



RTG Based Surface Realization from Dependency Representation

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENT FOR THE DEGREE OF

Master of Science

as a part of

**Erasmus Mundus Masters in Language and Communication
Technologies**

by

Shashi Narayan

Supervisors

Dr. Claire Gardent
Centre National de la Recherche Scientifique (CNRS)
LORIA, Vandoeuvre-les-Nancy
France

&

Mr. Mike Rosner
Dept. of Intelligent Computer Systems
University of Malta, Msida
Malta

**University of Malta and University of Nancy 2
Nancy 2011**

RTG Based Surface Realization from Dependency Representation

Shashi Narayan

MSc. Dissertation



Department of Intelligent Computer Systems
Faculty of Information and Communication Technology
University of Malta
2011

Supervisor(s):

Dr. Claire Gardent, CNRS LORIA, Nancy France
Mr. Mike Rosner, University of Malta, Malta

Submitted in partial fulfilment of the requirements for the Degree of
European Master of Science in Human Language Science and Technology (HLST)

RTG Based Surface Realization from Dependency Representation

Shashi Narayan

MSc. Dissertation



U.F.R. Mathématiques et Informatique
Université Nancy 2
2011

Supervisor(s):

Dr. Claire Gardent, CNRS LORIA, Nancy France
Mr. Mike Rosner, University of Malta, Malta

Submitted in partial fulfilment of the requirements for the Degree of
Master Sciences Cognitives et Applications, Spécialité Traitement Automatique des
Langues – Natural Language Processing

M.Sc. (HLST)
FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY
UNIVERSITY OF MALTA

Declaration

Plagiarism is defined as “the unacknowledged use, as one’s own work, of work of another person, whether or not such work has been published” (Regulations Governing Conduct at Examinations, 1997, Regulation 1 (viii), University of Malta).

I, the undersigned, declare that the Master’s dissertation submitted is my own work, except where acknowledged and referenced.

I understand that the penalties for making a false declaration may include, but are not limited to, loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Student Name:	Shashi Narayan
Course Code	CSA5310 HLST Dissertation
Title of work:	RTG Based Surface Realization from Dependency Representation

Signature of Student:



Date: 01/08/11

Abstract

Surface realization may be viewed as the back-end task of the natural language generation pipeline. It is the process of constructing natural language sentences from an abstract linguistic representation. Surface realizers, like GenI and RTGen, take abstract linguistic input in the form of flat logical formulae. These kinds of realizers present a number of important advantages but at the same time they raise an issue of complexity because of the use of realization from flat logical formulae. In this dissertation, we present an approach to surface realization with the abstract linguistic input in the form of dependency representation. Dependency structures can be adopted very easily to represent abstract linguistic representations. In this thesis, at first we study the relation between dependency structures and Tree Adjoining Grammar (TAG) derivation trees. In the next step, we propose an algorithm adapted from RTGen with an implementation for generation of TAG derivation forest from a given dependency representation. The proposed algorithm is guided by the relation between dependency structures and TAG derivation trees. The Regular Tree Grammar (RTG) encoding of the Feature Based Tree Adjoining Grammar (FB-TAG) with Earley chart parsing technique have been used to overcome the complexity issues. At the end, we present the analysis of our proposed method based on selected test cases from the Generation Challenge Campaign 2011. We see that the results are quite encouraging.

Keywords: Natural Language Generation [NLG], Surface Realization [SR], Tree Adjoining Grammar [TAG], Regular Tree Grammar [RTG], Dependency Representation, Flat Semantics Structure, TAG Derivation Tree, GenI, RTGen and Generation Challenge 2011.

Contents

Abstract	iii
Contents	iv
List of Figures	vi
List of Tables	vii
List of Algorithms	vii
Acknowledgements	ix
1 Introduction	1
2 Surface Realization and Input Representation	4
2.1 Natural Language Generation	4
2.2 Surface Realization	5
2.2.1 Abstract Linguistic Input	6
2.2.2 Knowledge Source: Lexicon and Grammar	7
2.2.3 Realization Algorithm	8
2.3 Surface Realization with TAG	9
2.3.1 GenI	11
2.3.2 RTGen	14
2.4 Input Representations	17
2.4.1 Dependency Structure Vs TAG Derivation Tree	18
3 Surface Realization from Dependency Representations	22
3.1 DRTGen Algorithm	22
3.1.1 Generation Challenge 2011 Surface Realization Shared Task Dependency Data	23
3.1.2 Feature based RTG encoding of TAG	24

<i>CONTENTS</i>	<i>CONTENTS</i>
3.1.3 Lexical Selection	25
3.1.4 Top Down Earley Chart Parsing Algorithm	25
3.1.5 String Extraction	34
3.2 DRTGen: Implementation Issues and Variations	36
3.2.1 Preprocessing of Input Dependency Structure	36
3.2.2 Multiple Adjunction	37
3.2.3 Guided with Dependency Structure	38
3.2.4 Intersective Modifiers : Packing	38
3.2.5 Handling Co-anchors	39
4 Implementation, Results and Discussion	41
4.1 DRTGen: Input Resources	41
4.2 DRTGen: A complete run	44
4.3 DRTGen: Running on Chunks	48
5 Conclusion and Future Research Ideas	51
Appendix A POS tag Mapping: Generation Challenge 2011 and XTAG	55
Bibliography	56

List of Figures

2.1	NLG Pipeline	5
2.2	Surface Realizer	6
2.3	Dependency structure for “John really likes Lyn”: [(sub,john,like), (obj,lyn,like), (mod,really,like)]	7
2.4	FB-LTAG Elementary Trees: the, man, runs and often	10
2.5	FB-LTAG Derived and Derivation Trees for “The man often runs”	11
2.6	FB-LTAG with compositional semantics “John often runs”	12
2.7	TAG elementary trees with Polarity	13
2.8	The elementary TAG tree for “caught” [Schmitz and Le Roux (2008)]	15
2.9	Study of Dependency vs Derivation : Derivation Process [Rambow and Joshi (1994)]	18
2.10	Study of Dependency vs Derivation : Derived Tree [Rambow and Joshi (1994)]	19
2.11	Derivation Tree	20
2.12	Dependency Tree	20
2.13	Study of Dependency vs Derivation [Rambow and Joshi (1994)]	20
3.1	DRTGen Architecture	22
3.2	FB-LTAG Elementary Trees: the, man, runs and often	30
3.3	FB-LTAG Derived and Derivation Trees for “The man often runs”	30
3.4	Elementary TAG tree with co-anchor	39
4.1	Generation Challenge: Actual Data	45
4.2	Generation Challenge Data: After processing	45
4.3	The president’s wife will be receiving a gift from the queen	45
4.4	Derivation Tree with Tn0Vn1-24	47
4.5	Derivation Tree with Tn0Vn1-26	47
4.6	DRTGen Output Derivation Tree	47
4.7	DRTGen output: Derived Tree (Figure 4.4)	48
4.8	DRTGen output with epsilon production (Figure 4.5)	50

List of Tables

3.1	DRTGen: Lexical Selection (“The man often runs”)	31
3.2	DRTGen: Realization of “The man often runs”	31
3.3	DRTGen Knowledge Base: rtgSelected/6	32
3.4	DRTGen Knowledge Base: agendaEntry/8 and chartEntry/8	33
4.1	DRTGen: Lexical Selection and Anchoring RTG rules	46
4.2	DRTGen: Initialization (With and Without Coanchor)	46
4.3	DRTGen run on NP-4	49
4.4	DRTGen run on S-6	49
A.1	POS Tag mapping	55

List of Algorithms

1	RTGen: Earley Chart Parsing Algorithm	17
2	DRTGen: Top Down Earley Chart Parsing Algorithm - Axiom and Goal	26
3	DRTGen: Top Down Earley Chart Parsing Algorithm - Prediction	27
4	DRTGen: Top Down Earley Chart Parsing Algorithm - Completion . . .	29
5	DRTGen Realization Algorithm: Pseudo code	35

Acknowledgements

I would like to thank Claire Gardent for introducing me to the interesting field of Natural Language Generation and for giving me an excellent opportunity to work under her guidance. Her knowledge and commitment with her politeness have always inspired me.

I am very thankful to Mike Rosner for his continuous guidance and support. Our Erasmus Mundus masters journey would not have been same without him. He was always there to listen to our problems.

I am very thankful to Laura for helping me throughout the project. It was great fun working together. I would also like to express my gratitude to all of my LCT friends for making my stay in Malta and Nancy very pleasant. I am extremely grateful to my parents and my lovely sisters for their unbounded love and affection.

SHASHI NARAYAN

University of Malta & University of Nancy 2
June 2011

Chapter 1

Introduction

Natural Language Generation (NLG) involves mapping from the communication goal to some surface utterances using domain dependent knowledge sources [Reiter and Dale (2000)]. The architecture of an NLG system may be viewed as a pipeline of various tasks like Content Determination, Document Planning, and Surface Realization. Surface realization is the back-end task in the NLG pipeline. It maps from the underlying content of text (abstract linguistic representation [Perez (2009)]) to a grammatically correct sentence that expresses the desired meaning.

There have been multiple high-quality publications in the area of surface realization (e.g., SURGE, KPML, RealPro [Lavoie and Rambow (1997)], GenI [Gardent and Kow (2005)], White (2004)'s CCG system, the HPSG ERG based realizer [Carroll and Oepen (2005)] and RTGen [Gardent and Perez-Beltrachini (2010)]). In this dissertation, we shall be mainly focusing on surface realizers based on reversible *Feature Based Lexicalized Tree Adjoining Grammars* or *FB-LTAG*. GenI and RTGen take abstract linguistic inputs in the form of flat logical formulae. These kinds of realizers present a number of important advantages but at the same time they raise an issue of complexity due to the use of flat logical formulae as their input [Kay (1996)] [Carroll and Oepen (2005)]. Because these capture semantic information only, the lexical ambiguity raised by this type of input given a large scale grammar is very high making these systems highly non-deterministic. These systems try to overcome these problems with various filtering techniques. In [Gardent and Kow (2005)], they propose an approach called *Polarity Filtering* to reduce the initial search space of GenI by discarding TAG trees that can not possibly yield a valid grammatical structure. Following up on a proposal by [Koller and Striegnitz (2002)], RTGen uses an interesting approach of Regular Tree Grammar (RTG) encoding of FB-TAG [Schmitz and Le Roux (2008)]. RTG-encoded TAG automatically guides system to take advantage of filtering [Gardent and Perez-Beltrachini (2010)] [C. Gardent and Perez-Beltrachini (2011)]

by discarding all the trees from the initial search space which can never be used to produce a well-formed TAG derivation tree. Another advantage of filtering techniques used in RTGen is that it considers all TAG elementary trees. In [Gardent and Kow (2005)], auxiliary trees were not considered during the polarity filtering stage.

The representation of an abstract linguistic input to surface realizers is not defined a priori and can be more or less specific. As mentioned above, the flat semantics representation used in GENI and RTGen leads to high non determinism due in particular to very high lexical ambiguity but also to the combinatorics induced by intersective modifiers.

In this dissertation, we explore a slightly easier surface realization task where the abstract linguistic input is in the form of *dependency representation*. Dependency representation provides closer representation to natural language syntax than free order flat semantics. The syntactically specific form of dependency structure provides us with more constraints on lexical selection (e.g., the part of speech of the input items is known) and hence reduced complexity. In this dissertation, we adapt the RTG based surface realizer developed by [Gardent and Perez-Beltrachini (2010)] for flat semantic input to syntactic dependency trees and propose an RTG (encoding of FB-TAG) based surface realizer for dependency representations called **DRTGen**.

The thesis consists of two major parts: *first*, we study the relation between dependency structures and TAG derivation trees and *second*, we adapt RTGen algorithm to generate TAG derivation forests from a given dependency structure. We also provide DRTGen with an implementation. The proposed algorithm is guided by the relation between dependency structures and TAG derivation trees. The RTG-encoding of FB-TAG in *DRTGen* incorporates the optimized filtering steps done in RTGen. We test our system with data from the *Generation Challenge 2011 Surface Realization Shared Task*¹.

This thesis is structured into five chapters organised as follows.

In **Chapter 2**, we briefly introduce the background concepts needed to understand this thesis. We describe the natural language generation task, the surface realization sub-task, and their main components. We present the RTG-encoding of FB-TAG. At the end, we introduce GenI, RTGen, their input representations and their computational complexity. We also do a background analysis on the relation between dependency structures and TAG derivation trees.

In **Chapter 3**, we propose an RTG based Surface Realizer from Dependency Representation (*DRTGen*). We formally present our algorithm. We discuss the implementation, performance and complexity issues.

¹<http://www.nltg.brighton.ac.uk/research/sr-task/>

Chapter 4 describes the data used to test our system and presents a complete run of DRTGen with its various steps of realization for a selected test dependency case.

In **Chapter 5**, we present our final conclusions of the dissertation and the direction of future research that can be done using the proposed approach.

Chapter 2

Surface Realization and Input Representation

This chapter briefly introduces the background concepts needed to understand this thesis. We describe the natural language generation task in Section 2.1. We give details of the surface realization sub-task and its main components in Section 2.2. We present a brief discussion of existing surface realization systems followed by the detailed analysis of the surface realization with FB-TAG and the RTG-encoding of FB-TAG in Section 2.3. At the end of this section, we introduce GenI, RTGen, their input representations and their computational complexities. We also do a background analysis on the relation between dependency structure and TAG derivation tree in Section 2.4.1.

2.1 Natural Language Generation

The problem of NLG can be divided into two major areas: *content selection* (“what shall I say?”) and *content expression* (“how shall I say it”). To process these issues, the architecture of NLG systems may be viewed as the pipeline of various subsequent tasks as shown in Figure 2.1.

The task “Text Planning” or “Content Planning” accepts communication goals as input and processes them to select content materials to express. This stage finally outputs the ordered content materials into a coherently flowing sequences. The task of text planning is generally language or domain independent. The next task “Sentence Planning” is an intermediate stage between “Text Planners” and “Surface Realizers”. This stage tries to fill the generation gap between those two stages. It processes the ordered content materials from “Text Planning” to introduce sentence boundaries and organize

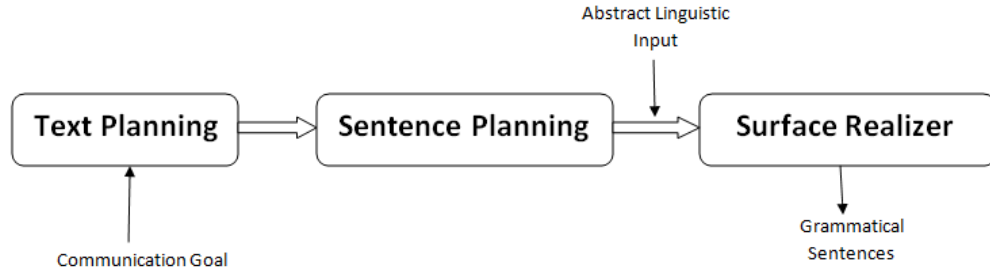


Figure 2.1: NLG Pipeline

the material internal to each sentence like resolving cross referencing, anaphora, word selection and other syntactic parameters. Its ideal output is a list of *abstract linguistic clauses or representations* with each unit clause fairly complete on syntactic specifications. These *abstract linguistic representations* are accepted by the task “Surface Realization” (discussed in detailed in Section 2.2) and realized into grammatically correct sentences.

2.2 Surface Realization

Surface Realization (SR) is the back-end task of the NLG pipeline (Figure 2.1). It is a mapping from the underlying content of text (*abstract linguistic representation*) to the grammatically correct sentences that expresses the desired meaning. The process of surface realization is sketched in Figure 2.2.

The problem of surface realization can be approached in two ways: the statistical approach (e.g., systems like NITROGEN [Langkilde and Knight (1998)]) and the symbolic approach (e.g., systems based on TAG [Gardent and Kow (2005), Gardent and Perez-Beltrachini (2010)], CCG [White (2004)] and HPSG [Carroll and Oepen (2005)]). Statistical systems induce rules from a large amount of training data, whereas symbolic approaches give an opportunity to work closely with language phenomena (grammatical rules, lexical constraints, paraphrase generations etc). With this motivation, the approach used for surface realization in this dissertation is the symbolic one. The method DRTGen proposed in this report is TAG based surface realization system. In the following subsection, we briefly present the main component of the symbolic approach based surface realization systems:

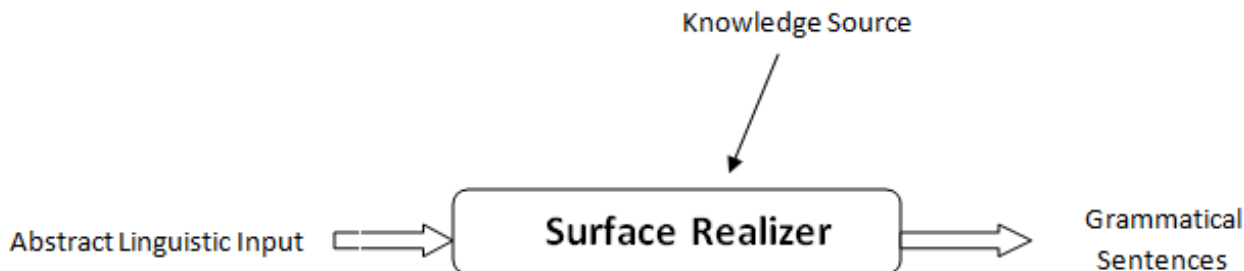


Figure 2.2: Surface Realizer

2.2.1 Abstract Linguistic Input

The abstract linguistic input to the surface realizer is not defined a priori and can be more or less specific [Copestake et al. (2005)]. Flat semantics representations consist of formulae with a direct translation to first order logic. It gives a syntactically underspecified representation of abstract linguistic input and consists of a set of literals where each literal consists of a predicate and some variables representing its relationship with other literals. One example of flat semantics is shown in (2.1). Please refer to [Gardent and Kallmeyer (2003)] and [Copestake et al. (2005)] for a detailed description of flat semantics. Several surface realizers [Koller and Striegnitz (2002); Carroll and Oepen (2005); White (2004); Gardent and Kow (2005); C. Gardent and Perez-Beltrachini (2011)] have been developed with *Flat Semantics* as their linguistic input. These realizers present a number of important advantages but at the same time they raise an issue of complexity because of the use of flat logical formulae as their input [Kay (1996)] [Carroll and Oepen (2005)]. Because these capture semantic information only, the lexical ambiguity raised by this type of input given a large scale grammar is very high making these systems highly non-deterministic.

Another simpler way to represent abstract linguistic input is *dependency structure*. Dependency structures are directed graphs or trees where nodes are labeled with words and edges with dependency relations such as “sub”, “obj” etc. Dependency structure provides a representation of linguistic input which is closer to syntax than the flat semantics representation and hence its use constitutes a more direct approach¹ towards surface realization. Another motivation to use dependencies as abstract linguistic input is their availability. Dependency trees can be easily and accurately obtained from Penn Treebank in sufficient quantities to test surface realizers. An example of dependency structure is shown in Figure 2.3.

¹guided by its structure, but still remains the problems of free ordering of sister nodes specially modifiers

In this dissertation, the proposed method, *DRTGen*, uses dependency structures as its linguistics input. It accepts a dependency structure in the form of a set of dependency edges where each dependency edge or dependency link consists of a tuple with at least the following information: Dependency Relation, Current Word and Parent Word. Additional grammar information such as POS tag, number, tense etc may be added to further constrain the output.

$$\{like(a), john(j), lyn(l), agent(a, j), pat(a, l), really(a)\} \quad (2.1)$$

Flat semantics for “John really likes Lyn”

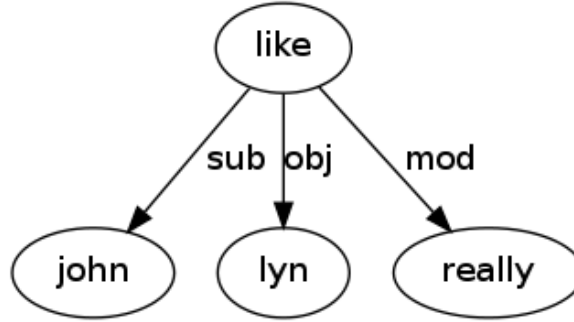


Figure 2.3: Dependency structure for “John really likes Lyn”: [(sub,john,like), (obj,lyn,like), (mod,really,like)]

2.2.2 Knowledge Source: Lexicon and Grammar

To generate sentences from some abstract linguistic input, surface realizers usually resort to some kind of lexical and grammatical knowledge e.g., grammar, lexicons, language models, Redwood type treebanks (for ranking). In particular, symbolic surface realizers such as those presented in [Koller and Striegnitz (2002); Carroll and Oepen (2005); White (2004); Gardent and Kow (2005); C. Gardent and Perez-Beltrachini (2011)] all use a lexicon and a grammar as its knowledge sources to map an abstract linguistic input to surface utterances. Both lexicon and grammar are language dependent. The lexicon is a list of lemmas and their associated grammatical properties. The lexicon provides the word choices to be made during realization and, in the case of lexicalised grammar, further guides realization with the corresponding grammatical rules to be applied. For example, in the case of a Tree Adjoining Grammar (*TAG*) grammar, the lexicon associates the names of *TAG* families with each lemma.

There have been many developments in grammatical formalisms. Keeping generation in mind and the grammar selected as knowledge source, a surface realization algorithm can be studied in two broad parts: *NLG Geared Realizers* and *Reversible Realizers*. NLG geared realizers (e.g., KPML [Bateman (1997)]) use grammars which are specially developed for generation purposes whereas reversible realizers use a grammar which can be used both for generation and for parsing. NLG geared grammars (e.g., functional grammar [Matthiessen et al. (1991)]) are augmented with additional compositional semantics, syntactic, pragmatic and discourse information. Reversible grammar are mainly augmented with compositional semantic information. Note that although NLG geared grammar looks like the efficient choice for the realizers but it comes with obvious drawbacks: first, time and expertise required for developing such grammar and second, irreversibility makes it impossible to use for parsing, and hence restricts evaluation of the realizer ¹. In this dissertation, the proposed method, *DRTGen*, is based on a reversible Tree Adjoining Grammar (**TAG**).

2.2.3 Realization Algorithm

As illustrated in Figure 2.2, a realization algorithm takes an abstract linguistic input and maps it to natural language sentences. In the process, it uses a lexicon to guide the selection of words or *lemmas* and a grammar to guide the sentence generation process. The selection of the grammar and the linguistic input highly influence the realization algorithm to be used. They define the approach of the algorithm and the goals to be achieved.

In this dissertation, the proposed method, *DRTGen*, adapts the TAG based realization algorithm used in surface realizer RTGen [Gardent and Perez-Beltrachini (2010)]. For a given dependency representation, the realization algorithm builds a set of TAG derivation/derived trees associated by grammar rules with the input dependency.

Morphological Realization

Morphological realization makes use of the morpho-syntactic properties (e.g., inflection for tense, gender, case etc) of words to produce a well inflected sentence out of the list of lemmas labeling the yield of the trees output by the surface realization process. Usually morphological realization is done at the end of the realization algorithm. It takes a

¹In case of reversible grammar, realized sentences (output of SR) can be checked with the parser for the grammar correctness or the test cases can be generated using the parser for the realizer testing

list of lemmas and feature structures decorating yields of each derived tree, looks up a morphological lexicon associating lemmas, word forms and morpho-syntactic features and produces for each lemma, the form which in the lexicon is associated with that lemma and whose features are compatible with those given by the derived tree.

The method DRTGen proposed in this dissertation, is adapted from RTGen, a *regular tree grammar (RTG) encoding of Feature Based Tree Adjoining Grammar (FB-TAG)* based surface realizer. RTG encoding of FB-TAG with the Earley chart parsing technique provide us a very efficient approach to deal with the complexities induced because of free order of abstract linguistic input representation [Gardent and Perez-Beltrachini (2010), Kay (1996) and Carroll and Oepen (2005)]. It also differs from the other existing reversible surface realizers like systems based on CCG [White (2004)] and HPSG [Carroll and Oepen (2005)]. TAG provides us with a symbolic selection of possible paraphrases, in contrast, existing reversible realizers use statistical information to select from the produced output the most plausible paraphrase [Gardent and Kow (2005)]. In the following Section 2.3, we introduce GenI and RTGen, the TAG based surface realizers.

The GenI and RTGen surface realizers use a Feature-Based Lexicalised TAG (FB-LTAG) as a grammar and take flat semantics logical formulae as their input representation. For a given flat semantics formula, the realization algorithm builds the set of TAG derived trees associated by the grammar with the input formula. At the later stage of morphological realization, each TAG derived tree is morphologically analyzed to produce natural language sentences.

2.3 Surface Realization with TAG

In this section, we present the surface realization algorithms adapted for the tree adjoining grammar (TAG) formalism. Although we assume the reader’s familiarity with TAG, we introduce the concepts of TAG and its variations in brief and then we focus on realization algorithm used in GenI and RTGen. The algorithm used in RTGen has been adapted for DRTGen.

The grammar formalism TAG was first proposed in [Joshi et al. (1975)]. A TAG consists of a set of elementary trees (initial and auxiliary trees) which can be combined with a substitution and an adjunction operation to produce TAG derived trees. The tree representation of TAG provides a larger domain of locality to handle the linguistic dependencies compared to context free grammars (CFG). Also TAG generates the tree language instead of the string language. We refer the reader to [Joshi and Schabes (1997)] for a detailed description of TAG.

Since the first introduction of TAG, several variations of TAG have been developed. Here we discuss two important variations of TAG: *Lexicalized TAG* (LTAG) and *Feature Based TAG* (FB-TAG). In lexicalized TAG formalism, each elementary tree contains at least one leaf node labeled with a terminal symbol or a lexical element called *anchor* of the tree. We refer the reader to [Abeille (1988)] and [Schabes (1990)] for complete knowledge of LTAG. The Feature based TAG formalism is an approach to constrain TAG’s substitution and adjunction operation. In FB-TAG, each non-terminal node of an elementary tree is optionally assigned with a set of grammatical features (attribute-value pairs). A substitution or an adjunction operation between two non-terminal nodes must satisfy the conditions set by the feature sets assigned to both nodes. We refer the reader to [Vijay-Shanker and Joshi (1988)] and [Vijay-Shanker (1993)] for the complete study of FB-TAG. GENI, RTGen and our proposed method use the combined form of LTAG and FB-TAG, also called *feature based lexicalized TAG* or *FB-LTAG*. We refer the reader to [XTAG-Research-Group (2001)] for the FB-LTAG for *English*.

At the end of this TAG introduction part, we would like to present to the reader two very important representations of TAG: *Derived trees* and *Derivation trees*. The derived tree is basically a phrase structure tree obtained by combining some or all of the elementary trees as per substitution and adjunction operation, whereas the derivation tree records the history of the derivation process to produce the derived tree. Derivation trees are similar to dependency trees. The TAG based surface realizer RTGen takes advantage of the concept of TAG derivation trees and tries to build the derivation trees from an abstract linguistic input. In the process it tries to achieve various kind of filtering for optimization. Note that the derivation tree can be easily converted to the derived tree and hence the corresponding string output.

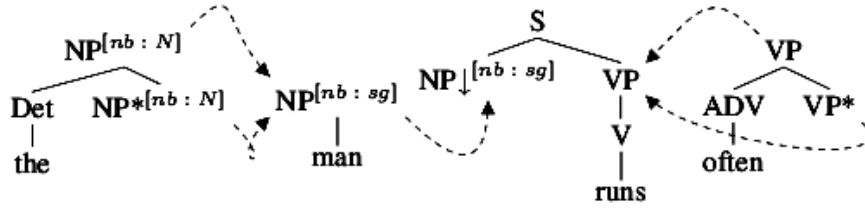


Figure 2.4: FB-LTAG Elementary Trees: the, man, runs and often

Figure 2.4 and Figure 2.5, an example from [C. Gardent and Perez-Beltrachini (2011)], summarize the concepts of TAG, its variations, substitution and adjunction operations, and derived and derivation trees representation. Figure 2.4 shows the lexicalized elementary trees (initial trees: T_{man} and T_{runs} ; auxiliary trees: T_{the} and T_{often}) of FB-LTAG and how they have been combined to form the derived and the derivation trees in Figure 2.5.

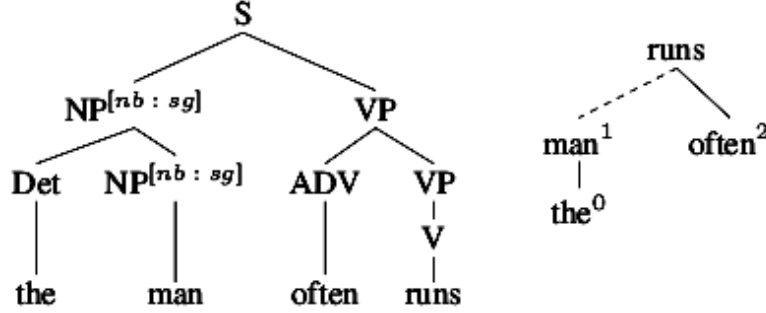


Figure 2.5: FB-LTAG Derived and Derivation Trees for “The man often runs”

The derivation tree in Figure 2.5 records the history of the derivation of the derived tree. The dotted line in the derivation tree represents the substitution operation whereas the straight line represents the adjunction operation. The number assigned to the node in the derivation tree represents the position of corresponding operation in the parent tree. In the derivation tree in Figure 2.5, T_{man} is substituted in T_{runs} at position 1. T_{the} and T_{often} are adjuncts to T_{man} and T_{runs} respectively.

2.3.1 GenI

GenI is a reversible FB-LTAG based surface realizer. It takes abstract linguistic input in the form of flat semantics. In this section, we first describe the variation of TAG used by GenI. Then we present the realization algorithm and at the end of the section, we discuss how GENI tackles the complexity problems.

Semantics construction in FB-LTAG

To achieve its realization goal, GenI uses a FB-LTAG equipped with a compositional semantics [Gardent and Kallmeyer (2003)]. In this representation of FB-LTAG, each lexicalized elementary tree of TAG is associated with a flat semantic representation.

Figure 2.6 shows an example of FB-LTAG equipped with compositional semantics. For each elementary tree, the arguments of its semantic representations occur at the substitution sites of the tree as unification variables (e.g., in Figure 2.6, s occurs at the substitution site of the elementary tree T_{run}) and the indices of semantic representation occurs at the root node or the adjunction sites of the tree (e.g., in Figure 2.6, r of the elementary tree T_{run} and j of the elementary tree T_{john} occur at the adjunction site and the root node of their tree respectively). This way, a TAG derived tree represents a complete

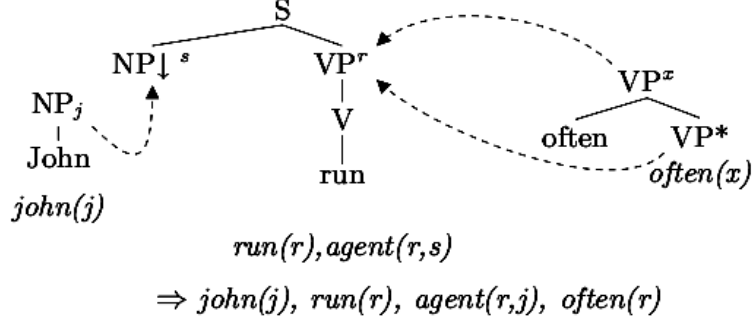


Figure 2.6: FB-LTAG with compositional semantics “John often runs”

flat semantics formula which is obtained by the unification of variables of the elementary tree’s flat semantics during the derivation process. We refer the reader to [Gardent and Kallmeyer (2003)] for a complete description of semantic construction in FB-LTAG.

Realization Algorithm

GenI uses a bottom-up, chart based realization algorithm described in [Gardent and Kow (2005)], [Gardent and Kow (2006)], [Gardent and Kow (2007a)] and [Gardent and Kow (2007b)]. The whole algorithm can be described in three major steps:

Lexical Selection For a given abstract linguistic input (flat semantics formula), it selects all elementary trees whose associated semantics subsumes part of the input formula. All selected initial trees are stored in an agenda and all selected auxiliary trees are stored in an auxiliary agenda to support the delayed adjunction operations.

Substitution Phase This stage selects a tree from the agenda and places it in the chart and tries to combine it using the substitution operation with the trees present in the chart. This stage continues till the agenda is empty.

Delayed Adjunction Phase This stage moves all complete chart trees from the previous stage to the agenda and adds all auxiliary trees from the auxiliary agenda to the chart. Then it selects a tree from the agenda and tries to combine it using the adjunction operation with the trees present in the chart. The newly derived items are added back to the agenda. This stage continues till the agenda is empty.

At the end, we select all complete TAG derived trees from the chart which cover the input flat semantics formula. These derived trees are then morphologically analyzed to retrieve the grammatical sentences.

Complexity Issues

Given a reasonable size TAG grammar, lexical ambiguity is a major source of inefficiency as many of the input literals will select a large number of elementary trees. For a given input literal, the *lexical selection* phase can end up with lots of unnecessary lexicalized elementary trees giving enormous load to the following steps. To address these shortcomings GenI implements various optimizations [Gardent and Kow (2006)].

Polarity Filtering: One of the very important optimization that takes place in GenI after the *lexical selection* phase is ***Polarity Filtering*** [Gardent and Kow (2005)]. It reduces the effect of lexical ambiguity. To understand it, we shall go through the example from [Kow (2007)] in Figure 2.7.

$$\{picture(p), cost(c, p, h), high(h)\} \quad (2.2)$$

Flat semantics for “The cost of the picture is high”

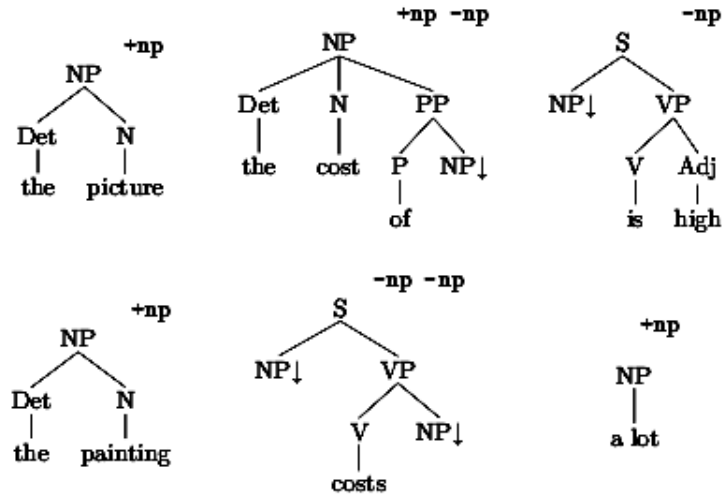


Figure 2.7: TAG elementary trees with Polarity

Figure 2.7 shows the set of elementary trees selected in the lexical selection phase for the flat semantics input formula given in 2.2. The synonymy appears to be a big problem for wide coverage lexicalized TAG. We select $T_{picture}$ and $T_{painting}$ for *picture* and T_{high} and T_{alot} for *high*. *cost* also selects two different trees but its not because of the synonymy but two different syntactic categories. The *polarity filtering* proceeds in two steps:

- Each of the initial trees of FB-LTAG is assigned a *polarity signature* reflecting its syntactic requirements, and
- All possible combinations of the elementary trees from the lexical selection with the net syntactic requirements *zero* are stored. All other combinations of the elementary trees are dropped out because such combinations can never lead to the syntactically complete derived trees.

Figure 2.7 shows the polarity signatures assigned to different trees. To find all possible combinations, *finite state automaton* (FSA) has been used. From Figure 2.7 we can see that the polarity filtering will reject the combination of $T_{painting}$, T_{cost} and T_{alot} because it has a net polarity of $+2np$. The combination $T_{picture}$, T_{cost} and T_{high} passes this test and proceeds to the next stage. *Polarity filtering* has proved to be very useful for the wide coverage lexicalized TAG. It improves the performance of the system by a huge margin. Also it is very easy to adapt in TAG scenario. But it has one major drawback: it does not consider any auxiliary trees for filtering. We refer the reader to [Gardent and Kow (2005)] for the detailed study of *Polarity filtering*.

Other Optimizations: GenI filters out all unusable trees after the *substitution phase* to reduce the search space. These unusable trees are mainly incomplete trees (with unfilled substitution) and complete trees with the root node other than category S . Another important optimization achieved automatically in GenI is ***delayed adjunction***. *Delayed adjunction* reduces the impact on the efficiency of intersective modifiers [Carroll and Oepen (2005)]. GenI adapts this technique very easily because it has a separate step for the adjunction process and all modifiers are defined as auxiliary trees in the TAG formalism.

2.3.2 RTGen

Like GenI, RTGen is a reversible FB-LTAG based surface realizer which takes the abstract linguistic input in the form of flat semantics representation. RTGen takes advantage of the *Regular Tree Grammar (RTG) encoding of TAG* defined by [Schmitz and Le Roux (2008)]. The main motivation behind RTGen is that RTG encoding of TAG can be used very effectively to handle the TAG derivations and to adapt more complete filtering approaches and hence lead to a better surface realization system. In this section, we first present the grammar formalism used for RTGen. We then present the realization algorithm used in RTGen.

RTG encoding of TAG

The derivation tree language of a TAG is context free. As pointed out in [Koller and Striegnitz (2002); Schmitz and Le Roux (2008)], TAG derivation trees are simpler to manipulate than TAG derived trees. To process a TAG based on its derivation trees, a new grammar formalism *Feature Based Regular Tree Grammar* or FB-RTG is proposed by [Schmitz and Le Roux (2008)]. They propose a way to translate from a *feature based tree adjoining grammar* to a *feature based regular tree grammar*. This RTG encoding of TAG has been used in RTGen for the surface realization algorithm.

$$(A, f) \rightarrow a((B_1, f'_1), (B_2, f'_2), \dots, (B_n, f'_n)) \quad (2.3)$$

where A, B_1, \dots, B_n are non-terminals,

f, f'_1, \dots, f'_n are feature structures, and

a is a terminal with rank n .

Thus each FB-TAG elementary tree can be converted to a FB-RTG rule. Equation 2.3 shows the format of a rule in FB-RTG. The left hand side of an RTG rule (A, f) , corresponds to the elementary tree information needed for combination into other trees. The right hand side of an RTG rule $(B_1, f'_1), \dots, (B_n, f'_n)$, corresponds to the information about the trees with particular operations (substitution or adjunction) that can be combined into this tree.

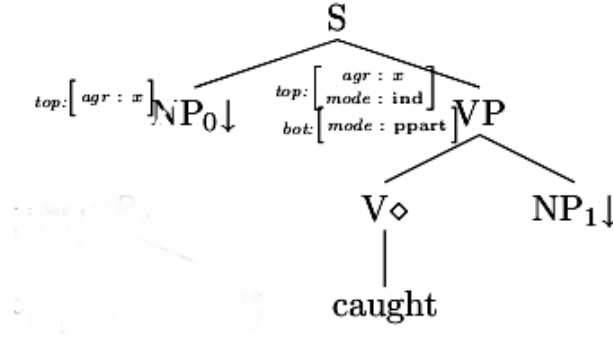


Figure 2.8: The elementary TAG tree for “caught” [Schmitz and Le Roux (2008)]

$$(S_{Sub}, \top) \rightarrow caught \left((NP_{Sub}, [top : [agr : x]]), \left(VP_{Adj}, \left[\begin{array}{ll} top & : \left[\begin{array}{ll} agr & : x \\ mode & : ind \end{array} \end{array} \right] \right. \right. \right. \\ \left. \left. \left. \begin{array}{ll} bot & : [mode : ppart] \end{array} \right] \right) \right), (NP_{Sub}, \top) \right) \quad (2.4)$$

For example, Equation 2.4 is an RTG rule corresponding to the elementary TAG tree for *caught* shown in Figure 2.8. It says that *caught* can be substituted into a *S* substitution node and it expects two substitutions, both of type *NP* and one adjunction of type *VP*. The corresponding feature requirements are also associated with each node. For each operation (substitution and adjunction), the associated feature structures must be unified. Other than converting all the elementary trees of TAG to RTG rules, the RTG encoding of TAG adds epsilon rules $X_{Adj} \rightarrow \epsilon$, for all category *X* to allow for the fact that the adjunction operations are optional in TAG derivation. We refer the reader to the second chapter of [Comon et al. (2007)] and [Schmitz and Le Roux (2008)] for the detailed description of RTG, FB-RTG and its translation from FB-TAG.

RTGen uses this *feature based RTG equipped with compositional semantics*. Note that the flat semantics associated with each elementary tree of FB-TAG can be directly transferred to a corresponding RTG rule in FB-RTG.

Realization Algorithm

RTGen uses *Earley's Parsing Algorithm* [Earley (1970)] adapted in [Kay (1996)] to support the generation from the flat semantics input representation using FB-RTG. We present this algorithm using the deduction framework [Shieber et al. (1995)] in Algorithm 1. The RTG rule format shown in equation 2.3 has been used here.

The algorithm starts with an axiom, $S' \rightarrow \bullet S_{sub}, \emptyset$ with associated semantics \emptyset . S_{sub} on the right side of the axiom makes sure that all possible derivation trees have the initial trees with the category *S* as their root nodes. The algorithm terminates with the goal $S' \rightarrow S_{sub} \bullet, \phi$, that means that S_{sub} has been analyzed completely and ϕ is exactly the same as the flat semantics input. The preconditions on the prediction and the completion steps the guide algorithm to generate correct derivation trees. At the same time they optimize the generation system by filtering out unnecessary chart items. The preconditions of the prediction inference make sure that only those chart items are initialized which have the same non-terminals, their feature structures are unifiable (unification: $\sigma = mgu(f_i, f')$) and their semantics doesn't overlap with each other (semantic filtering: $\psi \cap \varphi = \emptyset$). Similarly the preconditions of the completion inference block unnecessary

Algorithm 1 RTGen: Earley Chart Parsing Algorithm

$$\begin{aligned}
\textbf{Axiom} : & \frac{}{[S' \rightarrow \bullet S_{sub}, \emptyset]} \\
\textbf{Goal} : & [S' \rightarrow S_{sub} \bullet, \phi], \text{ where } \phi \text{ is the input semantics covered} \\
\textbf{Prediction} : & \frac{[(A, f) \rightarrow a(\alpha \bullet (B, f_i)\beta), \varphi]}{[(B, \sigma(f')) \rightarrow b(\bullet(B_1, \sigma(f'_1)), (B_2, \sigma(f'_2)), \dots, (B_n, \sigma(f'_n))), \psi]} \\
& \text{where } (B, f') \rightarrow b((B_1, f'_1), (B_2, f'_2), \dots, (B_n, f'_n)) \text{ is a rule in the grammar} \\
& \text{with associated semantics } \psi, \sigma = mgu(f_i, f') \text{ and } \psi \cap \varphi = \emptyset \\
\textbf{Completion} : & \frac{[(A, f) \rightarrow a(\alpha \bullet (B, f_i)\beta), \varphi] [(B, f') \rightarrow b(\beta') \bullet, \psi]}{[(A, \sigma(f)) \rightarrow a(\alpha(B, \sigma(f_i)) \bullet (C, \sigma(f_j))\beta'), \phi]} \\
& \text{where } \sigma = mgu(f_i, f'), \psi \cap \varphi = \emptyset \text{ and } \psi \cup \varphi = \phi
\end{aligned}$$

completion steps. Because of its chart based algorithm, RTGen also takes advantage of the sharing of intermediate results.

2.4 Input Representations

The representation of abstract linguistic input to surface realizer is not defined a priori and can be more or less specific. The *flat semantics* representation used in GenI and RTGen provides a syntactically underspecified representation of abstract linguistic input. This underspecification of the syntactic properties of the output leads these realizers to be highly non-deterministic and hence to higher complexity. GenI tries to overcome this using various optimisation techniques (delayed adjunction, polarity filtering) that are easily implementable in a TAG based framework [Gardent and Kow (2006)] whereas RTGen uses the RTG encoding of TAG and uses an Earley algorithm to reduce the combinatorics. In RTGen, the given set of elementary TAG trees, the TAG to RTG mapping makes it possible to determine whether this set can be combined into a derivation tree. This also takes the auxiliary tree into account which was not covered in GenI *polarity filtering*.

Another way to approach the problem of surface realization is with *dependency structures* as its input representation. Because of the structural similarity between the dependency structure and the TAG derivation tree, we approach this problem as realization from a dependency structure to TAG derivation trees and hence TAG derived trees. This realization is not straightforward because of the fundamental gaps between both repre-

sentations. Section 2.4.1 presents a study of the similarities and differences between both representations.

2.4.1 Dependency Structure Vs TAG Derivation Tree

The TAG derivation tree offers a first insight into the sentence semantics [Candito and Kahane (1998)]. It presents a very similar representation to dependency structure. But while a TAG derivation tree can be easily mapped to a derived tree (parse tree, hence surface utterances), the mapping between dependency structures and TAG derivation trees is much less straightforward. This section presents a comparative study of both representations. This study guides the realization algorithm proposed in Chapter 3.

- TAG derivation trees (Figure 2.11) are enriched with substitution, adjunction and lexicon (trees to select from TAG grammar) information. So TAG derivation trees can easily be converted to TAG derived trees (phrase structure trees)(see Figure 2.9 and Figure 2.10). But in dependency structures (Figure 2.12) this information is missing.

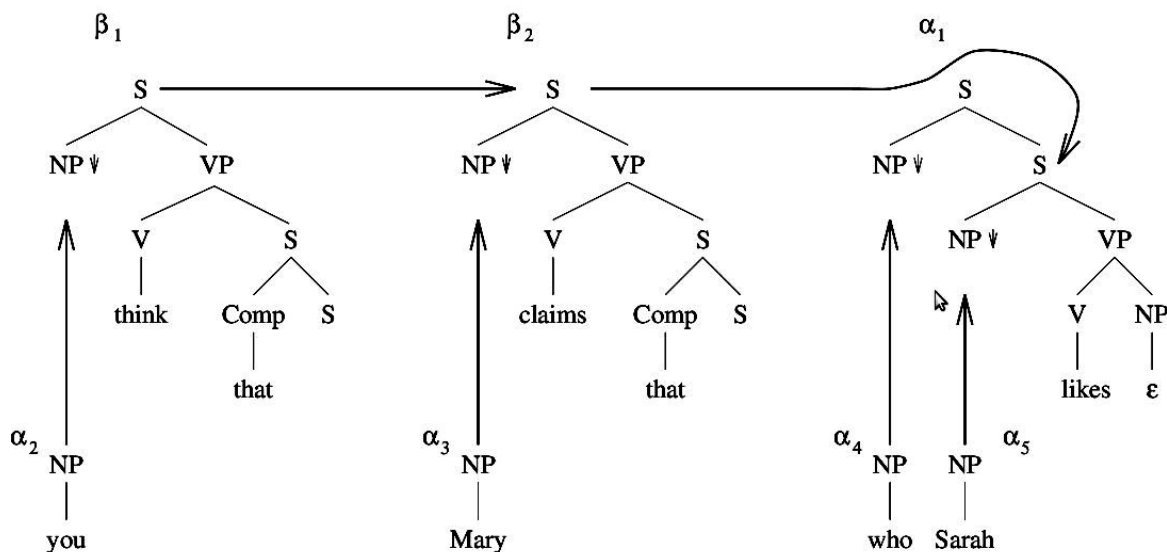


Figure 2.9: Study of Dependency vs Derivation : Derivation Process [Rambow and Joshi (1994)]

- Sometimes dependency structures are very similar to TAG derivation trees structurally. But this is not always true. For example Figure 2.12 and Figure 2.11 are the

dependency tree and the derivation tree for the sentence “**Who you think that Mary claims that Sarah likes?**” respectively. Both are structurally different (e.g., the root nodes are different.)

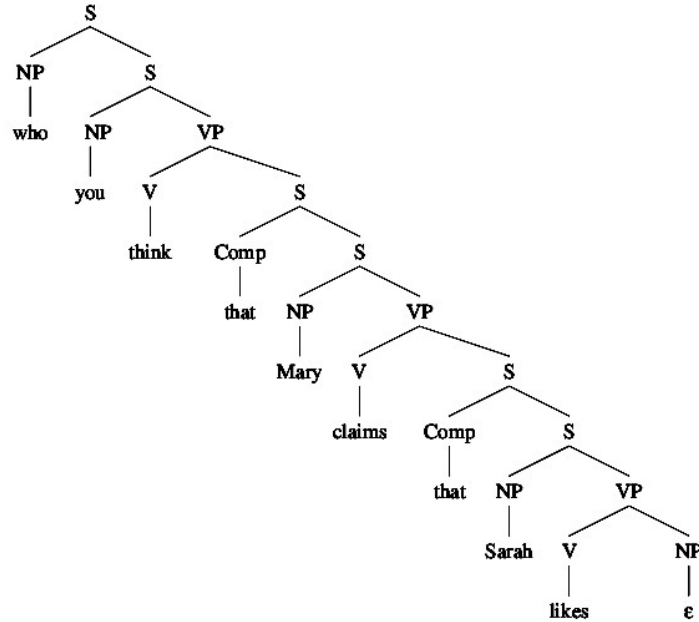


Figure 2.10: Study of Dependency vs Derivation : Derived Tree [Rambow and Joshi (1994)]

- In general, each of the derivation processes present in the TAG derivation tree corresponds to a dependency relation in the corresponding dependency structure. For the substitution process in the TAG derivation tree, the hierarchy of corresponding dependent nodes in the dependency structure is fixed (e.g., if a node n_1 is substituted in node n_2 , then in the dependency structure, node n_1 is child dependent of node n_2). This is not true for the adjunction process.
- There is no order between the sister nodes in the dependency structure. But the TAG derivation tree has direct correspondence with its derived tree.
- Sometimes in dependency structures, the non-important words are missing. For example in Figure 2.12, word “that” is missing. But it is present in the TAG elementary trees.
- In general, dependency structures are in tree formats, but sometimes to capture specific details (e.g., semantic information), they can be in graph formats.

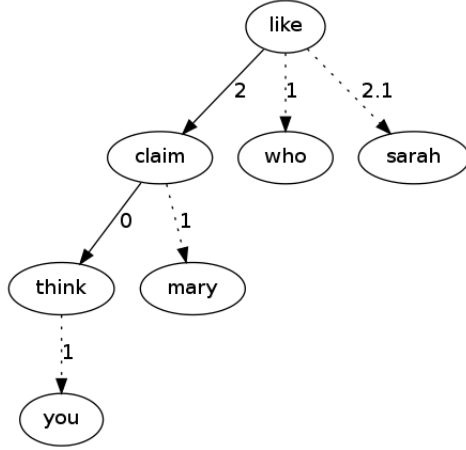


Figure 2.11: Derivation Tree

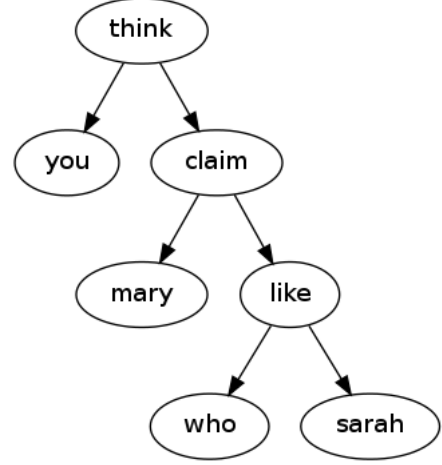


Figure 2.12: Dependency Tree

Figure 2.13: Study of Dependency vs Derivation [Rambow and Joshi (1994)]

Study of Mapping from Dependency Structure to TAG Derivation Trees

Several attempts [Joshi and Rambow (2003)], [Xia and Palmer (2001)] and [Rambow and Joshi (1994)], have been taken in the past to study the dependency structure and the derivation tree, and to narrow down the gap between them.

In [Xia and Palmer (2001)], they use a conversion algorithm inspired by *X-bar theory*. They use heuristic rules to retrieve phrase structure for a given dependency structure. We refer the reader to [Xia and Palmer (2001)] for the detailed study of this approach. In [Joshi and Rambow (2003)], they present a formalism for dependency grammar based on some key ideas from Tree-Adjoining Grammars: *Dependency Grammar based on TAG*. They present a dependency grammar (in few examples) in terms of elementary dependency trees anchored with the lexical items. These elementary trees correctly capture the dependencies associated with the lexical anchor. These trees may also include nodes that represent items on which the lexical anchor depends. They also describe operations for combining elementary or derived dependency trees, which are analogous to “substitution” and “adjoining” in TAG. Their dependency elementary trees are also lexicalized which can be directly mapped to the elementary trees of the lexicalized tree adjoining grammar (LTAG). Their approach seems very interesting but the lack of a resource such as “Dependency Grammar based on TAG” makes it very difficult to approach this way. We refer the reader to [Joshi and Rambow (2003)] for the detailed picture of *Dependency Grammar based TAG*.

In Chapter 3, we modify the RTG (encoding of FB-TAG) based surface realizer so as to generate from dependency representations. The resulting surface realizer (or **DRTGen**) takes an input dependency representation and tries to map it to TAG derivation trees which can then be mapped to the corresponding TAG derived trees. Also with the base algorithm, we present and discuss the specific input dependency representations made available by the Generation Challenge Surface Realisation task in 2011.

Chapter 3

Surface Realization from Dependency Representations

We propose an RTG based Surface Realizer from Dependency Representations or ***DRT-Gen***. DRTGen takes a dependency structure and converts it to a TAG derivation tree. The TAG derivation tree is then mapped to the corresponding derived tree. In this chapter, we formally present DRTGen. We discuss its implementation, performance and complexity issues.

3.1 DRTGen Algorithm

DRTGen adapts RTGen algorithm, described in Section 2.3.2, to generate TAG derivation forest from a given dependency structure. The proposed algorithm is guided with the relation between dependency structures and TAG derivation trees (Section 2.4.1). Figure 3.1 shows the major components of DRTGen.

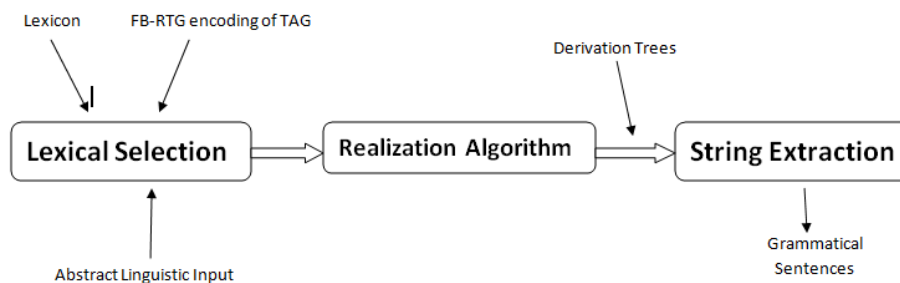


Figure 3.1: DRTGen Architecture

In this section, we present the major components of DRTGen in detail. In the first two sections, Section 3.1.1 and Section 3.1.2, we discuss the input resources to DRTGen. Section 3.1.3, Section 3.1.4 and Section 3.1.5 explain the major steps of *DRTGen* algorithm in detail. DRTGen has been implemented in *Prolog*. The data-structures (*knowledge-base* in *Prolog* terms) of DRTGen are presented in *Prolog* representation.

3.1.1 Generation Challenge 2011 Surface Realization Shared Task Dependency Data

For the input dependency structures, DRTGen uses the data from the *Generation Challenges 2011 Surface Realization Shared Task*¹. The Surface Realization task of Generation Challenge 2011 aims to compare and evaluate systems that map (one of) the common-ground input representations to surface word strings (fully realized sentences). The common-ground input representation is proposed to be *dependency representation*. The data provided by the SR Task was obtained from the Penn Treebank. Two types of dependency input representations are available: *shallow* and *deep*.

Shallow representation Words are represented as nodes (lemma, POS-tag and other grammatical feature of the word) in a *syntactic dependency tree*. Connecting edges are labeled with the syntactic dependency relations.

Deep representation The syntactic dependency relations are replaced with the semantic dependency relations (if available). The POS information has been removed. In the deep representation, the dependency structure may be a **graph**.

For both shallow and deep representations, relation or dependent sister nodes are arbitrarily ordered. Sentences have single sentence root.

For DRTGen, we only consider the shallow representation of input dependencies. The shallow dependency tree can be considered as a set of dependency edges where each dependency edge or dependency link consists of a quadruple with the minimum of this information: Dependency Relation, Current Lemma, Parent Lemma, and POS tag of the current Lemma (Equation 3.1). Additional grammar information may be associated with the current lemma (e.g., gender, tense and partic) to restrict the surface realization and improve the morphological analysis. For example, Equation 3.2 shows the shallow dependency tree representation (Generation Challenge 2011) accepted by DRTGen. The lemma associated with the dependency relation “sroot” is the root of the dependency tree.

¹<http://www.nltg.brighton.ac.uk/research/sr-task/>

Thus \mathcal{D} in Equation 3.2 is a set of dependency links associated with an input dependency representation to DRTGen.

$$\begin{aligned} \mathcal{D} &= [(dep_1, word_1, parent_1, pos_1), \dots, (dep_n, word_n, parent_n, pos_n)] \\ \text{DRTGen: Input Dependency Format} \end{aligned} \quad (3.1)$$

$$\begin{aligned} \mathcal{D} &= [(sroot, like, -, vb), (sub, john, like, nn), (obj, lyn, like, nn), (mod, really, like, rb)] \\ \text{DRTGen: An example for Input dependency} \end{aligned} \quad (3.2)$$

3.1.2 Feature based RTG encoding of TAG

DRTGen uses the same grammar formalism *feature based RTG* as in RTGen without the associated semantics. Although we are using the same approach described in Section 2.3.2, we are adding some extra information to the RTG rule. To deal with dependencies, we are adding an extra feature “dep” (for dependency relations) to the feature structures of the substitution sites of initial trees and the foot nodes of the auxiliary trees of TAG. For example in Equation 2.4, node NP_0 will get an additional attribute-value pair (“dep”=SUB) (subject) in its existing feature set and node NP_1 will get an additional attribute-value pair (“dep”=OBJ) (object) in its existing feature set. The RTG encoding of TAG described in [Schmitz and Le Roux (2008)] can be used for this improved TAG without any problem. The RTG encoding of the improved TAG will have a feature “dep” inside the feature structures d ’s of the associated non-terminals (Equation 2.3). But for the sake of explanation, we shall keep the “dep” value associated with each non-terminal separately as shown in Equation 3.3.

$$(A, (f, r)) \rightarrow a((B_1, (f'_1, r'_1)), (B_2, (f'_2, r'_2)), \dots, (B_n, (f'_n, r'_n))) \quad (3.3)$$

where A, B_1, \dots, B_n are non-terminals,

f, f'_1, \dots, f'_n are feature structures,

r, r'_1, \dots, r'_n are dependency relations, and

a is a terminal with rank n .

Note that in the RTG encoding of the improved TAG, the dependency relations (r') associated with the adjunction non-terminals on the right hand side will be empty (unifiable with any value). Also the dependency relations (r) associated with the non-terminal

on the left hand side will be empty for the initial trees. In all other cases, like the substitution non-terminals on the right hand side and the non-terminal on the left hand side for the auxiliary tree, the dependency relations will get a value from the feature “dep” added to the corresponding TAG elementary tree.

We are on the way to complete the augmenting of TAG with the “dep” feature. In the present scenario, many dependency relations are left empty (unifiable with any value).

In another improvement, we keep account of the node positions (in TAG elementary tree) associated to the non-terminals on the right hand side of a rule. [Schmitz and Le Roux (2008)] doesn’t keep the account of this, which makes it difficult to build the derived tree from RTG derivation. For ease of explanation, we are not showing this in Equation 3.3. This information is not required during the generation process. We need this during the sentence extraction process from RTG derivations.

3.1.3 Lexical Selection

Lexical selection is the first step of DRTGen. For a given input dependency \mathcal{D} (e.g., shown in Equation 3.2), it selects the RTG rules associated with each $(lemma, pos)$ pair present in \mathcal{D} . The selected RTG rules are stored in *Prolog knowledge base* with the representation shown in Table 3.3. Before storing them in RTG rules, we check for their validity by their argument requirements. For example, if a selected RTG rule has n substitution entries on the right hand side then in the dependency structure the corresponding lemma must have at least n child dependents with initial trees (Section 2.4.1). Only those RTG rules are considered which pass this test. Also note that POS tags used in Generation Challenge 2011 are different from POS tags used in XTAG (used in our lexicon). To overcome this issue, we are using a mapping between the Generation Challenge POS tag set and the XTAG POS tag set. Please refer to Appendix A for this mapping.

In terms of the deductive framework of DRTGen, lexical selection gives a set of anchored RTG rules upon which prediction and completions rules are applied.

3.1.4 Top Down Earley Chart Parsing Algorithm

The algorithm designed for *DRTGen* has been adapted from *RTGen*, described in Section 2.3.2. *RTGen* generates from a flat semantics input representation, and we adapted it to generate from a dependency tree representation. *DRTGen* uses *Top down Earley Chart Parsing* algorithm to optimize the realization from a dependency tree input. It incorporates various parsing techniques to optimize the system (e.g., chart based system,

sharing of intermediate results and dynamic programming).

Study of Dependency vs Derivation Trees

The proposed algorithm is guided with the relation between dependency structures and TAG derivation trees (Section 2.4.1). Some important studies that we reported in Section 2.4.1, are mentioned again for clarity:

- Corresponding to each dependent lemma pair in a dependency tree \mathcal{D} , we have a TAG derivation process between the same lemma pair in the derivation tree. The information that is missing, are “which derivation process” (substitution or adjunction) and “its relative position” in the TAG derivation tree. This fact makes sure that for each lemma, we just need to look among its direct dependent lemmas.
- The search among dependents can be refined further. The substitution sites of a lexicalized TAG elementary tree of a lemma can only be fulfilled with the child dependents of corresponding lemma in the dependency tree, whereas the adjunction sites of a lexicalized TAG elementary tree can be fulfilled with both parent and child dependents.
- The root of the input dependency tree may not be the same as the root of the output TAG derivation tree. So DRTGen considers all possible choices of root lemmas (initial trees with category s) in the input dependency tree.

Algorithm 2 DRTGen: Top Down Earley Chart Parsing Algorithm - **Axiom and Goal**

Axiom : $\overline{[S' \rightarrow \bullet S_{Sub}, \emptyset, (C_{dep} = -, P_{dep} = -)]}$, where C_{dep} and P_{dep}

are rest child and rest parent dependents respectively

Goal : $[S' \rightarrow S_{Sub}\bullet, \mathcal{D}, (-, -)]$, where \mathcal{D} is the input dependency covered

Deductive Parsing Framework for DRTGen

The core of DRTGen, *initialization* and *termination* (Algorithm 2), *prediction* (Algorithm 3), and *completion* (Algorithm 4) steps are explained using the deductive parsing framework [Shieber et al. (1995)] for the direct and declarative presentation.

Algorithm 3 DRTGen: Top Down Earley Chart Parsing Algorithm - **Prediction**

$$\mathbf{Subs} : \frac{[(A, (f, r)) \rightarrow a(\alpha \bullet (B_{Sub}, (f_i, r_i))\beta), D_A, (C_A, P_A)]}{[(B_{Sub}, (\sigma(f'), r')) \rightarrow b(\bullet(B_1, (\sigma(f'_1), r'_1)), \dots, (B_n, (\sigma(f'_n), r'_n)))) , D_{B_{Sub}}, (C_{B_{Sub}}, P_{B_{Sub}})]}$$

where $(B_{Sub}, (f', r')) \rightarrow b((B_1, (f'_1, r'_1)), \dots, (B_n, (f'_n, r'_n)))$ is a rule in the grammar

with associated dependencies $D_{B_{Sub}}, \sigma = mgu(f_i, f')$,

$d_{AB_{Sub}} \in C_A$ where $d_{AB_{Sub}} = (r_i, b, a)$, the dependency between A and B ,

$C_{B_{Sub}} = \mathcal{C}_{B_{Sub}}, P_{B_{Sub}} = \mathcal{P}_{B_{Sub}} - d_{AB_{Sub}}$ and $D_{B_{Sub}} \cap D_A = \emptyset$

$$\mathbf{Adj} : \frac{[(A, (f, r)) \rightarrow a(\alpha \bullet (B_{Adj}, (f_i, r_i))\beta), D_A, (C_A, P_A)]}{[(B_{Adj}, (\sigma(f'), r')) \rightarrow b(\bullet(B_1, (\sigma(f'_1), r'_1)), \dots, (B_n, (\sigma(f'_n), r'_n)))) , D_{B_{Adj}}, (C_{B_{Adj}}, P_{B_{Adj}})]}$$

where $(B_{Adj}, (f', r')) \rightarrow b((B_1, (f'_1, r'_1)), \dots, (B_n, (f'_n, r'_n)))$ is a rule in the grammar

with associated dependencies $D_{B_{Adj}}, \sigma = mgu(f_i, f')$,

$d_{AB_{Adj}} \in C_A \cup P_A$, where $d_{AB_{Adj}} = (r_i, b, a)$ or (r_i, a, b) , the dependency between A and B ,

$C_{B_{Adj}} = \mathcal{C}_{B_{Adj}} - d_{AB_{Adj}}, P_{B_{Adj}} = \mathcal{P}_{B_{Adj}} - d_{AB_{Adj}}$ and $D_{B_{Adj}} \cap D_A = \emptyset$

In Algorithms 2, 3 and 4, each fact is of the form $[Rule, D_{Rule}, Rule_{dep}]$. The $Rule$ corresponds to an RTG rule (Equation 3.3) in the process. The D_{Rule} is the dependency covered (a subset of \mathcal{D} , Equation 3.2) by this $Rule$ and the $Rule_{dep}$ is the rest dependents of the lemma (associated with $Rule$) in the input dependency tree and not covered by D_{Rule} . The $Rule_{dep}$ is of the format (C_{Rule}, P_{Rule}) whereas C_{Rule} and P_{Rule} represent the rest child and the rest parent dependent of the lemma respectively. Note that \mathcal{C}_{Rule} and \mathcal{P}_{Rule} are the complete child and parent dependent sets of the lemma associated with $Rule$ in input dependency tree \mathcal{D} .

DRTGen starts with an axiom $[S' \rightarrow \bullet S_{Sub}, \emptyset, (-, -)]$ (Algorithm 2). S_{Sub} on the right side of the rule makes sure that it considers all lexicons with initial elementary trees with category S as roots of the TAG derivation trees. To make sure it selects globally, both C_{dep} and P_{dep} are set to “-” (Prolog term, unifiable with everything). Also S' is the start symbol, so no dependents are available for them in dependency tree. It has not covered any dependencies yet, so D_{Rule} has been set to \emptyset . The goal of DRTGen is defined as $[S' \rightarrow S_{Sub}\bullet, \mathcal{D}, (-, -)]$ (Algorithm 2). It covers the dependency input \mathcal{D} .

The *prediction* step of DRTGen is guided with the input dependency tree. Depending on the node next to \bullet , the prediction step (Algorithm 3) has been divided into two inference rules: *substitution* and *adjunction*. If the node B_{Sub} next to \bullet is of type substitution, we just look inside the child dependent C_A and the dependency $d_{AB_{Sub}}$ ($d_{AB_{Sub}} \in C_A$) between the lemma a associated with the rule A and the lemma b associated with the predicted rule B_{Sub} will be of the form (r_i, b, a) where r_i is the dependency relation required with this substitution site. The newly predicted rules are introduced with the dependency covered $D_{B_{Sub}}$, the child dependents $C_{B_{Sub}} (= \mathcal{C}_{B_{Sub}})$ and the parent dependents $P_{B_{Sub}} (= \mathcal{P}_{B_{Sub}} - d_{AB_{Sub}})$. $D_{B_{Sub}}$ is generally \emptyset but it can have some dependencies in case of lemmas with co-anchors (Section 3.2.5). $\mathcal{C}_{B_{Sub}}$ and $\mathcal{P}_{B_{Sub}}$ are the complete child and parent dependent sets of the lemma associated with B_{Sub} in input dependency tree \mathcal{D} . Note that $d_{AB_{Sub}} \in \mathcal{P}_{B_{Sub}}$ and $P_{B_{Sub}}$ is the rest dependent lemma for B_{Sub} so we need to remove $d_{AB_{Sub}}$ from $\mathcal{P}_{B_{Sub}}$ and then assigned to $P_{B_{Sub}}$. This reduction stops the chances of repeating the covered dependencies. If the node B_{Adj} next to \bullet is of type adjunction, then we look inside both of the child and the parent dependent of A (Algorithm 3). Note that newly predicted rules also unify their feature structures using the most general unifier “mgu” $\sigma = mgu(f_i, f')$.

The *completion* step of DRTGen (Algorithm 4) is also guided with the input dependency tree, and allows multiple adjunction. Depending on the completed rules B used to complete the incomplete rule A , the completion can be presented with 3 different inference rules: *substitution completion*, *adjunction completion* and *epsilon completion*. Note that *epsilon completion* is a special case of *adjunction completion*. In case of *substitution*

Algorithm 4 DRTGen: Top Down Earley Chart Parsing Algorithm - **Completion**

$$\mathbf{Subs} : \frac{[(A, (f, r)) \rightarrow a(\alpha \bullet (B_{Sub}, (f_i, r_i))\beta), D_A, (C_A, P_A)] \left[(B_{Sub}, (f', r')) \rightarrow b(\beta')\bullet, D_{B_{Sub}}, (\emptyset, \emptyset) \right]}{[(A, (\sigma(f), r)) \rightarrow a(\alpha(B_{Sub}, (\sigma(f_i), r_i)) \bullet (C, (\sigma(f_j), r_j))\beta_1), D'_A, (C'_A, P'_A)]}$$

where $\sigma = mgu(f_i, f')$, $D_{B_{Sub}} \cap D_A = \emptyset$, $D_{B_{Sub}} \cup D_A \cup d_{AB_{Sub}} = D'_A$,

where $d_{AB_{Sub}} = (r_i, b, a)$, the dependency relation between A and B_{Sub} ,

$d_{AB_{Sub}} \in C_A$, $d_{AB_{Sub}} \notin D_{B_{Sub}}$, $P'_A = P_A$ and $C'_A = C_A - d_{AB_{Sub}}$

$$\mathbf{Adj} : \frac{[(A, (f, r)) \rightarrow a(\alpha \bullet (B_{Adj}, (f_i, r_i))\beta), D_A, (C_A, P_A)] \left[(B_{Adj}, (f', r')) \rightarrow b(\beta')\bullet, D_{B_{Adj}}, (\emptyset, \emptyset) \right]}{[(A, (\sigma(f), r)) \rightarrow a(\alpha(B_{Adj}, (\sigma(f_i), r_i)) \bullet (B_{Adj}, (\sigma(f_i), r_i))\beta_1), D'_A, (C'_A, P'_A)]}$$

where $\sigma = mgu(f_i, f')$, $D_{B_{Adj}} \cap D_A = \emptyset$, $D_{B_{Adj}} \cup D_A \cup d_{AB_{Adj}} = D'_A$,

where $d_{AB_{Adj}} = (r_i, a, b)$ or (r_i, b, a) , the dependency relation between A and B_{Adj} ,

$d_{AB_{Adj}} \in C_A \cup P_A$, $d_{AB_{Adj}} \notin D_{B_{Adj}}$, $C'_A = C_A - d_{AB_{Adj}}$ and $P'_A = P_A - d_{AB_{Adj}}$

[The rule completed but the bullet position not shifted, **allow multiple adjunction**]

$$\mathbf{eps} : \frac{[(A, (f, r)) \rightarrow a(\alpha \bullet (B_{Adj}, (f_i, r_i))\beta), D_A, (C_A, P_A)] \left[(B_{Adj}, (f', r')) \rightarrow \epsilon, \emptyset, (\emptyset, \emptyset) \right]}{[(A, (\sigma(f), r)) \rightarrow a(\alpha(B_{Adj}, (\sigma(f_i), r_i)) \bullet \beta_1), D_A, (C_A, P_A)]}$$

where $\sigma = mgu(f_i, f')$

completion by B_{Sub} , DRTGen looks for all rules A with $d_{AB_{Sub}} \in C_A$. But in the case of *adjunction completion* this is not true ($d_{AB_{Sub}} \in C_A \cup P_A$). Note that in the normal case of *adjunction completion*, it allows *multiple adjunction* (See Algorithm 4) by leaving the one instance of B_{Adj} uncovered. This process of *multiple adjunction* is terminated by *epsilon completion*. The *epsilon completion* also makes sure that the adjunction process is optional.

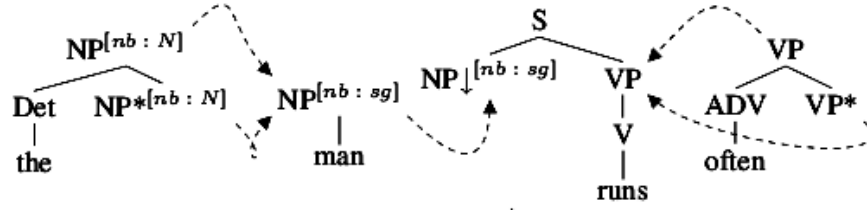


Figure 3.2: FB-LTAG Elementary Trees: the, man, runs and often

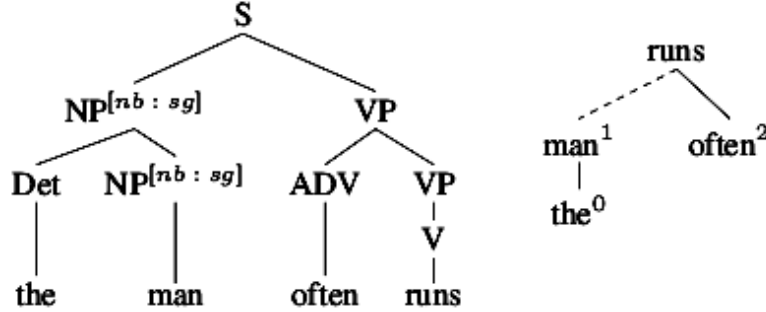


Figure 3.3: FB-LTAG Derived and Derivation Trees for “The man often runs”

For clear understanding of our deductive parsing framework for DRTGen, we present an ideal run of DRTGen using the deductive framework. We assume that our FB-TAG is the TAG presented in Figure 3.2. We try to generate from the dependency representation of “The man often runs”. Note that for this sentence, dependency structure will be the same as its derivation tree shown in Figure 3.3. $\mathcal{D} = [(t, m), (m, r), (o, r)]$ where ‘t’, ‘m’, ‘r’ and ‘o’ represents for ‘the’, ‘man’, ‘runs’ and ‘often’ respectively. (t, m) represents the dependency between the word ‘the’ and ‘man’ and so on. Table 3.1 and 3.2 show the realization process. We are not presenting the dependency relation and the feature structures associated with RTG rules. Also we are just showing those steps which are required for the derivation.

Id	RTG Rule
<i>A</i>	$NP_{Adj} \rightarrow the(NP_{Adj})$
<i>B</i>	$NP_{Sub} \rightarrow man(NP_{Adj})$
<i>C</i>	$S_{Sub} \rightarrow runs(NP_{Sub}VP_{Adj}S_{Adj})$
<i>D</i>	$VP_{Adj} \rightarrow often(VP_{Adj})$
<i>E</i>	$NP_{Adj} \rightarrow \epsilon$
<i>F</i>	$VP_{Adj} \rightarrow \epsilon$
<i>G</i>	$S_{Adj} \rightarrow \epsilon$

Table 3.1: DRTGen: Lexical Selection (“The man often runs”)

Id	Rule in progress	D_{Rule}	(C_{Rule}, P_{Rule})	detail
0	$S' \rightarrow \bullet S_{Sub}$	\emptyset	$(-, -)$	axiom
1	$S_{Sub} \rightarrow runs(\bullet NP_{Sub}VP_{Adj}S_{Adj})$	\emptyset	$([(m,r),(o,r)], \emptyset)$	prediction[0], <i>C</i>
2	$NP_{Sub} \rightarrow man(\bullet NP_{Adj})$	\emptyset	$([(t,m)], \emptyset)$	prediction[1], <i>B</i>
3	$NP_{Adj} \rightarrow the(\bullet NP_{Adj})$	\emptyset	(\emptyset, \emptyset)	prediction[2], <i>A</i>
4	$NP_{Adj} \rightarrow \bullet$	\emptyset	(\emptyset, \emptyset)	prediction[2], <i>E</i>
5	$NP_{Adj} \rightarrow the(NP_{Adj}\bullet)$	\emptyset	(\emptyset, \emptyset)	completion[3,4]
6	$NP_{Sub} \rightarrow man(NP_{Adj}\bullet)$	$[(t,m)]$	(\emptyset, \emptyset)	completion[2,5]
7	$S_{Sub} \rightarrow runs(NP_{Sub}\bullet VP_{Adj}S_{Adj})$	$[(t,m),(m,r)]$	$([(o,r)], \emptyset)$	completion[1,6]
8	$VP_{Adj} \rightarrow often(\bullet VP_{Adj})$	\emptyset	(\emptyset, \emptyset)	prediction[7], <i>D</i>
9	$VP_{Adj} \rightarrow \bullet$	\emptyset	(\emptyset, \emptyset)	prediction[7], <i>F</i>
10	$VP_{Adj} \rightarrow often(VP_{Adj}\bullet)$	\emptyset	(\emptyset, \emptyset)	completion[8,9]
11	$S_{Sub} \rightarrow runs(NP_{Sub}VP_{Adj}\bullet S_{Adj})$	$[(t,m),(m,r),(o,r)]$	(\emptyset, \emptyset)	completion[7,10]
12	$S_{Adj} \rightarrow \bullet$	\emptyset	(\emptyset, \emptyset)	prediction[12], <i>G</i>
13	$S_{Sub} \rightarrow runs(NP_{Sub}VP_{Adj}S_{Adj}\bullet)$	$[(t,m),(m,r),(o,r)]$	(\emptyset, \emptyset)	completion[11,12]
14	$S' \rightarrow S_{Sub}\bullet$ (Goal)	$[(t,m),(m,r),(o,r)]$	$(-, -)$	completion[1,13]

Table 3.2: DRTGen: Realization of “The man often runs”

rtgSelected/2 rtgSelected(Rule, Lemma)
<p>where</p> <p>Rule = rule/5</p> <p>Rule = rule(RuleId, TreeName, Anchor, LHS, RHS)</p> <p>RuleId = Prolog Atom, Rule Id</p> <p>TreeName = Prolog Atom, TAG tree name</p> <p>Anchor = anchor/3</p> <p>Anchor = anchor(FamilyName, BotFS, Semantics)</p> <p>FamilyName = Prolog Atom, TAG family name</p> <p>BotFS and Semantics of Anchor are not used in DRTGen</p> <p>LHS = var/5</p> <p>LHS = var(Type, Cat, BotFS, TopFS, NodeName)</p> <p>Type = Prolog Atom, auxiliary or initial</p> <p>Cat = Prolog Atom, Syntactic category (s, np etc)</p> <p>BotFS = Prolog Term, fs/n, n is automatically defined</p> <p>TopFS = Prolog Term, fs/n</p> <p>NodeName = Prolog Atom, node name in TAG tree</p> <p>RHS = Prolog Term, a list of rhs/2</p> <p>rhs/2</p> <p>rhs(Id, Var)</p> <p>Id = Prolog Atom, RHS id</p> <p>Var = Prolog Term, var/5</p> <p>Lemma = lemma/1</p> <p>Lemma = lemma(LemmaName)</p> <p>LemmaName = Prolog Atom, Lemma associated to this rule</p>

Table 3.3: DRTGen Knowledge Base: rtgSelected/6

agendaEntry/8 agendaEntry(AgendaId, Info, Anchor, LHS, RHS_Rest, RHS_Done, DepList, DepLemma_Rest)
chartEntry/8 chartEntry(ChartId, Info, Anchor, LHS, RHS_Rest, RHS_Done, DepList, DepLemma_Rest)
where AgendaId = Prolog Atom, Id in Agenda ChartId = Prolog Atom, Id in Chart Info = info/3 Info = info(RuleId, TreeName, Lemma) Anchor = anchor/3 LHS = var/5 RHS_Rest = [rhs/2 [...]] RHS_Done = [[(ChartId, Info, NodeName) [...]] [...]] DepList = Subset of \mathcal{D} , Dependency covered DepLemma_Rest = [ChildDepLemma, ParentDepLemma] ChildDepLemma = Subset of \mathcal{D} , Child dependent ParentDepLemma = Subset of \mathcal{D} , Parent dependent The terms which are not defined here, are already defined in Table 3.3

Table 3.4: DRTGen Knowledge Base: agendaEntry/8 and chartEntry/8

Prolog implementation of DRTGen

DRTGen has been implemented in *Prolog*. Algorithm 5 presents the description of DRTGen in pseudo codes. The Prolog knowledge-base *rtgSelected* and, *agendaEntry* and *chartEntry* are explained in Table 3.3 and Table 3.4 respectively.

The term *rtgSelected/2* shown in Table 3.3 consists of an RTG rule and its associated lemma. The *LHS* of the RTG *Rule* is a variable (*var/5*) whereas the *RHS* of RTG *Rule* is a list of *rhs(Id, var/5)*. The *var/5* defines a particular node in the TAG elementary tree and consists of *Type* (auxiliary tree or initial tree), *Cat* (syntactic category), *BotFS* (bottom feature structure), *TopFS* (top feature structure) and *NodeName* (position of this node in the corresponding elementary tree). The *var/5* of *LHS* defines if a given rule (or the corresponding elementary tree) is of type initial or auxiliary. The *var/5* in the list of nodes in *RHS* defines if its a substitution or an adjunction site.

The content of the term *agendaEntry* and *chartEntry* shown in Table 3.4 are the same except *AgendaId* and *ChartId*. They are used to show that a particular item belongs to an *agenda* or a *chart*. These items show the progress of an RTG rule described in

Algorithm 2, 3 and 4. *AgendaId* and *ChartId* shows the count of this item in the *agenda* and the *chart* respectively. The *Info* gives the details of the lemma and the TAG tree name associated with this item. The *LHS* is same as the *LHS* of *rtgSelected/2*. The *RHS_Rest* keeps track of the right hand side nodes after \bullet (same as the *RHS* of *rtgSelected/2* in format). The *RHS_Done* keeps track of the right hand side nodes before \bullet . It is a list of lists. For each of the completed nodes, *RHS_Done* has a list of information like *ChartId* which has completed this node, *Info* of that chart item and *NodeName* of node completed. For a completed substitution node, the corresponding list has just one entry. But for a completed adjunction node, the corresponding list may have multiple entries (multiple adjunction). The *DepList* is the dependency covered (D_{Rule} a subset of \mathcal{D}) by this rule. *DepLemma_Rest* are the remaining dependents of the lemma associated with this rule. *DepLemma_Rest* is a pair of *ChildDepLemma* C_{Rule} and *ParentDepLemma* P_{Rule} .

The *lexical selection* selects the RTG rules associated with the lemmas present in the input dependency tree \mathcal{D} and stores them in the *Prolog* knowledge base as *rtgSelected*. DRTGen starts with **INITIALIZE AGENDA** (Algorithm 5) by selecting the *rtgSelected* associated with the possible root lemmas of the derivation tree output. Note that the functionality of **INITIALIZE AGENDA** is the same as an axiom in the deductive framework for DRTGen. It directly converts the *rtgSelected*(*rule*(*RuleId*, *TreeName*, *Anchor*, *LHS*, *RHS*), *Lemma*) to *agendaEntry*(*AgendaId*, *info*(*RuleId*, *TreeName*, *Lemma*), *Anchor*, *LHS*, [], *RHS*, D_{Lemma} , (\mathcal{C}_{Lemma} , \mathcal{P}_{Lemma})) and pushes them in the *agenda*. After the initialization step, DRTGen calls the **PROCESS AGENDA**. The “process agenda” pops an item from the agenda and processes using the prediction and completion steps described in Algorithm 3 and 4. This continues until the agenda is empty. We refer the reader to Algorithm 5 for the detailed steps of DRTGen. Note that in the *process complete items* of Algorithm 5, only those items are considered which has covered all of its dependents. This way it filters out all the sub-trees which can never lead to the complete derivation. In the *process incomplete items* of Algorithm 5, the predicted new items are checked for their availability in the chart and if the predicted new item is already in the chart, we check for the completion of the item using this predicted new item, instead of pushing them to the agenda. This way DRTGen rules out the repetition of the same processes.

3.1.5 String Extraction

String Extraction is the last step of the DRTGen realization system. It traverses through the *chart* looking for the goal items ($D_{item} = \mathcal{D}$ and $Item_{dep} = (\emptyset, \emptyset)$) (Algorithm 2). For a given goal item (the root tree of the derivation tree), it traverses through its *RHS_Done* to find out its children trees in the dependency tree. This way it builds up the derivation

Algorithm 5 DRTGen Realization Algorithm: Pseudo code

INITIALIZE AGENDA

- Select *rtgSelected* with $LHS = (init, s, -, -, -)$, possible root nodes
- Convert all selected to *agendaEntry* and **Push** them in *Agenda*

PROCESS AGENDA

if *Agenda* is *Empty* **then**

Terminate

else

Pop an *item* from *Agenda*

if *item* \notin *Chart* **then**

Push *item* in *Chart*

end if

if *item* is *Complete* (*RHS_Rest* is *empty*) **then**

Process Complete Item

if Dependents of *item* covered (*DepLemma_Rest* is *empty*) **then**

 Complete items in *Chart* using the completion rules (**Algorithm 4**)

Push newly generated item in *Agenda*

 Back to **PROCESS AGENDA**

else

 Back to **PROCESS AGENDA**

end if

else

Process Incomplete Item

 Predict new items with this *item* using the prediction rules (**Algorithm 3**)

for all *new_item* \in *New Predicted Items* **do**

if *new_item* \in *Chart* **then**

if *new_item* is *Complete* **then**

 Complete *item* with *new_item* using the completion rules (**Algorithm 4**)

Push newly generated item in *Agenda*

 Back to **PROCESS AGENDA**

else

 Back to **PROCESS AGENDA**

end if

else

Push *new_item* in *Agenda*

 Back to **PROCESS AGENDA**

end if

end for

end if

end if

tree. The *RHS_Done* is enriched with the node position (*NodeName* in the corresponding TAG elementary tree) information for each completed operation. This information has been used for the mapping from the TAG derivation tree to the TAG derived tree. Yields of TAG derived trees are further analyzed morphologically using the grammatical information (e.g., pos, number, gender, partic and tense, shallow dependency representation of Generation Challenge 2011) associated with each lemma to produce final grammatical string or natural language sentence.

On multiple occasions, a lemma in the dependency tree has multiple dependent modifiers (for example, in “beautiful kind french wife”: “beautiful”, “kind” and “french” are dependent modifiers of “wife”). DRTGen handles them using *multiple adjunction* and *packing* (Section 3.2.4). Similar situations may occur with dependent prepositional phrases. In the derived tree, the sister dependents with multiple adjunction may appear in random order. DRTGen expects that the problem of the *ordering of modifiers* or the *ordering of prepositional phrase* are resolved at the *String Extraction* level. In the current version of our system, this problem has not been analyzed intensively and the *String Extraction* level outputs the strings (with multiple adjunction on same node) in the same order as they appear in the input dependency tree.

3.2 DRTGen: Implementation Issues and Variations

In the previous section we presented the core of an *RTG based Surface Realizer from Dependency Representation* or *DRTGen*. In this section, we present its variation and other important implementation issues.

3.2.1 Preprocessing of Input Dependency Structure

DRTGen builds on the fact (Section 2.4.1) that corresponding to each dependent lemma pair in a dependency tree \mathcal{D} , we have a TAG derivation process between the same lemma pair in the corresponding derivation tree. Only thing is missing is “which derivation process” (substitution or adjunction) and “its relative position” in the TAG derivation tree. But different formalism can adopt different ways to represent dependency structures. In that case, we need to narrow down this gap by preprocessing of the input dependency representation.

In the shallow dependency representation of the Generation Challenge 2011, we found some mismatches. So we have preprocessed the input dependency to narrow down this gap. Also some other kinds of rewriting have been done. We are presenting them in the

following descriptions.

Case of auxiliary verb In the presence of the auxiliary verb, the dependency relation *SBJ* is shown on the auxiliary verb and not on the main verb. But in case of TAG derivation tree, the main verb takes the *SBJ* argument. So these dependencies on the auxiliary verb needed to be transferred to the main verb. For example the dependency tree [will [go] [i]] has been converted to [go [i] [will]].

Case of 's In TAG, the apostrophes s ('s, A TAG adjoining tree) is an adjunct to the main noun to be modified and the elementary tree of 's accepts the modifier. But in the shallow dependency representation, 's is a dependent of the modifier noun. For example “The president 's wife” in shallow representation is presented as [wife [president ['s] [the]]]. To accept this by TAG, we process these kinds of format and convert them to of [wife ['s [president the]]] format.

Grouping Named Entity In the Generation Challenge data, named entities are with dependency relation “name_1”, “name_2”, “name_3” etc. For each occurrence, we have combined them together and kept them as one lemma with dependency relation “name”.

Handling Repeated Lemma Even if a lemma is repeated in a dependency tree, its dependent and its position in the dependency make it unique. To make it more explicit, and to recognize each lemma independently in DRTGen, we combine each instance of the repeated lemma with a count. For example, if “apple” occurs twice in a dependency tree then new rewritten nodes in that dependency tree for apple are (“apple”, 1) and (“apple”, 2). Using this new representation, DRTGen can directly search for a particular lemma even if its repeated. This representation defined each lemma of dependency tree uniquely.

3.2.2 Multiple Adjunction

One of the important feature of DRTGen is its power of multiple adjunction (Algorithm 4). This power enables DRTGen to handle the cases of multiple dependent modifiers and multiple dependent prepositional phrase without rewriting of the input dependency structure. For example in “beautiful kind french wife”, “beautiful”, “kind” and “french” are dependent modifiers of “wife” and in sentence “I played for two hours with a bat”, “with a bat” and “for two hours” are dependent prepositional phrases to “play”. These situations are handled in DRTGen very easily.

3.2.3 Guided with Dependency Structure

DRTGen is guided by the structure of the input dependency representation (Algorithm 3 and 4). For each lemma, DRTGen just traverses through its uncovered dependents. This provides a top-down guidance for the surface realization process. In each rule in chart or agenda, it maintains the list of dependents left to cover. The dependent list gets updated after each prediction or completion steps.

3.2.4 Intersective Modifiers : Packing

The problem of intersective modifiers is a well known problem for generation systems. [Kay (1996)] describes it in the scenario of generation from flat semantics showing how intersective modifiers, because they all modify the same item, may lead to an exponential number of intermediate structures being constructed. The problem remains the same in the dependency structure approach (no order in sister nodes). As this is derived by allowing multiple adjunction, for a given set of n modifiers all modifying the same structure. The system can end up generating all possible 2^n intermediate structures. Lets go back to our example of “beautiful kind french wife”. In the worst case, it will produce all 8 intermediate structures “wife”, “beautiful wife”, “kind wife”, “french wife”, “beautiful kind wife”, “beautiful french wife”, “french kind wife” and “beautiful kind french wife”.

To resolve this problem, DRTGen uses an strategy with which it can look if a multiple adjunction is in progress. If yes, it completes that item which has the highest number of multiple adjunction applied. For example, the chart has an item “wife”, then “beautiful” combines to “wife” with multiple adjunction and produces “beautiful wife”. Now “kind” can combine with both “wife” and “beautiful wife”. But by looking at these two instances, DRTGen knows that the multiple adjunction is in process so it just combines with “beautiful wife” and produces “beautiful kind wife”. The process continues with “french” and finally produces “beautiful kind french wife”.

To achieve this, DRTGen uses a special data-structure for *RHS_Done* (Table 3.4). It is a list of lists to keep track of the completed right hand side nodes of an item in the chart or the agenda. For each of the completed nodes, *RHS_Done* has a list of information (*ChartId*, *Info*, *NodeName*). For a completed substitution node, corresponding list has just one entry. But for a completed adjunction node, the corresponding list may have multiple entries because of the multiple adjunction. *RHS_Rest* stores the nodes of right hand side of corresponding RTG rule which are still not complete. DRTGen computes the sum of the number of left nodes present in *RHS_Rest* and the number of the lists present in *RHS_Done*. If the count is bigger than the count of the nodes present in the *RHS*

of corresponding RTG rule, then DRTGen knows that the multiple adjunction has been initialized. So instead of combining with every possible instance, it looks for the instance with *RHS_Done* which is super-set of all other *RHS_Done* of possible instances. DRTGen completes the selected item only.

3.2.5 Handling Co-anchors

Some of the lexicalized TAG elementary trees have more than one associated lemma. The primary lemma (the anchor) defines the corresponding elementary tree and other lemma (the co-anchor) is associated with the tree structure to give other functionality. Figure 3.4 shows such an example of the TAG elementary tree. In Figure 3.4, “go” is an anchor and “to” is a co-anchor. DRTGen assigns covered dependencies ($\neq \emptyset$) to these types of trees during initialization. The tree in Figure 3.4 is assigned to $D_{Rule} = [(-, go, to, verb)]$ (- unifies with any value) during initialization. If D_{Rule} is not the member of the dependency input \mathcal{D} , the associated tree will not be initialized. During generation process, the corresponding tree considers dependents of the anchor and all of the co-anchors.

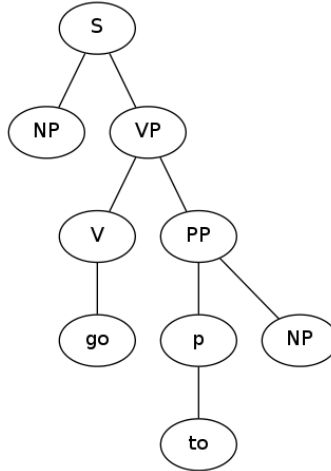


Figure 3.4: Elementary TAG tree with co-anchor

In this chapter, we present DRTGen which takes the dependency structure as an abstract linguistic input representation and outputs the surface sentences. The syntactically specific form of dependency representation provides us with a more constrained approach to surface realization than flat semantics.

In summary, the lexical selection using lemma and associated POS tags, the valid-

ification of selected lexicon with the dependency structure, realization process governed with the structure of the dependency structure and the advantages of chart parsing are the key techniques behind DRTGen.

Chapter 4

Implementation, Results and Discussion

In Chapter 3, we proposed an *RTG based Surface Realizer from Dependency Representation* or *DRTGen*. In this chapter, we illustrate the working of the algorithm with an example. We start with a brief explanation of the input resources used in Section 4.1. We discuss the details of the grammar, the lexicon and the input dependencies used. In Section 4.2, we show a complete run of DRTGen with an example. We present and analyze the results at the different stages of DRTGen.

4.1 DRTGen: Input Resources

The architecture of DRTGen (shown in Figure 3.1) uses two input resources: The *dependency structures* to realize and a *feature based RTG encoded TAG*. For the *feature based RTG encoded TAG*, we use the formalism described in Section 2.3.2 and 3.1.2 [Schmitz and Le Roux (2008)] to convert the TAG elementary trees to RTG rules. For the compact representation of TAG, the TAG elementary trees are not lexicalized. We convert this TAG representation to RTG rules. We use the lexicon as a separate file which maps from a lemma to its associated TAG elementary trees. During the lexical selection process 3.1.3 of DRTGen, both the lexicon and the RTG rules are combined. For a given lemma, we choose associated RTG rules using the lexicon file. We anchor RTG rules with the lemma and store in the *Prolog* knowledge-base. The following section gives a small description of different input resources used:

Shallow dependencies from Generation Challenges 2011 SR Task

For the input dependency structures, we have used the shallow data from *Generation Challenges 2011 Surface Realization Shared Task*. This dataset has been built from the Penn Treebank. There are around 40000 dependency trees available in this dataset. To analyse and explain the performance of the system easily, we tested our system with the chunked data. We chunked the input data in NPs, PPs and Clauses of different sizes. The chunking was performed by retrieving the Penn Treebank (PTB) and their alignment between words and dependency tree nodes provided by the organisers of the SR Task. We shall present the numeric details in the following sections.

The obtained dependency tree is converted to a list, *Prolog* format compatible with DRTGen’s Prolog implementation.

The Lexicon

The lexicon can be viewed as the mapping between the lexical items and the TAG elementary trees. The lexicon used for DRTGen is obtained from the existing XTAG lexicon. Both lexicons (the lexicon used by DRTGen and the XTAG lexicon) differ in the way they describe POS tags, verb families, features, feature values, equations and co-anchors. Various mapping files have been used to map the XTAG lexicon to the lexicon accepted by DRTGen. DRTGen accepts the lexicon file as a Prolog knowledge-base file with multiple entries of *lexEntry/6* (Equation 4.1). Equation 4.2 shows an example of *lexEntry/6*. The basic lexicon (directly mapped from the XTAG lexicon) has **71327** (lemma, TAG family) pairs for **17315** unique lemmas. The degree of lexical ambiguity in this lexicon is 4.11.

$$\textit{lexEntry}(\textit{LexLemma}, \textit{FamilyName}, \textit{Equation}, \textit{Coanchors}, \textit{Filters}, \textit{XTAGPOS}) \quad (4.1)$$

$$\begin{aligned} \textit{lexEntry}(\textit{"break"}, \textit{"n0VDN1"}, [], [(\textit{"D1"}, \textit{"a"}, \textit{"det"}), (\textit{"N1"}, \textit{"leg"}, \textit{"n"})], \\ [\textit{family} : \textit{"n0VDN1"}, \textit{"v"}]) \end{aligned} \quad (4.2)$$

Due to small size of the basic lexicon, it is not enough to cover the lemmas from the Surface Realization task of the Generation Challenge 2011. To overcome this problem, we automatically built up the lexicon for missing lemmas based on some heuristics and statistical approaches. From the basic lexicon, we extracted a mapping file which maps a part of speech to the set of TAG trees/families associated with this part of speech in the lexicon together with the frequency of the association. For each word in the input data

that is not in the lexicon, we then create an entry that associates that word, with the part(s) of speech assigned to that word in the input data and the 5 most frequent TAG trees/families associated with that part of speech in the XTAG lexicon. For example, a missing lemma associated with the parts of speech “n” will be assigned the (alphaNXN, alphaN, betaNn, n0N1, s0N1) TAG trees/families. Thus to cover this lemma, these five entries have been inserted in the automatically built up lexicon. For verbs (part of speech “v”), we took a more differentiated approach to account for the fact that there are roughly 80 verb families in the XTAG grammar¹. So for the lemmas with part of speech “v”, we learned a statistical model using the basic lexicon and the training dependency data. The lemmas present in the dependency trees (the shallow representation of the surface realization task of the Generation Challenge 2011) which are also present in the basic lexicon, have been used to learn a model for the TAG families/trees and their associated frequent dependent (parent and child) dependency relation patterns. For example, for the TAG family “n0Vn1”, the dependency relation patterns can be shown with two vectors “Parent Dependent” \mathcal{V}_{PD} [sroot : 29331, im : 11483, vc : 11106, nmod : 10381, sub : 7203, obj : 7040, conj : 5104, appo : 4343, pmod : 3144, adv : 2072] and “Child Dependent” \mathcal{V}_{CD} [p : 58386, sbj : 53729, obj : 51402, adv : 27997, vc : 23765, NULL : 15185, tmp : 14366, oprd : 8490, coord : 6395, loc : 5775]. Using the dot product similarity over this model, a missing lemma $\langle v_{PD}, v_{CD} \rangle$ in the dependency tree input with the parts of speech “v” can be assigned to a set of most probable TAG families/trees. The final complete lexicon contains **196391** entries of (lemma, TAG family) pairs for **40130** unique lemmas. The degree of lexical ambiguity in this lexicon is 4.89.

The Grammar

The grammar used for the testing of DRTGen is *SemXTAG*. It is a feature based TAG augmented with unification based semantics [Gardent and Kallmeyer (2003)]. Note that DRTGen ignores the associated semantics. This grammar is compiled from a *XMG* (eX-tensible MetaGrammar). According to this formalism, the basic units (classes) are combined using inheritance, conjunction and disjunction to produce *SemXTAG* elementary trees. We convert *SemXTAG* elementary trees to RTG rules using the formalism described in [Schmitz and Le Roux (2008)]. This generates a *Prolog* knowledge-base with multiple entries of *rule/5*, each of *rule/5* corresponds to a TAG elementary tree. Equation 4.3 shows the format of *rule/5*. Detailed explanation of each entry of *rule/5* is explained in Table 3.3.

¹We tried to generalise this approach to all parts of speech, but the results for non verbs were worse than those obtained using the simple frequency heuristics just described.

$$rule(TreeId, TreeName, Anchor, LHS, RHS) \quad (4.3)$$

SemXTAG has the same coverage as big as XTAG. Like XTAG, it contains around 1300 elementary trees and covers auxiliaries, copula, raising and small clause constructions, topicalization, relative clauses, infinitives, gerunds, passives, adjuncts, ditransitives and datives, ergatives, it-clefts, wh-clefts, PRO constructions, noun-noun modification, extraposition, sentential adjuncts, imperatives, and resultatives.

4.2 DRTGen: A complete run

This section present a complete run of DRTGen. We select an abstract linguistic input in the form of shallow dependency representation for surface realization by DRTGen. We present the results and their analysis for the major steps taken by the realizer. The selected dependency test case for the sentence “The president’s wife will be receiving a gift from the queen.” is shown in Figure 4.1. We have tested this input with a small grammar (a small fragment of SemXTAG).

Preprocessing of Input Dependency

The input dependency data shown in Figure 4.1 has been processed to narrow down the gap between dependency structures and derivation trees. The processed dependency is shown in Figure 4.2. In the current example, we have apostrophe s dependent of the modifier noun “president”. In the processed version, “s” is dependent of head noun “wife”. The “sbj” dependency relation has been shifted from the auxiliary verb “will” to the main verb “receive”. The lemma repetition is also handled during preprocessing stage. It assigned each lemma a number to define uniquely among all other entries. unique among its repeated lemma. For example in Figure 4.3, “the” is repeated twice. So we assign the first lemma to be (the, 1) and the other one to be (the, 2) Figure 4.2. This way DRTGen recognizes each lemma and its dependent unambiguously.

The processed dependency output is converted to a list of dependency relations. \mathcal{D} (e.g., [(vc, (received,1), (be,1), v)), ...] to match the Prolog requirements of DRTGen.

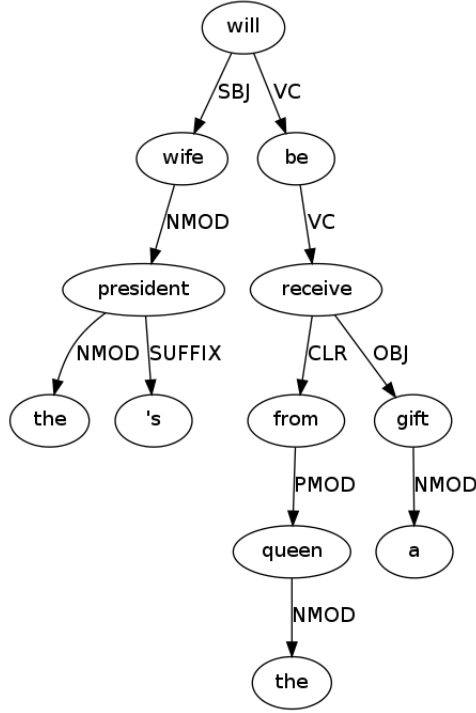


Figure 4.1: Generation Challenge: Actual Data

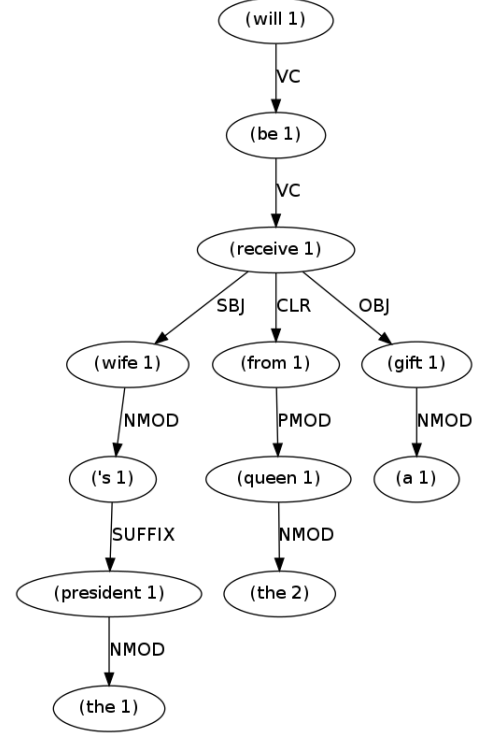


Figure 4.2: Generation Challenge Data: After processing

Figure 4.3: The president's wife will be receiving a gift from the queen

Lexical Selection and Anchoring of RTG rules

Table 4.1 presents the result of lexical selection. Each lemma is associated with a set of selected TAG families from the lexicon. For each TAG family, we might have multiple entries for TAG elementary trees with different type and category. For example “receive” is associated with “n0Vn1” in lexicon and in RTG, there are 17 rules for “n0Vn1” with 6 of them are initial trees with category s, 3 of them are initial tree with category np and 8 of them are auxiliary trees with category np.

We see that lexical ambiguity is a very big problem for surface realizer. Even with the small grammar, we get so many instance for a word. So to optimize, we have to have an efficient filter mechanism at various steps.

Lemma	Associated TAG families
will	betaVv [(aux, vp): 1]
be	betaVv [(aux, vp): 1]
receive	n0Vn1 [(init, s): 6, (init, np): 3, (aux, np): 8]
wife	noun [(init, np): 1]
s	betanGn [(aux, np): 1]
president	noun [(init, np): 1]
the	betaDn [(aux, np): 1]
from	betavPn [(aux, vp): 1], betanPn [(aux, np): 1]
queen	noun [(init, np): 1]
gift	noun [(init, np): 1]
a	betaDn [(aux, np): 1]

Table 4.1: DRTGen: Lexical Selection and Anchoring RTG rules

Without Coanchor	With coanchor
(Tn0Vn1-23, lemma(receive, 1))	(Tn0Vn1-23, lemma(receive, 1))
(Tn0Vn1-26, lemma(receive, 1))	(Tn0Vn1-26, lemma(receive, 1))
(Tn0Vn1-48, lemma(receive, 1))	
(Tn0Vn1-49, lemma(receive, 1))	
(Tn0Vn1-50, lemma(receive, 1))	
(Tn0Vn1-54, lemma(receive, 1))	

Table 4.2: DRTGen: Initialization (With and Without Coanchor)

Initialization of DRTGen

To avoid running with all selected RTG rules as possible root nodes in TAG derivation tree, we apply a simple filter that only possible root nodes are initial trees with category “s”. This drops down our search space to just 6 instances (6 instances of (init, s) for receive).

During the initialization itself, it refines its search space further. Four of the selected trees are with co-anchor “by”, so these tree will be initialized with a dependency (–, receive, by, v). But this dependency is not present in \mathcal{D} . So these all 4 trees are dropped out (Table 4.2).

TAG Derivation Trees

DRTGen goes ahead with the two final possible root trees. Figure 4.6 shows the final output derivation trees generated by DRTGen. As we can see the structures are very

similar to dependency input but it is enriched with derivation process details. Figure 4.7 shows the derived tree generated from the derivation trees shown in Figure 4.4. Figure 4.8 shows the derivation with all epsilon productions for Figure 4.5.

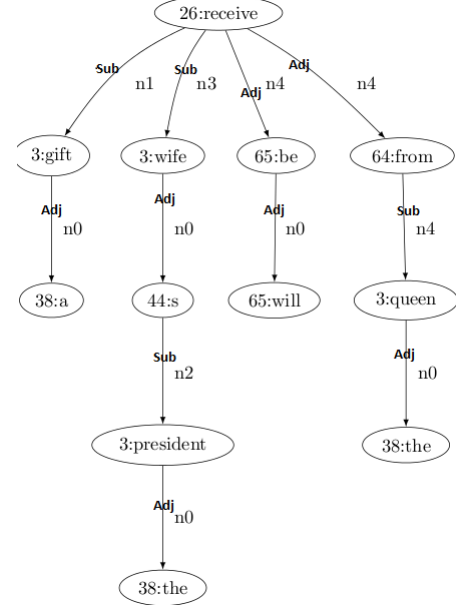
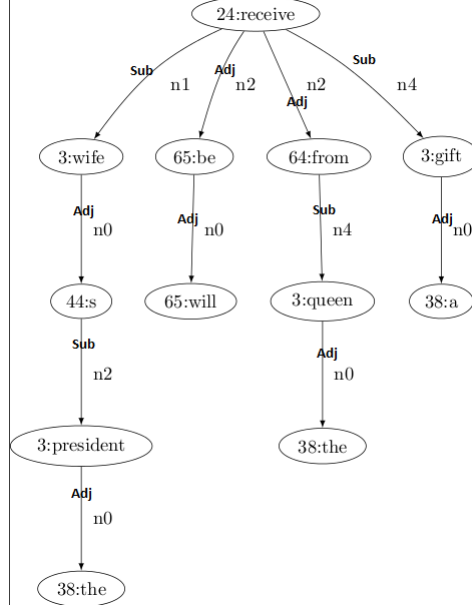


Figure 4.4: Derivation Tree with Tn0Vn1-24

Figure 4.5: Derivation Tree with Tn0Vn1-26

Figure 4.6: DRTGen Output Derivation Tree

These two output derivation trees are converted to the corresponding derived tree and we get the output strings “The president ’s wife will be receiving a gift from the queen” (Figure 4.4) and “A gift the president ’s wife will be receiving from the queen” (Figure 4.5). Note that we could not get the “receiving” from “receive” because right now we are completely dependent on the Generation Challenge Data, and the grammatical information provided with the input was not enough to morphologically analyze words correctly.

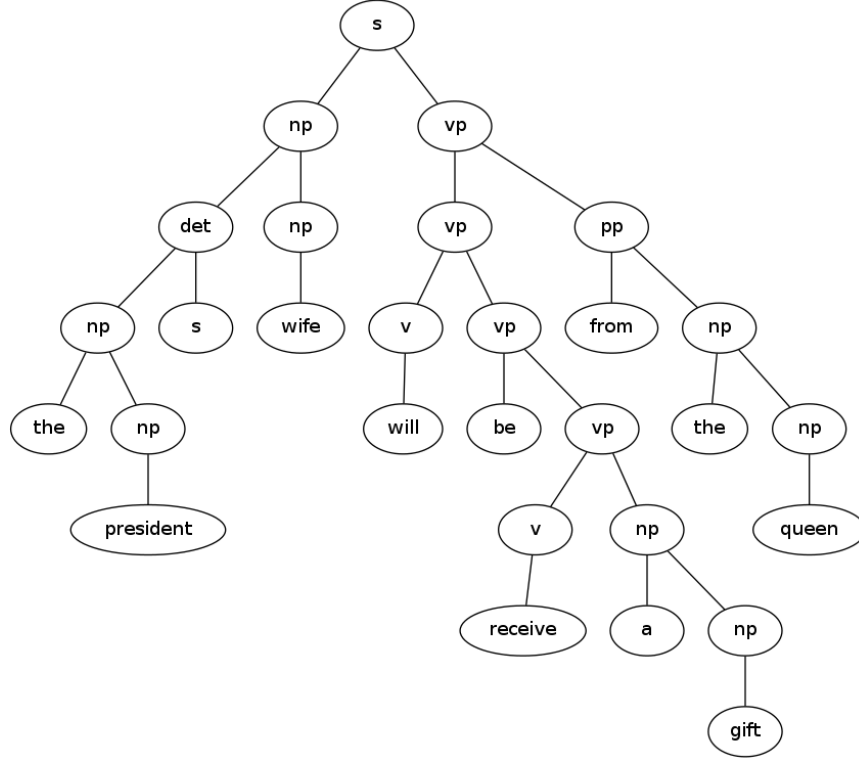


Figure 4.7: DRTGen output: Derived Tree (Figure 4.4)

4.3 DRTGen: Running on Chunks

The performance of DRTGen on the actual shallow dependencies (without rewriting) was not very good. On the careful result analysis, we found that there were few mismatches between the dependency representation and the input expected by our grammar. Because these mismatches were very frequent, they caused a considerable percentage of the input data to fail during generation. In the later experiment, to analyze and explain the performance of the system easily, we tested our system with the chunked data. We chunked the input data in NPs, PPs and Clauses of different sizes. The chunking was performed by retrieving the Penn Treebank (PTB) and their alignment between words and dependency tree nodes provided by the organisers of the SR Task.

Table 4.3 shows the experiments done with NP chunks (size 4, NP-4). We were able to extract a total of 24995 NP-4 dependency trees. The numbers shown in the table are the failure counts during generation process. There were other failures like lexicon missing or TAG family missing, we ignore them in this experiment as we mainly focus on the

NP-4	Generation Failure
SR Actual Data	8361
Converted SR Data	5255

Table 4.3: DRTGen run on NP-4

S-6	Generation Failure
SR Actual Data	2514
Converted SR Data (just for 's)	2433
Converted SR Data (just for VC)	2172
Converted SR Data (both for VC and 's)	2081

Table 4.4: DRTGen run on S-6

performance of our generation algorithm. On the actual SR data (without any rewriting), generation failed on 33% of these input. Converting the input data to the correct input format to resolve the mismatch induced by possessive reduces generation failure to 21%. For this data of NP-4, there were 3264 instances of possessive ('s) that were rewritten. We also did a similar experiment with sentential clause of size 6 (S-6). Results are presented in Table 4.4. We were able to extract 4049 instances of S-6. This data set had 163 instances of 's rewriting and 1211 instances of auxiliary-main verb folding.

We see that the rewriting of the input data just for 's and auxiliary verb cases improves the accuracy of our system by a significant margin. This gives an idea of the impact of possible mismatches between SR dependency representation and expected input structure by our grammar on our generation system. We still need to do large scale testing of DRTGen and find other possible mismatches, that can lead to an interesting research topic itself.

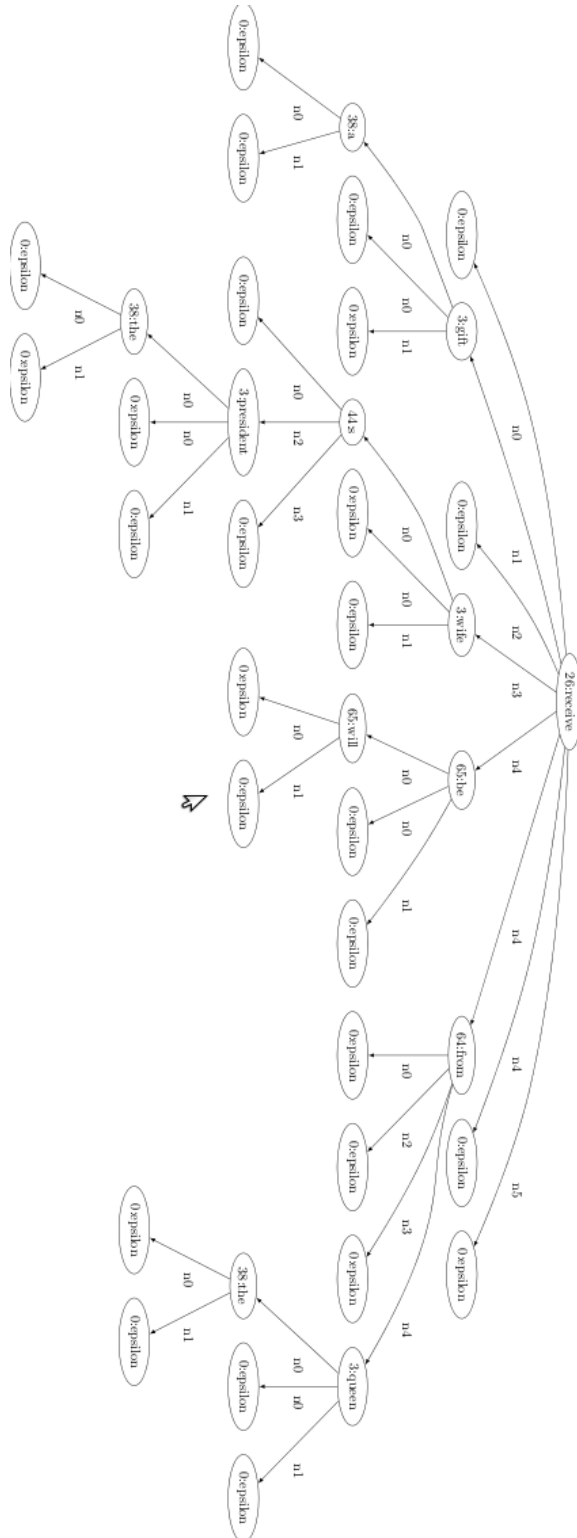


Figure 4.8: DRTGen output with epsilon production (Figure 4.5)

Chapter 5

Conclusion and Future Research Ideas

We presented a surface realizer “DRTGen” based on a Regular Tree Grammar encoding of Tree Adjoining Grammar. DRTGen differs from RTGen in several ways:

- It takes as input syntactic dependency trees rather than semantic formulae . As shown in Chapter 3, Equation 3.3, this change can be obtained by modifying the grammar and substituting flat semantic representations with dependency tuples.
- As in RTGen, the core of the algorithm is based on an Earley chart parsing procedure. DRTGen differs from RTGen however in that it integrates stronger top-down guidance. The prediction and the completion steps are closely instructed with the local dependency to cover and the global dependencies covered. This constrained approach towards generation makes it more efficient compare to RTGen.
- Another difference between RTGen and DRTGen is that DRTGen allows multiple adjunctions thereby reducing the differences between the dependency trees provided by the Generation Challenge and our grammar derivation trees. This was obtained by modifying the completion rule as shown in Figure 4 (Chapter 3).
- DRTgen also includes a special treatment of co-anchors again so as to handle the divergences between the GenChall dependency trees (which handle co-anchors as standard lemmas) and TAG derivation trees (where coanchors do not appear since they are part of elementary trees). As explained in Section 3.2.5, co-anchor information is extracted from the elementary TAG trees and taken into account when calculating the dependencies covered by a completion step.

- As shown in Section 3.2.4, DRTgen also permits a more efficient treatment of intersective modifiers than RTGen by keeping track of structures involving multiple adjunctions and always completing the item with the highest number of multiple adjunctions.

To test our system on the data provided by the Generation Challenge 2011 Surface Realization Shared Task, we furthermore developed a large scale lexicon partly, by conversion from the already available XTAG lexicon and partly, through learning from the Generation Challenge data.

The approach to surface realization from dependency representation presented in this thesis opens up multiple research ideas. Based on the research done in this dissertation, we summarize following future works:

Improving String Extraction The multiple modifiers modifying the same word can be adjoined in any order without any restriction during multiple adjunction. This can lead to random order of modifiers in the derived tree and hence a surface sentence with randomly ordered modifiers. This problematic situation is also true for the case of prepositional phrases attached to the same node. In the current version of our system, this problem has not been analyzed intensively and the *String Extraction* level outputs the strings (with multiple adjunction on same node) in same order as they appear in the input dependency tree. We need to investigate different efficient techniques to order the multiple modifiers and prepositional phrases. For the start, we can look into statistical techniques for ordering prenominal modifiers [Mitchell (2010)].

Also the current morphological analysis of a word is dependent on the grammatical information available in the input dependency data. But this information is not very consistent. To improve our *String Extraction*, we can modify DRTGen so that it carries out full morphological realization taking into account both the morphological information present in the input and the constraints imposed by the feature structures in the grammar.

Further optimization of Realization Algorithm In the current version of DRTGen, except few dependencies “SUB” and “OBJ”, other dependencies have not been considered properly. One reason behind this is, we are still in the process of augmenting our TAG with dependency relations but we believe that certain dependencies can be used to constrain our realization system in particular, dependencies licensed by function words. Dependencies licensed by complements and adjuncts are more difficult to handle and will require a detailed analysis of how the distinction between complements and adjuncts embodied in the Generation Challenge data differ from that made in the TAG grammar.

Large scale testing of DRTGen The surface realizer DRTGen has so far been tested on a small number of test cases from the shallow representation of dependency data (Generation Challenge 2011). To make its importance and presence count, it would be necessary to test the realization system with the complete dependencies data from shallow representation of Generation Challenge 2011 data. This will also lead to the full fledged evaluation of the system.

To achieve this, several issues need to be addressed:

- The realizer must be extended with a robustness mechanism for handling missing entries i.e., words that are not in the lexicon.
- The mismatches between TAG derivation trees (the trees produced by our grammar) and the shallow dependency trees provided by the Generation Challenge need to be identified and handled in a systematic manner. Here we plan to use tree based error mining technique [Chi et al. (2004)] to identify tree fragments occurring predominantly in input for which no sentence could be generated.
- The multiple output produced by DRTGen need to be ranked using parse disambiguation and realization ranking techniques such as proposed e.g., in [Charniak and Johnson (2005); Velldal (2008)].
- DRTGen needs to be tested with other existing systems and more specially its parent system RTGen. We need to build up the abstract linguistic input in both flat semantics and dependency trees to achieve this goal. This will also lead to the close study between flat semantics and dependency representations.

Deep Dependency Representation The current version of DRTGen assumes the input in the form of dependency tree. It would be interesting to test our system with the graph representation of deep dependency representation.

Evaluation So far DRTGen has been tested manually for the selected test case and small chunk data. It would be very promising to test it with various well known automatic metrics (e.g., BLEU Score). We could also test our results using “DebGram: Manual Annotation System”¹ by making it available for public annotation.

¹<http://talcloria.fr/mastertalc>

Real World Application Dependency structures seems like a very handy way to represent abstract linguistic information. Surface realization using these data representation can be put for testing in real world application. It could be used for wider generation system.

Appendix A

POS tag Mapping: Generation Challenge 2011 and XTAG

Table A.1: POS Tag mapping

XTAG	Generation Challenge 2011
a	[Adjectives] JJ, JJR, JJS
adv	[Adverb] RB, RBR, RBS, WRB
comp	[Complementizer] IN, WDT, DT [for, if, that, whether]
conj	[Conjunction] CC
det	[Determiner] DT, PDT, PP\$, WDT, CD, PRP\$
g	[Genitive] POS
i	[Interjective] UH
n	[Noun] NN, NNS, NNP, NP, NPS, PP, WP, WP\$, EX, PRP
n1	[Deverbalized Noun]
np	[Noun Phrase]
prep	[Preposition] IN
prep1	[Preposition1]
prep2	[Preposition2]
prep3	[Preposition]
prep4	[Preposition]
pl	[Verb Particle] RP
punct	[Punctuation]
punct1	[Punctuation]
punct2	[Punctuation]
v	[Verb] VB, VBD, VBG, VBN, VBP, VBZ, MD
det1	[Determiner1]
det2	[Determiner2]

Bibliography

- A. Abeille. Parsing french with tree adjoining grammar: some linguistic accounts. In *12th International Conference on Computational Linguistics*, Budapest, Hungary, 1988.
- J. A. Bateman. Enabling technology for multilingual natural language generation: the kpml development environment. *Natural Language Engineering*, 1997.
- B. G. C. Gardent and L. Perez-Beltrachini. Using Regular Tree Grammar to enhance Surface Realisation. *Natural Language Engineering*, 2011.
- M.-H. Candito and S. Kahane. Can the tag derivation tree represent a semantic graph?: An answer in the light of meaning-text theory, 1998.
- J. Carroll and S. Oepen. High efficiency realization for a wide-coverage unification grammar. *The 5th International Joint Conference on Natural Language Processing*, 2005.
- E. Charniak and M. Johnson. Coarse-to-fine n-best parsing and MaxEnt discriminative reranking. In *43rd Annual Meeting of the Association for Computational Linguistics*, Ann Arbor, Michigan, 2005.
- Y. Chi, Y. Yang, and R. R. Muntz. Hybrid tree miner: An efficient algorithm for mining frequent rooted trees and free trees using canonical form. In *16th International Conference on and Statistical Database Management, IEEE Computer Society*, Santorini Island, Greece, 2004.
- H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. 2007.
- A. Copestake, D. Flickinger, I. A. Sag, and C. J. Pollard. Minimal recursion semantics-an introduction. *Research on Language and Computation*, 2005.
- J. Earley. An efficient context-free parsing algorithm. *Association for Computing Machinery*, 1970.

- C. Gardent and L. Kallmeyer. Semantic construction in FTAG. In *10th Conference of the European Chapter of the Association for Computational Linguistics*, Budapest, Hungary, 2003.
- C. Gardent and E. Kow. Generating and selecting grammatical paraphrases. In *Proceedings of the 10th European Workshop on Natural Language Generation*, Aberdeen, Scotland, 2005.
- C. Gardent and E. Kow. Three reasons to adopt tag-based surface realisation. In *The Eighth International Workshop on Tree Adjoining Grammar and Related Formalisms (TAG+8)*, Sydney, Australia, 2006.
- C. Gardent and E. Kow. Spotting over-generation suspects. In *Proceedings of the 12th European Workshop on Natural Language Generation*, Dagstuhl, Germany, 2007a.
- C. Gardent and E. Kow. A symbolic approach to near-deterministic surface realisation using tree adjoining grammars. In *Proceedings of the Association for Computational Linguistics*, Praha, Czech Republic, 2007b.
- C. Gardent and L. Perez-Beltrachini. Efficient rtg based surface realisation for tag. In *Proceedings of the International Conference on Computational Linguistics*, Beijing, China, 2010.
- A. Joshi and O. Rambow. A Formalism for Dependency Grammar Based on Tree Adjoining Grammar. In *Proceedings of the Conference on Meaning-Text Theory*, Paris, France, 2003.
- A. Joshi and Y. Schabes. Tree-adjoining grammars. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, pages 69 – 124. Springer, Berlin, New York, 1997.
- A. Joshi, L. Levy, and M. Takahashi. Tree Adjunct Grammars. *Computer Systems Science*, 1975.
- M. Kay. Chart Generation. In *Proceedings of the 34th Annual meeting on Association for Computational Linguistics*, Santa Cruz, California, 1996.
- A. Koller and K. Striegnitz. Generation as dependency parsing. In *the 40th Annual meeting on Association of Computational Linguistics*, Philadelphia, 2002.
- E. Kow. *Surface realisation: ambiguity and determinism*. PhD thesis, University of Henri Poincare - Nancy 1, 2007.

- I. Langkilde and K. Knight. Generation that exploits corpus-based statistical knowledge. In *36th Annual meeting of the Association for Computational Linguistics joint with 17th International Conference on Computational Linguistics*, Montreal, Canada, 1998.
- B. Lavoie and O. Rambow. Realpro-a fast portable sentence realizer. *Proceedings of the Fifth Conference on Applied Natural Language Processing*, 1997.
- C. M. I. M. Matthiessen, J. A. Bateman, and T. Patten. *Text generation and systemic-functional linguistics: Experiences from english and japanese*. London: Pinter Publishers, 1991.
- M. Mitchell. A flexible approach to class based ordering of prenominal modifiers. In *Preceeding of Empirical Methods in Natural Language Generation*. Springer, 2010.
- L. H. Perez. Using regular tree grammars to optimise surface realisation. Master’s thesis, Erasmus Mundus Master - Free University of Bozen-Bolzano, 2009.
- O. Rambow and A. Joshi. A Formal Look at Dependency Grammars and Phrase-Structure Grammars, with Special Consideration of Word-Order Phenomena. *Leo Wanner, ed., Current Issues in Meaning-Text Theory, Printer London*, 1994.
- E. Reiter and R. Dale. *Building natural language generation systems*. Cambridge University Press, 2000.
- Y. Schabes. *Mathematical and Computational Aspects of Lexicalized Grammars*. PhD thesis, University of Pennsylvania, 1990.
- S. Schmitz and J. Le Roux. Feature unification in tag derivation trees. In *Proceedings of the 9th International Workshop on Tree Adjoining Grammars and Related Formalisms*, Tübingen, Germany, 2008.
- S. M. Shieber, Y. Schabes, and F. C. N. Pereira. Principles and implementation of deductive parsing. *J. Log. Program*, 1995.
- E. Velldal. *Empirical Realization Ranking*. PhD thesis, Department of Informatics, University of Oslo, 2008.
- K. Vijay-Shanker. Using descriptions of trees in a tree adjoining grammar. *Computational Linguistics*, 1993.
- K. Vijay-Shanker and A. K. Joshi. Feature structure based tree adjoining grammar. In *12th International Conference on Computational Linguistics (COLING)*, Budapest, Hungary, 1988.

- M. White. Reining in CCG chart realization. In *The Third International Conference on Natural Language Generation*, Brighton, UK, 2004.
- F. Xia and M. Palmer. Converting dependency structures to phrase structures. In *Proceedings of the first international conference on Human language technology research, Association for Computational Linguistics*, San Diego, CA, 2001.
- XTAG-Research-Group. A lexicalized tree adjoining grammar for english: Technical report. Technical report, IRCS, University of Pennsylvania, 2001.