Proposition** (Element predicate)

  Constructor from a parsed XML file.

- boolean **equals** (Object o)

  Tests whether this Proposition and the specified object are equal.

- boolean **translatesTo** (Proposition p, HashMap translateArgs)

  Tests whether this Proposition translates to the specified Proposition, given a translation dictionary.

- void **prettyPrint** (PrintWriter out)

  Prints this proposition human-readably onto the specified PrintWriter.

- String **toString** ()

  Prints this proposition to a string.

E.3.1 Detailed Description

This class implements an MRS elementary predication.

Definition at line 15 of file Proposition.java.

E.3.2 Constructor & Destructor Documentation

Proposition::Proposition (Element *predicate*)

Constructor from a parsed XML file.

This constructs a new Proposition from an Element node contained in a parsed XML file.

Parameters:

predicate the element node representing this proposition.

Definition at line 78 of file Proposition.java.

E.3.3 Member Function Documentation

boolean Proposition::equals (Object *o*)

Tests whether this Proposition and the specified object are equal.

Two propositions are equal, if they have the same name (i.e. represent the same predicate), and their semantic argument and scopal argument lists are both equal (i.e., contain the same argument variables).

Parameters:

o the Object to which to compare this Proposition.

Returns:

true, iff the specified Object is a Proposition, its name is the same as this proposition's name, and the semantic and scopal argument lists are equal; false otherwise.

Definition at line 113 of file Proposition.java.
References getScopeArgs(), and getSemArgs().

Integer Proposition::getHandle ()

Gets the label of this elementary predication.

Returns:

the label of this proposition.

Definition at line 30 of file Proposition.java.
Referenced by PropositionSet::getHandles(), and PropositionSet::setHead().

String Proposition::getName ()

Gets the name of this elementary predication.

This returns the predicate.

Returns:

the predicate of this proposition

Definition at line 23 of file Proposition.java.
Referenced by PropositionSet::containsAllSemantics(), PropositionSet::containsProposition(), PropositionSet::isEqual(), Variable::prettyPrint(), and translatesTo().

ArrayList Proposition::getScopeArgs ()

Gets all scopal arguments of this elementary predication.

Returns:
an ordered ArrayList of all the scopal argument variables of this proposition.

Definition at line 43 of file Proposition.java.
Referenced by equals(), PropositionSet::getHandles(), and translatesTo().

ArrayList Proposition::getSemArgs ()

Gets all (ordinary) semantic arguments of this elementary predication.

Returns:
an ordered ArrayList of all the semantic argument variables of this proposition.

Definition at line 37 of file Proposition.java.
Referenced by PropositionSet::createGraph(), equals(), PropositionSet::removePropsWithSemArg(), and translatesTo().

boolean Proposition::hasScopeArgs ()

Tests whether this elementary predication has a scopal argument.

Returns:
true, if this proposition has at least one scopal argument; false otherwise.

Definition at line 49 of file Proposition.java.

void Proposition::prettyPrint (PrintWriter *out*)

Prints this proposition human-readably onto the specified PrintWriter.
The format looks like: "h0:every(1,h2,h3)".

Parameters:
out the PrintWriter where to print this proposition.

Definition at line 231 of file Proposition.java.
Referenced by toString().

String Proposition::toString ()

Prints this proposition to a string.
Overwrite the generic **toString**() (p. 94) method, and calls **prettyPrint**() (p. 94) instead.

Returns:
a String representing this proposition.

See also:

Proposition::prettyPrint (p. 94)

Definition at line 254 of file Proposition.java.
References prettyPrint().
Referenced by Variable::findAssignment().

boolean Proposition::translatesTo (Proposition *p*, HashMap *translateArgs*)

Tests whether this Proposition translates to the specified Proposition, given a translation dictionary.

If arguments are not included in the dictionary yet, they will be introduced into it with their correspondants in p as the values.

Parameters:

p the Proposition to compare this object to.

translateArgs the translation dictionary.

Returns:

true, if this Proposition can be translated to the given Proposition with respect to the translation dictionary; false otherwise.

Definition at line 135 of file Proposition.java.
References getName(), getScopeArgs(), and getSemArgs().

E.4 PropositionSet Class Reference

This class implements an MRS structure.

Public Methods

- Integer **getGlobalHandle** ()

  Gets the global top handle.

- void **setGlobalHandle** (Integer handle)

  Sets the top handle.

- **PropositionSet** ()

  Constructor.

- **PropositionSet** (Collection c)

  Constructor.

- **PropositionSet** (int i)

  Constructor.

- **PropositionSet** (int i, float f)

  Constructor.

- **PropositionSet** (Element semantics)

  Constructs the PropositionSet from an element node of a parsed XML file.

- void **setUnion** (PropositionSet set2)

  Computes the set union of this PropositionSet with the specified one.

- PropositionSet **setUnionCopy** (PropositionSet set2)

  Computes the set union of this PropositionSet with the specified one.

- void **replaceArg** (Integer oldArg, Integer newArg)

  Replaces the argument variable oldArg in all occurences with newArg.

- void **prettyPrint** (PrintWriter out)

  Prints this PropositionSet human-readably onto the specified PrintWriter.

- boolean **isEqual** (PropositionSet ps)

  Tests whether the two PropositionSets are the same.

- boolean **containsAllSemantics** (PropositionSet ps)

  Tests whether this PropositionSet contains all the semantic literals in the specified set, regardless of arguments.

- boolean **containsProposition** (**Proposition** p)

  Checks whether this set contains the specified **Proposition** (p. 92).

- int **maxNumberOfArgs** ()

  Computes an upper bound to the number of argument variables occuring in this set's propositions.

- Iterator **getPropositionsByName** (String name)

  Gets all propositions in this set that have the specified name.

- Iterator **getPropositionsByHandle** (Integer handle)

  Gets all propositions in this set that have the same label.

- boolean **hasHead** ()

  Tests whether this PropositionSet has a head.

- **Proposition setHead** (**Proposition** headp)

  Set the specified **Proposition** (p. 92) *as the head of this PropositionSet.*

- ArrayList **removeScopalAdjunctions** (Integer handle)

  Removes and returns all scopal predicates that scope over the specified handle.

- PropositionSet **removePropsWithSemArg** (Integer arg)

  Removes and returns all predicates in this set that contain the specified semantic argument variable.

- **Graph createGraph** ()

  Creates a semantic graph from this PropositionSet.

E.4.1 Detailed Description

This class implements an MRS structure.

It consists of elementary predications and a top handle.

Definition at line 13 of file PropositionSet.java.

E.4.2 Constructor & Destructor Documentation

PropositionSet::PropositionSet (Element *semantics*)

Constructs the PropositionSet from an element node of a parsed XML file.

Parameters:

semantics the element node of the XML file representing this proposition set.

Definition at line 42 of file PropositionSet.java.

E.4.3 Member Function Documentation

boolean PropositionSet::containsAllSemantics (PropositionSet *ps*)

Tests whether this PropositionSet contains all the semantic literals in the specified set, regardless of arguments.

Parameters:

ps the PropositionSet to compare this one to.

Returns:

true, if this set contains all the predicates in the specified set; false otherwise.

Definition at line 172 of file PropositionSet.java.
References Proposition::getName(), and getPropositionsByName().

boolean PropositionSet::containsProposition (Proposition *p*)

Checks whether this set contains the specified **Proposition** (p. 92).

The argument variable names can vary between the found proposition and the one specified as the argument, as long as they are consistently translated.

Parameters:

p the **Proposition** (p. 92) to be checked for.

Returns:

true, if this set contains (a translation of) the specified **Proposition** (p. 92); false otherwise.

Definition at line 194 of file PropositionSet.java.
References Proposition::getName(), and getPropositionsByName().

Graph PropositionSet::createGraph ()

Creates a semantic graph from this PropositionSet.

Returns:

the semantic graph associated with this set.

Definition at line 416 of file PropositionSet.java.
References Graph::addEdge(), Graph::addVertex(), and Proposition::getSemArgs().
Referenced by DFRealizer::realize().

Integer PropositionSet::getGlobalHandle ()

Gets the global top handle.

Returns:

the top handle variable.

Definition at line 20 of file PropositionSet.java.
Referenced by isEqual().

Iterator PropositionSet::getPropositionsByHandle (Integer *handle*)

Gets all propositions in this set that have the same label.

Constructs a new Iterator containing all appropriate propositions.

Parameters:

handle the label for which to find the propositions.

Returns:

a new Iterator containing all propositions with the specified label.

Definition at line 302 of file PropositionSet.java.

Iterator PropositionSet::getPropositionsByName (String *name*)

Gets all propositions in this set that have the specified name.

This constructs a new Iterator containing all appropriate predicates.

Parameters:

name the predicate to look for.

Returns:

an Iterator containing all propositions with the specified name.

Definition at line 253 of file PropositionSet.java.

Referenced by containsAllSemantics(), containsProposition(), and isEqual().

boolean PropositionSet::hasHead ()

Tests whether this PropositionSet has a head.

The head is a proposition which is labelled by the top handle.

Parameters:

true if this set has a head; false otherwise.

Definition at line 352 of file PropositionSet.java.

boolean PropositionSet::isEqual (PropositionSet *ps*)

Tests whether the two PropositionSets are the same.

In contrast to the inherited equals() method, the objects contained in the set do not have to be identical. Instead, it is checked whether the predicates contained in the second set are the same as in this set, with the possibility of arguments being consistently translated.

Parameters:

ps the PropositionSet with which to compare this object.

Returns:

true, if the two sets contain the same predicates and are equal under consistent translation of argument variables; false otherwise.

Definition at line 140 of file PropositionSet.java.

References getGlobalHandle(), Proposition::getName(), and getPropositionsByName().

int PropositionSet::maxNumberOfArgs ()

Computes an upper bound to the number of argument variables occuring in this set's propositions.

Returns:
the maximum number of argument variables occuring in this proposition set.

Definition at line 207 of file PropositionSet.java.
Referenced by SemTagTree::SemTagTree().

void PropositionSet::prettyPrint (PrintWriter *out*)

Prints this PropositionSet human-readably onto the specified PrintWriter.
The print format looks like this: "[h0, h0:every(1,h2,h3) & h2:man(1)]".

Parameters:
out the PrintWriter where to print this PropositionSet.

Definition at line 118 of file PropositionSet.java.

PropositionSet PropositionSet::removePropsWithSemArg (Integer *arg*)

Removes and returns all predicates in this set that contain the specified semantic argument variable.

Parameters:
arg the semantic variable for which to find predicates.

Returns:
a PropositionSet of predicates containing the argument variable.

Deprecated:
This method is deprecated and should not be used in Realizers. It might not work properly.

Definition at line 397 of file PropositionSet.java.
References Proposition::getSemArgs(), and PropositionSet().

ArrayList PropositionSet::removeScopalAdjunctions (Integer *handle*)

Removes and returns all scopal predicates that scope over the specified handle.
Find scopal predicates that take handle as argument. Do this recursively (i.e. through handles of such scopal propositions). List the results in order, so that the innermost predicate comes last. Remove the results from this PropositionSet.

Parameters:

handle the handle for which to find scoping predicates.

Returns:

a sorted ArrayList of all predicates scoping over the handle.

Deprecated:

This method is deprecated and should not be used in Realizers. It might not work properly.

Definition at line 380 of file PropositionSet.java.

void PropositionSet::replaceArg (Integer *oldArg*, Integer *newArg*)

Replaces the argument variable oldArg in all occurences with newArg.

All propositions contained in this set are traversed and checked for occurences of oldArg.

Parameters:

oldArg the argument variable to be replaced.

newArg the new argument variable.

Definition at line 89 of file PropositionSet.java.
Referenced by makeDistinctHandles().

void PropositionSet::setGlobalHandle (Integer *handle*)

Sets the top handle.

Parameters:

handle the new top handle.

Definition at line 25 of file PropositionSet.java.
Referenced by setUnionCopy().

Proposition PropositionSet::setHead (Proposition *headp*)

Set the specified **Proposition** (p. 92) as the head of this PropositionSet.

This also sets the top handle as a side effect.

Parameters:

headp the new head **Proposition** (p. 92).

Returns:

the head **Proposition** (p. 92).

Definition at line 362 of file PropositionSet.java.
References Proposition::getHandle().

void PropositionSet::setUnion (PropositionSet *set2*)

Computes the set union of this PropositionSet with the specified one.
Stores all of set2's propositions destructively in this set.

Parameters:
set2 the second PropositionSet with which to compute the set union.

Definition at line 59 of file PropositionSet.java.

PropositionSet PropositionSet::setUnionCopy (PropositionSet *set2*)

Computes the set union of this PropositionSet with the specified one.
Constructs a new PropositionSet and stores the union results into it.

Parameters:
set2 the second PropositionSet with which to compute the set union.

Returns:
the set union of this set and the specified set2.

Definition at line 72 of file PropositionSet.java.
References setGlobalHandle().

E.5 SemTagGrammar Class Reference

This class implements a TAG grammar with semantics interface.

Public Methods

- **SemTagGrammar** ()

  Dummy constructor.

- **SemTagGrammar** (Document doc)

  Constructs a SemTagGrammar from an xml document complying with grammar-sem.xsd.

- TagTree **findTagTree** (String name)

  Finds a **SemTagTree** (p. 108) *in this Grammar by its name.*

- TagTree **findTagTreeCopy** (String name)

  Finds a copy of a TagTree in this Grammar.

- Elements **readFromFile** (String path)

  Reads a new Grammar from a file.

- boolean **printToFile** (String path)

  Prints this grammar to the file specified by its path.

- HashSet **getTagTreesBySemantics** (**Proposition** p)

  Finds tag trees in this grammar by the semantics it expresses.

E.5.1 Detailed Description

This class implements a TAG grammar with semantics interface.

Definition at line 19 of file SemTagGrammar.java.

E.5.2 Constructor & Destructor Documentation

SemTagGrammar::SemTagGrammar (Document *doc*)

Constructs a SemTagGrammar from an xml document complying with grammar-sem.xsd.

Parameters:

doc the document element representing the xml document.

Definition at line 32 of file SemTagGrammar.java.

E.5.3 Member Function Documentation

TagTree SemTagGrammar::findTagTree (String *name*)

Finds a **SemTagTree** (p. 108) in this Grammar by its name.

Parameters:

name the name of the TagTree.

Returns:

the TagTree. if no such TagTree exists in this Grammar, null is returned.

Definition at line 64 of file SemTagGrammar.java.
Referenced by findTagTreeCopy().

TagTree SemTagGrammar::findTagTreeCopy (String *name*)

Finds a copy of a TagTree in this Grammar.
Looks for a tag tree with the specified name and deepclones it.

Parameters:

name The name of the TagTree.

Returns:

A copy of the TagTree. If no such TagTree exists in this Grammar, null is returned.

Definition at line 85 of file SemTagGrammar.java.
References findTagTree().

HashSet SemTagGrammar::getTagTreesBySemantics (Proposition *p*)

Finds tag trees in this grammar by the semantics it expresses.

Parameters:

p the proposition which should be expressed.

Returns:

a hash set of all the trees in this grammar whose semantics contains the proposition p.

Definition at line 227 of file SemTagGrammar.java.
Referenced by TreeHandle::TreeHandle().

boolean SemTagGrammar::printToFile (String *path*)

Prints this grammar to the file specified by its path.

Parameters:

path the path to the file where this grammar should be printed.

Returns:

false, if an I/O error occurs, else true.

Definition at line 196 of file SemTagGrammar.java.
References SemTagTreeFamily::getName().

Elements SemTagGrammar::readFromFile (String *path*)

Reads a new Grammar from a file.

Part of the de.dfki.smartkom.generator.treeEditor.Grammar interface implementation.

Parameters:

path the path to the file specifying a new grammar.

Returns:

the SemTagGrammar specified in the file at location path.

Definition at line 109 of file SemTagGrammar.java.
References SemTagGrammar().

E.6 SemTagNode Class Reference

A Tag Node implementation that also provides a semantics interface.

Public Methods

- Integer **getArg** ()
- **SemTagNode** ()

  Dummy constructor.

- **SemTagNode** (String id, Vector children, FeatureStructure topfeats, FeatureStructure botfeats, String cat, Integer a)

  Constructor.

- void **innerXmlPrint** (PrintWriter out, int indent, Hashtable coReferences)

  Prints the TagNode and its children and FeatureStructures into a PrintWriter.

E.6.1 Detailed Description

A Tag Node implementation that also provides a semantics interface.
Definition at line 16 of file SemTagNode.java.

E.6.2 Constructor & Destructor Documentation

SemTagNode::SemTagNode (String *id*, Vector *children*, FeatureStructure *topfeats*, FeatureStructure *botfeats*, String *cat*, Integer *a*)

Constructor.

Parameters:

id the identifier of this node.

children the Vector of child nodes.

topfeats a top feature structure.

botfeats a bottom feature structure.

cat the category of this node

a the semantic argument associated with this node.

Definition at line 45 of file SemTagNode.java.

E.6.3 Member Function Documentation

Integer SemTagNode::getArg ()

Returns:

The semantic argument that is associated with this node.

Definition at line 24 of file SemTagNode.java.

void SemTagNode::innerXmlPrint (PrintWriter *out*, int *indent*, Hashtable *coReferences*)

Prints the TagNode and its children and FeatureStructures into a PrintWriter.

Output is formatted like the example in grammar.xml.

This method should only be invoked after a call to findCoReferences(new Hashtable()) from the root node of this tree. Otherwise, a null pointer exception will occur.

Parameters:

out the PrintWriter (file) where to write the node

indent the current indentation

coReferences all coReferences in the TagTree this node belongs to

Definition at line 164 of file SemTagNode.java.

E.7 SemTagTree Class Reference

A Tag Tree implementation that also provides a semantics interface.

Public Methods

- **Function getMapping** ()
- **PropositionSet getSemantics** ()
- boolean **hasScopalArgument** ()
- Integer **removeScopalArgument** ()

  Remove the scopal argument.

- **SemTagTree** (String id)

  Constructor of dummy TAG trees.

- **SemTagTree** (DocumentImpl doc)

  Constructor.

- **SemTagTree** (Element elementaryTree)

  Constructs a new SemTagTree from an xml document Element.

- SemTagTree **cloneSTT** ()

  Clone the SemTagTree.

- boolean **substitute** (String nodeID, TagTree tree2) throws Exception

  Substitution of tree2 at nodeID.

- boolean **adjoin** (String nodeID, TagTree tree2) throws Exception

  Adjunction of tree2 at nodeID.

- boolean **adjoinRoot** (String nodeID, TagTree tree2) throws Exception

  Adjunction of tree2 into nodeID, keeping the properties of the adjunction site in the former root node of tree2 for further adjunctions into the same node.

- boolean **adjoinFoot** (String nodeID, TagTree tree2) throws Exception

  Adjunction of tree2 into nodeID, keeping the properties of the adjunction site in the former foot node of tree2 for further adjunctions into the same node.

- void **printToPW** (PrintWriter out)

  Print this tree to a print writer in xml format.

E.7.1 Detailed Description

A Tag Tree implementation that also provides a semantics interface.

Definition at line 29 of file SemTagTree.java.

E.7.2 Constructor & Destructor Documentation

SemTagTree::SemTagTree (Element *elementaryTree*)

Constructs a new SemTagTree from an xml document Element.

The xml schema is grammar-sem.xsd.

Parameters:

elementaryTree The element node representing this elementary tree.

Definition at line 103 of file SemTagTree.java.
References PropositionSet::maxNumberOfArgs().

E.7.3 Member Function Documentation

boolean SemTagTree::adjoin (String *nodeID*, TagTree *tree2*)

Adjunction of tree2 at nodeID.

If tree2 is not a SemTagTree, normal adjunction without semantic composition is executed. Note that the resulting tree (this tree!) is not a well-formed SemTagTree any more, but rather only a TagTree. For this reason, mixing of SemTagTrees and simple TagTrees should be usually avoided.

This implementation tests non-destructively for unification. In case of a success, results are stored destructively into the calling SemTagTree. If unification fails, the SemTagTrees are yet unchanged (they can be used in further operations). However, in the current implementation, if something goes wrong with the semantic composition only, the trees are already changed. This case should not occur usually, though.

Parameters:

nodeID the node identifier of the adjunction node in the calling TagTree

tree2 the TagTree to adjoin

Exceptions:

Exception if this TagTree is final or if adjunction constraints are violated; or if the nodeID is not a valid identifier in this TagTree.

Returns:

true, if the adjunction is successful; false otherwise

Definition at line 235 of file SemTagTree.java.
Referenced by Executor::execute(), and Variable::isValidOperation().

boolean SemTagTree::adjoinFoot (String *nodeID*, TagTree *tree2*)

Adjunction of tree2 into nodeID, keeping the properties of the adjunction site in the former foot node of tree2 for further adjunctions into the same node.

In this version, the identifier and adjunction constraints of the adjunction node are given to the former root node of tree2, so that further adjunctions in the "same" node are possible.

Parameters:

nodeID the adjunction site

tree2 the inner tree

Returns:

true in case of successful adjunction, false otherwise.

See also:

SemTagTree::adjoin (p. 109)

Definition at line 293 of file SemTagTree.java.

boolean SemTagTree::adjoinRoot (String *nodeID*, TagTree *tree2*)

Adjunction of tree2 into nodeID, keeping the properties of the adjunction site in the former root node of tree2 for further adjunctions into the same node.

In this version, the identifier and adjunction constraints of the adjunction node are given to the former root node of tree2, so that further adjunctions in the "same" node are possible.

Parameters:

nodeID the adjunction site

tree2 the inner tree

Returns:

true in case of successful adjunction, false otherwise.

See also:

SemTagTree::adjoin (p. 109)

Definition at line 264 of file SemTagTree.java.

SemTagTree SemTagTree::cloneSTT ()

Clone the SemTagTree.

Constructs a deep clone of the SemTagTree, also cloning all SemTagNodes, and the semantics. This is currently implement by printing the tree and reading it in again, so the working directory should be writable.

Returns:

The cloned SemTagTree.

Definition at line 149 of file SemTagTree.java.
References printToPW(), and SemTagTree().
Referenced by Executor::execute(), and Variable::isValidOperation().

Function SemTagTree::getMapping ()

Returns:

The function phi mapping nodes of the TAG tree onto argument variables in the associated semantics.

Definition at line 40 of file SemTagTree.java.

PropositionSet SemTagTree::getSemantics ()

Returns:
The set of semantic expressions associated with this (partial) TAG tree.

Definition at line 53 of file SemTagTree.java.
Referenced by DFRealizer::setVariable().

boolean SemTagTree::hasScopalArgument ()

Returns:
true, if this TAG tree has a scopal (handle) argument that needs to be filled; i.e. if this is a scopal auxiliary tree. false otherwise.

Definition at line 64 of file SemTagTree.java.

void SemTagTree::printToPW (PrintWriter *out*)

Print this tree to a print writer in xml format.

Prints the SemTagTree according to the xml schema grammar-sem.xsd onto the specified printwriter. This calles recursive methods for printing semantics, nodes and feature structures.

Parameters:
out The PrintWriter where to print me.

Definition at line 411 of file SemTagTree.java.
References PropositionSet::innerXmlPrint().
Referenced by cloneSTT().

Integer SemTagTree::removeScopalArgument ()

Remove the scopal argument.

This method should be called once the scopal argument is filled.

Returns:
The previous scopal argument.

Definition at line 75 of file SemTagTree.java.

boolean SemTagTree::substitute (String *nodeID*, TagTree *tree2*)

Substitution of tree2 at nodeID.

If tree2 is not a SemTagTree, normal substitution without semantic composition is executed. Note that the resulting tree (this tree!) is not a well-formed SemTagTree, but rather only a TagTree.

Parameters:
nodeID The substitution site in this tree.

tree2 The inner TAG tree.

Exceptions:

Exception if the TagTree is final; if the categories of substituted and substituting node do not match; or if the nodeID is not valid (no such node exists).

Returns:

true, if the substitution is successful, false otherwise.

Definition at line 195 of file SemTagTree.java.
Referenced by Executor::execute(), and Variable::isValidOperation().

E.8 SemTagTreeFamily Class Reference

A tag tree family is a set of trees.

Public Methods

- **SemTagTreeFamily** ()

  Dummy Constructor.

- **SemTagTreeFamily** (String n)

  Dummy Constructor.

- **SemTagTreeFamily** (Element elem)

  Constructs a tree family from an element of an xml document.

- String **toString** ()
- TagTree **findTagTree** (TagTree tagTree)

  Find tag tree.

- TagTree **findTagTree** (String name)
- String **getName** ()
- void **setName** (String newName)

E.8.1 Detailed Description

A tag tree family is a set of trees.

These trees are retrievable by their names.

Definition at line 19 of file SemTagTreeFamily.java.

E.8.2 Constructor & Destructor Documentation

SemTagTreeFamily::SemTagTreeFamily (String *n*)

Dummy Constructor.

Parameters:

n the name of the tree family.

Definition at line 33 of file SemTagTreeFamily.java.

SemTagTreeFamily::SemTagTreeFamily (Element *elem*)

Constructs a tree family from an element of an xml document.

This recursively calls the constructor of **SemTagTree** (p. 108).

Parameters:

elem the xml document element node representing this tree family.

Definition at line 42 of file SemTagTreeFamily.java.

E.8.3 Member Function Documentation

TagTree SemTagTreeFamily::findTagTree (TagTree *tagTree*)

Find tag tree.

Parameters:

tagTree the tag tree to find.

Deprecated:

This will most likely not work, use findTagTree(String name) instead, which also makes much more sense.

See also:

SemTagTreeFamily::findTagTree(String name)

Definition at line 67 of file SemTagTreeFamily.java.

String SemTagTreeFamily::toString ()

Returns:

the name of this tree family.

Definition at line 56 of file SemTagTreeFamily.java.

Appendix F

Realizer Class Documentation

F.1 Package de.dfki.smartkom.generator.realizer

Classes

- class **DFRealizer**

  Class implementing a (top down) depth-first realizer for some semantics represented as a **PropositionSet** (p. 96).

- class **Edge**

  This class implements an undirected Edge that can hold some data object.

- class **Executor**

  This class implements an executor of TAG derivation scripts.

- class **Graph**

  This class implements a generic graph data structure, to be used by a syntactic realizer.

- class **TreeHandle**

  A handle class which delivers possible TAG trees for a certain semantics in some order.

- class **Variable**

  This class implements a realization variable.

- class **Vertex**

  This class implements a Vertex that can hold some data object.

F.2 DFRealizer Class Reference

Class implementing a (top down) depth-first realizer for some semantics represented as a **PropositionSet** (p. 96).

Public Methods

- void **setMemoization** (boolean b)

  Sets the memoization parameter.

- void **setGenerateAll** (boolean b)

  Sets the parameter determining whether all possible realizations should be generated instead of just one.

- **DFRealizer** (**SemTagGrammar** g)

  Constructor.

- HashSet **realize** (**PropositionSet** p)

  Realizes the given **PropositionSet** (p. 96).

F.2.1 Detailed Description

Class implementing a (top down) depth-first realizer for some semantics represented as a **PropositionSet** (p. 96).

Definition at line 13 of file DFRealizer.java.

F.2.2 Constructor & Destructor Documentation

DFRealizer::DFRealizer (SemTagGrammar *g*)

Constructor.

Parameters:

g the TAG grammar that this realizer uses.

Definition at line 96 of file DFRealizer.java.
References memoizedTreeSets.

F.2.3 Member Function Documentation

HashSet DFRealizer::realize (PropositionSet *p*)

Realizes the given **PropositionSet** (p. 96).

Parameters:

p the semantics that should be realized.

Returns:

a HashSet containing (all or one) realizations of the given semantics.

Definition at line 107 of file DFRealizer.java.
References PropositionSet::createGraph(), visitedEdges, and visitedVertices.

void DFRealizer::setGenerateAll (boolean *b*)

Sets the parameter determining whether all possible realizations should be generated instead of just one.

Calling this method with argument "false" results in generating only the first possible realization of some input.

Parameters:

b the argument specifying whether all possible realizations should be generated or not.

Definition at line 79 of file DFRealizer.java.

void DFRealizer::setMemoization (boolean *b*)

Sets the memoization parameter.

Parameters:

b the argument specifying whether memoization should be done or not.

Definition at line 55 of file DFRealizer.java.

F.3 Edge Class Reference

This class implements an undirected Edge that can hold some data object.

Public Methods

- **Edge** (**Vertex** v1, **Vertex** v2, Object d)
  Constructs a new undirected edge.
- **Vertex oppositeVertex** (**Vertex** v)
  Gets the **Vertex** (p. 131) *opposite the one specified on this edge.*
- **Vertex**[ ] **getEndPoints** ()
  Gets the end points of this edge.
- boolean **isDirected** ()
  Tests whether this edge is directed.

F.3.1 Detailed Description

This class implements an undirected Edge that can hold some data object.

See also:
Graph (p. 122)

Definition at line 9 of file Edge.java.

F.3.2 Constructor & Destructor Documentation

Edge::Edge (Vertex *v1*, Vertex *v2*, Object *d*)

Constructs a new undirected edge.

Parameters:
v1 the first end point of the new edge.
v2 the second end point of the new edge.
d some data to be associated with the new edge.

Definition at line 24 of file Edge.java.

F.3.3 Member Function Documentation

Vertex [] Edge::getEndPoints ()

Gets the end points of this edge.

Returns:
an array containing the two vertices this edge is incident upon.

Definition at line 53 of file Edge.java.
Referenced by Graph::endVertices().

boolean Edge::isDirected ()

Tests whether this edge is directed.

Returns:

true, if this is a directed edge; false if it is an undirected edge.

Definition at line 62 of file Edge.java.

Vertex Edge::oppositeVertex (Vertex *v*)

Gets the **Vertex** (p. 131) opposite the one specified on this edge.

Parameters:

v the **Vertex** (p. 131) whose opposite should be found.

Returns:

the **Vertex** (p. 131) which is opposite v on this edge; if the specified **Vertex** (p. 131) v is not an end point of this edge, null is returned.

Definition at line 38 of file Edge.java.
Referenced by Vertex::addEdge().

F.4 Executor Class Reference

This class implements an executor of TAG derivation scripts.

Public Methods

- **Executor** (ArrayList s, boolean all, HashSet realizations)
  Constructor.
- boolean **execute** () throws Exception
  Executes a realization script produced by the realizer.
- **SemTagTree getExecutionResult** ()
  Gets the last execution result this Executor produced.

F.4.1 Detailed Description

This class implements an executor of TAG derivation scripts.

I.e., the script represents a derivation tree, of which some operations are ordered.

Definition at line 12 of file Executor.java.

F.4.2 Constructor & Destructor Documentation

Executor::Executor (ArrayList *s*, boolean *all*, HashSet *realizations*)

Constructor.

Parameters:

s the execution script.

all whether all realizations need to be found.

realizations the set of realizations for the semantics.

Definition at line 27 of file Executor.java.

F.4.3 Member Function Documentation

boolean Executor::execute ()

Executes a realization script produced by the realizer.

Exceptions:

Exception if one of the operations in the execution script fails.

Returns:

true, if execution was successful; false otherwise.

Definition at line 40 of file Executor.java.

References SemTagTree::adjoin(), SemTagTree::cloneSTT(), Variable::inner, Variable::innerTree, Variable::nodeId, Variable::operation, Variable::outer, Variable::outerTree, and SemTagTree::substitute().

SemTagTree Executor::getExecutionResult ()

Gets the last execution result this Executor produced.

Returns:

the **SemTagTree** (p. 108) that was produced by the last successful call to **execute**() (p. 120); null, if the last call was unsuccessful.

Definition at line 100 of file Executor.java.

F.5 Graph Class Reference

This class implements a generic graph data structure, to be used by a syntactic realizer.

Public Methods

- **Graph** ()

  Constructor of an empty graph.

- int **numVertices** ()

  Gets the number of Vertices contained in this graph.

- int **numEdges** ()

  Gets the number of Edges contained in this graph.

- Iterator **vertices** ()

  Gets all the Vertices of this graph.

- Iterator **edges** ()

  Gets all the Edges of this graph.

- void **addVertex** (**Vertex** v)

  Adds a **Vertex** (p. 131) *to this graph.*

- **Edge addEdge** (**Vertex** v1, **Vertex** v2, Object d)

  Constructs a new **Edge** (p. 118) *and adds it to this graph.*

- int **degree** (**Vertex** v)

  Gets the degree of the specified **Vertex** (p. 131)*.*

- Iterator **adjacentVertices** (**Vertex** v)

  Gets all vertices adjacent to the specified one.

- Iterator **incidentEdges** (**Vertex** v)

  Gets all edges incident to the specified **Vertex** (p. 131)*.*

- **Vertex**[ ] **endVertices** (**Edge** e)

  Gets the two vertices that are the endpoints of **Edge** (p. 118) *e.*

- boolean **areAdjacent** (**Vertex** v, **Vertex** w)

  Tests whether the two specified vertices are adjacent.

- **Vertex getStartNode** ()

  Gets a start vertex for this graph.

F.5.1 Detailed Description

This class implements a generic graph data structure, to be used by a syntactic realizer.

Definition at line 10 of file Graph.java.

F.5.2 Member Function Documentation

Edge Graph::addEdge (Vertex *v1*, Vertex *v2*, Object *d*)

Constructs a new **Edge** (p. 118) and adds it to this graph.

Parameters:

v1 the first vertex adjacent to the edge.

v2 the second vertex adjacent to the edge.

d some data associated with the edge.

Returns:

the newly constructed edge.

Definition at line 58 of file Graph.java.
References Vertex::addEdge().
Referenced by PropositionSet::createGraph().

void Graph::addVertex (Vertex *v*)

Adds a **Vertex** (p. 131) to this graph.

Parameters:

v the **Vertex** (p. 131) to be added.

Definition at line 48 of file Graph.java.
Referenced by PropositionSet::createGraph().

Iterator Graph::adjacentVertices (Vertex *v*)

Gets all vertices adjacent to the specified one.

Parameters:

v a **Vertex** (p. 131) in this graph.

Returns:

an Iterator containing all the vertices adjacent to v.

Definition at line 77 of file Graph.java.
References Vertex::getAdjacentVertices().

boolean Graph::areAdjacent (Vertex *v*, Vertex *w*)

Tests whether the two specified vertices are adjacent.

Parameters:

v a first **Vertex** (p. 131) in this graph.

w a second **Vertex** (p. 131) in this graph.

Returns:

true, if the two vertices are neighbors; false otherwise.

Definition at line 96 of file Graph.java.
References Vertex::hasNeighbor().

int Graph::degree (Vertex *v*)

Gets the degree of the specified **Vertex** (p. 131).

Parameters:

v a **Vertex** (p. 131) in this graph.

Returns:

the degree of **Vertex** (p. 131) v.

Definition at line 71 of file Graph.java.
References Vertex::degree().

Iterator Graph::edges ()

Gets all the Edges of this graph.

Returns:

an Iterator containing all the edges this graph consists of.

Definition at line 42 of file Graph.java.

Vertex [] Graph::endVertices (Edge *e*)

Gets the two vertices that are the endpoints of **Edge** (p. 118) e.

Parameters:

e an **Edge** (p. 118) in this graph.

Returns:

an array containing the two endpoints of e.

Definition at line 89 of file Graph.java.
References Edge::getEndPoints().

Vertex Graph::getStartNode ()

Gets a start vertex for this graph.

Just picks one **Vertex** (p. 131) arbitrarily.

Returns:

a **Vertex** (p. 131).

Definition at line 104 of file Graph.java.

Iterator Graph::incidentEdges (Vertex *v*)

Gets all edges incident to the specified **Vertex** (p. 131).

Parameters:

v a **Vertex** (p. 131) in this graph.

Returns:

an Iterator of all edges incident to v.

Definition at line 83 of file Graph.java.

References Vertex::getIncidentEdges().

int Graph::numEdges ()

Gets the number of Edges contained in this graph.

Returns:

the number of edges this graph consists of.

Definition at line 32 of file Graph.java.

int Graph::numVertices ()

Gets the number of Vertices contained in this graph.

Returns:

the number of vertices of this graph.

Definition at line 27 of file Graph.java.

Iterator Graph::vertices ()

Gets all the Vertices of this graph.

Returns:

an Iterator containing all the vertices this graph consists of.

Definition at line 37 of file Graph.java.

F.6 TreeHandle Class Reference

A handle class which delivers possible TAG trees for a certain semantics in some order.

Public Methods

- **TreeHandle** (**Proposition** p, **SemTagGrammar** g)

  Constructs a new TreeHandle that handles all trees which are assigned semantics p by grammar g.

- **TreeHandle** (HashSet origSet, Object constraints)

  Constructs a new TreeHandle for the trees listed in origSet.

- boolean **hasNext** ()

  Tests whether there are more trees in this tree handle.

- **SemTagTree next** ()

  Gets the next tree in this handle.

- **SemTagTree current** ()

  Gets the current tree in this handle.

F.6.1 Detailed Description

A handle class which delivers possible TAG trees for a certain semantics in some order.

The specific order is capsuled in this class. In this implementation, the trees are returned in no specific order.

Definition at line 11 of file TreeHandle.java.

F.6.2 Constructor & Destructor Documentation

TreeHandle::TreeHandle (Proposition *p*, SemTagGrammar *g*)

Constructs a new TreeHandle that handles all trees which are assigned semantics p by grammar g.

Parameters:

p the semantic proposition whose trees are to be handled.

g the grammar where to find the trees to be handled.

Definition at line 24 of file TreeHandle.java.

References SemTagGrammar::getTagTreesBySemantics().

TreeHandle::TreeHandle (HashSet *origSet*, Object *constraints*)

Constructs a new TreeHandle for the trees listed in origSet.

Parameters:

origSet the trees that should be handled. The HashSet is copied, so that we don't influence each other.

constraints ignored dummy argument.

Definition at line 35 of file TreeHandle.java.

F.6.3 Member Function Documentation

SemTagTree TreeHandle::current ()

Gets the current tree in this handle.

Returns:

the **SemTagTree** (p. 108) that was returned by the last call to **next**() (p. 127).

See also:

TreeHandle::next() (p. 127)

Definition at line 62 of file TreeHandle.java.

boolean TreeHandle::hasNext ()

Tests whether there are more trees in this tree handle.

Returns:

true, if any more trees are left; false otherwise.

Definition at line 44 of file TreeHandle.java.
Referenced by Variable::findAssignment().

SemTagTree TreeHandle::next ()

Gets the next tree in this handle.

Returns:

the next **SemTagTree** (p. 108).

Definition at line 52 of file TreeHandle.java.
Referenced by Variable::findAssignment().

F.7 Variable Class Reference

This class implements a realization variable.

Public Methods

- **Variable** (**DFRealizer** r, **Vertex** v, HashMap memoizedTreeSets)
  Constructor.
- boolean **isAssigned** ()
  Tests whether this variable has been assigned a value.
- boolean **findAssignment** ()
  Finds a new assignment for the variable.
- boolean **findValidAssignment** ()
  Finds a valid assignment for the variable.
- boolean **isValidOperation** ()
  Tests whether the operation assigned to this variable is valid.
- String **prettyPrint** ()
  Prints this variable to a string.

Static Public Attributes

- final int **OP_SUBST** = 1
  The substitution TAG operation.
- final int **OP_ADJ** = 2
  the adjunction TAG operation.

F.7.1 Detailed Description

This class implements a realization variable.
Definition at line 10 of file Variable.java.

F.7.2 Constructor & Destructor Documentation

Variable::Variable (DFRealizer *r*, Vertex, HashMap *memoizedTreeSets*)

Constructor.

Parameters:
r the **DFRealizer** (p. 116) this Variable belongs to.

v the **Vertex** (p. 131) in the semantic graph this variable is associated with.

memoizedTreeSets a map of literals to memoized tree sets for faster finding of trees.

Definition at line 64 of file Variable.java.
References Vertex::degree(), Vertex::getAdjacentVertices(), and Vertex::getData().

F.7.3 Member Function Documentation

boolean Variable::findAssignment ()

Finds a new assignment for the variable.
The new assignment is destructively stored in this variable.

Returns:
true, if a new assignment could be found; false otherwise.

Definition at line 172 of file Variable.java.
References Vertex::getData(), TreeHandle::hasNext(), TreeHandle::next(), and Proposition::toString().
Referenced by findValidAssignment().

boolean Variable::findValidAssignment ()

Finds a valid assignment for the variable.
This method also checks for the possibility of trivial assignments.

Returns:
true, if a valid assignment could be found; false otherwise.

Definition at line 403 of file Variable.java.
References findAssignment(), and isValidOperation().
Referenced by DFRealizer::setVariable().

boolean Variable::isAssigned ()

Tests whether this variable has been assigned a value.

Returns:
true, if this variable was assigned a value; false otherwise.

Definition at line 81 of file Variable.java.
References OP_ADJ, and OP_SUBST.
Referenced by isValidOperation(), and prettyPrint().

boolean Variable::isValidOperation ()

Tests whether the operation assigned to this variable is valid.

This performs only local tests.

Returns:

true, if the operation is valid; false if this variable hasn't been assigned a value or the operation is invalid.

Definition at line 428 of file Variable.java.

References SemTagTree::adjoin(), SemTagTree::cloneSTT(), isAssigned(), OP_ADJ, OP_SUBST, and SemTagTree::substitute().

Referenced by findValidAssignment().

String Variable::prettyPrint ()

Prints this variable to a string.

The format is: < loves – trivial > or < every ; i:every-1 ; o:man-1 ; ADJ ; man-1-1 >

Returns:

the String containing a printed version of this variable.

Definition at line 474 of file Variable.java.

References Proposition::getName(), and isAssigned().

F.8 Vertex Class Reference

This class implements a Vertex that can hold some data object.

Public Methods

- Object **getData** ()

  Gets the data object contained in this Vertex.

- **Vertex** (String id, Object d)

  Constructs a new Vertex with the specified id and data object.

- void **addEdge** (**Edge** e)

  Adds an **Edge** (p. 118) *incident to this Vertex.*

- int **degree** ()

  Gets the degree of this Vertex.

- Iterator **getAdjacentVertices** ()

  Gets the vertices adjacent to this one.

- Iterator **getIncidentEdges** ()

  Gets the edges incident on this Vertex.

- ArrayList **getEdgeList** ()

  Gets the edges incident on this Vertex as a list.

- boolean **hasNeighbor** (Vertex w)

  Tests whether the specified Vertex w is a neighbor of this Vertex.

F.8.1 Detailed Description

This class implements a Vertex that can hold some data object.

See also:
Graph (p. 122)

Definition at line 10 of file Vertex.java.

F.8.2 Constructor & Destructor Documentation

Vertex::Vertex (String *id*, Object *d*)

Constructs a new Vertex with the specified id and data object.

Parameters:
id the identifier of this Vertex.

d some data to be associated with this Vertex.

Definition at line 29 of file Vertex.java.

F.8.3 Member Function Documentation

void Vertex::addEdge (Edge *e*)

Adds an **Edge** (p. 118) incident to this Vertex.

Parameters:

e the new incident edge.

Definition at line 39 of file Vertex.java.
References Edge::oppositeVertex().
Referenced by Graph::addEdge().

int Vertex::degree ()

Gets the degree of this Vertex.

Returns:

the degree of this Vertex.

Definition at line 48 of file Vertex.java.
Referenced by Graph::degree(), and Variable::Variable().

Iterator Vertex::getAdjacentVertices ()

Gets the vertices adjacent to this one.

Returns:

an Iterator containing all neighbors of this Vertex.

Definition at line 55 of file Vertex.java.
Referenced by Graph::adjacentVertices(), and Variable::Variable().

Object Vertex::getData ()

Gets the data object contained in this Vertex.

Returns:

the data associated with this Vertex.

Definition at line 22 of file Vertex.java.
Referenced by Variable::findAssignment(), DFRealizer::setVariable(), and Variable::Variable().

ArrayList Vertex::getEdgeList ()

Gets the edges incident on this Vertex as a list.

Returns:
an ArrayList containing all incident edges.

Definition at line 70 of file Vertex.java.

Iterator Vertex::getIncidentEdges ()

Gets the edges incident on this Vertex.

Returns:
an Iterator containing all incident edges.

Definition at line 62 of file Vertex.java.
Referenced by Graph::incidentEdges().

boolean Vertex::hasNeighbor (Vertex *w*)

Tests whether the specified Vertex w is a neighbor of this Vertex.

Parameters:
w the Vertex to test for adjacency.

Returns:
true, if the specified Vertex is adjacent; false otherwise.

Definition at line 79 of file Vertex.java.
Referenced by Graph::areAdjacent().

Bibliography

Anne Abeillé and Marie-Hélène Candito. FTAG: A lexicalized Tree Adjoining Grammar for French. In Anne Abeillé and Owen Rambow, editors, *Tree Adjoining Grammars: Formalisms, Linguistic Analysis, and Processing*, pages 305–330. CSLI Publications, Stanford, CA, 2000.

Jon Barwise and John Perry. Semantic innocence and uncompromising situations. *Midwest Studies in the Philosophy of Language*, VI, 1981.

Jon Barwise and John Perry. *Situations and Attitudes.* MIT Press (Bradford Books), Cambridge, 1983.

Tilman Becker, Wolfgang Finkler, Anne Kilger, and Peter Poller. An efficient kernel for multilingual generation in speech-to-speech diaglogue translation. In *Proceedings of the 38th ACL and 17th Coling*, Montreal, Canada, 1998.

Tilman Becker, Anne Kilger, Patrice Lopez, and Peter Poller. The Verbmobil generation component VM-GECO. In Wolfgang Wahlster, editor, *Verbmobil: Foundations of Speech-to-Speech Translation.* Springer, Berlin, 2000.

Johan Bos. Predicate logic unplugged. In *Proceedings of the 10th Amsterdam Colloquium*, Amsterdam, NL, 1995.

Stephan Busemann. Best-first surface realization. In *Proceedings of the 8th INLG Workshop*, Sussex, UK, 1996.

Ann Copestake, Dan Flickinger, Ivan A. Sag, and Carl Pollard. Minimal Recursion Semantics. An Introduction. Draft, Stanford University, 1999.

Ann Copestake, Alex Lascarides, and Dan Flickinger. An algebra for semantic construction in constraint-based grammars. In *Proceedings of the 39th ACL*, pages 132–139, Toulouse, France, 2001.

Cassandre Creswell and Owen Rambow. English dependency treebank coding manual. Available online at: `http://www.cis.upenn.edu/~creswell/dependency/ecm.html`, 23.3.2003.

Mary Dalrymple, editor. *Semantics and Syntax in Lexical Functional Grammar.* MIT Press, 1999.

Mary Dalrymple. *Lexical-Functional Grammar*, volume 34 of *Syntax and Semantics.* Academic Press, 2001.

Mary Dalrymple, John Lamping, Fernando Pereira, and Vijay Saraswat. Linear logic for meaning assembly. In *Proceedings of Computational Logic for Natural Language Processing*, Edinburgh, 1995.

Laurence Danlos. G-TAG: A lexicalized formalism for text generation inspired by Tree Adjoining Grammar. In Anne Abeillé and Owen Rambow, editors, *Tree Adjoining Grammars: Formalisms, Linguistic Analysis, and Processing.* CSLI Publications, Stanford, CA, 2000.

Ralph Debusmann. A declarative grammar formalism for dependency grammar. Master's thesis, Computational Linguistics, Universität des Saarlandes, Germany, 2001.

Christine Doran, Beth Ann Hockey, Anoop Sarkar, B. Srinivas, and Fei Xia. Evolution of the XTAG system. In Anne Abeillé and Owen Rambow, editors, *Tree Adjoining Grammars: Formal, Computational and Linguistic Aspects.* CSLI, Stanford, 2000.

Denys Duchier and Ralph Debusmann. Topological dependency trees: A constraint-based account of linear precedence. In *Proceedings of the 39th ACL*, Toulouse, France, 2001.

Anette Frank and Josef van Genabith. Linear logic based semantics for LTAG – and what it teaches us about LFG and TAG. In Miriam Butt and Tracy Holloway King, editors, *Proceedings of LFG'01*, University of Hong Kong, 2001.

Robert Frank. *Syntactic locality and tree adjoining grammar: grammatical acquisition and processing perspectives.* PhD thesis, University of Pennsylvania, 1992.

Claire Gardent and Laura Kallmeyer. Semantic construction in Feature-Based TAG. In *Proceedings of the 10th EACL*, Budapest, Hungary, 2003.

Michael T. Goodrich and Roberto Tamassia. *Data structures and algorithms in Java.* Worldwide series in computer science. John Wiley & Sons, Inc., 1997.

Saul Gorn. Explicit definitions and linguistic dominoes. In J. Hart and S. Takasu, editors, *Systems and Computer Science*, pages 77–115. University of Toronto Press, Toronto, CA, 1967.

Aravind K. Joshi. The relevance of Tree Adjoining Grammar to generation. In G. Kempen, editor, *Natural Language Generation: New Directions in Artificial Intelligence, Psychology, and Linguistics.* Dordrecht: Kluwer, 1987.

Aravind K. Joshi, Laura Kallmeyer, and Maribel Romero. Flexible composition in LTAG, quantifier scope and inverse linking. In *Proceedings of the 5th IWCS*, pages 179–194, Tilburg, NL, 2003.

Aravind K. Joshi and Yves Schabes. Tree-Adjoining Grammars. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages. Volume 3: Beyond Words*, pages 69–123. Springer, 1997.

Aravind K. Joshi and K. Vijay-Shanker. Compositional semantics with Lexicalized Tree Adjoining Grammar (LTAG): How much underspecification is necessary? In *Proceedings of the 3rd IWCS*, pages 131–145, Tilburg, NL, 1999.

Laura Kallmeyer. Enriching the TAG derivation tree for semantics. In *6. Konferenz zur Verarbeitung natürlicher Sprache. Proceedings*, DFKI GmbH, Saarbrücken, Germany, 2002a.

Laura Kallmeyer. Using an enriched TAG derivation structure as basis for semantics. In *Proceedings of TAG+6*, pages 127–136, Venice, Italy, 2002b.

Laura Kallmeyer and Aravind K. Joshi. Factoring predicate argument and scope semantics: Underspecified semantics with LTAG. In *Proceedings of the 12th Amsterdam Colloquium*, pages 169–174, Amsterdam, NL, 1999.

Martin Kay. Chart generation. In Aravind Joshi and Martha Palmer, editors, *Proceedings of the 34th ACL*, pages 200–204, San Francisco, 1996.

Anne Kilger and Wolfgang Finkler. Incremental generation for real-time applications. Technical Report RR-95-11, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH, 1995.

Alexander Koller and Kristina Striegnitz. Generation as dependency parsing. In *Proceedings of the 40th ACL*, University of Pennsylvania, Philadelphia, 2002.

Raymond Kozlowski. DSG/TAG: An appropriate grammatical formalism for flexible sentence generation. In *Proceedings of the 40th ACL, student session*, University of Pennsylvania, Philadelphia, 2002.

Anthony Kroch and Aravind K. Joshi. The linguistic relevance of Tree Adjoining Grammar. Technical Report MS-CS-85-16, Department of Computer and Information Sciences, University of Pennsylvania, 1985.

Markus Löckelt. Liliput – ein parametrisierbarer Constraint-Solver für endliche Wertebereiche und sein Einsatz in der Generierung natürlicher Sprache. Master's thesis, Computer Science, Universität des Saarlandes, Germany, 2000.

Kathleen F. McCoy, K. Vijay-Shanker, and Gijoo Yang. A functional approach to generation with TAG. In *Proceedings of the 30th ACL*, pages 48–55, University of Delaware, 1992.

David D. McDonald and James D. Pustejovsky. TAG's as a grammatical formalism for generation. In *Proceedings of the 23rd ACL*, pages 94–103, University of Chicago, IL, 1985.

M. Meteer et al. Mumble-86: Design and implementation. Technical report, University of Massachusetts, 1987. COINS Technical Report 87-87a.

Stefan Müller. *Deutsche Syntax deklarativ. Head-Driven Phrase Structure Grammar für das Deutsche.* Number 394 in Linguistische Arbeiten. Max Niemeyer Verlag, Tübingen, 1999.

Peter Norvig. *Paradigms of AI Programming: Case Studies in Common LISP*. Morgan Kaufmann, 1992.

Carlos Prolo. Systematic grammar development in the XTAG project. In *Proceedings of the TAG+6 Workshop*, Venice, Italy, 2002.

Owen Rambow, Cassandre Creswell, Rachel Szekely, Harriet Tauber, and Marilyn Walker. A dependency treebank for english. In *ISLE Workshop on Dialogue Tagging for Multi-Modal Human Computer Interaction*, University of Edinburgh, UK, 2002.

Ehud Reiter and Robert Dale. Building applied natural language generation systems. *Natural Language Engineering*, 3(1):57–88, 1997.

Yves Schabes and Stuart M. Shieber. An alternative conception of tree-adjoining derivation. *Computational Linguistics*, 20(1):91–124, 1994.

Antje Schweitzer, Norbert Braunschweiler, and Edmilson Morais. Prosody generation in the SmartKom project. In Bernard Bel and Isabel Marlien, editors, *Proceedings of Speech Prosody 2002*, pages 639–642, Aix-en-Provence, France, 2002.

Stuart M. Shieber. The problem of logical-form equivalence. *Computational Linguistics*, 19(1):179–190, 1994.

Stuart M. Shieber, Gertjan van Noord, Robert C. Moore, and Fernando C. N. Pereira. A semantic-head-driven generation algorithm for unification-based formalisms. In *Proceedings of the 27th ACL*, pages 7–17, Vancouver, Canada, 1989.

Matthew Stone and Christine Doran. Paying heed to collocations. In *Proceedings of the 8th INLG Workshop*, pages 91–100, Sussex, UK, 1996.

Matthew Stone and Christine Doran. Sentence planning as description using Tree Adjoining Grammar. In *Proceedings of the 35th ACL*, pages 198–205, Madrid, Spain, 1997.

Matthew Stone, Christine Doran, Bonnie Webber, Tonia Bleam, and Martha Palmer. Microplanning with communicative intentions: The SPUD system. Available online at `http://www.cs.rutgers.edu/~mdstone/pubs/spud.pdf`, 2001.

Kristina Striegnitz. Pragmatic constraints and contextual reasoning in natural language generation: a system description. Report; Available online at `http://www.coli.uni-sb.de/cl/projects/indigen/papers/bulzoni.ps.gz`, 2000.

Kristina Striegnitz. A chart-based generation algorithm for LTAG with pragmatic constraints. Report; Available online at `http://www.coli.uni-sb.de/~kris/papers/chart.ps.gz`, 2001.

Dimitri van Heesch. Doxygen documentation system, 1997. Available online at: `http://www.stack.nl/~dimitri/doxygen/download.html`.

K. Vijay-Shanker and Aravind K. Joshi. Feature structures based Tree Adjoining Grammars. In *Proceedings of the 12th Coling*, Budapest, Hungary, 1988.

K. Vijayashanker. *A Study of Tree Adjoining Grammars.* PhD thesis, University of Pennsylvania, Philadelphia, PA, 1987.

Wolfgang Wahlster, editor. *Verbmobil: Foundations of Speech-to-Speech Translation.* Springer, Berlin, 2000.

Wolfgang Wahlster, Norbert Reithinger, and Anselm Blocher. Multi-modal communication with a life-like character. In *Proceedings of EuroSpeech2001*, Aalborg, DK, 2001.

XTAG Research Group. A lexicalized tree adjoining grammar for english. Technical Report IRCS-01-03, IRCS, University of Pennsylvania, 2001.