جامعة البحرين
المكتبة
LIBRARY
University of Bahrain

An ATMS-Based Architecture for Stylistics-Aware Text Generation

Hasan Kamal



Ph.D.
Institute for Communicating and Collaborative Systems
Division of Informatics
University of Edinburgh
2001

Abstract

This thesis is concerned with the effect of surface stylistic constraints (SSC) on syntactic and lexical choice within a unified generation architecture. Despite the fact that these issues have been investigated by researchers in the field, little work has been done with regard to system architectures that allow surface form constraints to influence earlier linguistic or even semantic decisions made throughout the NLG process. By SSC we mean those stylistic requirements that are known beforehand but cannot be tested until after the utterance or — in some lucky cases — until a proper linearised part of it has been generated. These include collocational constraints, text size limits, and poetic aspects such as rhyme and metre to name a few.

This thesis introduces a new NLG architecture that can be sensitive to surface stylistic requirements. It brings together a well-founded linguistic theory that has been used in many successful NLG systems (Systemic Functional Linguistics, SFL) and an existing AI search mechanism (the Assumption-based Truth Maintenance System, ATMS) which caches important search information and avoids work duplication.

To this end, the thesis explores the logical relation between the grammar formalism and the search technique. It designs, based on that logical connection, an algorithm for the automatic translation of systemic grammar networks to ATMS dependency networks. The generator then uses the translated networks to generate natural language texts with a high paraphrasing power as a direct result of its ability to pursue multiple paths simultaneously. The thesis approaches the crucial notion of choice differently to previous systems using SFL. It relaxes the choice process in that choosers are not obliged to deterministically choose a single alternative allowing SSC to influence the final lexical and syntactic decisions. The thesis also develops a situation-action framework for the specification of stylistic requirements independently of the micro-semantic input. The user or application can state what surface requirements they wish to impose and the ATMS-based generator then attempts to satisfy these constraints.

Finally, a prototype ATMS-based generation system embodying the ideas presented in this thesis is implemented and evaluated. We examine the system's stylistic sensitivity by testing it on three different sets of stylistic requirements, namely: collocational, size, and poetic constraints.

Acknowledgements

I acknowledge with gratitude the support, help and encouragement I received from many individuals in conducting my research at the University of Edinburgh and completing this thesis.

First and foremost, my sincerest thanks are due to my supervisor Dr. Chris Mellish for providing individualised and patient supervision. His countless questions and comments helped me convey my ideas in a more coherent way. I have been privileged to have had him as my first supervisor.

I would also like to thank Dr. Mick O'Donnell who had been my second supervisor at some stage before he left Edinburgh. Even away, he continued to provide an abundant source of help.

I also owe many thanks to my thesis external examiner Dr. Richard Power and internal examiner Dr. Graeme Ritchie. They were extremely thorough and constructive in their comments. The final version of this thesis owes much to their careful reading.

During the years I spent at UoE I had the opportunity to learn from many excellent teachers. In particular, I benefited from Henry Thompson, Chris Brew, Geraint Wiggins, and Roberto Zamparelli.

My thanks to all friends and colleagues at Room E17 in South Bridge: those who already left and those who are still there, working hard. In particular, I thank Hua Cheng, Daqing He, Virginia Biris-Brilhante, Yannis Kalfoglou, Daniela Carbogim, Chris Lin, Joao Cavalcanti, Sonia Schulenburg, John Atkinson and Siu Wai Leung.

During the research time, I and my family were sponsored by the University of Bahrain, State of Bahrain.

Finally, my thanks must go to my family here and back home. They have always believed in me and encouraged me to pursue my studies. I am greatly indebted to my wife Sahar and my children Yousif, Maryam, and Salman. They have been a source of love, joy, and comfort during difficult times.

Declaration

I hereby declare that I composed this thesis entirely myself and that it describes my own research.



H Y Kamal
Edinburgh
April 30, 2001

Acronyms

Term	Meaning
ATMS	Assumption-based Truth Maintenance System
CNF	Conjunctive Normal Form A logical formula is in CNF if it is a conjunction of disjunctions of literals.
DNF	Disjunctive Normal Form A formula is in DNF if it is a disjunction of conjunctions of literals.
GF	Generation Fault
IE	Inference Engine
LC	Lexical Choice
NLG	Natural Language Generation
RMS	Reason(ing) Maintenance System
SAS	Situation-Action Specification
SFG	Systemic Functional Grammars
SFL	Systemic Functional Linguistics
SNAC	System Network to ATMS Converter
SSC	Surface Stylistic Constraints
SSF	Surface Stylistic Fault
SSR	Surface Stylistic Requirements
STAGE	STylistics-Aware GEnerator
TMS	Truth Maintenance System
WAG	Workbench for Analysis and Generation A system for the analysis and generation of sentences using Systemic Grammars (developed by Michael O'Donnell).

Contents

Abstract ii

Acknowledgements iii

Declaration iv

Acronyms v

List of Figures xiv

1 Introduction 1

1.1 Natural Language Generation 1

1.2 Motivation for this Research 3

1.3 Solution Proposed and Contributions 5

1.4 Organisation of the Thesis 7

1.5 Thesis Scope and Key Assumptions 9

1.6 Summary and Outlook 10

2 Lexical and Linguistic Choice in NLG 11

2.1 What is Lexical Choice? 12

2.2 When does it Take Place? 13

2.3 Different Models of Lexical Choice 14

2.3.1 Trivial Model 14

2.3.2 Structure-Mapping 15

2.3.3 Classification 15

2.3.4 Complex Models 16

2.3.5 Statistical Model 17
2.4 Factors Influencing Lexical Choice 18
2.4.1 Denotational and Connotational Constraints 18
2.4.2 Grammatical Constraints 19
2.4.3 Communicative Goal Constraints 20
2.4.4 User Model Constraints 20
2.4.5 Salience/Perspective Constraints 20
2.4.6 Deep Stylistic Constraints 21
2.4.7 Surface Stylistic Constraints 22
2.5 Lexical Choice and Other NLG Decisions 25
2.6 Flexible NLG Architectures 27
2.6.1 Integrated Architecture 27
2.6.2 Feedback Architecture 28
2.6.3 Blackboard Architecture 28
2.6.4 Revision-Based Architecture 29
2.6.5 Summary 30
2.7 Stylistics-aware Generation: Promising Directions 31
2.8 Summary and Outlook 32

3 Systemic Functional NLG **33**
3.1 Introduction 33
3.2 Systemic Functional Linguistics SFL 34
3.3 Systemic Functional Grammars SFG 35
3.3.1 The System 35
3.3.2 Realisation 39
3.3.3 Rank and Constituency 42
3.4 How the Lexico-grammatical Resource is Used 44
3.5 The Semantic and Lexico-grammatical Interface 44
3.5.1 Preselection-based Approach 46
3.5.2 System-based Approach 47
3.5.3 Feature-based Approach 48

3.6 Systemic Sentence Generation 49
3.7 Lexical Choice in Systemic NLG 51
3.8 Limitations of Current Generation Algorithms 53
3.9 Discussion: Choosing not to Choose 55
3.10 Summary and Outlook 57

4 The ATMS Framework **58**
4.1 Introduction 58
4.2 Truth Maintenance Systems (TMS) 59
4.3 Families of TMS 61
4.4 The ATMS: Basic Concepts 63
4.4.1 Nodes 64
4.4.2 Justifications 64
4.4.3 Dependency Networks 64
4.4.4 Labels 65
4.5 ATMS Algorithms for Label Maintenance 68
4.6 How the ATMS Works 70
4.7 The Encoding Problem 71
4.8 Solution Construction 72
4.9 Efficiency Considerations 74
4.9.1 ATMS Complexity 75
4.9.2 Implementation Optimisations 75
4.9.3 The IE Duty 76
4.10 ATMS for NLG: why? 78
4.10.1 Backtracking Generation 79
4.10.2 Chart Generation 81
4.11 Summary and Outlook 82

5 From SFG to ATMS **84**
5.1 Introduction 84
5.2 The Need to Find Multiple Selection Expressions 85

5.3 The Organisation of the Rest of the Chapter 88
5.4 Logical Interpretation of System Networks 89
5.5 Logical Specification of the ATMS 92
5.6 From Systemic Grammars to ATMS Representation 92
5.6.1 Simple System Representation 94
5.6.2 Simultaneous Systems Representation 96
5.6.3 Disjunctively Entered Systems 97
5.6.4 Conjunctively Entered Systems 98
5.7 Dealing with Conjunctively Entered Systems 100
5.7.1 Logical Formula for the Whole Network 101
5.7.2 Factorising the DNF Formula 104
5.7.3 Pros and Cons of the Factorisation Algorithm 109
5.8 Creating System Networks without Conjunctive Gates 110
5.8.1 Implications of Conjunctive Gates 110
5.8.2 The meaning of system simultaneity 111
5.8.3 Networks without right-facing braces 112
5.9 Compilation of Complete Grammatical Resources 115
5.10 Summary and Outlook 119

6 ATMS-Based NLG **122**

6.1 Introduction 122
6.2 ATMS-Based Generation 123
6.2.1 Tailoring System Network Snapshots 124
6.2.2 Creating Instances of a Network 127
6.2.3 Concept-Function Association 130
6.2.4 Lexicalisation of Function Bundles 131
6.2.5 Interfacing Expansion Triangles 133
6.3 The Overall System Architecture 134
6.4 Plain vs. Stylistics-aware Generation 136
6.5 Complete Example 136
6.5.1 Compilation of the Grammar 138

6.5.2 The Generation Procedure 142
6.6 Summary and Outlook 145

7 Stylistics-Aware Generation **146**
7.1 The Other Dimension of the Process 147
7.2 Surface Stylistic Constraints 150
7.3 Hard vs. Soft Stylistic Constraints 152
7.4 How to Capture Surface Stylistic Properties 153
7.5 Using Functional Constituency 153
7.5.1 Function Bundle Lexicalisation 154
7.5.2 Border Meta-Functions 155
7.5.3 Order Rules 156
7.5.4 Complete Sequences 157
7.5.5 Constituency Level 157
7.6 Situation-Action Framework 158
7.7 Situation-Action Specification (SAS) 159
7.8 Accommodating the Stylistics-Aware Mode 161
7.9 Examples of SSCs 161
7.9.1 Word Adjacency Constraints 162
7.9.2 Poetry Metre Constraints 166
7.9.3 Text Size Constraints 168
7.10 How the Situation-Action Approach Works 170
7.11 Summary and Outlook 175

8 System Implementation and Evaluation **177**
8.1 Implementation Notes 177
8.1.1 SNAC Input/Output 178
8.1.2 STAGE Input/Output 179
8.1.3 The ATMS Component 180
8.2 System Evaluation 181
8.2.1 Translation Phase 181

8.2.2 Generation Phase: Empirical Results . . . 183
8.3 Discussion . . . 194
8.4 Limitations . . . 196
8.5 Summary and Outlook . . . 198

9 Conclusions **199**
9.1 Contributions of this Thesis . . . 199
9.2 Future Directions . . . 202
9.3 Concluding Remarks . . . 205

Bibliography **206**

A The Lexico-grammatical Resources **218**
A.1 The Systemic Grammar Networks . . . 218
A.1.1 Clause Network . . . 218
A.1.2 Group Network . . . 224
A.2 SNAC Translations of the Grammar . . . 227
A.2.1 The Format of a Network Snapshot . . . 227
A.2.2 The Clause Network Snapshot . . . 229
A.2.3 The Group Network Snapshot . . . 229
A.3 The Lexicon . . . 231

B Generation Examples **233**
B.1 Micro-Semantic Input . . . 233
B.2 Example Choosers . . . 235
B.2.1 Clause Network Choosers . . . 235
B.2.2 Group Network Choosers . . . 236
B.3 Generation Trace . . . 236

C Situation-Action Specifications **251**
C.1 Word Adjacency Constraints . . . 251
C.2 Poetry Metre Constraints . . . 252
C.3 Text Size Constraints . . . 254

List of Figures

1.1 Natural language generation phases 3

2.1 Flexible NLG architectures (reproduced from [De Smedt *et al.* 96]) . . . 27

3.1 The English GENDER system and the POLARITY system 36
3.2 Related English clause systems 36
3.3 Different system types and their semantics 38
3.4 The English pronoun network 39
3.5 A clause network fragment with realisation rules in boxes 41
3.6 The rank and constituency relationship 43
3.7 The steps of realising a constituent 45
3.8 The steps of generating a sentence 46
3.9 Inter-stratal mapping (from [Teich 99]) 47
3.10 The MOOD TYPE system and its chooser 48
3.11 A decision tree for the chooser of a typical Number system (from [Mann 82]) 49
3.12 Different domains using the UM's taxonomy (from [Bateman *et al.* 90]) 51

4.1 Problem solving systems 60
4.2 The graphical convention for dependency networks 65
4.3 An example dependency network 65
4.4 A fragment of the English clause network 66
4.5 The dependency network of figure 4.3 with computed node labels 67
4.6 ATMS incremental label update for some node n 68
4.7 Graphical representation of a dependency network 70
4.8 Solution construction exploiting the nature of ATMS 73

4.9 The search space for the simple generation task 79
4.10 The branches of the search space visited by ATMS-based search 80

5.1 A simple system network 85
5.2 A tailored version of the system network of figure 5.1 88
5.3 An exhaustively labelled system 89
5.4 Different systemic configurations and their logical interpretations 90
5.5 An exhaustively labelled network with several connected systems 91
5.6 An example dependency network 93
5.7 An example simple system 93
5.8 Dependency network representations of simple systems 95
5.9 The meaning of disjunctive entry conditions 97
5.10 A typical conjunctively entered system 98
5.11 An impermissible form of ATMS justifications 98
5.12 A simple clause network 99
5.13 The corresponding dependency network of the clause network 100
5.14 A flattened representation of the system network of figure 5.1 103
5.15 A staged dependency network with exactly the same NETWORK label of that of figure 5.14 103
5.16 A network with simple systems only 105
5.17 A network with a conjunctively entered system 107
5.18 A dependency network corresponding to the network of figure 5.17 . . . 108
5.19 The English pronoun network (from [Winograd 83]) 108
5.20 The dependency network of the pronoun network 109
5.21 A conjunctive gate 111
5.22 A network with two simultaneous systems 112
5.23 System networks equivalent to that of figure 5.22 112
5.24 A system network and its corresponding braces-free version 113
5.25 A braces-free version of the English pronoun network 114
5.26 A system network with nested AND gates 115
5.27 A braces-free version of the network of figure 5.26 115

5.28 A simplified clause network 119
5.29 A snapshot compilation of the simplified clause network 120
5.30 A possible instantiation of the simplified clause network 121

6.1 The generation algorithm 124
6.2 An inter-stratal preselection resulting in multiple selection expressions . 126
6.3 A solution triangle for a tailored network 129
6.4 Interfacing of realisation triangles 133
6.5 An overview of the ATMS-based NLG system 135
6.6 An example semantic input representation 135
6.7 Clause and Group network fragments 137
6.8 A snapshot of the clause network of figure 6.7 139
6.9 A snapshot of the group network of figure 6.7 140
6.10 The dependency network constructed from the clause snapshot 141
6.11 Sample sentences with different lexical and syntactic choices 145

7.1 The stylistic dimension of the generation process 148
7.2 The functional constituency of a surface form 153
7.3 The process of promoting and demoting surface stylistic properties . . . 159
7.4 Plain-mode interfacing of realisation triangles 171
7.5 Interfacing of realisation triangles in the stylistics-aware mode 176

8.1 An overview of the ATMS-based NLG system 178
8.2 Effect of nondeterminism on generation time 188

Chapter 1

Introduction

This chapter gives an overview of the thesis. It starts by describing the context and motivation for this work. Then it briefly describes our solution and summarises the contributions made in this project. It also presents the structure of the rest of the thesis by briefly describing the content and purpose of each chapter. Finally, it states what is and what is not addressed and what assumptions we make in this work.

This thesis is concerned with the effect of surface stylistic constraints (SSC) on syntactic and lexical choice within a unified generation architecture. Despite the fact that these issues have been investigated by researchers in the field, little work has been done with regard to system architectures that allow the surface form constraints to influence earlier linguistic or even semantic decisions made throughout the NLG process.

1.1 Natural Language Generation

Natural Language Generation (NLG) is a relatively new field compared to Natural Language Understanding (NLU) which analyses human-produced texts and puts the meaning in computer-internal representation. NLG works in the other direction: from computer-internal representation to natural language. The aim of the NLG process is to produce high-quality comprehensible texts from some computer representation of data using language grammars and lexicons. This form of NLG which is known as "generation from first principles" is a complex process compared to other simpler

methods such as template-based techniques which in essence fill in blanks of predefined structures.

Examples of successful NLG applications include the automatic production of technical documentation from product specifications [Rösner & Stede 94], generation of weather reports from collected readings [Bourbeau *et al.* 90], summarisation based on the contents of relational databases [O'Donnell *et al.* 00], and preparation of medical documents from patient records [Binsted *et al.* 95]. As the NLG technology matures, more and more applications that require computer-human interaction will have text generation components of some sort. This means that generators will not only have to produce understandable texts but they will also have to be stylistically appealing.

Natural language generation is a complex process which is usually decomposed into manageable modules. Generally speaking, the NLG process is divided into two levels: the strategic level (deciding 'what to say') and the tactical level (deciding 'how to say it') [Thompson 77]. More recently, applied NLG systems tend to break the process into four smaller modules as shown in figure 1.1. According to [Mellish 95, Reiter 94], the task of each module can be more or less summarised as follows:

1. Content Determination: This concerns determining what meaning is to be conveyed. The result of content determination is a representation in terms of non-linguistic concepts. No decision is made at this stage about the sequence of events, words, or phrases to use to express the details.

2. Sentence Planning: This involves chunking the meaning into sentence-sized units, deciding what will be embodied in nouns and what in verbs, and deciding on the global syntactic structure of the text (e.g. thematisation, tense).

3. Surface Realisation: This is concerned with determining the syntactic structure and word order based upon the grammar rules of the target language. At this stage, it is possible to represent individual words in terms of a root, together with grammatical information preparing them to be inflected correctly at a later stage.

4. Morphology and Post-Processing: This is concerned with the production of the actual inflected words (for the output) using the rules of word formation of the target language. Other context-dependent processing of the words might take place at

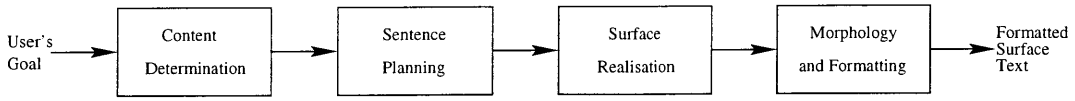


Figure 1.1: Natural language generation phases

this stage (e.g. capitalising the first letters of certain words).

The above breakdown of the generation process is seen from a processing perspective. Others look at the NLG process from a functional perspective where the whole process is divided into smaller linguistic operations. Cahill and Reape identify seven such tasks in their attempt to locate where different systems tend to perform these linguistic tasks during generation [Cahill & Reape 99]. They give the following non-exhaustive list of component tasks: lexicalisation, aggregation, rhetorical structuring, referring expression generation, ordering, segmentation, and salience/theme processing. Although not every NLG system incorporates all these tasks, they give an idea of what it is like to go from a conceptual representation to a natural language text.

A generation system is characterised by what modules it has and how these modules are allowed to interact with each other. The interaction constraints, or their absence, define the system architecture. For example, the flow of control shown above is known as the standard pipeline architecture. Most NLG systems are based on this *consensus architecture* as was first pointed out by [Reiter 94]. This claim is recently validated by the RAGS group in their search for a reference architecture for NLG systems [Cahill *et al.* 99]. There are, however, other possibilities as we will discuss later on in Chapter 2 (Section 2.6).

1.2 Motivation for this Research

At the end of the day, texts generated by machines are for humans to read. Therefore, it is always desirable to have natural, interesting and appealing texts; just the way we would like them to be. No matter how sophisticated the underlying modules and intermediate representations are, the quality of an utterance is judged by the surface

form (i.e. how it stands as a whole). Moreover, human writers sometimes revise an utterance just because its surface form has stylistic faults although it is otherwise perfect (i.e. grammatical and comprehensible). Stylistic constraints on generation can be divided into deep constraints such as the formality, euphemism, and force of an utterance, and surface constraints such as the length, collocational restrictions, and poetic aspects (e.g. rhyme and metre) of an utterance. As one can see, the surface stylistic characteristics of an utterance are directly affected by the lexical choices and syntactic decisions made earlier on.

Surface stylistic requirements constrain the surface forms and render them either good or bad. Generally speaking, it is not until the final utterance has been fully generated that we come to know what surface problems it has or hasn't. Lexical choice interacts highly with the syntactic distribution and morphological shape of words. For example, the French masculine pronoun *le* and feminine pronoun *la* are abbreviated to *l'* when they are followed by a word that starts with *e*. To avoid ambiguity, the choice between *la* and *Sarah* for example depends on the next word; something that might not be known until after the surface form is actually linearised.

Moreover, revision of individual words is not totally a surface matter as might first appear. It may lead to necessary revision of other parts because of syntactic and semantic constraints. For example, suppose that for some stylistic reasons (e.g. informal text) the verb "own" is to be replaced with "have" in "the house owned by Tom is beautiful". The resulting sentence "the house had by Tom is beautiful" is either grammatically imperfect or sounds more formal. Other alternatives like "Tom has a beautiful house" disturb other semantic issues such as the theme of the sentence.

Little work has been done to incorporate surface constraints in the overall generation process. In Section 2.4.7, we will discuss why this has been the case so far. [Evelyne & Pierette 94] characterise the NLG research effort in the past and explain why lexical choice has not received enough attention:

> "Lexical choice has often been side-stepped, not because it is a daunting issue, but rather because the interest in natural language generation first focused on syntactic, morphological and discourse aspects of language."

Among the work on lexical choice, most of it is dedicated to the paradigmatic side of lexical choice. We think it is time to incorporate both paradigmatic and syntagmatic aspects in lexical choice. Roughly speaking, by paradigmatic lexical choice we mean choosing a lexical entry that best realises the semantic piece; and by syntagmatic we mean choosing a lexeme — among the applicable ones — that with the surrounding lexical items satisfy the surface stylistic requirements.

Moreover, lexical choice is intimately related to architectural constraints in addition to other linguistic and stylistic constraints. The style of text to be generated influences our choice of a system architecture and vice versa [Paiva 99]. No wonder then that researchers dealing with surface constraints have started questioning the appropriateness of the pipeline architecture. [Reiter 00] argues for an alternative architecture in order to satisfy the text size constraint in the STOP project. Other applications that have strict surface constraints such as poetry generation required the abandonment of the pipeline, in spite of its advantages, in favour of the non-modular integrated architecture [Manurung *et al.* 00]. Others prefer to make a compromise by keeping the pipeline and providing a feedback mechanism between text planning and linguistic realisation [Rubinoff 00].

Our work in this thesis is an attempt in this direction. We are motivated by the influence that the surface constraints have on earlier stages such as lexical choice, syntactic choice, and — even though not within the scope of our current work — content selection. Because the surface stylistic characteristics of a text do not start to appear until after most of the syntactic and lexical choices have been made, our aim in this thesis is an NLG architecture that facilitates the efficient revision of just what is necessary as well as preventing the generation of structures that will give rise to unstylistic utterances.

1.3 Solution Proposed and Contributions

Lexical choice — like other generation tasks — is not an autonomous module within the NLG process. It interacts highly with the syntactic and semantic decisions made during the process. We propose a new NLG architecture that handles this interaction

between different NLG tasks. In particular, it accounts for the effect of surface stylistic constraints on earlier lexical and syntactic decisions.

We do this by bringing together a well-founded linguistic theory (Systemic Functional Linguistics, SFL) that has been used in many successful NLG systems and an existing AI search mechanism (the Assumption-based Truth Maintenance System, ATMS). The linguistic theory is chosen because of the way it handles lexical, syntactic, and semantic knowledge in one formalism using the same representation language. The search mechanism is chosen because it caches important search information and avoids work duplication. Moreover, it can pursue multiple solution paths simultaneously without explicit backtracking. We establish the logical relationship between the two formalisms. Fortunately, the two representations turned out to be logically similar which makes the ATMS a natural way to go.

The outcome of this work is an ATMS-based architecture for systemic natural language generation that is sensitive to the surface stylistic requirements specified at the outset of the generation process. We list below the contributions this thesis makes:

- The thesis identifies the limitations of current generation architectures with regard to the interaction between SSC and earlier linguistic tasks such as syntactic and lexical choice. In particular, systemic generators are studied to see why they do not account for SSC although it is claimed that their underlying linguistic formalism can be used for representation at all levels: ideational, grammatical, and lexical.

- The logical relationship between systemic grammar networks and ATMS dependency networks is established. Inspired by the works of [Mellish 88], [Brew 91], and [Calder 99] which interpret systemic grammar networks in logic, we take the idea further. We explore the logical connection between the two representations (i.e. SFG and ATMS) in an attempt to represent system networks using dependency (or causal) networks.

- An algorithm for the automatic translation of system networks to ATMS dependency networks is designed. The translation algorithm takes conventional systemic grammars (i.e. system networks and realisation statements) and maps

them to an equivalent ATMS representation. The translation algorithm can be used in applications other than text generation, since there is no reason which makes the system networks' framework specific to linguistic applications [Mellish 88]. System networks simply specify how combinations of features may imply or be inconsistent with other combinations.

- A new ATMS-based generation architecture is developed that uses translated networks to generate natural language texts with a high paraphrasing power as a direct result of its ability to pursue multiple paths simultaneously.

- A framework for user-specified stylistic requirements is developed enabling the generation system to work in one of two modes: plain or stylistics-aware mode. In the latter mode, the user can use the framework's vocabulary to write his own surface stylistic requirements which the generator then attempts to satisfy.

- A prototype ATMS-based generation system embodying the ideas presented in this thesis is implemented and evaluated. From the NLG side, this attempt can be seen as a new system architecture to experiment with; and from the ATMS side, it represents yet another application showing the versatility of the ATMS as a reasoning component.

In the remainder of this thesis, and in particular in Chapters 5, 6, 7, and 8, we present our ideas and implementations of them to support the above claims. In Chapter 9, we revisit the claims and justify the contributions made.

1.4 Organisation of the Thesis

In this section we provide an overview of the structure of this thesis. We briefly describe the content and purpose of each chapter.

- *Chapter 1: Introduction.* In the introduction we have given an overview of the thesis. We have described the context and motivation for this work as well as the main contributions in this thesis.

- *Chapter 2: Lexical and Linguistic Choice in NLG.* In this chapter, we take a tour of the body of related literature in order to show how various aspects of the lexical choice problem have been addressed in the NLG field. We also focus on literature related to choice of system architecture and then to linguistic and stylistic choice within a given architecture.

- *Chapter 3: Systemic Functional NLG.* Systemic natural language generation is discussed in Chapter 3. We first introduce the linguistic foundation: Systemic Functional Linguistics (SFL). Then, we show how the linguistic theory is used to generate natural language texts. We characterise the generation algorithms of the existing systemic generation systems. This chapter concludes by discussing why it would be expensive for existing systems to generate texts that have certain surface stylistic requirements and what plausible alternatives there are to overcome this problem.

- *Chapter 4: The ATMS Framework.* In this chapter we introduce the Assumption-based Truth Maintenance System (ATMS): what it is and how it works. In particular we focus on label updating and propagation of information between nodes. Some efficiency considerations are mentioned. We also give explanatory examples and discuss the problem of encoding. We present different techniques for using the ATMS to construct solutions.

- *Chapter 5: From SFG to ATMS.* In Chapter 5 we establish the logical relationship between the systemic functional grammar (SFG) networks and the ATMS dependency networks. This relationship helps us solve the problem of encoding; an essential question that needs to be answered in any application trying to use the ATMS as its reasoning engine. We then design — based on the logical connection between the two representations — a translation algorithm that takes general systemic grammar networks and transforms them into dependency networks, a representation that the ATMS can reason with.

- *Chapter 6: ATMS-Based NLG.* In this chapter we present our ATMS-based NLG architecture. We show how the systemic grammars and semantic input are presented to the generation system which then uses them to generate text relying heavily on the ATMS as its search mechanism.

- *Chapter 7: Stylistics-aware Generation.* In Chapter 7 we state what we mean by stylistics-aware generation. We design a framework for user-defined stylistic requirements which can be used to parameterise the generation process described in Chapter 6. We experiment with some sets of surface stylistic requirements, namely: word adjacency constraints, poetry metre specification and text size limitation.

- *Chapter 8: System Implementation and Evaluation.* In this chapter we present the implementation details of a prototype generation system that demonstrates the ideas discussed in the previous three chapters. Also, we evaluate the system's two main phases: translation and generation. We evaluate the translation stage in terms of the grammar size and the outcome of the translation process. Then, different modes of generation are compared and the effect of the particular set of stylistic constraints on the performance of the generator is discussed.

- *Chapter 9: Conclusions.* Finally in Chapter 9 we conclude the thesis by revisiting the contributions made in this work. We also point towards future work on the framework by suggesting possible extensions to our work in this thesis.

1.5 Thesis Scope and Key Assumptions

- The linguistic theory that this work is based upon is Systemic Functional Linguistics SFL. The generation system depends on the ATMS to do the reasoning. The idea is to combine a well-established linguistic theory with an existing AI search technique in a new NLG architecture that takes into consideration the effect of surface stylistic constraints on syntactic and lexical choices. No attempt was made to improve on any one of them.

- The work in this thesis focuses on techniques for sentence generation. Hence, the input to the generation process reflects this fact in that no history of previous utterances is provided, although this can obviously affect the stylistic appearance of a longer span of text.

- The types of surface stylistic requirements we consider here are for exposition purposes only. The aim is to show how stylistic constraints, such as poetry metre

and sentence length restrictions, can seamlessly be incorporated in the generation process using the same ATMS representation language. These examples should guide the user to write his own stylistic preferences.

1.6 Summary and Outlook

This chapter has given an overview of this work. We started with a brief introduction to the natural language generation process, its modules, and how they are standardly related in the consensus pipeline architecture. Generating text that has required surface properties, being our motivation in this project, we then discussed how this is directly related to lexical choice and indirectly to deeper generation decisions. The theme of the next chapter is, therefore, lexical choice. We see how this problem has been approached in the literature, its relation to the generation architecture, and how it interacts with other NLG tasks.

Chapter 2

Lexical and Linguistic Choice in NLG

As the introductory chapter has made clear, we are interested in the interaction between lexical choice, syntax, and surface stylistic constraints (SSC) and in system architectures that allow SSC to influence earlier linguistic decisions. In this chapter we take a tour of the body of related literature in order to show how various aspects of the lexical choice problem have been addressed in the NLG field. Regarding the lexical choice problem per se, we discuss its definition, when and how it is carried out, and what factors influence this task. We then discuss the interaction between lexical choice and the other NLG tasks as well as the relationship between it and the architectural paradigm. We then discuss why we think little work has been done with regard to the problem at hand and what needs to be done.

NLG is a process of choice at many levels. At the heart of the process is lexical choice, as the chosen words will shape the final utterance. At the surface, they stand on behalf of the deeper syntactic and semantic decisions made during the generation process, and surface stylistic judgements will have to be based on that linearised lexical evidence. In case there is a stylistic fault in the utterance, caused by a particular choice of words in a particular syntactic structure, fixing (or maybe avoiding) it depends on the architecture of the generation system.

Although the theme of this chapter is lexical choice, we find it inevitable to discuss other closely related issues such as syntactic choice, surface stylistic requirements, and architectural constraints. After presenting the lexical choice task, we discuss how it influences and is influenced by these NLG decisions.

2.1 What is Lexical Choice?

Given a communicative intention, a language generator must select information from a set of non-linguistically represented concepts, order this information, and linearly realise the representation in the form of sentences under the guidance of the language grammar. The process of selecting words that will represent the concepts is termed lexical choice.

To give a more specific definition of lexical choice we need to make a distinction between open-class words and closed-class words. Open-class words such as nouns, verbs, adjectives, and adverbs are also called content words; whereas closed-class words such as articles, pronouns, and conjunctions are called function words. Open classes are large and constantly expanding while closed classes are small and stable. The application of grammar rules entirely dictates the choice of most closed-class words. However, the choice of open-class words cannot be dictated by the grammar rules but only restricted by them. Having said that, lexical choice is, therefore, the choice of open-class words.

[Reiter & Dale 00] distinguish between CONCEPTUAL LEXICALISATION and EXPRESSIVE LEXICALISATION. They refer to the process of converting raw data or information into linguistically-expressible concepts as conceptual lexicalisation. Expressive lexicalisation, on the other hand, is concerned with realising a given conceptual representation using the words of the target language. [Cahill *et al.* 99] use the term "lexicalisation" to refer to the CONCEPTUAL LEXICALISATION and "lexical choice" to refer to EXPRESSIVE LEXICALISATION. According to this distinction, we will be concerned with expressive lexicalisation or simply lexical choice which we take to mean deciding between lexical alternatives. In particular, we will be concerned with *stylistic* lexical choice (i.e. lexical choice under surface stylistic constraints).

Related to the notion of lexical choice is the distinction between paradigmatic and

syntagmatic relations. Two open-class words are said to be paradigmatically related if they can appear in place of each other in correct phrases; they are syntagmatically related if they appear close to each other in correct phrases [Robin 90]. As we will see later on, most systems perform paradigmatic lexical choice. This is selecting one word among a number of open-class words where the syntactic structure has already been specified. Choosing between *old* and *elderly* in the phrase: "The old/elderly man ..." is paradigmatic lexical choice. It is worth mentioning here that most of the research effort has focused on techniques for paradigmatic lexical choice as we will see in Section 2.3.

Lexical choice, however, is not as simple as it may first sound [Nirenburg & Nirenburg 88]. One-to-one mappings between domain concepts and language words are both undesirable and not always possible. They are undesirable because a domain concept may be expressed by multiple words and, conversely, a single word may express many concepts [Elhadad *et al.* 97]. In multi-lingual generation, a one-to-one mapping is sometimes not possible as there is not always a direct translation of a word from one language to another [Stede 96a].

2.2 When does it Take Place?

The place of lexical choice within a complete generation system is controversial. For instance, [Reiter 91] considers lexical choice a content determination task. Some work, in systemic functional grammar, views lexical choice as just a very detailed part of syntactic choice which is carried out during surface realisation.

[Elhadad *et al.* 97] review previous generation systems with regard to the position of lexical choice in generation based on a two-stage architecture: content planning and syntactic realisation. They identify the following options:

1. Within the syntactic realisation stage.
2. After content planning and before syntactic realisation (i.e. in between).
3. Divided between the two stages (i.e. some at each stage).
4. Within the content planning stage.

As a matter of fact, these are all the reasonable possibilities for placing the lexical choice component in a two-stage architecture. Table 2.1 summarises their findings as to where different systems perform lexical choice.

	option 1	**option 2**	**option 3**	**option 4**
Example Systems	ANA PHRED PAULINE	COMET SPOKESMAN EPICURE	PENMAN	FN (Reiter's) Danlos System

Table 2.1: Where different generation systems perform lexical choice

In a RAGS' survey of nineteen applied NLG systems, [Cahill & Reape 99] concluded that systems vary in where they perform lexical choice, although there seems to be a tendency towards lexical choice being the last or one of the last tasks to take place.

Form the above discussion, we can see that lexical choice is a controversial NLG task which can take place almost anywhere in the process: within the planning components or within the realisation components. Even worse, some researchers believe that lexical choice "does not constitute an autonomous module within the process of generation ... [and that it] can influence a conceptual choice and vice versa" [Evelyne & Pierette 94]. Therefore, the effect and presence of lexical choice extend throughout the generation process.

2.3 Different Models of Lexical Choice

2.3.1 Trivial Model

Early approaches to lexical choice assumed a one-to-one mapping between words and concepts. For example, if the input to the lexicaliser has the logical form predicate ANIMAL, it outputs "animal" as a realisation of the concept ANIMAL. Although this is worthwhile from an engineering point of view, it simplifies the task to the extent of eliminating it from any theoretical interest [Reiter 91].

The one-to-one mapping between words (or phrases) and concepts does not solve the choice problem; it only avoids it. While, in principle, one could have a grain-size for concepts so that each word gets mapped to a concept, this only shifts the choice task to

the module that loads the knowledge base with concepts. This is clearly not the best solution since lexical choice is obviously a linguistic matter. Moreover, the one-to-one mapping is both undesirable and not always possible as mentioned in Section 2.1.

2.3.2 Structure-Mapping

Structure-mapping systems take, as input, a semantic structure that needs to be communicated to the user and they search for pieces of the input structure that are equivalent to a concept with an associated word. These pieces are then removed from the input and the word is added to the output. The matching and substitution process is repeated until the input is completely reformulated as a set of lexical units. For example, the structure {`human, sex:male, age-status:adult, wealth:high`} might be reduced to {`wealth:high`} after matching {`human, age-status:adult, sex:male`} with the surface unit "man". The remaining input structure is further substituted with "rich" and we finally get the surface realisation "rich man". Examples of systems that use this model can be found in [Nogier & Zock 91, Sondheimer *et al.* 89, Iordanskaja *et al.* 88].

The problem with this model is that an item cannot be missed out even if it is implied by others, e.g. {`dangerous, human-eating`}. Also, nothing can be added even if it is true and gives a shorter description. For example, {`unmarried, human`} cannot be realised as "bachelor" unless `sex:male` is also in the input. Moreover, this model does not account for the user model. For instance, if the hearer has a lexical gap then it is not possible to have a set of words to correspond to a single item of the input. {`human, male, unmarried`}, for example, is lexicalised as "bachelor" even if this word is not in the user's lexicon.

2.3.3 Classification

Classification systems map a concept to one of the near-synonymous words that represent a possible realisation of that concept. The first classification systems for lexical choice were discrimination nets [Goldman 75, Pustejovsky & Nirenburg 87]. They perform lexical choice by providing for each semantic primitive a decision tree with possible words attached to its leaves. Every word sense has associated with it a set of value

restrictions that have to be satisfied by the input conceptual representation. If a conceptual unit satisfies the restrictions of a word sense then that concept will be rendered using that word. For example, the discrimination net for the primitive concept INGEST can be related to different verbs such as *eat*, *drink*, and *inhale*. When trying to map a concept to one of these words, the discrimination net is traversed based on a sequence of queries regarding the object being ingested. Eventually, the concept will be realised as *eat* if the object is solid, *drink* if it is liquid, and *inhale* if it is gaseous.

Recently, new models have been developed based on the basic classification approach such as Edmonds' clustered model. This model claims to be more efficient in representing near synonyms by having measures that cut off the ontology at a coarse grain which "alleviates an awkward proliferation of language-dependent concepts on the fringes of the ontology" [Edmonds 99].

Classification systems only provide one word as a realisation of a given conceptual input. They offer no solution when the most specific lexical concept conveys hardly any information. Also, they can never generate a word more specific than the input even though one more specific word would be shorter than an exact description providing several words.

2.3.4 Complex Models

In a complex model, the lexical choice task is not restricted to picking the words from the lexicon only. Instead, in this model, lexical choice is an involved process that goes through several stages before it decides on the lexemes to be selected.

For example, in [Elhadad *et al.* 97], the lexical choice process involves two steps: syntagmatic decisions and paradigmatic decisions. The syntagmatic decision step first carries out *phrase planning* to set the scene for the second stage which they call *lexicalisation proper*. They consider phrase planning part of the lexical chooser because it can merge two content units in a single linguistic unit which can then be served by the lexicalisation proper which does a paradigmatic choice of words.

[Stede 96b]'s model of lexical choice can also be considered a complex one since the lexical choice process also involves several steps: a matching process, application of alter-

nation rules, and selection of preferred choices based on salience and connotations. The lexicon he uses is rich and is purpose-built for the task of generating many paraphrases describing an event. The lexical entries consist of several additional components in addition to the conventional morphosyntactic features of a word. In particular, verb entries contain pointers to alternations and extension rules. These alternation rules simply mean to systematically derive more complex verb configurations from simpler ones. This means that lexical choice first gets some initial words during the matching process. This set of words is then enriched by applying the alternation rules. Finally, the options are ranked based on salience and connotational criteria with the top ones being preferred.

2.3.5 Statistical Model

A statistical model for lexical choice is usually used in connection with another model. The 'other' model's job is to arrive at a small set of near-synonyms for a concept. Statistical methods are then used to choose the most typical or natural candidate in a given context.

The limitation of this approach, however, is that it only solves part of the lexical choice problem. That is why it has to be coupled with another model. [Edmonds 97] experimented with this model at some stage and he admits that it is only part of the original problem:

> "while this lexical problem is weaker than the general lexical choice problem, it can be thought of as an important sub-problem, because it would enable a system to determine the effects of choosing a non-typical word in place of the typical word." [Edmonds 99]

Unlike the other models which only consider the paradigmatic aspect of lexical choice[1], this approach does have some syntagmatic lexical choice. Therefore, the advantage of this approach is that it treats lexical choice in the context of other words already appearing in the surface form.

[1] Although [Elhadad *et al.* 97]'s lexical chooser makes syntagmatic decisions in its initial stage, the kind of syntagmatic decisions it considers have to do more with planning *unlexicalised* lexeme-size semantic entities than with inter-lexical constraints of surface words.

2.4 Factors Influencing Lexical Choice

Constraints influencing lexical choice come from a wide variety of linguistic and non-linguistic sources. These include, but are not restricted to, syntax, semantics, pragmatics, and the lexicon. In this section, we review the range of lexical choice constraints available in the literature of NLG. This does not mean that NLG systems take all of these constraints into account. Different systems will consider different subsets of these constraints. Still, some constraints are only acknowledged to influence lexical choice, but are never incorporated in implemented systems. For each type of constraint, we try to state a definition, give examples, and show how it affects the way a generator chooses its words. We also classify each factor according to the paradigmatic/syntagmatic relations. In fact, we will notice that most of the research effort is directed towards the paradigmatic constraints on lexical choice. This supports our claim that little work has been done regarding the syntagmatic constraints that the surface form imposes on lexical choice.

2.4.1 Denotational and Connotational Constraints

By denotation we mean the relation between the lexical unit and whatever object or concept it is used to refer to [Matthews 97]. Simply put, denotation refers to the core meaning of a word. This is the most obvious factor that influences our choice of words because the last thing we want is to pick a word that is totally inappropriate. If all factors are reduced to preferences, the denotational factor remains the only and most important constraint that cannot be overlooked. Denotational constraints are not usually listed with other factors on lexical choice, only because they are there by default. However, denotational constraints alone are not enough to push the lexical choice process forward; words are selected based on other additional factors. When trying to express a concept, we sometimes look at the thesaurus for a more appropriate candidate although we have one word at hand. As a matter of fact, the task of lexical choice arises when the generator has a number of lexical units for expressing a concept at its disposal. Needless to say, all of these lexical units have the same core meaning:

> "So, lexical choice—genuine lexical choice—is about making choices be-

> tween options, rather than merely finding the words for concepts, as is the case in many past text generation systems" ([Edmonds 99], pg. 135)

On the other hand, connotation is used variously to refer to differences in meaning that cannot be reduced to differences in denotation [Matthews 97]. [DiMarco & Stede 93] characterise the difference between denotation and connotation in terms of truth conditions: "if two words differ semantically ... then substituting one for the other in a sentence or discourse will not necessarily preserve truth conditions; the denotations are not identical. If two words differ (solely) in stylistic features ..., then inter-substitution does preserve truth conditions, but the connotation — stylistic and interpersonal effect of the sentence — is changed".

[Stede 96b, Stede 96a] follows this broad definition of connotation. Actually, connotational and stylistic constraints are used interchangeably in Stede's work. Other researchers in the field, however, take the stance that restricts the meaning of connotation to that of social implications. As an example of this viewpoint, [Busemann 93] gives two German words *Putzfrau* and *Raumpflegerin* which both mean *cleaning woman* but only the latter is used officially as the former carries a depreciatory connotation with it.

Both denotational and connotational constraints are paradigmatic constraints that influence the system's choice of one word from alternatives without worrying much about the surrounding lexemes.

2.4.2 Grammatical Constraints

If the grammatical form of an utterance has already been determined, then the choice of words expressing a given concept is restricted to those that can appear in such a grammatical form. The predetermination of the syntactic form is not uncommon since, under some stylistic conditions, one might prefer to use parallel structures with surrounding sentences [Iordanskaja *et al.* 91]. On the other hand, many words have idiosyncratic grammatical properties. If the generator has decided to choose a specific word then the potential syntactic structures are restricted to those that can accommodate the pre-selected word. For instance, the word *has* in the context of TOM HAVING

THE CAR cannot be passivised. This means that the choice of the syntactic form is restricted to the active voice yielding something like "Tom has the car". So, syntax influences lexical choice and vice-versa.

2.4.3 Communicative Goal Constraints

The communicative goal of a system might simply be: inform the hearer. However, systems that generate text under pragmatic constraints might have richer communicative goals such as: inform the hearer, warn, encourage, discourage, and/or confuse him.

It is obvious how these communicative goals influence the choice of words that a generator can make in order to realise a given concept. [Hovy 88] in PAULINE and [Elhadad 92] in ADVISOR II consider these type of constraints and their choices of lexical items differ depending on the communicative goal(s) that they are trying to satisfy.

2.4.4 User Model Constraints

The output of an NLG system is for humans to read and understand. Therefore, it helps if a generator has a representation of a user model and it chooses the lexical units based on what the user knows about the domain, his lexicon, and interests. A text generated for expert readers in a certain field would be drastically different from another text directed towards a general audience even though the underlying conceptual knowledge base is identical. For example, [McKeown *et al.* 93] use "different strategies in COMET for selecting words with which the user is familiar." By the same token, text generated for native speakers of a language is different from text that is generated with non-native speakers in mind. Different people have different lexicons and different mental categories. Hence, choosing the right words according to the user's lexicon and mental classification prevents false implications being communicated [Reiter 91].

2.4.5 Salience/Perspective Constraints

The choice of words is usually dictated by the way we look at things or how much salience we assign to different participants. Whether we choose to say 'I sold my car to

him' or 'he bought my car' depends on which participant is in focus [Jacobs 87]. The same applies to all transfer events that can be reported from different points of views such as *take/give*, *borrow/lend*, and *send/receive*.

2.4.6 Deep Stylistic Constraints

Style is generally defined as the choice between the various ways of expressing the same message. [Stede 93] lists the following stylistic features that influence our choice of words in the process of realising some concept(s): formality, slant, euphemism, archaic/trendy, floridity, abstractness and force. We call these *deep stylistic constraints* as we differentiate between them and *surface stylistic constraints*. Deep stylistic constraints are those that can be checked early in the generation process while surface constraints do not arise until after surface realisation takes place – as will be shown next. Deep constraints influence the paradigmatic lexical choice and, in general, no consideration is paid to the surrounding lexemes of the surface form. Below we define and give examples of different deep stylistic features, most of them are based on [Stede 93].

1. **Formality:** Linguists have thoroughly investigated the formality dimension of words. It is common for dictionaries to rate words as formal or colloquial. Examples of formal/informal pairs are: *father/dad*, and *motion picture/movie*.

2. **Slant:** The choice of words depends on the attitude of the speaker towards the subject. The speaker may manifest his opinion through the use of *slanted* words. Although {`human, male, adult`} is usually realised as *man*, one may choose instead *gentleman* or *jerk* for example.

3. **Euphemism:** Euphemism is used in order to avoid some emotional words in certain social situations (e.g. *pass away* instead of *die*).

4. **Trendy/Archaic:** If the text is to be trendy then not only should obsolete and archaic words be avoided but also fashionable trendy words should be used instead.

5. **Floridity:** Flowery words are used when the speaker wants to make some kind of

impression on the hearer. [Hovy 88] gives *entertain the thought* as a more flowery expression for *consider*. In PAULINE, Hovy attempts to enhance the quality of the generated text in order to achieve various effects.

6. **Abstractness:** How abstract/concrete one wants to be determines one's choice of words. An example would be *unemployed person* (abstract) and *out of work* (concrete).

7. **Force:** Some words are more forceful than others. Consider *destroy* vs. *annihilate* or *big* vs. *monstrous* .

2.4.7 Surface Stylistic Constraints

Surface stylistic constraints on lexical choice are those that cannot be tested until or after surface realisation [Reiter 00]. Some may even be delayed until morphology and post processing. There are cases when one would choose other words after one has already written a complete sentence because of some syntagmatic factors constraining the words that can collectively appear in the surface form. Reasons for this may be the adjacency of words that share the same root, or other rhythmical reasons such as the ease of reading the text due to adjacent words that sound similar. The following are examples showing the kind of surface stylistic constraints we mean.

1. **Inter-lexical Constraints:** Due to unfortunate lexical choices, the utterance may be awkward or ambiguous when words are linearised although each word is good in isolation. For example, the French pronouns *le, la* cannot precede words starting with *e*. When that happens, both are abbreviated to *l'*. So that, if we want to generate (in French) an unambiguous utterance, the choice between the feminine pronoun *la* and *Sarah* depends on the next word [Reiter & Dale 00]. Although simple, this example shows that there are cases where generators cannot make a final decision on lexical choice until after the surface form has been linearised and its words inflected.

2. **Collocational Constraints:** A well known phenomenon of inter-lexical constraints in natural languages is collocations. Collocational constraints require that certain words co-occur in a sentence while others do not. [Stede 93] gives

the example of three adjectives which mean very much the same thing but the occurrence of any one of them depends on whatever comes next. We find *rancid* butter, *putrid* fish, and *addled* eggs but no alternative combination. As a result, the choice of word x for concept a can enforce the choice of word y for concept b.

Models that are based on Meaning Text Theory (MTT) have, in the lexicon, linguistic knowledge governing the usage of words in texts. Among other information, an MTT lexicon includes Lexical Functions (LFs) which are pointers to other words that are semantically or collocationally related to the entry word. For example, the functions $\mathbf{Syn}_{\supset}$, $\mathbf{Syn}_{\subset}$, and $\mathbf{Syn}_{\cap}$ are LFs that designate, respectively, synonyms with broader, narrower, and intersecting meanings [Mel'čuk 88]. Hence, $\mathbf{Syn}_{\supset}(assist1) = help$.

Collocational constraints are a kind of syntagmatic constraint on lexical choice since words are chosen partly based on other words being also chosen. However, collocational constraints represent only a subset of the syntagmatic constraints.

3. **Variousness:** We define variousness as using different words for the same concept in a short span of text. Granville's system (PAUL) was an attempt to produce interesting texts by varying the lexical items chosen for a specific concept. His idea of *lexical substitution* is to classify synonyms and near-synonyms into distinct sets based on their connotative qualities. For instance, PAUL partitions the synonyms of ODOUR into three sets: POSITIVE: {`aroma, emanation, fragrance`}, NEGATIVE: {`foulness, stench, stink`}, and NEUTRAL: {`odour, scent, smell`}. Once the decision has been made on a particular connotation,PAUL then selects *randomly* from within the proper set of synonyms making sure it does not "repeat itself unless every item on the list has already been used"[Granville 84].

4. **Consistency:** Although variousness is a positive stylistic feature that makes the generated text more interesting, too much of it might distract the reader's attention and turn the text into a piece of agony. Referring to person X in a span of text as boss, foreman, superintendant, head, master and chief is surely not helping much in producing less awkward text. So, consistency obviously restricts the system's choice of words.

5. **Text Size Constraint:** In some generation applications there may be some size constraints on the text that can be generated. For example, [O'Donnell 97] describes a system for variable length on-line document generation "whereby the user specifies how long the document should be". Such a requirement influences lexical choice as some words result in longer utterances because of the way each word packages information. The accumulative effect of such verbose choices can result in longer texts. However, the exact length of text is not known until after the text is generated and compared to the size limit that it is allowed to occupy. In the STOP project, Reiter discusses how even marginal things like punctuation, inflection, and font type can play a role in keeping the text within the allowed limit [Reiter 00]. Also, [Bouayad-Agha *et al.* 91], in their patient information leaflets generation project, regard text length and lexical choice, for example, as matters of style which "require rewording of the text". These constraints are imposed by the user (the patients in this case): the "patients might object that the sentences are too long, or that technical words [...] are used instead of familiar ones". The length of a sentence's constituents is sometimes related to the structural complexity of it, so more complex sentences tend to be longer [Lin 96]. In general, complex sentences are more difficult for humans to process.

6. **Poetic Constraints:** In poetic writings, issues like metre, rhyme, and alliteration all restrict the words that an NLG system can select in relation to what has already been chosen. Applications that require this kind of text are starting to appear in the NLG literature such as poetry [Manurung *et al.* 00], punning riddles [Binsted & Ritchie 94], and story generation [Bailey 99, Binsted & Ritchie 96].

Researchers acknowledge that considering the interaction between the constraints on lexical choice is something that is deemed to improve the quality of the generated texts. For instance, Busemann urges for a new approach to lexical choice which he calls 'a holistic view of lexical choice' [Busemann 93]. So, why is it that the kind of surface stylistic constraints we are interested in here have not received enough attention although they are acknowledged by the researchers in the field? We think that some of the reasons might be:

- The inflexibility of the widely used pipeline architecture especially given that we do not come to know all surface stylistic problems until after the utterance has been generated.

- For some generators, this is simply not an issue, even if they have an architecture different from the pipeline, since their applications do not have predefined surface requirements.

- There are also historical reasons why lexical choice has not received enough attention:

> "Lexical choice has often been side-stepped, not because it is a daunting issue, but rather because the interest in natural language generation first focused on syntactic, morphological and discourse aspects of language." [Evelyne & Pierette 94]

2.5 Lexical Choice and Other NLG Decisions

In this section, we discuss the interaction between lexical choice and syntax and argue that the widely used pipeline architecture cannot handle such interaction. In the next section, we explore other architectural options which we collectively refer to as flexible NLG architectures.

Lexical choice interacts significantly with other NLG tasks such as syntactic choice and even content selection. A generation gap, due to the unavailability of a lexical item to realise a concept in a semantic input, might require that the syntactic choice made to realise it is abandoned. It may even require the semantic input to be chunked in a different way. This brings up the issue of system architecture and whether it permits any choices to be redone.

Revision of individual words is not totally a surface matter as might first appear. It may lead to necessary revision of other parts because of syntactic and semantic constraints. For example, suppose that for some stylistic reasons (e.g. informal tone) the verb "possess" is to be replaced with "have" in "the house possessed by Tom is green". The resulting sentence "the house had by Tom is green" is either grammatically imperfect

or sounds as formal as the original utterance. Other alternatives like "Tom has a green house" disturb other semantic issues such as the theme of the sentence and may clash with other stylistic constraints (e.g. rhyme constraints). It also brings with it ambiguity: is it really a greenhouse, or a house that is green? To avoid ambiguity, one might say "Tom has a house whose colour is green", "Tom has a house that is green", or "Tom has a green-coloured house" which are all verbose utterances. Eventually, we might need to relax the *informality* requirement a bit. The sentence "The house owned by Tom is green" although more formal than the *have-verb-option* is still less formal than the *possess-verb-option.* Moreover, it keeps the theme and rhyme of the sentence intact. This by no means means that the problem is solved; the *own-verb-option* might bring with it a new set of problems and we might find ourselves in the same cycle of trying to satisfy involved constraints.

There is a relationship between lexical choice and system architecture because lexical choice is one aspect of stylistic variation [DiMarco & Hirst 93] and style has been shown to influence and be influenced by our choice of a system architecture [Paiva 99]. The pipeline — the only architecture introduced so far — is a common architecture used in many generation systems (hence called the consensus architecture). Information in a sequential architecture flows in one way through the pipeline. In a rigid sequential architecture there is no direct feedback between the subtasks of the generation process.

This architecture has the disadvantage of inflexibility. Each module has to reflect its assumptions on the intermediate representation, hoping that things will go well eventually. Sometimes, however, the local decisions made at each stage do not add up to a good overall outcome [De Smedt *et al.* 96]. We saw in the previous examples how generating an utterance that is stylistically unacceptable requires undoing previous decisions. The pipeline, being a one-way architecture, cannot allow for this interaction. It only allows an early decision to constrain other modules that come afterwards but does not account for the influence of later choices' constraints.

Next, we present some flexible generation architectures. We discuss the problems they try to solve by having such flexibility and what the pros and cons of each architecture are. After this discussion, we restate our problem and point towards promising directions as to which linguistic theory and generation architecture are appropriate for our

needs.

2.6 Flexible NLG Architectures

Despite the simplicity of the pipeline architecture, it is a rigid framework. NLG applications that have special needs opt for other architectures that allow for some flexibility and/or revision. Moreover, experimenting with new system architectures has always been a good thing. As a matter of fact, "continued development of grammars and *generation methods*" is considered a longer-term goal by researchers in the field [Hovy 96].

[De Smedt *et al.* 96] identify five architectures of natural language generators: the sequential (pipeline), integrated, interactive (feedback), blackboard, and the revision-based architecture. They are depicted in figure 2.1 except for the pipeline architecture which was shown earlier (see Section 1.1).

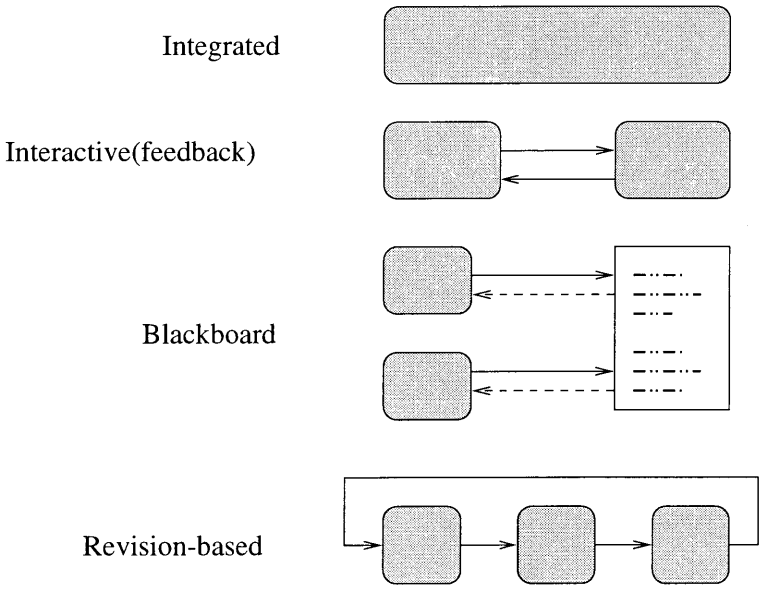


Figure 2.1: Flexible NLG architectures (reproduced from [De Smedt *et al.* 96])

2.6.1 Integrated Architecture

The integrated architecture differs from the others in that it does not decompose the generation process into clear-cut subtasks although decomposition is favoured both from a psychological and an engineering point of view [De Smedt *et al.* 96]. As a

result, it has the advantage of flexibility but it lacks the modularity feature. In general, this architecture supports "generation of pragmatic effects more efficiently than other systems" [Kantrowitz & Bates 92].

Sometimes it is the only way to go if later stages are allowed to influence the decisions of early stages. For example, [Manurung *et al.* 00]'s poetry generator has to give up modularity for flexibility. Their poetry generation system uses an integrated NLG architecture. [Kantrowitz & Bates 92] in the GLINDA system, also opted for the integrated architecture for a similar generation task: generation of poetic utterances in a virtual reality environment.

2.6.2 Feedback Architecture

The interactive architecture allows feedback between consecutive stages. For example, in [Rubinoff 92, Rubinoff 00]'s IGEN system, interaction between the strategic and the tactical component is handled via *annotations* that provide feedback from the linguistic component to the content planner. Each lexical unit is enriched with annotations about what extra information it carries, what it makes explicit/implicit, and whether it is a concise construction or not. For example, if the plan is to inform the user of something and to make him happy at the same time, IGEN would generate "it is drizzling" instead of "it is raining". Assuming that the linguistic component cannot choose between *raining* and *drizzling*, it gets back to the planner providing it with the annotations for each linguistic choice. The planner then prefers *drizzling* because it thinks that it can make the user happy by downplaying the *unpleasant* information.

The drawback of this approach is that it considers each lexical unit in isolation. It does not consider the context in which this word might appear nor the interaction between it and the surrounding words. Furthermore, it does not account for the yet-to-come syntactic and morphological constraints on lexical choice.

2.6.3 Blackboard Architecture

The blackboard architecture allows its modules to post information on the blackboard without knowing which particular module is going to use it next. So, the order of

module execution is quite unpredictable and subject to execution time circumstances.

Although any system adopting the blackboard architecture would gain its power from the flexibility of the order in which each module is triggered, there needs to be extra knowledge to control priorities when more than one knowledge source is triggered. To the best of our knowledge, the only proposal to use a blackboard architecture to do revision in NLG is by Wong and Simmons[2]. However, [Robin 94] states that "they are not specific about what type of information is to be posted on the blackboard during revision". He also concludes that the inherent flexibility of such an architecture would allow manipulations that affect several layers at once. This makes it hard to follow the propagation of the effects of any single change to the intermediate representation.

2.6.4 Revision-Based Architecture

In a revision-based architecture, the system follows a defined sequence of stages and, at the end, it decides whether to revise or not; and if yes, where to start the revision process from. WEIVER is a revision-based generator for Japanese [Inui *et al.* 92]. The purpose of revision is to solve some surface problems, namely: structural ambiguities and sentence complexity (such as its length, embedding depth, and modification relations). The motivation behind the system is that Japanese allows flexible word order in some sentences giving rise to ambiguities in interpretation. For example, "*John-ga Poochie-to satteiku Mary-wo miteita*" can mean "John looked at (Mary departing with Poochie)" or "With Poochie, John looked at Mary departing". WEIVER revises to resolve such ambiguities by producing a different surface form free of structural ambiguities.

The surface realiser is responsible for generating the first draft and the subsequent surface changes such as word order, punctuation, lexical choice, and syntactic structure. WEIVER requires human assistance to detect the ambiguities. It "repeats the revision cycle until (it) produce(s) an acceptable text". [Inui *et al.* 92] acknowledge the fact that even the slightest surface change introduces new problems of a different nature. However, they assume that "these can be detected and solved in subsequent cycles" of revision. What they do not specify clearly is where this guarantee comes from. It seems

[2] This paper is no longer available.

that an acceptable text for them is one which does not contain structural ambiguities and whose sentences are of a limited number of words. On this assumption, one can see the revision cycle coming to an end, although there is no guarantee that tackling one problem at a time might not trigger a series of new problems of a different type which are not under the system's current consideration. Moreover, they assume the independence of these surface problems as the *revision planner* "selects one of the problems detected by the evaluator" using some heuristics called revision rules. It then sends the surface generator a message that describes the change required by the chosen revision rule. When one surface problem is dealt with successfully, the revision planner turns its attention to another surface problem. So, WEIVER only revises within surface realisation making it difficult to follow up the implications of the surface amendments that go beyond the surface level.

[Robin 94] uses the revision approach differently. His approach is an *information-adding* one as opposed to the *information-preserving* kind of revision. The purpose of the system is to generate summary reports presenting new information in combination with related historical information. In STREAK, generation proceeds in two main steps: draft, and then revision. The initial draft contains only the new facts. In the second step, the draft is incrementally revised in order to include secondary information such as historical facts. During revision, the system searches for appropriate insertion points called *hook locations*. The system then uses these locations to add "as many ADSS (additional deep semantic specifications) as it can", until it reaches some predefined constraint such as a maximum number of words or depth of embedding to ensure a readable text.

2.6.5 Summary

In summary, the above discussion of flexible systems — that allow some sort of revision of previous decisions — shows that revision is reasonable only under certain circumstances: when each decision can be considered in isolation from the rest in the process (e.g. the WEIVER system), when revision always starts from a predefined stage — whether it is a short or long distance back (e.g. the IGEN system), or when the system is designed for a specialised application (e.g. the STREAK system). Even

then, it has to pay the cost in the form of complex intermediate representation and/or involved control mechanism.

2.7 Stylistics-aware Generation: Promising Directions

> Style is created through subtle variation, seemingly minor modulations of exactly what is said, the words used to say it, and the syntactic constructions employed, but the resulting effect on communication can be striking. [DiMarco & Hirst 93]

In this research, we are concerned with sentence generation. The input is assumed to be a sentence-sized semantic representation and the output is an utterance that should have certain surface stylistic requirements. Whether the generated surface form meets these surface constraints depends on what paradigmatic decisions we have made and what the syntagmatic consequences of these decisions are. For example, choosing between active or passive is a paradigmatic choice which has different syntagmatic consequences: *actor* being before *pred* in the active case or *actee* before *pred* in the passive. Now choosing between different applicable lexemes for *actor*, for instance, is again a paradigmatic choice with its own syntagmatic consequences since any particular choice of lexeme constrains and is constrained by what comes before and after it.

[DiMarco & Hirst 93] identify four parameters that determine the stylistic feel of an utterance: lexical, syntactic, thematic and semantic aspects. They also, believe that "together, form and content create style". This togetherness property is reminiscent of the way the SFL linguistic theory handles lexical, syntax, and semantics. They are all inter-twined using one knowledge representation language [O'Donnell 94]. In this respect, the SFL is a plausible formalism for our stylistics-aware approach to NLG.

However, most systemic generators are based on the pipeline architecture which we think is rigid for our task, especially if anything were to go wrong down the line which would require us to undo one or more of the early decisions. Therefore, we need a search mechanism that can perform revision (whether explicitly or implicitly) efficiently. An AI search technique known as the Assumption-based Truth Maintenance System (ATMS) claims to be able to pursue multiple solution paths simultaneously

without much backtracking [deKleer 86]. This is a direction that is worth exploring.

2.8 Summary and Outlook

In this chapter, we gave an overview of the literature related to the task of lexical choice. We showed the significant interaction between it, the choice of system architecture, and the other syntactic and semantic decisions of NLG. We also briefly claimed that a linguistic theory called SFL is deemed to be appropriate for the kind of holistic approach to both lexical and syntactic representation. In the next chapter we present how SFL is used in systemic NLG systems. We pay particular attention to the notion of paradigmatic syntactic choice. We discuss the shortcomings of the existing systemic generation algorithms and what needs to be done in order to account for the surface constraints in the overall process. Our proposal for a new generation architecture is based on an AI search technique known as the Assumption-based Truth Maintenance System (ATMS) which is introduced in a subsequent chapter.

Chapter 3

Systemic Functional NLG

This chapter is concerned with systemic NLG. That is, NLG that uses Systemic Functional Linguistics (SFL) as its theoretical foundation. It first introduces SFL in general. Then it presents the main concepts of the Systemic Functional Grammar (SFG) formalism. Next, it discusses how these grammars are used in text generation. It describes how lexical choice is approached in traditional systemic generators. It characterises the generation algorithms of existing systemic generation systems. It discusses the limitations of the current implementations with regard to surface stylistic requirements and proposes a new way of dealing with these limitations.

3.1 Introduction

Although a linguist has much freedom to theorise about the different aspects of language, language is a complex phenomenon and no single theory can capture all its aspects:

> "Present-day linguists believe that there is, and should be, more than one way of studying language ... [it is] a strength of linguistics that there are these different approaches to the subject. Controversy is always a healthy sign. Language is so complex that no one approach can cover all its aspects." ([Berry 75], p.12)

Moreover, the way from theory to implementation (or computational applications) is long and implementors, under constraints to be explicit, usually make some compromises and simplifications. Some implementations may even pick only parts of the theory. Language, being complex, and linguistic theories being vague, it seems very natural to have many linguistic theories and, for each theory, different implementational approaches.

Having put things into perspective, we present, in this chapter, both a linguistic theory and its implementation. We then show that current implementations do not account for an aspect of language which we call the surface stylistic requirements. We then point the way towards improving on the current implementations of the theory. In fact, no other linguistic theory claims to do any better in this regard as the difficulty with surface stylistic faults is that we do not come to know them all until after we have generated the actual surface form.

3.2 Systemic Functional Linguistics SFL

Systemic Functional Linguistics (SFL), being a linguistic theory rooted in anthropology and sociology, views language as a social resource which the speaker and hearer use to communicate meaningfully. The theory in its current shape was developed by Michael Halliday in England in the early sixties. It has developed from the scale-and-category linguistics of that time which, in turn, emerged from the work of Professor J. R. Firth. Halliday refined the notion of system so that it is centred around the idea of paradigmatic choice which is motivated by the communicative goal that the speaker is trying to achieve. The choices that the speaker makes then result in certain syntagmatic consequences which build the syntactic structure of the utterance.

We do not intend to give a historical background of the theory and the motivation behind it. We will only present the main concepts that are needed to understand the computational aspects of it which have been implemented by various NLG systems. The suitability of SFL for natural language generation is established by the many successful generation systems using it as their theoretical basis, such as PENMAN, SLANG, and WAG.

In the remainder of this chapter, we will introduce the Hallidayan SFL grammar formalism. Then, we show how it is used by NLG systems. We then focus on lexical choice and discuss how it is dealt with in traditional systemic generators. Towards the end of this chapter, we discuss the limitations of the current generation algorithms with respect to our problem: surface stylistic constraints. We also discuss what is needed to compensate for the limitations of the current implementations of the theory.

3.3 Systemic Functional Grammars SFG

SFL views language as a social resource which the speaker and hearer use to communicate meaningfully. Hence, the SFL formalism is concerned with the choices available for the language user. Therefore, it emphasises the paradigmatic axis of language. The other axis of language is the syntagmatic (or structural) axis which is represented in terms of realisation statements.

Our discussion in this section will include two main parts. The first part is concerned with the paradigmatic axis of SFL. This includes basic notions such as systems, networks, delicacy, and dependency. The second part focuses on the syntagmatic axis. Related to this axis are the notions of realisation statements, constituency, and rank.

Together the paradigmatic and syntagmatic axes form the grammatical resource of language. It is worth mentioning at this point that Systemic Functional Grammars (SFG) do include vocabulary (or the lexis) and hence are referred to as lexico-grammars.

3.3.1 The System

The system is the basic means that expresses the paradigmatic choice a language offers in a particular context. A system is basically a list of choices that are available in the grammar of the language. A language can have many systems related to each other in different ways. For example, English has the GENDER system which offers a choice between *masculine, feminine,* or *neuter.* The POLARITY system enables us to choose between either *positive* or *negative.* Both the GENDER and POLARITY systems are shown diagrammatically in figure 3.1. The choices available in a system are called *features.* Sometimes, they are referred to differently in the literature (eg.

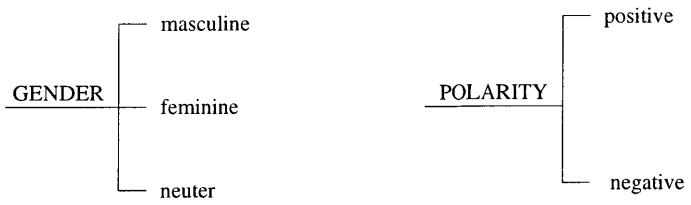


Figure 3.1: The English GENDER system and the POLARITY system

terms [Berry 75], or *classes* [Patten 88]).

According to [Berry 75], a system has three characteristics:

- The features of a system are mutually exclusive. So, we can only choose one and only one feature of a system.
- A system is finite. That is, it has a finite number of features. Therefore, all the features that are mutually exclusive of each other are available in the system. Others which do not have this property are not included in that particular system.
- The meaning of a feature in a system is relative to the meanings of the other features in that very system. For example, *plural* has a different meaning in the system (*singular, dual, plural*) from that in the system (*singular, plural*).

But how do we arrive at a particular system? In a linguistic context, there are certain circumstances that must hold before the features of a system (and hence the system itself) become available. This particular set of circumstances is called the *entry condition* of the system.

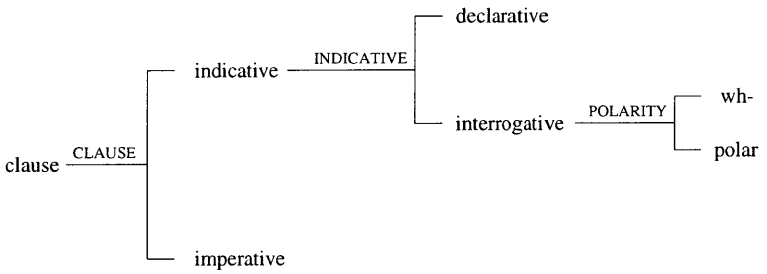


Figure 3.2: Related English clause systems

For example, provided that we choose *indicative* in figure 3.2, the entry condition of

the next system is satisfied. This way, the system (*declarative, interrogative*) becomes enterable. As a matter of fact, it is obligatory to get into a system whose entry condition has been satisfied [Berry 75].

Delicacy

Related to the notion of system is the notion of delicacy. Generally speaking, delicacy increases from left to right. A system Y to the right of another one X is more delicate. In figure 3.2, the CLAUSE system is the least delicate and POLARITY is the most delicate of all. The scale of delicacy is important from the meaning (or choice) point of view. As we move from left to right, we make finer and finer distinctions in meaning. That is why the systems become more and more delicate.

Dependency

Dependency is related to delicacy in that not only is a system to the right more delicate than one to the left, but also in that it *depends* on it. A system Y is *directly dependent* on another system X if it is immediately to the right of X. If it is more than one place to the right of X on the scale of delicacy then it is *indirectly dependent* on X. For example, the POLARITY system of figure 3.2 is directly dependent on the INDICATIVE system but indirectly dependent on CLAUSE. Direct dependency can be either simple or complex. These dependencies correspond to simple and complex entry conditions respectively. Complex dependency can, in turn, be conjunctive or disjunctive. This means that entry conditions can be simple and complex and that a complex entry condition can be either conjunctive or disjunctive.

Simultaneity

Sometimes there are systems that are not dependent on each other (i.e. they are independent of each other). These are called *simultaneous systems.* A system is simultaneous with another system if it is independent of the other system but has the same entry condition as the other one.

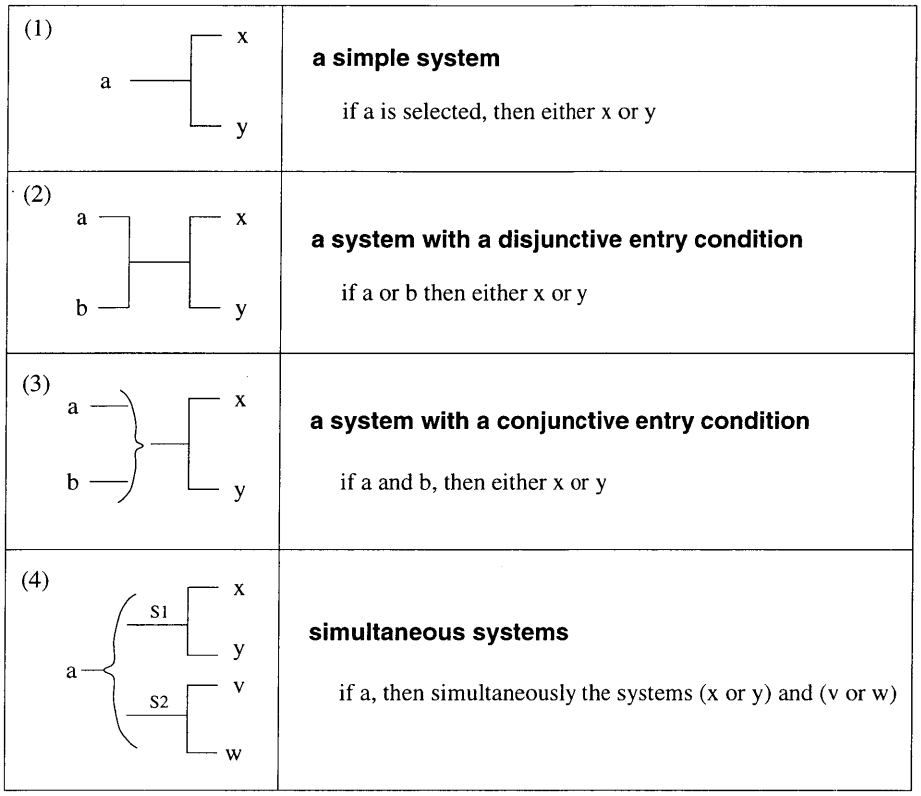


Figure 3.3: Different system types and their semantics

System Networks

As mentioned earlier, a language can have many systems related to each other in different ways depending on the scale of delicacy and dependency relationships. Related systems connected to each other form what is called a *system network.*

The basic constructs of a system network are shown in figure 3.3. (1) represents a simple system with the entry condition *a.* (2) shows a disjunctive entry condition to the system and (3) shows a conjunctive entry condition. The systems of (2) and (3) have complex dependency on the less delicate systems that are to the left of them. Finally, (4) shows two simultaneous systems S1 and S2. Any system network consists of (some of) these basic building blocks.

Some researchers take the stance that all the lexico-grammatical resources of the language represent one (large) system network [Tung *et al.* 88]. Others believe that there could be many system networks in the lexico-grammar. Accordingly, they de-

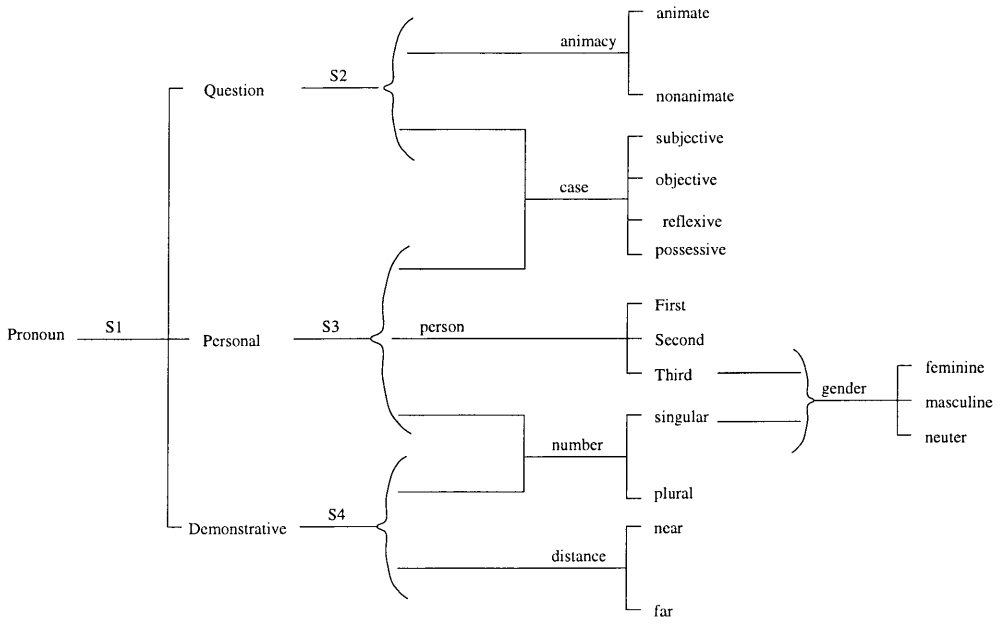


Figure 3.4: The English pronoun network

fine a system network as "a set of systems which are closely related from a semantic point of view" [Berry 75]. Hence, there is the MOOD network and the THEME network for example. Figure 3.4 shows the English PRONOUN network based on that in [Winograd 83].

3.3.2 Realisation

> "The scale of realisation is perhaps the most important scale of all. It shows how the different levels of language are related to each other; and within each level it shows how the different categories, such as structure and system, are related to each other." ([Berry 77], p.18)

The other axis of language is the syntagmatic axis which the SFL accounts for by means of realisation statements. Simply put, the syntagmatic axis represents the utterance (or written text in NLG) as it appears in space. That is, a linearised sequence of bits of language.

A feature in a system network may have attached to it a set of statements called realisation statements or rules. The operations carried out by the realisation statements build the syntactic structures. Each realisation statement makes some modification or

imposes some restriction on the structure being produced. The realisation rules can be thought of as the structural implications of the features that they are attached to [Patten 88].

There are three groups of realisation statements: those that build structure by introducing *functions* or collapsing them together (e.g. insert, conflate), those that constrain order of functions (e.g. order, partition), and those that associate features with grammatical functions (e.g. preselect, classify). By repeated use of the structure-building operations, the grammar is able to construct sets of *function bundles* [Mann 83a]. This way the paradigmatic features of the grammar are realised linearly by constituency structures. These constituents specify the usual linguistic functions such as *Subject, Agent, Actor, Goal* and *Predicate.*

Below, we discuss different realisation statements. Note that we only discuss those realisation statements that have been implemented in our generation system[1]. We will be referring to figure 3.5 in the examples given in relation to each realisation operation.

Insertion

The *insertion* realisation statement simply inserts or introduces a function into the structure. In work that uses SFL for parsing and generation, *insertion* is referred to as *inclusion* to avoid biased terminology towards generation [O'Donnell 94]. It is usually denoted by the symbol "+". For example, the feature *clause* has the realisation rule (+ Subj) meaning that clauses have subjects in them.

Conflation

The conflation realisation statement takes two functions. It means that these two linguistic functions will be represented by one entity in the structure. For example, the feature *modal* has the realisation rule *(Mod/Fin)* meaning that in modal clauses the *Mod* and the *Fin* are the same. Conflation is denoted by the symbol "/" in the network of figure 3.5.

[1] These are the same realisation statements that are implemented by O'Donnell's WAG system since our grammar is based on that of WAG's

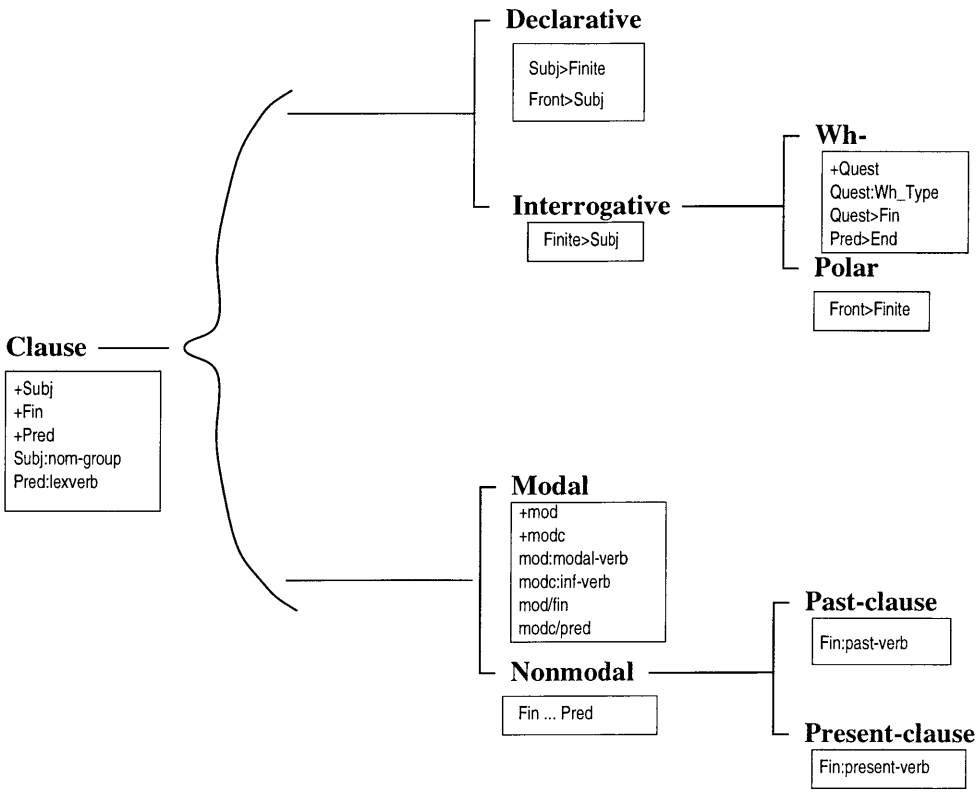


Figure 3.5: A clause network fragment with realisation rules in boxes

Order

The order realisation statement *(Function1 > Function2)* requires that the two functions are realised with immediate adjacency. Figure 3.5, for example, shows that in *declarative* clauses *Subj* precedes *Finite* as in "*John will* come". In *Interrogative* clauses, however, *Finite* precedes *Subj* as in "*Will John* come?". A special case adjacency is where an item is a leftmost or rightmost constituent and therefore adjacent to the borders. For example, in SLANG the feature *clause* has the realisation rule ($\sharp\wedge$ Theme) meaning that in all clauses Theme is always at the beginning [Patten 88]. To indicate these cases, some systemicists use special realisation rules such as "order-at-front" and "order-at-back" [Mann 83b]. WAG uses the meta functions *Front* and *End* to indicate that an item is a leftmost (e.g. *Front > Subj*) or rightmost (e.g. *Pred > End*).

Partition

The partition realisation statement *(Function1...Function2)* constrains *Function1* to be realised to the left of *Function2* but not necessarily adjacent to it. For example, attached to the feature *Nonmodal* of figure 3.5 is the partition realisation rule *Fin ... Pred*, meaning that *Fin* occurs before *Pred* in the syntactic structure. Depending on whether the clause is declarative or interrogative, the *Fin* function might be adjacent to *Pred* or separated from it by the function *Subj*.

Preselection

The preselection realisation statement is used to interface the different system networks. It is this realisation statement that makes the grammar recursive since it requires an inserted function to be filled by a "substructure which itself is licensed by a selection expression containing the preselected feature" [Henschel 97]. Preselection is usually denoted by the symbol ':'. For example, in the network of figure 3.5 *Subj* is preselected as *Nom-group* (i.e. *Subj:Nom-group*). This means that in order to realise the *Subj* function, we need to start with a path (in the group rank network) which includes the *Nom-group* feature.

Lexification

This realisation rule forces a particular lexical item to be used to realise a function. This is usually denoted by the symbols '='. For instance, the feature *speaker-subject* may be given the realisation statement (*Subject* ="I").

3.3.3 Rank and Constituency

> "If we are concerned with syntagmatic relations, then it is possible to make a very rough generalisation to the effect that clauses tend to consist of phrases[2] and phrases tend to consist of words - and words tend to consist of morphemes. In other words, the relation between them is one of constituency." ([Hudson 71], p.67)

[2] That is, groups in the terminology we adopted.

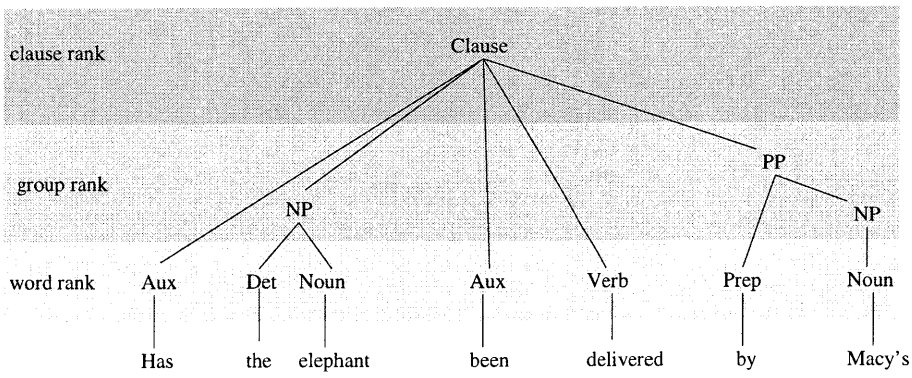


Figure 3.6: The rank and constituency relationship

Constituency is defined as "one structure having others as parts" [Winograd 83]. In general, a sentence (or a clause to use the SFL terminology) is a unit or constituent which consists of other constituents. The constituents of the clause are usually groups or words and the group constituents tend to be words. This constituency relationship is called the *rank* relationship. Figure 3.6 shows the constituency relationship of the sentence "Has the elephant been delivered by Macy's?" [Winograd 83].

The rank scale for the English grammatical units that are usually implemented in NLG systems are:

- clause-complex
- clause-simplex
- group
- word

A complete systemic approach would also represent words in terms of smaller constituents: morphemes. However, systemic implementations tend to skip this step and fetch the words directly from the lexicon.

Sometimes, a unit may include among its constituents, a unit of rank equal to or higher than its rank rather than units of lower ranks only, as mentioned above. This is called *rank-shifting*. For example, Deictic in a nominal group is usually of word rank (i.e. realised by a single word) as in "this idea". However, in "the software engineer's idea"

Deictic is a nominal group acting as a word.

3.4 How the Lexico-grammatical Resource is Used

The system networks together with the realisation statements represent the lexico-grammatical resource of the language. Traversing a system network from left to right and getting into any system whose entry conditions are satisfied results in what is known as a *selection expression.* A selection expression is a set of features forming a complete path through the system network. The collection of realisation statements attached to the features in the selection expression are the syntagmatic consequences of the paradigmatic choices we have made during traversal.

Recall that the notion of constituency is fundamental in the SFL theory. A sentence is a unit that consists of several constituents of different ranks which themselves have their own constituency structure. Using the lexico-grammar to generate sentences draws on this basic idea. We start with the clause rank and traverse the clause network. The resulting collection of realisation statements is then carried out to construct the structure of the topmost unit (i.e. the sentence or clause). Each of the clause's immediate constituents that are not of the word rank are realised in exactly the same way. At any point in this process, if a unit is of the word rank, we just realise it by fetching the appropriate lexeme from the lexicon.

[Winograd 83] gives the steps of the generation of a single constituent, which are more or less the same steps followed by most implemented systemic generators. These are shown in figure 3.7.

Figure 3.8 shows how the sentence "When will John come?" is generated using the network of figure 3.5 and following the steps of figure 3.7.

3.5 The Semantic and Lexico-grammatical Interface

We showed how a selection expression results in a set of realisation statements which, if carried out, can construct the syntactic structure and eventually result in a natural language text. But what guides us in constructing the selection expression in the first

1. Choose all of the features for the constituent (i.e. construct a selection expression).
2. Add to the structure all the functions required by these features (i.e. execute the *insertion* realisation statements).
3. Combine functions conflated by the features (i.e. do the *conflation* operations).
4. Assign features to each function (i.e. as specified by the *preselection* operations).
5. Fill in the realisation of each function (i.e. recursively generate each function).
6. Order the functions as specified by the *order* realisation statements.
7. Do any agreement or post-processing required.

Figure 3.7: The steps of realising a constituent

place? In other words, what makes us choose one particular feature from the set of alternatives offered by an entered system?

[Hudson 71] in his non-computational work uses the term 'semantic factors' to refer to what influences our choices of features in traversing a system network to realise a unit. In fact, the semantic and lexico-grammatical interface is part of the SFL theory. In systemic functional linguistics, systemic choice is meant to be motivated by the social context or situation:

> " If we are to relate the notion 'can do' to the sentences and words and phrases that the speaker is able to construct in his language – then we need *an intermediary step*, where the behaviour potential is as it were converted into linguistic potential ... There is nothing new in the notion of associating grammatical categories with higher level categories of a 'socio-' semantic kind." ([Halliday 73], p.51, 56)

Figure 3.9 shows the mapping between the three levels (or strata to use the SFL terminology). In an interactive context (say in a dialogue) the features *exchanging, demanding*, and *information* maybe chosen. This selection, in turn, is reflected in the grammar by certain features being selected (in this case: *interrogative* from the MOOD system) [Teich 99].

Step	Operation	Result
1	Selection Expression	(clause, interrogative, wh, modal)
2	Insert Functions	$(Subj, Fin, Pred, Mod, Modc, Quest)$
3	Conflate Functions	$(Subj, Fin/Mod, Pred/Modc, Quest)$
4	Do Preselection	($Subj$: nom-group, Fin/Mod: modal-verb, $Pred/Modc$: lex-verb, infinitive, $Quest$: wh-type)
5	Fill Constituents Recursively	$Subj$: traverse the group network; yielding "john", Fin/Mod: get "will" from the lexicon, $Pred/Modc$: get "come" from the lexicon, $Quest$: get "when" from the lexicon
6	Order Functions	$Quest > Fin/Mod > Subj > Pred/Modc$
7	Postprocessing	"When will John come?"

Figure 3.8: The steps of generating a sentence

Implementation wise, different NLG systems may do the inter-stratal mapping differently. Next, we show some of the different approaches for the implementation of semantic-grammatical interface.

3.5.1 Preselection-based Approach

In this approach, the mapping is done by means of some preselection realisation statements that are attached to the features of the semantic strata systems. The role of the preselection operations is to realise the semantic features by preselecting lexico-grammatical features particularly (but not necessarily) at the clause level.

[Patten 88] takes this approach in his SLANG generation system. He distinguishes this type of preselections from the lexico-grammar ones by calling them *inter-stratal preselections.* For example, the semantic feature *stated-cond* has realisation rules which preselect the clause-rank features *unmarked-declarative-theme* and *noncondition.* Other semantic features might directly preselect features from the group or word rank. For

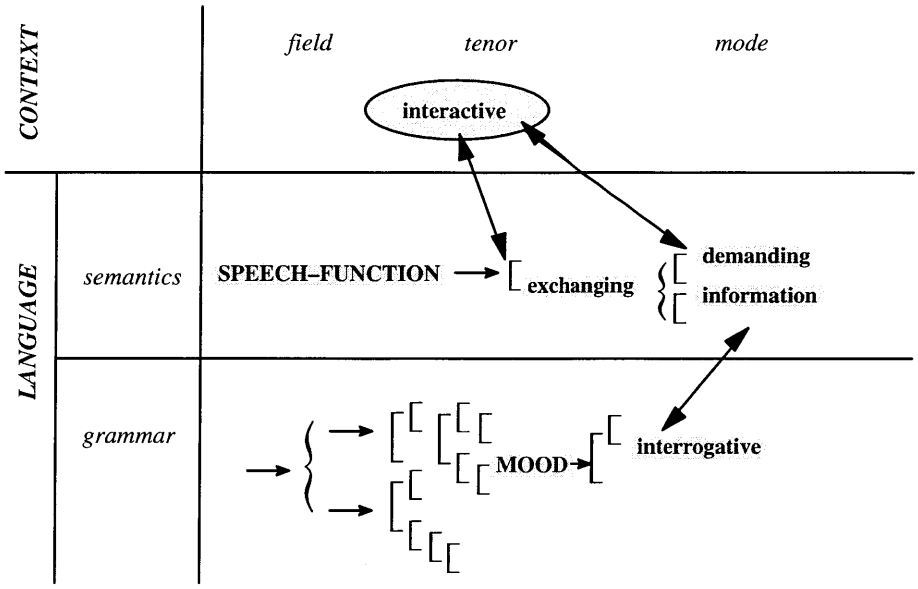


Figure 3.9: Inter-stratal mapping (from [Teich 99])

instance, the semantic feature *child-centred* might preselect *singular* from the nominal group network.

3.5.2 System-based Approach

The system-based approach attaches enquiry procedures to each system of the grammar. The task of these procedures is to help choose one of the features of the system that they are attached to by investigating the conceptual representation details.

The first NLG system to implement this approach was PENMAN. This mapping approach is called the *chooser-inquiry interface.* The chooser starts its work when the system it is associated with has been entered [Matthiessen 87]. Choosers are explicit procedures that examine a particular case and raise some questions (or inquiries) to help them understand that case and hence make the correct systemic choice.

Figure 3.10 shows the MOOD system and the chooser associated with it [Matthiessen 87]. A chooser (or choice expert) may have to go through several enquiries in order to come to terms with which features to choose. This results in a decision tree (similar to that of figure 3.11) with each of its leaves nominating the selection of a particular feature.

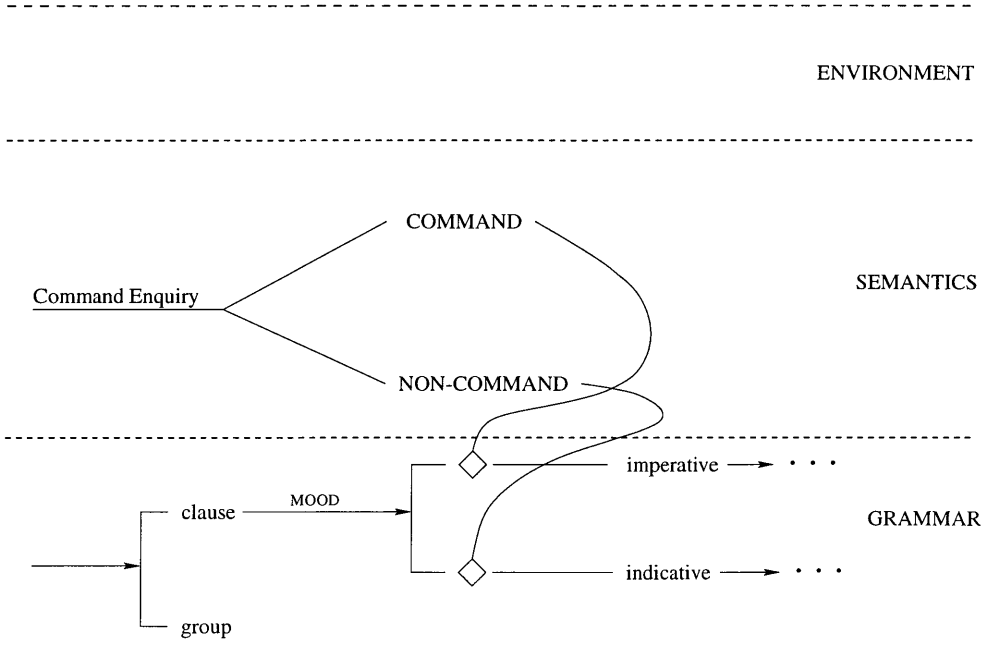


Figure 3.10: The MOOD TYPE system and its chooser

3.5.3 Feature-based Approach

The feature-based approach is an attempt to declarativise the procedural implementation of the chooser-inquiry framework [Kasper & O'Donnell 90]. Instead of attaching a choice expert to a system, *selection constraints* are assigned to the features of the system. [O'Donnell 94] uses this approach for the inter-stratal mapping in his WAG system:

> "In the WAG system, the mapping between two strata is represented by associating higher-level constraints with features of the lower stratum. Each feature of a network is assigned a *selection constraint.* If the feature is to be chosen, then its selection constraints must be satisfied." ([O'Donnell 94], p.80)

According to O'Donnell, this approach has some advantages over the chooser-inquiry formalism. It is a declarative formalism which allows bidirectionality. That is, the same formalism can be used for generation and analysis. Moreover, this formalism has its own constraint language which makes it portable, as opposed to the procedural implementation of PENMAN's choosers in Lisp.

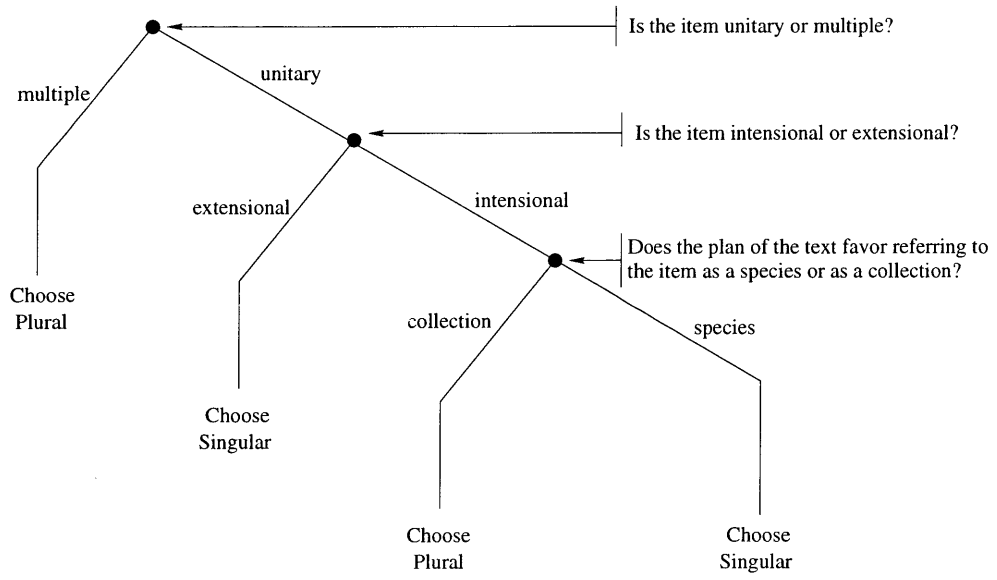


Figure 3.11: A decision tree for the chooser of a typical Number system (from [Mann 82])

3.6 Systemic Sentence Generation

There are many NLG systems inspired by the systemic functional theory. Examples of such systems include PENMAN [Mann 83b], WAG [O'Donnell 94], SLANG [Patten 86], KPML [Bateman 97a], and KOMET [Teich 95]. PENMAN is one of the earliest and most comprehensive generators of English sentences. PENMAN was under development from the early 1980s onwards at the Information Sciences Institute, University of Southern California. It provides a computational environment for generating English sentences, starting with non-linguistic input representation.

Some aspects of the PENMAN system (i.e. the chooser-inquiry interface) have already been mentioned in Section 3.5.2. However, we present an overview of it to give an idea of what constitutes a full sentence generator since it has become a reference model and later generators tend to compare their architecture with PENMAN's. At a gross level, PENMAN consists of the following three components:

- **The grammatical resource:** PENMAN's grammatical resource is called Nigel which is basically a systemic grammar of English. It is a network of over 700 systems, each representing a single minimal grammatical alternation. Some of

the Nigel-like systems were shown at different points in this chapter. Nigel also includes realisation statements which construct the syntactic structure based on the systemic choice of features.

- **The chooser-inquiry Interface:** This is one of the most important components of PENMAN. Its main task is to mediate between the semantic and grammatical resources. Its importance lies in the fact that any choice made affects the generated sentence later on, during the generation process. For a chooser to function sensibly, it must ask questions about particular entities of the conceptual input. For example, to choose between singular and plural, the chooser of that system must be referring to a particular entity in order to know whether it is unitary or multiple. In the PENMAN generator, choosers have access to the Function Association Table (FAT) [Mann & Matthiessen 83] which keeps a record of the associations between the grammatical functions and the corresponding semantic entities. Additionally, one of the tasks of the choosers is to make such an association and put it in the Function Association Table for other choosers to use when answering inquiries. For example, to realise the grammatical function THING which is associated to say *bicycle-x* (i.e. a particular semantic entity), the nominal-group network is traversed. Upon entering the Possessiveness system, say, the chooser of that system must associate the grammatical function DETERMINER with the conceptual entity representing the possessor (say *Tomy*).

- **The Upper Model (UM):** The upper model provides a domain-independent classification system of the world. It allows domain-dependent applications to use PENMAN as their text generation component by relating their world's entities to the objects of the upper model classification [Bateman *et al.* 90]. Figure 3.12 illustrates this idea diagrammatically. In fact, the choosers of PENMAN refer to the UM to answer some of their enquiries. Now, all that a new application has to do is relate its domain objects to the ontology of the UM.

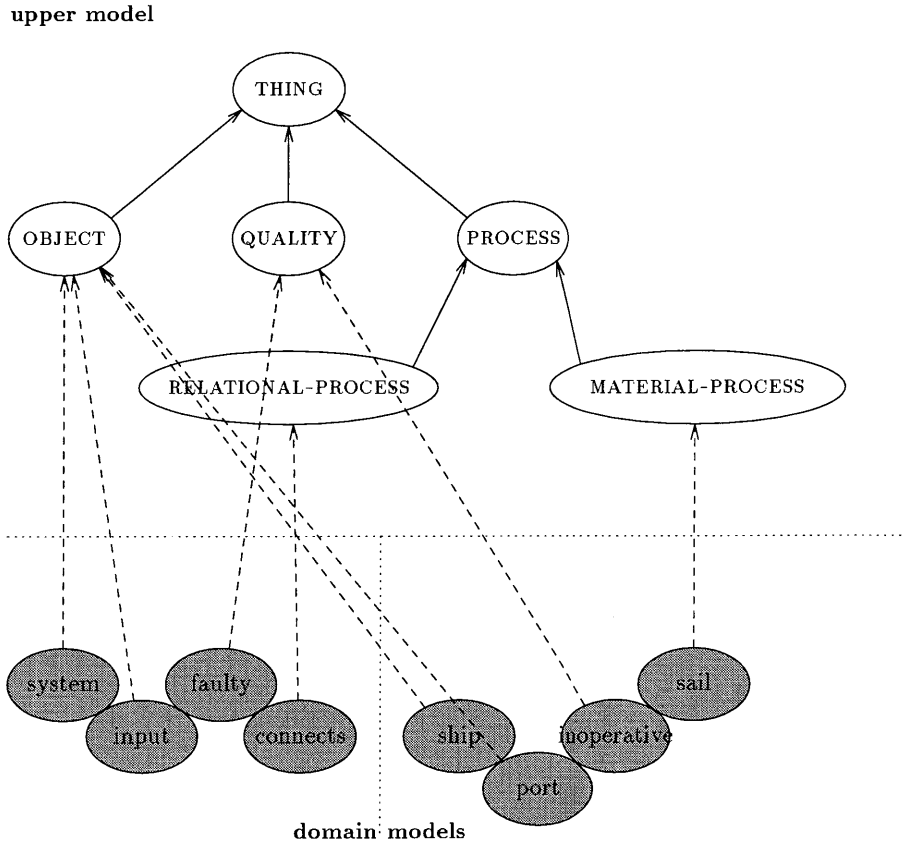


Figure 3.12: Different domains using the UM's taxonomy (from [Bateman *et al.* 90])

3.7 Lexical Choice in Systemic NLG

The design intention of systemic functional linguistics requires that both the lexicon and the grammar form together to make the lexico-grammatical resource which is used to realise the input semantic configurations. From an SFL point of view, the lexicon is regarded as a delicate extension to the grammar. The title of [Hasan 87] " The grammarian's dream: lexis as most delicate grammar" summarises most systemicists' view of the lexicon within the whole formalism. Of course, this stance is taken by Halliday the founder of the SFL theory:

> "Within this stratum there is no hard and fast division between vocabulary and grammar; the guiding principle in language is that the more general meanings are expressed through the grammar, and the more specific mean-

ings through the vocabulary." ([Halliday & Hasan 76], p.5)

Although different PENMAN-like systems follow different lexicalisation techniques, they have been faithful to the theory with respect to the timing of lexical choice. They tend to do lexical choice at the end (i.e. at the most delicate systems). During the expansion or realisation of a constituent, most of the choices made are syntactic. As soon as the delicacy reaches the word rank, PENMAN, for example, starts performing choice of a different kind: lexical choice. This state of affairs is described by [Mann 83b] as follows:

> "For every functionally characterised constituent there is the possibility either (i) of re-entering the system network to make further choices to determine its internal structure or (ii) of *going to the lexicon.*" ([Mann 83b], p.67)

As a result of this systemic viewpoint, lexical choice was not given enough attention. The notion of a system and its paradigmatic choice has always been in the focus. Lexical choice is considered a consequence of these early choices.

Therefore, one of the limitations of the traditional systemic approach to lexical choice is that it can only account for the paradigmatic dimension. Traditionally, lexicalisation of a function is not considered in relation to the surrounding lexemes and hence the other dimension (i.e. the syntagmatic dimension) was not accounted for. Lexical choice in systemic functional generation is done for a unit of word rank in isolation of the context in which it will appear. In case of a syntagmatic stylistic fault in the surface form, a conventional systemic generator is likely to reach a dead-end.

Even later work on PENMAN, which specifically addresses lexical choice, restricts itself to the paradigmatic aspect of the problem. For example, [Sondheimer *et al.* 89], in their work on lexical choice within PENMAN, explicitly point out aspects that are not addressed:

> "We discuss only lexical choice based on purely referential (or ideational) considerations. That is, we do not deal here with the discourse-related issues *surrounding lexical choice*, such as: (a) choice between synonyms,

> based on considerations of speech register, style, lexical cohesion, etc. ... (d) choice based on conventional associations between lexical items (e.g. collocations)." ([Sondheimer *et al.* 89], p.2)

Another limitation of the traditional systemic lexical choice is that no bidirectional interaction between lexical choice and syntactic choice is possible despite the fact that both the lexis and grammar are intertwined in one formalism. Lexical choice is always done at the end and it is always constrained by the syntactic choices made earlier. The "result is that word choice cannot constrain or influence syntactic choice" [Edmonds 95]. A word that best describes a concept but cannot fit into the syntactic structure will be disregarded. Ideally, it should be allowed to influence the syntactic decisions as well.

Having discussed the problem with the systemic approach to lexical choice, we discuss, in the next section, the limitations of the systemic generation algorithms in general. We identify the main problem and suggest a way of compensating for this shortcoming.

3.8 Limitations of Current Generation Algorithms

Current systemic generators are designed to deterministically generate text. As Bateman puts it [Bateman 97b]:

> "The generation algorithm [...] supports efficient implementations: there is no backtracking or non-determinism to incur performance overheads. The traversal of the network directs the generation process precisely as required by the input specifications. The price for this efficiency is the reduced range of interactions possible across different structural ranks and the requirement that choosers always produce a determinate response when triggered." ([Bateman 97b])

The assumption that the generated text will be of 'adequate quality' explains the reluctance of the systemic generators to examine the generated text from a stylistic point of view. Here we identify two problems with these generators that render them inadequate for dealing with the surface stylistic constraints.

- On the one hand, SFL is usually driven by the social situation (and in an NLG context by the semantic input). This suggests that once the situation is explained in every detail then finding our way through system networks is almost deterministic. This approach seems to ignore the surface form and how it, in some situations, restricts content determination, lexical choice, and syntactic structures.

- On the other hand, when the semantic input is under-specified, the choosers in systemic generators resort to selecting arbitrary paths or marking some as default choices. This entails the danger of unexpected surface problems that might arise later on.

What makes it hard to account for the surface form constraints in the overall process is the fact that we do not come to know them all until after the sentence has been generated. Failing to meet the surface stylistic requirements means that backtracking is the inevitable expensive alternative in the endeavour of exploring different paths through the system networks. Needless to say, efficient techniques are necessary in order to keep the computational costs within an acceptable range. That is, backtracking[3] should involve undoing only those decisions that need to be changed and no more.

Even in the early stages of PENMAN development there was an acknowledgement of the fact that generated texts might need some improvement and hence alternative paths may need to be explored:

> "It is surprising how much progress has been made in text generation based on generators which do no more than produce "first draft" text ... PENMAN does not try to anticipate the major determinants of readability in the text it is producing. Sentence length, levels of clause embedding and the like are difficult to anticipate but trivial to measure after the text has been generated. Very simple measures of text quality, including these and also some comparative measures ... seem to be quite adequate as a basis for suggesting helpful revisions in text plans ... The improvement Module has

[3] In Section 4.10.1 of the next chapter, we discuss the disadvantages of backtracking generation.

> not been implemented; its design includes particular critic processes, repair proposing processes and repair introduction processes." ([Mann 83b], p.8)

However, no details were given as to how this improvement might be achieved. Also, and to the best of our knowledge, such revision modules have never been implemented within the PENMAN project.

Next, we summarise the limitations of the current SFL implementations with regard to lexical choice in particular and generation architecture in general. We briefly present our ideas for enabling a systemic generator to deal with surface stylistic constraints.

3.9 Discussion: Choosing not to Choose

We opt to use SFL as the linguistic formalism for our generation system for many reasons:

- SFL lends itself very easily to natural language generation.
- There exist a large number of NLG systems inspired by the linguistic theory.
- With the surface stylistic problem in mind, the SFL seems very appealing because of the way it handles language at different levels in one unified formalism. In particular, semantics, syntax, and the lexicon can all be represented using system networks.

However, in the discussion in Sections 3.7 and 3.8, we identified the following limitations with the current implementations of SFL:

- **Determinism:** Teich characterises the main task of PENMAN-like systems as follows:

  > "The generation question put for PENMAN-style generators is , when to choose which features from the system network. Or, more precisely, which functional alternative to choose in a particular context." ([Teich 99], p.60)

Therefore the main task of systemic generators is choice. However, this is done in a deterministic fashion based on input semantics only. Given a semantic input specification, choosers always produce the same selection expression. Even for cases where the alternatives are all equally good, they just default on one of the features.

- **One-way Architecture:** Later tasks are only influenced by earlier ones. They are not allowed to contribute to the generation process by feeding their own constraints. In fact, these later choices are not real choices; they are merely consequences of earlier choices as Berry points out:

  > "As we move nearer the surface on the scale we may appear to be making further decisions but in reality we are simply implementing earlier decisions. All the choices we make between functions, structures and classes of formal items are the direct result of the choices we made between terms [i.e. features] from systems." ([Berry 77], p.43)

- **Isolated Syntagmatic Operations:** The partial realisation operations are done in isolation of the context they will appear in. This applies to lexical choice, ordering operations and syntactic structuring as well.

Making unfortunate choices leading to unwanted consequences is a known problem in systemic generation. [O'Donnell 94] points out that the solution to this problem is either: simultaneous generation or backtracking generation which is very expensive according to his experience in the WAG system. Simultaneous generation means that multiple paths are pursued simultaneously. When one of them fails, it is taken out of consideration and the search continues with the remaining promising paths.

The Assumption-based Truth Maintenance System (ATMS) is an AI search technique that can work in multiple contexts. It claims to be able to pursue multiple solution paths simultaneously without much backtracking [deKleer 86]. Our idea is to use the ATMS to enhance the main task of any systemic generator: choice. In case, a chooser cannot decide between many features, it does not have to choose. The very nature of the ATMS will then be utilised to pursue the paths emerging from each kept alternative.

The proposed ATMS-based architecture is deemed to overcome the limitations of the traditional pipeline architecture of systemic generators as we will show in subsequent chapters. By keeping all plausible alternatives, the architecture allows for non-deterministic generation. The same semantic input may result in the generation of different paraphrases depending on the surface stylistic constraints.

Although the proposed ATMS framework is not a two-way architecture in the sense that control is passed back to earlier stages, it does allow for a two-way interaction so that later stages can influence the decisions of earlier stages. For example, a lexical choice that results in a problematic syntagmatic arrangement may urge the selection of a different syntactic structure.

Finally, the syntagmatic dimension of the systemic formalism is given more attention in our architecture. The realisation rules are compositionally treated in such a way as to predict any source of surface stylistic faults.

3.10 Summary and Outlook

In this chapter, we discussed systemic functional NLG. We first discussed the grammatical formalism of the SFL theory. This included both the paradigmatic and the syntagmatic dimension of systemic functional grammars. We showed how the lexicogrammatical resource is used to generate natural language utterances. We then discussed different approaches to the inter-stratal mapping between semantics and lexicogrammar. We showed what constitutes a typical systemic generator taking PENMAN as an example. The systemic approach to the lexical choice problem along with its limitations were also discussed. Finally, we discussed the limitations of the systemic generation architecture and proposed a new architecture that overcomes these shortcomings. The next chapter introduces the ATMS framework: what it is, how it works, and why it is plausible for our generation task.

Chapter 4

The ATMS Framework

This chapter introduces the Assumption-based Truth Maintenance System (ATMS) formalism. First, it presents the generic notion of truth maintenance and discusses the need for belief revision in AI problem solving. It presents the families of TMSs and then focuses on the ATMS family. It discusses some of ATMS's basic concepts such as nodes, justifications, and label maintenance algorithms. It gives explanatory examples and discusses the problem of encoding. Different ideas on how to use the ATMS to construct solutions are also presented. The chapter ends with a discussion of efficiency considerations.

4.1 Introduction

In the preceding chapters, we argued that in order to satisfy the surface stylistic requirements in systemic generation, some sort of interaction between lexical choice and syntactic choice must be allowed. In particular, lexical choice — being done at the very end — should be allowed to influence earlier NLG decisions. This can be achieved either through what [O'Donnell 94] calls backtracking generation which was shown (by O'Donnell himself) to be prohibitively expensive, or through simultaneous generation which would be just as expensive if all possibilities were generated.

We also pointed out that the ATMS seems plausible for efficient multiple contexts generation as it allows common parts to be factored out and built only once, which the

basic simultaneous generation will not necessarily do. As a matter of fact, our generation problem is similar to the type of problems generic truth maintenance systems (TMS) are designed for:

> " A common example is the use of plausible assumptions when reasoning about an incompletely specified problem. This may lead to difficulties when it is discovered that some initially plausible choice is, in fact, inconsistent with what we later find out. Reason Maintenance Systems are intended to address this problem by providing machinery that allows the consequences of assumptions to be determined and the set of assumptions revised, if needed."([Smith & Kelleher 88], p.5)

Our solution to the Surface Stylistic Constraints (SSC) problem involves a new NLG architecture based on the ATMS framework. But before we present our ideas and implementations of them (see in particular Chapter 5, 6, and 7), we introduce the basic concepts of the ATMS and before that the notion of Truth Maintenance in general.

4.2 Truth Maintenance Systems (TMS)

As we will see later on, the ATMS is one type of a group of families of TMS. Before we focus on the ATMS, we introduce the generic notion of TMS. We discuss the need for TMS and the advantages that they offer to AI reasoning in general.

A Truth Maintenance System (TMS) is defined as a program that maintains the consistency of a database. It does so by keeping track of dependencies between items in the database collection [Shoham 94, Russell & Norvig 95]. A TMS is always used within problem solving systems in conjunction with an inference engine (IE) which, while solving the problem, informs the TMS about the facts or assumptions that can be stated and the deductions that can be made from them. Figure 4.1 shows how the IE and TMS are put together to form a problem solving system. This arrangement seems appealing as it allows each part to focus on certain aspects of the problem solving. The IE focuses on the particulars of the task at hand and the TMS on the bookkeeping of beliefs and assumptions [Forbus & de Kleer 93]. The IE, being the *real* problem solver,

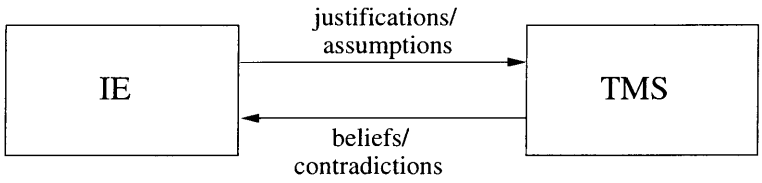


Figure 4.1: Problem solving systems

includes all the domain knowledge and inference procedures. It passes inferences made during problem solving to the TMS. The job of the TMS is to determine under what conditions any conclusion holds. The problem solver then interrogates the TMS to know exactly what holds at any point in time. To this end, problem solving can be viewed as a "process of accumulating justifications and changing beliefs until some goal is satisfied"[deKleer 86]. In order for the IE and TMS to communicate, they need to establish a common vocabulary between them. This communication is expressed in terms of nodes and justifications. It is worth noting, however, that the TMS does not know the meaning of nodes and inferences. It does the house-keeping on a purely syntactic basis[1].

Doyle's pioneering work in the late 70s marks the actual emergence of the Truth Maintenance System [Doyle 79]. [Smith & Kelleher 88] traced some of Doyle's ideas in his first TMS back to the late 60s and early 70s where they were found dispersed vaguely in different works.

TMSs are referred to differently in the literature. Although Doyle called his first system a TMS, he then admitted that it was a bit of a misnomer. Consistency maintenance or belief revision are better names for describing the nature of work such systems do. Reason(ing) Maintenance Systems (RMS) is also a widely used term for the notion of TMS.

The motivation behind TMS is the need to perform belief revision. Sometimes we make assumptions at a certain stage in problem solving, but we might discover later on that these assumptions were wrong. Deciding what effect this new information has on the rest of our beliefs and separating the affected beliefs from the intact ones is a

[1] Some TMS families have access to the semantics of the problem representation. For example, the so called Logical-based TMS knows the difference between p and $\neg p$.

costly task that needs to be done efficiently.

Belief revision is a common task among many search problems, so why not have a separate belief revision component (i.e. TMS) that does the bookkeeping for the problem solver? The TMS, being a specialist in belief revision, caches expensive inferences in order to avoid having to redo work that has already been done.

According to [Smith & Kelleher 88], the advantages of having such an architecture (which was shown in figure 4.1) are as follows:

- Increased design clarity: since the problem solver is now separate from the bookkeeper.
- Improved search efficiency: the TMS is a specialised reasoning module which can improve the overall efficiency of the system.
- Decreased redundancy of computation: the TMS can cache some of the inferences so that it avoids expensive computations further down the line.
- Access to the consequences of choices: the TMS maintains a dependency network for all that is passed to it by the problem solver. This way, any choice can be traced to its consequences or antecedents.

Because of these advantages, TMSs have become a common and important piece of AI problem solving and have been used in many applications. For example, they are used in fault diagnosis [deKleer & Williams 86], qualitative reasoning [Forbus 87], uncertainty modelling [Haenni 98], user modelling [Jones & Millington 88], and expert systems [Morgue & Chehire 91].

4.3 Families of TMS

There are two main families of TMS exemplified by Doyle's truth maintenance system and de Kleer's truth maintenance system. Doyle's system is a justification-based TMS (JTMS), while de Kleer's version is an Assumption-based TMS (ATMS). In JTMS, dependency between nodes is stored in terms of justifications while in ATMS, dependency is stored in terms of assumptions.

[deKleer 86] gives the following list of TMS families in an increasing order of complexity:

- Justification-Based Truth Maintenance System (JTMS):
  This is a simple TMS where one can examine the consequences of the current set of assumptions. It operates on a set of nodes (representing beliefs) and justifications which represent the causal relationship between the nodes. It is justification-based because of the way it records dependencies between the nodes (i.e. only the immediate relationship between a node and its cause is recorded). A JTMS works in a single context, so that if the problem solver needs to switch between contexts, a JTMS must rederive the database truth assignment for every context.

- Logical-Based Truth Maintenance System (LTMS):
  Like the JTMS, the LTMS reasons with only one set of current assumptions at a time. It is more powerful than JTMS in that it recognises the propositional semantics of sentences, i.e. it understands the relations between p and $\neg p$, p and q and $p \wedge q$, and so on.

- Assumption-Based Truth Maintenance System (ATMS):
  This allows a system to maintain and reason with a number of simultaneous, possibly incompatible, current sets of assumptions. Otherwise it is similar to JTMS in that it does not recognise the meaning of sentences. Compared to the JTMS, the ATMS requires a large computational overhead. However, it can switch contexts instantaneously and it does not need to backtrack to recover from inconsistencies.

- Non-Monotonic Justification-Based Truth Maintenance System (NMJTMS):
  This is similar to a JTMS except that it accepts non-monotonic[2] justifications.

[2] This definition of non-monotonicity is de Kleer's own. However, in the non-monotonic reasoning literature, non-monotonicity is defined as the property where on learning a new fact we may be forced to change our belief about other facts [Ginsberg 87]; whereas in monotonic reasoning, as the set of beliefs grows, so does the set of conclusions that can be inferred from them. According to this widely accepted definition, all families of TMS do some sort of non-monotonic reasoning. In fact, [Doyle 79] has made it clear that his TMS is trying to provide an efficient framework for default reasoning which is one type of non-monotonic logic according to [Shoham 87].

Non-monotonic TMSs accept justifications of the form "whenever P is true, Q is likely".

- Clause Management Systems (CMS):
  A CMS is similar to an ATMS but can represent any propositional calculus formula. The concept of a clause management system was first proposed by R. Reiter as a generalisation of de Kleer's original ATMS [Reiter & deKleer 87].

The last two types of TMS are not in wide use because of their inherent complexities [Forbus & de Kleer 93]. In particular, the CMS, which can handle general clauses and not only Horn clauses, is very expensive [Kohlas *et al.* 98] and hence not appropriate for large problems. It is quite important to choose the right truth maintenance system for the task at hand, as each TMS is designed to answer a particular pattern of questions more appropriately and efficiently than others.

In this thesis, we adopt de Kleer's version of the truth maintenance system. It is called the Assumption-based Truth Maintenance System (ATMS), since dependency between the nodes in a network is recorded in terms of assumptions. In the next section, we introduce the ATMS basic concepts.

4.4 The ATMS: Basic Concepts

[Shoham 94] differentiates between the ATMS and other simpler types of truth maintenance systems based on the kind of questions they can answer. Basic TMS answers the question: "Given the following justifications and premises, is this particular conclusion warranted?". An ATMS, on the other hand, answers the complementary general question: "Given the following justifications, under what assumptions would the following conclusion be warranted?". So, an ATMS not only gives a yes/no answer but also more informative answers in the form of different assumption sets, each independently warranting the conclusion. To this end, the ATMS can be viewed as a method for capturing multiple TMSs in a single structure. Below, we introduce the ATMS basic ideas of nodes, justifications, dependency networks and labels.

4.4.1 Nodes

A TMS node holds an important problem solver datum. Implementation-wise, the data structure for a datum points to the corresponding TMS node, and the data structure for a node points to the corresponding problem solver datum.

There are various kinds of nodes in an ATMS:

- A *premise node* holds unconditionally.
- A *contradiction node* is one that never holds according to the IE.
- An *assumption node* may or may not hold.

In addition to these basic node types there are derived nodes. A *derived* node is one which is neither a premise nor assumption node but which is derivable from collections of them. A derived node holds unconditionally if it depends solely on premise nodes; otherwise it may or may not hold.

4.4.2 Justifications

Inferences made by the IE and communicated to the TMS are in the form of justifications. Generally speaking, a justification is just an *if-then* rule describing the causal relation between a consequent and its antecedents. At any point in time, introducing a new justification may cause any belief to change. An ATMS justification takes the form: $\langle antecedents \rangle \rightarrow \langle consequent \rangle$. For efficiency reasons, ATMS justifications are restricted to propositional Horn clauses. Hence, the $\langle consequent \rangle$ is just a single node c and the $\langle antecedents \rangle$ part is a conjunction of nodes $a_1 \wedge ... \wedge a_n$. That justification can also be written as: $\neg a_1 \vee \neg a_2 \vee ... \vee \neg a_n \vee c$. If $n = 0$ then the node c is called a premise.

4.4.3 Dependency Networks

ATMS nodes and justifications together form a configuration known as the *dependency network* of the ATMS. A dependency network encodes the causal relation between

the ATMS nodes. Dependency networks are sometimes referred to as *causal networks* [Haenni 98] or *belief networks* [Laskey & Lehner 88]. Throughout this thesis, we will adopt the graphical convention shown in figure 4.2 to represent ATMS depen-

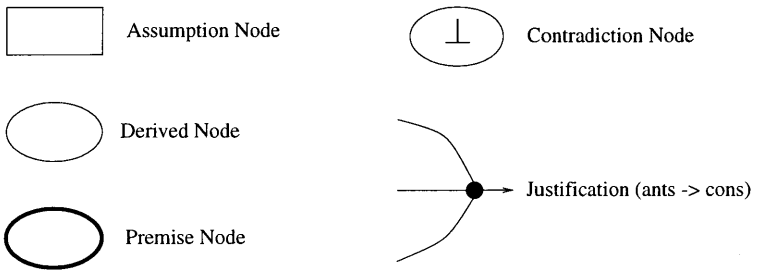


Figure 4.2: The graphical convention for dependency networks

dency networks. Figure 4.3 is a dependency network that corresponds to the clause system network shown in figure 4.4. The dependency network states that an utterance is a clause if it is either indicative or imperative. In turn, a clause is indicative if it is either declarative or interrogative. The dependency network, additionally, encodes the fact that a proposing or initiating utterance contradicts with the assumptions *interrogative* and *imperative*.

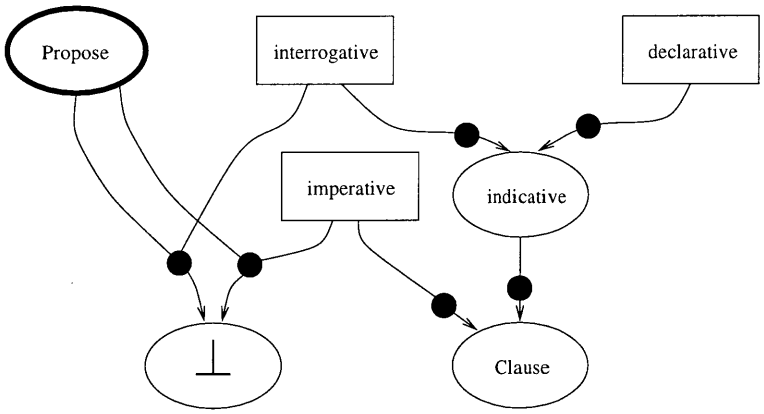


Figure 4.3: An example dependency network

4.4.4 Labels

A label is used in conjunction with a node to store the ATMS' representation of current belief in this node. It is a parsimonious description of the consistent contexts in which

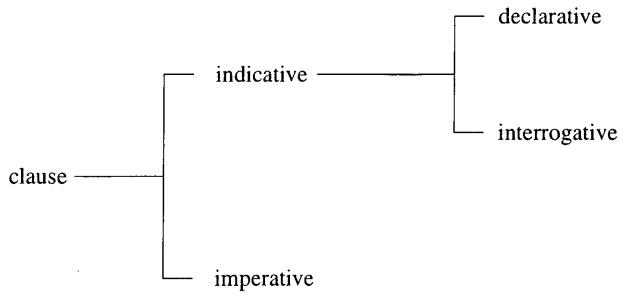


Figure 4.4: A fragment of the English clause network

the node holds [Forbus & de Kleer 93]. The ATMS precomputes the node labels in order to be ready to answer the IE queries. As a result, at any point in time, the label of a node consists of the sets of assumptions under which the node holds.

Next, we present some of the terminology that is related to node labels and which we will be using in our discussion about label maintenance.

- An *environment* is defined as a set of assumptions.
- A node *holds* in an environment if it is logically derivable from that environment and the justifications provided so far.
- A *nogood* is an environment in which some contradiction holds.
- A *consistent* environment is one which is not *nogood.*
- A *context* of an environment is the set of nodes that hold in that environment.

In order for the ATMS to answer queries about whether or not a node holds in some environment, it records in the label of each node all the consistent environments in which that node holds. In fact, it records the minimal consistent environments in which a node holds, because if a node follows from an environment then it follows from any superset of it. Also, *nogoods* are not included in node labels as such contexts are usually of no interest. Figure 4.5 shows the same dependency network of figure 4.3 but with the labels shown for each node as they would have been computed by the ATMS algorithms. In fact, the ATMS would prune out the nogoods. For example, the label of the INDICATIVE node does not include $\{interrogative\}$ since it is a nogood environment (cf. the label of the $\perp$ node).

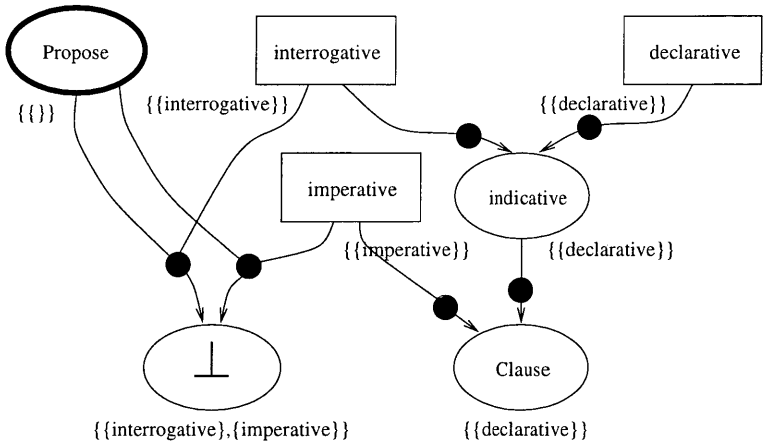


Figure 4.5: The dependency network of figure 4.3 with computed node labels

An empty label for a node indicates that the node does not hold in any consistent environment. This case happens either when there is no path from the assumptions to the node or when all the potential environments are *nogood*. The case where a label consists of the empty environment only indicates that the node holds in every environment. Premise nodes have labels consisting of the empty environment since every environment is a superset of the empty set. For example, the premise node *Propose* in figure 4.5 has a label that consists of the empty environment.

Formally speaking, let the set S stand for the set of ATMS nodes. The set A of assumptions of an ATMS is a subset of S (i.e. $A \subset S$). Let C be the set of clauses of the ATMS. C consists of premises and justifications asserted so far. An environment is defined as a subset of the assumptions $E \subset A$. Now, for each node n, the ATMS maintains a label or a set of environments $E_1, ..., E_k$ such that it has the following properties:

1. n holds in each E_i. This is called the Soundness property.
2. $\perp$ cannot be derived from any E_i given C. This is called the Consistency property.
3. Every consistent environment E in which n holds is a superset of some E_i. This is called the Completeness property.
4. No E_i is a proper subset of any other environment. This is called the Minimality property.

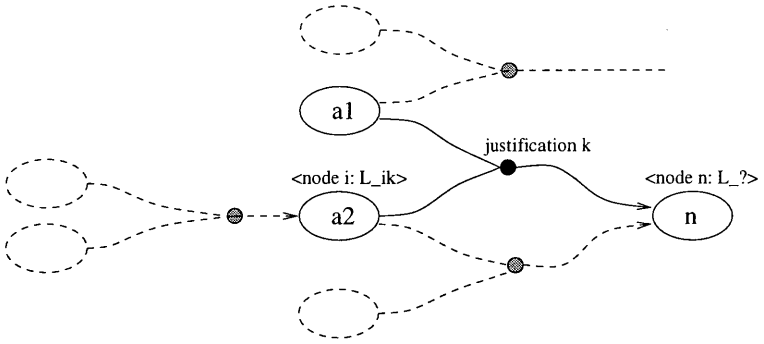


Figure 4.6: ATMS incremental label update for some node n

4.5 ATMS Algorithms for Label Maintenance

Recall that the job of the ATMS is to record under what sets of assumptions any conclusion holds. For this it attaches to each node a label which is a set of environments (i.e. a set of sets of assumptions) from which the node follows. If a node leads to a contradiction, then the ATMS knows that all the environments labelling it lead to a contradiction. The way the ATMS records this knowledge is by having a special *false* node (or $\perp$) whose label contains all the contradictory sets of assumptions. Actually, the *false* node label contains all the minimal inconsistent sets of assumptions. This means that all the assumptions in a minimal inconsistent set cannot hold simultaneously but any subset of such a set can.

It is worth mentioning here that the label of each node is *minimal* in the sense that it has the fewest disjuncts (i.e. environments) and that each environment has the fewest conjuncts of assumptions. Therefore, the ATMS needs to constantly check that no superfluous assumption appears anywhere in a set in any label.

The ATMS operates in an incremental way. At any point in time, it has an access to a correct set of node labels. Whenever a new justification is added, it uses the current labels to compute the incremental changes incurred by the addition of that justification. This is depicted in figure 4.6 which is better viewed in conjunction with Algorithm 1 shown below.

In general, there are three main ATMS activities for label maintenance: computing a locally correct label, propagating label changes throughout the dependency network,

and pruning nogood environments from all labels. Next, we present these algorithms in pseudo-code to give an idea of the ATMS internal activities. For detailed and more efficient versions of these algorithms see [Forbus & de Kleer 93]. We present the algorithms here because we will need to refer to the details later on when we discuss efficiency considerations in Section 4.9.

Algorithm 1: Computing a Locally Correct Label for a Node

1. Compute a tentative label L' for the node n. Let L_{ik} be the label of the ith node of the kth justification for some node n. Then, $L' = \{\bigcup_i e_i \mid e_i \in L_{ik}\}$
2. Remove from this tentative label:
   - any set which has another in the label L' as a subset.
   - any set which has a nogood environment as a subset.

Algorithm 2: Propagation of Label Changes Throughout the Network

1. If the node (say n) is newly created, compute its label as described in algorithm 1. This becomes the label of n.
2. If the node already exists (and has its own label), compute a new label for it using algorithm 1. Then blend the new label with the previous one. That is, take the union of the two labels and remove from it any set that has another as a subset. Stop if the label has not changed.
3. If it has changed, then check whether it is a contradiction node. If it is, then call algorithm 3 (shown next).
4. If the node is not a contradiction node and other nodes depend on it, then propagate *recursively* the changes to those nodes, and from them to others; and so on. That is, call Algorithm 2 for all the consequences of n.

Algorithm 3: Pruning of Nogood Environments

1. Mark as *nogood* the newly discovered environment found to be inconsistent.
2. Remove it and any superset of it from every label in the network.

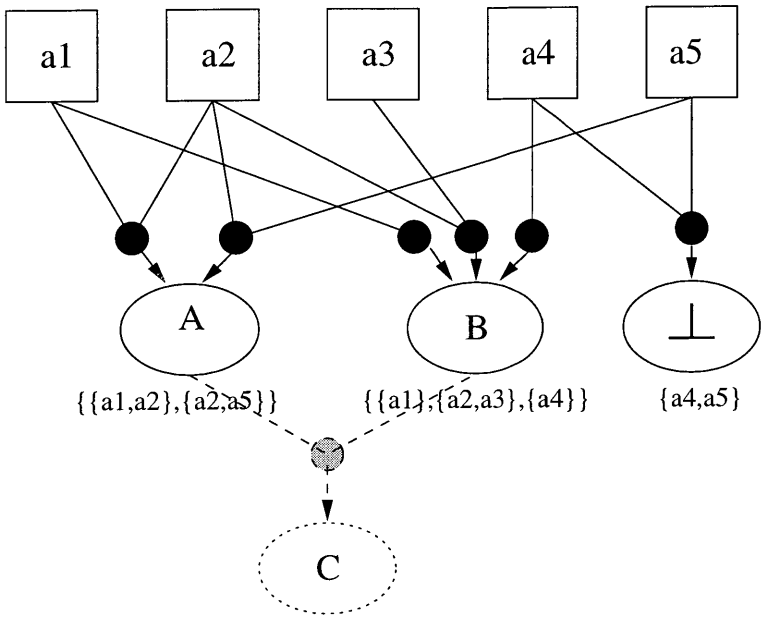


Figure 4.7: Graphical representation of a dependency network

4.6 How the ATMS Works

The ATMS maintains a dependency network consisting of nodes and justifications as shown in figure 4.7. So, all of a1 to a5 are assumptions and the nodes A and B are derived nodes. The dependency network shown in the figure represents graphically the following justifications:

$a1 \wedge a2 \rightarrow A$

$a2 \wedge a5 \rightarrow A$

$a1 \rightarrow B$

$a2 \wedge a3 \rightarrow B$

$a4 \rightarrow B$

$a4 \wedge a5 \rightarrow false$

To have an idea of how the ATMS works we consider a fragment of an example large enough to explain the basic notions of the false node and also label maintenance such as pruning and propagation.

To see what happens when new justifications are added, assume that we have the

setting shown in the solid part of figure 4.7 and that the IE is about to add another justification stating that A and B imply C (i.e. $A \wedge B \rightarrow C$) as shown by the dotted part of the figure. If it is the first time that C is encountered, a new node for it is created with an empty label. The ATMS then records the justification and creates a new label for C from the already existing labels of A and B which are {{a1,a2},{a2,a5}} and{{a1},{a2,a3},{a4}} respectively. It does so by proceeding as follows:

- **Step 1:** Compute a tentative label L' for C as described in Algorithm 1. This gives the following sets: {a1, a2},{a1, a2, a3}, {a1, a2, a4},{a1, a2, a5}, {a2, a3, a5},{a2, a4, a5}.
- **Step 2:** Remove any set which has another as a subset (e.g. {a1, a2, a3}, {a1, a2, a4},{a1, a2, a5}), and all sets having contradictory sets of assumptions (e.g. {a2, a4, a5}). This yields {{a1,a2}, {a2, a3, a5}} as the correct label for C.

There is a set of standard operations that define the interface between the IE and the ATMS. Although different implementations name these operations differently, what matters is the kind of service the ATMS offers when each operation is performed. The following are examples of such operations:

- create an assumption node
- create a derived node
- create a false node
- explain the node or equivalently return its label
- add a new justification
- add a new contradiction
- inquire whether a node is a premise, assumption, or derived node

4.7 The Encoding Problem

Although the interface to the ATMS is simple and is defined through a set of standard operations, deciding what and how to tell the ATMS is not a straight for-

ward task. In the truth maintenance context, this is known as the encoding problem and "experience suggests that providing an appropriate encoding can be very hard" [Smith & Kelleher 88]. In the case of the ATMS, the encoding problem involves two main questions:

- What data is going to be recorded in the ATMS, and, for every type of data what type of node should represent it. In particular, what information should be represented as ATMS assumptions.
- Having decided on what should be represented as assumptions, premises, and derived nodes, the other question is how we should connect these nodes in the ATMS dependency network. That is, which of the problem solver inferences are to be passed to the ATMS as not all inferences made may need to be reported.

For example, if we are to use the ATMS to reason with systemic grammar networks, then what should be represented by ATMS nodes: system labels, features and/or realisation statements? And if features are to be represented by assumptions, then should all features in a network be represented as assumptions or only the features at the leaves of the network? In some cases, it may be possible to infer the complete path by knowing only the last feature. Is it cheaper to construct the whole selection expression starting with the leaves or let the ATMS return the full selection expressions? And how about realisation statements? Are they to be represented by assumptions, premises or derived nodes? And if not by any type of nodes then what aspect of their effect is to be communicated to the ATMS and how?

These are all important questions and answering them is in essence solving the encoding problem which "has a vital bearing on the actual performance of the system" [Smith & Kelleher 88].

4.8 Solution Construction

Part of solving the encoding problem for a specific domain involves defining what constitutes a solution and how to read off such solutions. Although "the exact definition of a solution is task specific and is determined by the inference engine" [Forbus & de Kleer 93],

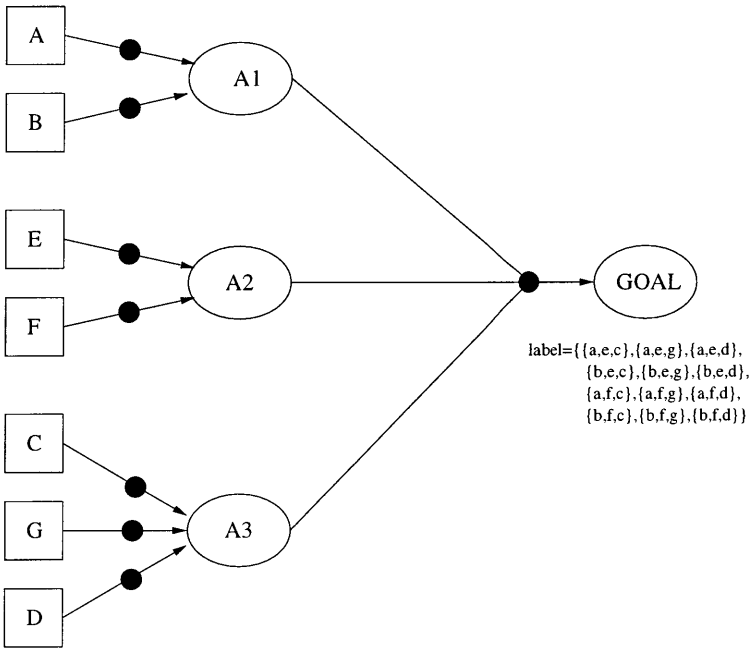


Figure 4.8: Solution construction exploiting the nature of ATMS

there are some known methods for constructing solutions. [Forbus & de Kleer 93] give two techniques for identifying solutions in an ATMS-based problem solver. They are both applicable when the task at hand is reducible to the following model: at different stages of problem solving we are faced with choice sets (which we may represent as assumptions), and any complete solution must pick one and only one choice from each set. Fortunately, this model fits our generation problem (i.e. traversing systemic grammar networks).

The first technique, which we elaborate on below, involves the introduction of new *goal* nodes and justifications in a prescribed manner. The solution can then be read off directly from the label of the *goal* node.

The second technique relies on a specialised solution construction procedure which takes a set of choice sets and returns a set of environments representing the solutions. This method produces the same solutions as the first one but without the need to introduce additional nodes and justifications. To this end, some ATMS implementations have a utility function (which might be called `interpretation`) whose job is to return all possible solutions given a set of choice sets.

As an example for the first technique of solution construction, consider the dependency network in figure 4.8 which shows a case where we have three choice sets {A,B}, {E,F}, and {C,G,D}. Any complete solution should pick one and only one element from these choice sets. In an NLG context, these choice sets might stand for action lexemes {*destroy, annihilate*}, actor lexemes {*man, chap*}, and global syntactic structures {*declarative, interrogative, imperative*} for instance. The figure also shows some nodes (A1, A2, A3, GOAL) and justifications ($A \rightarrow A1, B \rightarrow A1, E \rightarrow A2, F \rightarrow A2, C \rightarrow A3, G \rightarrow A3, D \rightarrow A3$) which have nothing to do with the task-specific nodes and justifications. According to the way the ATMS works, the *goal* node holds if at least one assumption of each choice set holds. This means that all of {A,B,E,F,G}, {A,E,C,G,D}, and {A,E,F,C,G,D}, for instance, can be legitimately in the label of the *goal* node. However, because of the minimality property of the ATMS labels, the label of the *goal* node contains one and only one choice from each choice set which is exactly what we want since, for example, a sentence cannot be declarative and interrogative at the same time.

This makes the setup of figure 4.8 an obvious way of constructing solutions. The *goal* node label in this case contains all the possible combinations. It is worth mentioning that the *goal* node along with its predecessors (A1, A2, A3), which we call *pre-goal* nodes, are all dummy nodes which have nothing to do with the task-specific nodes and justifications. Needless to say, the task-specific justifications would prevent some environments from appearing in the goal node label, as problem solving usually renders some environments contradictory. These *goal* and *pre-goal* nodes can be constructed upon request by a specialised procedure which takes the list of choice sets as an argument.

4.9 Efficiency Considerations

In this section, we discuss some ATMS efficiency considerations. There are two factors that influence the overall efficiency of a problem solver: the ATMS internal behaviour, and the IE decisions as these specify the kind and amount of work to be done by the ATMS. We discuss these two parts here and how the overall efficiency can be improved.

4.9.1 ATMS Complexity

The ATMS performs four basic operations for the inference engine – these are:

- create a new assumption
- create a new derived node
- record a justification
- return a node's label

The key to any ATMS implementation is to perform label updating as efficiently as possible after any of these operations. Fortunately, some operations require no changes to the ATMS labels, specifically – the creation of a new derived node and fetching of a node label.

However, the creation of a new assumption doubles the search space and consequently the size of each node's label, since the space size is 2^n where n is the number of assumptions. Likewise, the introduction of a new justification affects the behaviour of the ATMS but in a complex way. The effect of a new justification depends on what the antecedents and the consequent are. For example, if the consequent label changes then these changes have to be propagated to each node for which it is an antecedent, as outlined in Algorithm 2.

[deKleer 86] claims that the ATMS is designed in such a way that even when the number of assumptions is very large (say $n = 1000$), it is still practical to use. According to him, there are two reasons for this efficiency. Firstly, it is only sufficient to record the smallest environment in which a node holds since it is in every superset context as well. Secondly, where most environments are nogoods, these inconsistent environments are usually small subsets of assumptions. Therefore, a great deal of the search space needs never be checked for consistency.

4.9.2 Implementation Optimisations

The ATMS spends most of its time doing set unions and subset tests. Existing ATMS implementations use different techniques to speed up processing time. Also, label

updating algorithms use many heuristics to improve performance. For example, in one paper, [deKleer 86] describes many such optimisation methods. For instance, sets can be represented by using bit-vectors in order to improve both the cost and the number of set operations. Hash tables can be built using these bit-vectors as keys. The set of all environments and the set of all nogoods can be organised by length to speed the set operations. Also, since assumptions consume space resources, he describes a procedure for garbage collecting assumptions.

When optimisations of serial ATMS implementations had gone to the limit, researchers started exploiting developments in hardware design. For example, [Dixon & deKleer 89] show "how the combination of a conventional serial machine and a massively parallel processor can dramatically speed up the ATMS algorithms".

Other attempts to improve the efficiency of the ATMS include using a form of best-first search, for problems that only require approximate solutions. This is done by assigning weights to assumptions so that only the *most likely* interpretations are explored [Provan 88].

4.9.3 The IE Duty

Having decided to use an ATMS as a basis for a problem solver and whatever the speed of that ATMS implementation is, there is still something the IE can do to improve the overall efficiency of the problem solver. As Smith puts it:

> "Since the ATMS pays no attention to the meaning of the inferences passed to it, it is entirely the problem-solver's responsibility to see that they are both correct and useful. Correctness should be easy to ensure, because the problem-solver does have access to the meaning of the inferences, but it may be much more difficult to ensure that the inferences passed to the ATMS will allow the problem-solver to get useful information back."([Smith 88], p.159)

For example, in the solution construction network presented in figure 4.8, creating the *goal* and *pre-goal* nodes at the very end of the problem solving process greatly

improves the efficiency. There are three reasons why one would do that. Firstly, if these nodes are created at the outset of the process, then the size of the goal label will be exponential in the size of the choice sets. That is why it is always better to create these nodes after most or all of the *nogoods* have been determined. Secondly, this way, the IE (the generation part in our case) does not have to know in advance the choice sets. As a matter of fact, most choice sets would not be known until after some stages of the generation process take place. Thirdly, the ATMS does not need to waste time updating (rather shrinking) the *goal* node label after every task-specific justification is added but just creates this label once.

Additionally, one can improve on this by controlling how the goal node and its justifications are created. For example, providing the individual justifications (A $\rightarrow$ A1, B $\rightarrow$ A1, ...etc) before the GOAL justification ($A1 \wedge A2 \wedge A3 \rightarrow GOAL$) is more efficient as the consequent nodes A1, A2, will not be antecedents in any justification recorded so far and hence there is no need to propagate the change to its consequences. Moreover, when it comes to the costly justification ($A1 \wedge A2 \wedge A3 \rightarrow GOAL$), the order of the antecedents does matter. As a general observation, nodes representing choice sets with more of their environments participating in nogoods should come first in the antecedents' list of a justification. To understand what we mean by ordering the antecedent nodes, let us consider how the *goal* node label in figure 4.8 is computed. Suppose that the environments {F,G} and {F,C} are *nogoods*. When the justification ($A1 \wedge A2 \wedge A3 \rightarrow goal$) is encountered, then in order to construct the *goal* node label, the ATMS needs to perform set unions of A1 and A2 first and then the intermediate result with A3. The first operation results in four environments: {A, E}, {A, F}, {B, E}, {B, F}. None of them will be crossed out based on the information about *nogood* environments. The next operation will union these environments with {C}, {G}, {D} creating 12 environments which all need to be checked against the *nogoods* to determine which one to delete. On the other hand, if ($A2 \wedge A3 \wedge A1 \rightarrow GOAL$) is given instead, the first operation will give rise to six environments; two of which will be crossed out giving eight environments in the next final step. Therefore, the order of the antecedent nodes of a justification does have an effect on the amount of work the ATMS must do.

Generally speaking, creating the goal node and its justifications at the end of problem solving, and using the information in the false node label can help improve the efficiency of the generation process using the ATMS approach.

4.10 ATMS for NLG: why?

Recall that our aim in this work is the efficient generation of sentences that have certain surface stylistic requirements (SSR). Generating syntactically correct sentences is not enough though; the sentences have to satisfy the surface requirements or otherwise they are considered faulty. Therefore, the nature of our generation task requires multiple path exploration because the generation process is prone to failure, especially given that choice in current systemic grammars is not motivated by any SSR. Even if it was, the particular SSR might not be appropriate for a given generation task.

When systemic choice fails to give an acceptable surface form, [O'Donnell 94] suggests either backtracking generation, or simultaneous generation (such as chart techniques) to explore other alternatives. In this section, we discuss these two options and compare them to the ATMS search mechanism. Before that, we summarise the main advantages of the ATMS which are:

- work needs to be done only once, as inferences valid in several contexts can be carried out in a single operation. That is, it is sufficient to record only the smallest environment in which a node holds since it is in every superset context as well.

- sources of inconsistencies are cached in an economical way (i.e. minimal sets causing contradictions) so that futile paths are avoided as well as any expensive computations incurred by them. In cases where most environments are nogoods, these inconsistent environments are usually small subsets of assumptions. Therefore, a great deal of the search space needs never be checked for consistency.

4.10.1 Backtracking Generation

A backtracking algorithm proceeds in a depth-first manner as far as possible before it backtracks to the last choice point and tries the next alternative path. Here, we compare the behaviour of such an algorithm with that of the ATMS using a simple generation problem.

Suppose that we have a very simple grammar with one system only having the features (*nontopical*, *topical*). The implication of choosing *nontopical* is to order *Predicate* before *Subject*. *Topical* on the other hand orders *Subject* before *Predicate*. Suppose that the lexical choices for *Subject* are *Tom* and *the·boy*, and those for *Predicate* are *ar·rived* and *came*. Note that the dots here separate the syllables in a function's lexical realisation. Suppose further that we have the following SSR:

- no sentence ends with the letter *e*.
- no sentence consists of more than three syllables.

In effect, there are three choice points in this generation problem and the resulting search space is shown in figure 4.9. The search space has eight possible solutions, only four of them are stylistically licensed.

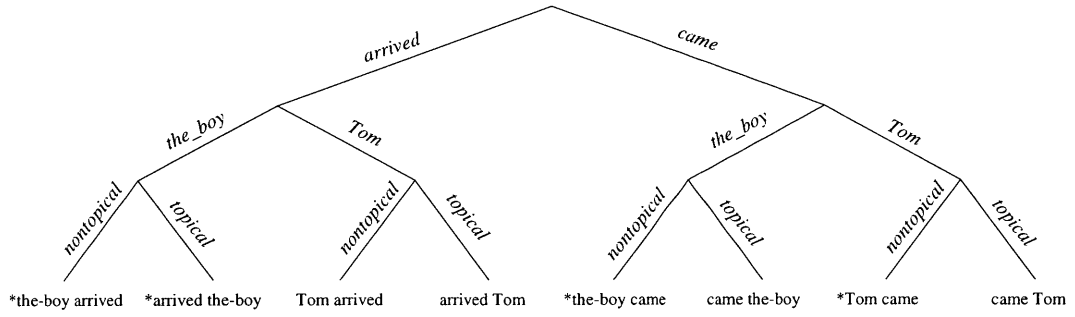


Figure 4.9: The search space for the simple generation task

According to its definition, backtracking generation proceeds to the last point it can reach and when it discovers that it has made a wrong choice somewhere, it backtracks to the last choice point and picks the next alternative. This way, it wastes effort by rediscovering contradictions. For example, although the choices *ar·rived* and *the·boy* are contradictory (since together they form a four-syllable utterance) it still generates

"the boy arrived" and "arrived the boy" which both fail. Moreover, when search reaches (*came, the·boy, nontopical*) and abandons it because it ends with *e*, it still visits (*came, Tom, nontopical*) although it will fail for the very same reason, since what is causing the problem is the combination of choices (*came, nontopical*).

An ATMS instead recognises the sources of inconsistencies and caches that information for later checking. So before it gets into any search branch it checks its records for sources of inconsistencies. This way it avoids any unnecessary computations related to these futile paths. The saving that the ATMS offers in this example is depicted in figure 4.10, which shows the paths that would be visited by an ATMS-based search algorithm.

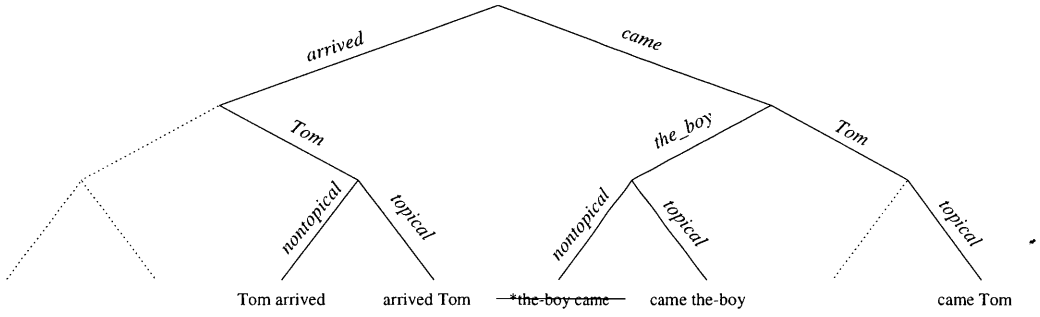


Figure 4.10: The branches of the search space visited by ATMS-based search

The other disadvantage of backtracking generation is that intermediate results are lost once the algorithm leaves that particular context. For example, the checking of the last letter of *Tom* although done for the path (*ar·rived, Tom, topical*), still has to be done again for the path (*came, Tom, topical*). Imagine how efficiency would be affected if this operation was an expensive one.

The ATMS on the other hand can recognise that the computation of the last letter depends only on *Subject* (*Tom* in this case) and *topical*. It performs this computation at this point, and it can use it at later points in the search space, e.g. at (*came, Tom, topical*).

4.10.2 Chart Generation

A chart-based search algorithm overcomes one of the main advantages of backtracking. We have already mentioned in the previous section that backtracking is inefficient because the same pieces of work are redone many times during search. Chart techniques were first used in parsing for efficient analysis of texts. Completely parsed constituents once found are stored in the chart, or the **well-formed substring table**, for later use. This way, work is done once and only once. In most cases, it is cheaper to get a value from the chart than to compute it again. The idea of charts has also been applied to NLG such as in the work of [Kay 96], [Nicolov 99], [Shemtov 98], and [Haruno *et al.* 93, Haruno *et al.* 96].

The main advantage of chart-based search is the notion of memoization which means that well-formed substrings are stored in the chart so that work related to their construction is done only once. The noun phrase *the boy*, for instance, is constructed once and can be used in any appropriate structure. If the solution *Tom came* fails, *the boy came* is built using the already constructed substrings *the boy* and *came*. This solution fails though because it ends with the letter *e*.

The ATMS is also motivated by the same idea. Therefore, the phrase *the boy* is constructed once. However, the ATMS nodes can cache all sorts of finer level decisions, such as ordering information and lexical properties. Chart techniques cache results only at the level of whole phrases. For example, when *Tom came* fails, the ATMS can record the reason of failure (e.g. $ends\_with(came, e) \wedge nontopical \rightarrow \bot$). Now, although it does not have to construct *the boy* again, it will not try the path (*came, the boy, nontopical*), because this path contains a contradictory combination (i.e. it is a superset of a nogood).

Another feature of the ATMS is that it can work in multiple contexts. Unlike chart techniques, an ATMS-based generator can pursue independent subgoals and the combinations of their results with efficient label operations.

We are inclined, at this point, towards the ATMS and not the chart option, because of the above reasons as well as the following:

- The grammar we use (i.e. systemic functional grammar) uses a similar representation to the ATMS dependency networks. This makes the ATMS a plausible option. In the next chapter we explore the logical relation between systemic networks and ATMS dependency networks. Chart-based techniques are traditionally related to context-free grammars and the algorithm details assume such a linguistic formalism (e.g. the fundamental rule).

- Although the SSR are not part of the lexico-grammatical resources, they can be represented using the same formalism (i.e. ATMS nodes and justifications) as we will see in the subsequent chapters. This way they constrain the generation process and reduce the search space considerably.

- As NLG is still considered in its infancy compared to NL analysis, it is always beneficial to experiment with new system architectures. As a matter of fact, "continued development of grammars and *generation methods*" is considered a longer-term goal by researchers in the field [Hovy 96]. In this project, we develop an ATMS-based generation architecture to see what advantages it brings to the NLG field.

4.11 Summary and Outlook

In this chapter, we have presented the ATMS framework. We started with the generic notion of Truth Maintenance System and the motivation behind such systems. We then focused on the ATMS: what it is and how it works. We discussed the encoding problem and presented some ideas on how to construct solutions using the ATMS. We then shed some light on some efficiency considerations of the ATMS. Towards the end, we discussed why we think the ATMS is plausible for systemic NLG.

This chapter concludes the first part of this thesis: the background chapters. In the second part, we start introducing our ideas and their implementations. In the next chapter, we explore the logical relationship between the Systemic Functional Grammars (SFG) and the ATMS dependency network representations. Our transformation algorithm which takes system networks and puts them into ATMS representations draws upon this logical relation between the two formalisms. In Chapter 6, we take

the translation idea further. We show how the translated networks are used in a new ATMS-based generation architecture.

Chapter 5

From SFG to ATMS

This chapter explores the logical relationship between Systemic Functional Grammars (SFG) and the ATMS. In particular, it relates the logical specification of system networks to that of ATMS dependency networks. This relationship helps us solve the problem of encoding; an essential question that needs to be answered in any application attempting to use the ATMS as its reasoning engine. We first motivate the idea of multiple path traversal of system networks, and justify the use of the ATMS to do that kind of network traversal. We then focus on the logical specification of both formalisms. The chapter culminates with a logically-founded translation algorithm that takes general systemic grammar networks and transforms them into dependency networks, a representation that the ATMS can reason with.

5.1 Introduction

The ATMS is a common tool for maintaining multiple sets of beliefs efficiently, thus allowing search in multiple contexts at the same time. We present our ideas of how to use the ATMS to find all selection expressions of a system network or part of it. But first, we discuss why one would want to pursue several system network paths in the first place (Section 5.2) and what implications such an approach might have on the outcome of the overall process. In an SFG context, following multiple paths simultaneously raises the issues of determinism and nondeterminism in NLG which we

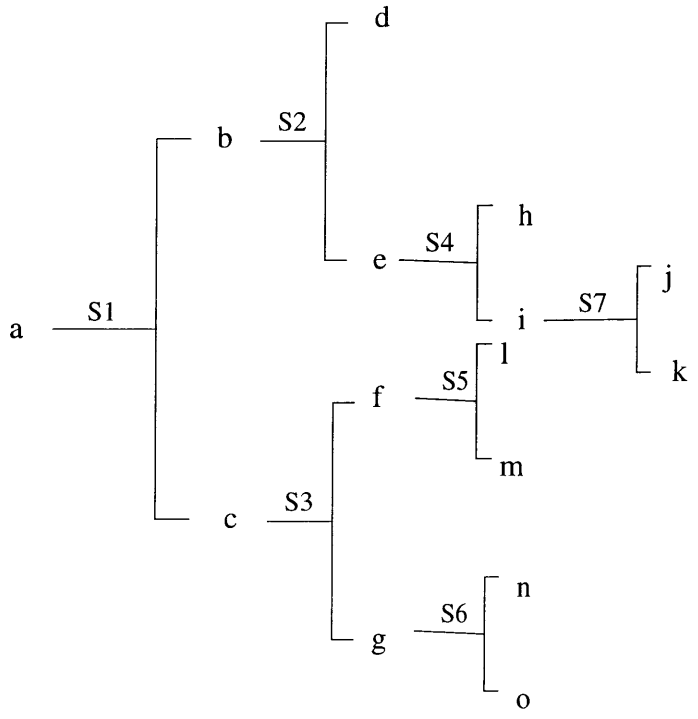


Figure 5.1: A simple system network

also discuss briefly in Section 5.2 along with a justification for our use of the ATMS as a search technique. The rest of the chapter (from Section 5.4 onwards) is dedicated to exploring the connection between SFG network and ATMS dependency network specifications.

5.2 The Need to Find Multiple Selection Expressions

System networks are used to represent systemic functional grammars. They specify how combinations of features may imply or be inconsistent with other combinations [Mellish 88]. They are largely used in linguistic applications, but they can be used in other applications, as there is nothing intrinsically linguistic in their nature.

To give a better idea of what a system network is, we will show how they are used in current generation systems. Given a system network such as the one in figure 5.1, to use this network for generation, one would start from the first feature a, get into the system $S1$ for which a is an entry condition, and then choose one of $S1$'s children. If one has no reason to prefer one over the other, then one chooses randomly or defaults

to the last feature (or the first in some generators). The process of getting into a system as soon as its entry condition is satisfied and then choosing one of its children is repeated until no more enterable systems exist. The result of this process is known as the *selection expression.* This is a set of features forming a complete path through the system network.

Each feature may have attached to it a set of realisation statements. Therefore, each selection expression brings with it an accumulation of realisation statements which if executed build the syntactic structure and produce the final surface form. Among the many selection expressions of a given system network, traditional systemic generators are only interested in one of them. This is due to the simplifying assumption that they can always choose from alternatives in any generation task. This way of generation is called *deterministic generation.*

Generally speaking, deterministic generation always forces a decision to be made at any point while non-deterministic generation, on the other hand, tries to follow all open alternatives. Their respective advantages are efficiency and flexibility. Deterministic generators are efficient mainly because they work quickly and simply. For similar conceptual inputs, the generator produces the same utterance. There is no need for backtracking to undo some of the earlier decisions. Nondeterministic generators are more flexible in that if they discover that they have made a wrong choice, they may revise earlier decisions. Consequently, for similar conceptual inputs, a nondeterministic generator can respond differently depending on other non-ideational considerations, e.g. stylistic aspects.

One limitation of deterministic generation is that "there is not always sufficient information to make the decision available" [O'Donnell 94]. Although systemicists argue that if the grammar is well constructed then choice is always possible, in reality this is not always the case. There will be situations where we cannot make an informed decision as all the choices at that point seem equally good [Bateman 97b, O'Donnell 94]. In such situations, one can only make uninformed decisions (either default or arbitrary choices). This may lead to dead-ends as choices are sometimes dependent on each other and if we made the wrong decision at one choice point we might not be able to find a valid alternative at a later point. Even if no dead-ends are encountered, the quality

of the generated text might be degraded as a result of an uninformed decision taken earlier.

In systemic functional generation, it cannot be stressed enough that making a wrong choice has its effect on subsequent choice points that follow from it. One might be under the illusion that one is still making choices towards the leaves of the system networks but actually one is not. Most of the later choices one makes are direct results of earlier decisions [Berry 77].

In this project we value the surface form and we search for an architecture that allows the surface stylistic requirements to be satisfied, especially since they cannot possibly be anticipated in earlier choices in the grammar as surface problems do not show clearly until after the surface form has been generated. To avoid making wrong choices, we would like to pursue all possibilities until we have good reasons to abandon one path or another. Considering the network of figure 5.1 again, suppose that S_1, S_3 and S_4 are systems where no choices are obvious. This means that we are going to pursue paths starting from their children. What we will have then is a reduced version of the original system network, which is still a system network, albeit a smaller one.

We call this smaller network (e.g. network of figure 5.2) a *tailored* system network and the process of removing the irrelevant parts – *tailoring* the system network. We would like now to pursue the different paths and see what consequences each one of them has. That is why we are interested in finding multiple selection expressions (or equivalently, all the selection expressions of a *tailored* system network).

The direct result of pursuing multiple paths is a set of selection expressions instead of just a single one. These selection expressions can result in many surface forms (i.e. nondeterministic generation) which differ significantly in their surface stylistic characteristics because of differences in syntax, word order, morphology, appearance or absence of certain function words, lexical items, ... etc.

In the non-deterministic paradigm, a generator does not have to make a decision when all the alternatives are equally good. It either follows all alternatives simultaneously, or follows one path tentatively and when it discovers that it was not the right decision it backtracks to an earlier choice point (see Section 4.10.1 for a discussion on back-

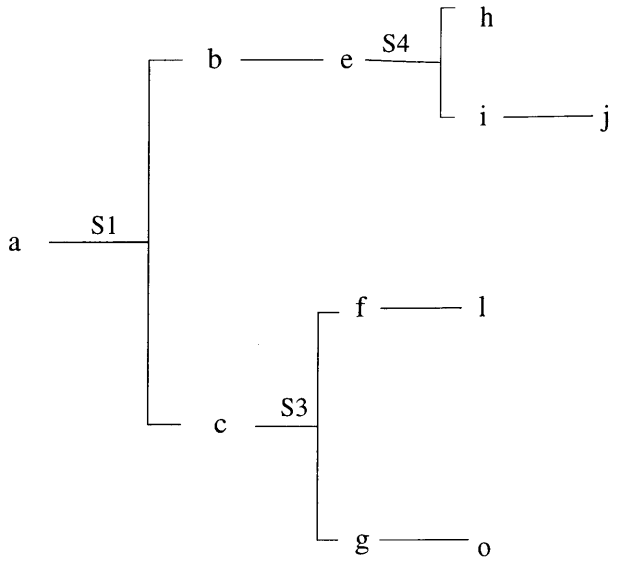


Figure 5.2: A tailored version of the system network of figure 5.1

tracking). At some stage, WAG attempted the backtrack strategy but, according to O'Donnell's experience, this generation strategy is very inefficient and as a result he switched to deterministic generation [O'Donnell 94].

Our work can be considered as non-deterministic simultaneous generation. However, it uses a new architecture based on the ATMS. The reason for this attempt is the great similarity between systemic networks and the ATMS dependency networks as we will see later on in this chapter. In particular, we will show that the logical specifications of both representations are very similar. This makes the ATMS a plausible tool to build our generation system on. The second reason is the ability of the ATMS to work in multiple contexts efficiently, through the revision of only what is necessary and the early elimination of parts of the search space that will give rise to unfavoured results further down the line.

5.3 The Organisation of the Rest of the Chapter

In the rest of this chapter, we first present the logical specification of systemic grammar networks including our interpretation of them. Then, we discuss the logical representation of ATMS dependency networks. We then introduce our ideas of translating systemic grammars into the ATMS representation. Next, we present our translation

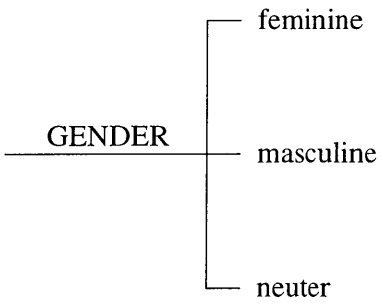


Figure 5.3: An exhaustively labelled system

algorithm which transforms systemic networks into ATMS dependency networks. Finally, we discuss the compilation of a complete systemic grammatical resource into an ATMS-ready representation.

5.4 Logical Interpretation of System Networks

In 1986, Patten pointed out that systemic grammar "has never been rigorously formalised in the way that traditional grammars have"[Patten 86]. [Patten & Ritchie 86] come up with a formal model for systemic grammars but, as [Patten 86] acknowledges, this attempt remains exploratory in nature. After a decade, Henschel makes the observation that "the implementations of systemic grammars remain rather old-fashioned [...] due to the fact that systemic grammar has not developed its own logic"[Henschel 97]. Nevertheless, there have been attempts to specify the logical content of system networks such as [Mellish 88], [Brew 91], and lately in [Calder 99].

Figure 5.4 shows the basic constructs of systemic networks and their logical interpretation as proposed by [Mellish 88]. In practice, these four basic systems can be connected in any way as long as the resulting network is an acyclic directed graph. In the figure, *AMO* stands for "at most one of" meaning that the daughters of a system are mutually exclusive. We opt to follow [Brew 91]'s exhaustive labelling. Exhaustive labelling is a labelling scheme in which each line in the system network has some label. However, we will not be using boolean conjunctions and disjunctions of atomic names as labels. Instead we will only use atomic values as label edges.

We have given the logical interpretation of isolated systems. Now we consider system

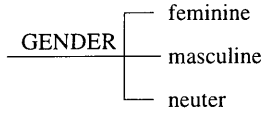
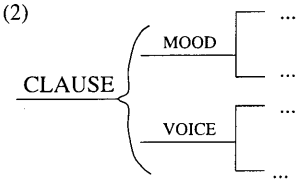
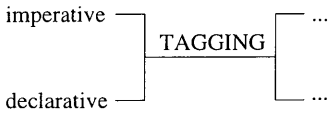
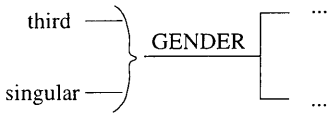
SFG CONFIGURATION	LOGICAL INTERPRETATION
(1)  **a simple system**	GENDER $\equiv$ feminine v masculine v neuter AMO {feminine,masculine,neuter}
(2)  **simultaneous systems**	CLAUSE $\equiv$ MOOD ^ VOICE
(3)  **a disjunctively entered system**	imperative v declarative $\equiv$ TAGGING
(4)  **a conjunctively entered system**	third ^ singular $\equiv$ GENDER

Figure 5.4: Different systemic configurations and their logical interpretations

networks involving many connected systems. For example, the network of figure 5.5 graphically states that we first choose the feature a and that we also get into the system S_1 that is connected[1] to it. In turn, S_1 requires that we either choose b and the system S_2 that is connected to it, or c and the system S_3 that is connected to it. Finally, S_2 states that we can choose between d and e, and S_3 between f and g. There are no systems associated with leaf features.

The above intuitive interpretation is logically expressed as follows (where $NETWORK$

[1] In SFL terminology, the feature a is the entry condition of S_1. The general reading would then be: choose a and get into all the systems that are enterable at this point.

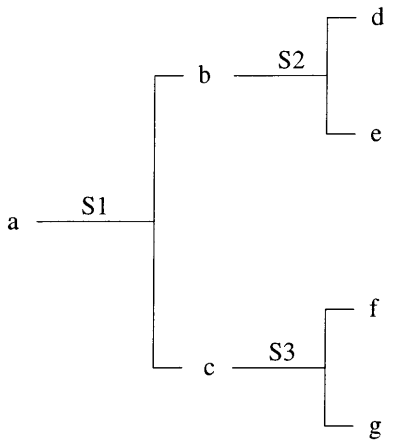


Figure 5.5: An exhaustively labelled network with several connected systems

stands for the logical interpretation of the network as a whole):

$$NETWORK \equiv a \wedge S_1 \tag{5.1}$$

$$S_1 \equiv (b \wedge S_2) \vee (c \wedge S_3) \tag{5.2}$$

$$S_2 \equiv d \vee e \tag{5.3}$$

$$S_3 \equiv f \vee g \tag{5.4}$$

Note that although not mentioned explicitly, we assume that the usual AMO restrictions on features of the same system still hold for the formulae (5.1) to (5.4). Now, substituting recursively for S_1, S_2, and S_3 into the first formula we get the following logical proposition:

$$NETWORK \equiv (a \wedge b \wedge d) \vee (a \wedge b \wedge e) \vee (a \wedge c \wedge f) \vee (a \wedge c \wedge g) \tag{5.5}$$

Formula (5.5) says that a network configuration is equivalent to $P_1 \vee P_2 \vee ... \vee P_n$ where P_1 to P_n are all the permissible paths in that network. Proposition (5.5) states that there exists a logical formula for a given system network; which happens to be in DNF in this case. Interestingly enough, these are all the selection expressions of the system network. [Calder 99] also presents an algorithm for computing the logical formula of a whole network in CNF. Note that the above steps of substitution give the DNF of the formula of the whole network directly, since selection expressions are in fact logical formulas in DNF. This logical equivalence supports our intuitive interpretation of the system networks.

5.5 Logical Specification of the ATMS

Having presented the logical specification of systemic networks in the previous section, we now briefly discuss the logical representation of the ATMS. This is a necessary step as our goal is to map one representation into the other. A full account of the ATMS representation and algorithms has been given in Chapter 4.

All families of Truth Maintenance Systems (TMS) can be formulated within propositional calculus and the ATMS is no exception [Forbus & de Kleer 93]. The ATMS nodes and justifications form a dependency network similar to the one shown in figure 5.6.

The only way to seriously communicate with the ATMS (other than node creation messages) is via justifications. In essence, a justification specifies that nodes $n_1, n_2, ..., n_m$ justify or imply node n. This is logically written as the material implication:

$$n_1 \wedge n_2 \wedge ... \wedge n_m \rightarrow n$$

which can also be written as

$$\neg n_1 \vee \neg n_2 \vee ... \vee \neg n_m \vee n$$

For efficiency reasons, justifications are not allowed to be arbitrary clauses; they are restricted to propositional Horn clauses only [Lamma & Mello 93]. In most ATMS implementations, a justification takes a fixed format: $<$ *antecedents* $\rightarrow$ *consequent* $>$, where the *antecedents* part is the conjunction of the nodes $n_1, n_2, ..., n_m$ and *consequent* is a single node n. This is deemed sufficient to answer the most fundamental question of whether a node logically follows from a given node configuration.

5.6 From Systemic Grammars to ATMS Representation

In order to be able to use systemic grammars in our ATMS-based generator, we translate the system networks to a representation which the ATMS can reason with: dependency networks. Following our logical interpretation of system networks, we show how each type of system can be re-written as a group of implications. As mentioned

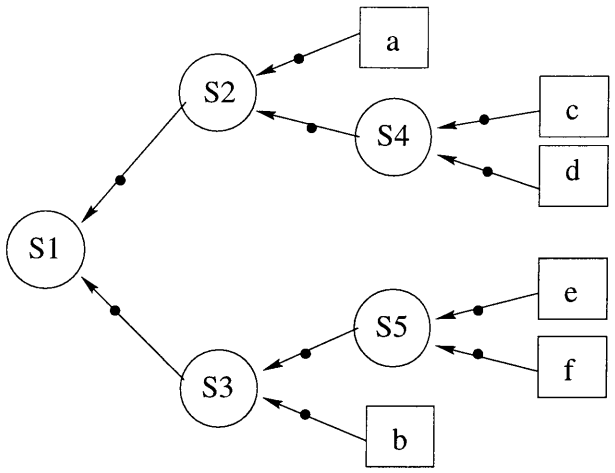


Figure 5.6: An example dependency network

previously, ATMS justifications are in fact material implications written in a specific format. That should tell us which systems can be mapped directly to ATMS dependency network fragments and which systems cannot.

For the relation $S \equiv a \vee b$ where S is a system having the features a and b (see figure 5.7), we opt for the interpretation: "if *exactly one* of the features a or b is chosen then the system S holds". In fact, the interpretation in the other direction is

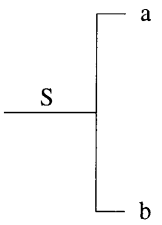


Figure 5.7: An example simple system

still valid (albeit implicitly) as the ATMS makes a kind of "closed world" assumption that this is the only way S can be inferred. In other words, it assumes that besides a and b there are no other causes for S. Logically, this can be expressed through an implication from the effect to the cause. For example, if it rains then the grass gets wet ($it\_rained \rightarrow wet\_grass$). However, if the grass is wet this does not necessarily mean that it has rained *unless* we assume that the only way the grass can get wet is by rain. In this case, we can safely say $wet\_grass \rightarrow it\_rained$ if we wish to express that fact explicitly. In the discussion below, although one direction of the equivalence is

considered, we assume that the other direction is implicit because of this closed-world assumption the ATMS makes.

In the following, we show, whenever possible, how the basic constructs of system networks are represented using ATMS justifications. Our criterion for a possible mapping is that the labels computed by the ATMS at certain points are the same as the set of selection expressions computed by the system network.

5.6.1 Simple System Representation

According to our interpretation of system networks (see Section 5.4), a simple system representation is:

$$S \equiv f_1 \vee ... \vee f_n$$
$$AMO\{f_1, ..., f_n\}$$

where $f_1, ..., f_n$ are features of the system S. For reasons that have to do with the strict format of the ATMS justifications we will content ourselves with the one way implication: $f_1 \vee ... \vee f_n \rightarrow S$ without loss of generality as just discussed earlier in this section. The LHS of that implication is not permissible as the antecedents' part of an ATMS justification. However, $f_1 \vee ... \vee f_n \rightarrow S$ can be written as separate ATMS justifications of the form: $f_i \rightarrow S$. We show next the steps of transforming the justification $f_1 \vee ... \vee f_n \rightarrow S$ into a set of permissible ATMS justifications.

$$\begin{aligned} & f_1 \vee ... \vee f_n \rightarrow S \\ \text{or} \quad & \neg(f_1 \vee ... \vee f_n) \vee S \\ \text{or} \quad & (\neg f_1 \wedge ... \wedge \neg f_n) \vee S \\ \text{or} \quad & (\neg f_1 \vee S) \wedge ... \wedge (\neg f_n \vee S) \\ \text{or} \quad & (f_1 \rightarrow S) \wedge ... \wedge (f_n \rightarrow S) \end{aligned}$$

Hence, $(f_1 \vee ... \vee f_n \rightarrow S) \equiv ((f_1 \rightarrow S) \wedge ... \wedge (f_n \rightarrow S))$. Moreover, the AMO constraint can easily be represented by a set of ATMS jusfifications as follows:

$$\{ f_x \wedge f_y \rightarrow \perp \text{ for all sister features } f_x \text{ and } f_y \text{ of the system S } \}$$

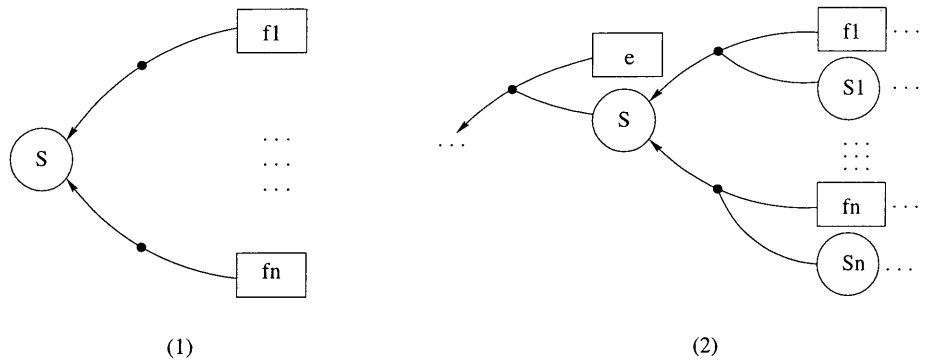


Figure 5.8: Dependency network representations of simple systems

Graphically, an isolated simple system is represented as shown in part (1) of figure 5.8. Part (2) shows the case where the system is part of a bigger system network. In this case, the system S is reachable only through its entry condition e, and its features $f_1, ..., f_n$ are themselves entry conditions to the systems $S_1, ..., S_n$ respectively. Note that if any of $f_1, ..., f_n$ is a leaf (i.e. not an entry condition to any further systems) then it is simply represented as shown in part (1).

The graphical convention we follow here uses rectangles to represent assumptions and ovals to represent derived nodes. As far as the mapping from system networks to dependency networks is concerned, systems are mapped to derived nodes and features to assumptions. By mapping features into assumptions, we can get all the selection expressions of a network represented in the label of the root node since the ATMS calculates a label for each node. The label of a node is the minimal sets of assumptions from which it logically follows. This corresponds to the disjointness information that system networks also provide. Note that in the graphical representation of dependency networks we do not show the AMO justifications. In fact, the AMO constraint on features of the same system is automatically imposed by the *minimality* property of the ATMS's labels. This allows us to omit the explicit representation of the AMO justifications across the systems in the ATMS network representation. Beware, however, that this indicates a slight difference of interpretation. The minimal representation in an ATMS label conveys a non-minimal state of affairs since any non-contradictory environment that subsumes the minimal label of a node will also support that node; whereas in a systemic network, the mutual exclusivity of features means that the set

means exactly itself and not any subsuming sets.

The only reason we omit the explicit representation of the AMO constraint is to improve the efficiency of the particular ATMS implementation we are currently using since the assertion of any justification may trigger a series of label update operations. Otherwise, the mutual exclusivity requirement can easily be represented as ATMS justifications as discussed earlier in this section.

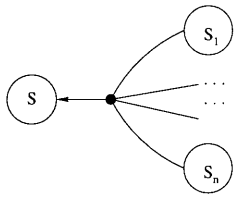
5.6.2 Simultaneous Systems Representation

Having presented simple systems with simple entry conditions in ATMS forms, we now discuss how the other basic systemic constructs can be represented using the same form. In this section, we discuss how we can map the simultaneity configuration to the ATMS representation. In the next section, we show how the disjunctively entered systems can be mapped to an ATMS representation. In section 5.6.4, we discuss why the conjunctively entered systems cannot be mapped directly into an ATMS form.

A simultaneity point in a system network is represented logically as:

$$S \equiv S_1 \wedge ... \wedge S_n$$

In the direction we have chosen, this can be directly written as an ATMS justification: $S_1 \wedge ... \wedge S_n \rightarrow S$ which gives the following dependency representation:



Assuming that the labels of the simultaneous systems $S_1, ..., S_n$ are computed correctly regardless of what each S_i leads to, the label of S is now computed based on the very justification $S_1 \wedge ... \wedge S_n \rightarrow S$. That is, the label of S is one that represents the selection expressions of all possible paths of getting to $S_1, ...,$ and S_n simultaneously.

5.6.3 Disjunctively Entered Systems

The generic disjunctive entry condition shown in part (1) of figure 5.9 states that the system S is reachable via any of its disjuncts $d_1, ..., d_n$. That is, either $d_1, d_2, ...,$ or d_n is a sufficient entry condition to the system S. In a way, this is nothing but a representation of simple entry conditions to simple systems which happen to be equivalent, as shown by part (2) of figure 5.9.

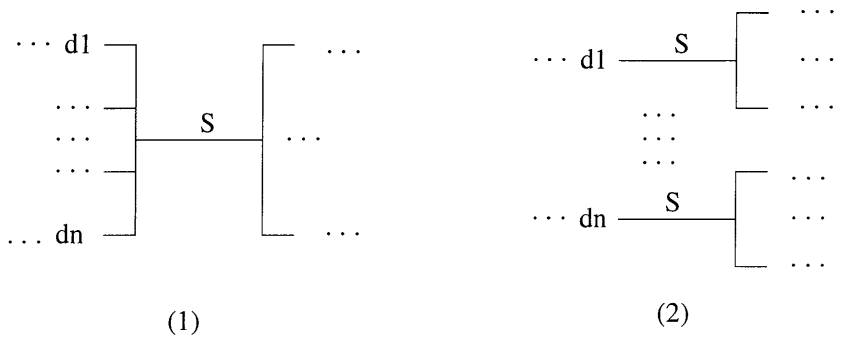
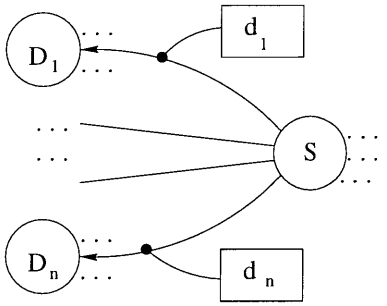


Figure 5.9: The meaning of disjunctive entry conditions

Assuming that the disjuncts $d_1, ..., d_n$ are features in some systems $D_1, ..., D_n$ respectively, this state of affairs can be mapped to the following ATMS representation.



This dependency network representation states that for the system D_1, for example, any of its children features can be selected as usual. However, if d_1 is selected then the system S must be entered because d_1 is nothing but an entry condition to S. The same applies to the systems of the remaining disjuncts $d_2, ..., d_n$.

5.6.4 Conjunctively Entered Systems

The generic conjunctive entry condition shown in figure 5.10 states that the system S is reachable only upon getting to (or selecting) all of its entry condition conjuncts $c_1, ..., c_n$. Assuming that the conjuncts $c_1, ..., c_n$ are features in some systems $C_1, ..., C_n$

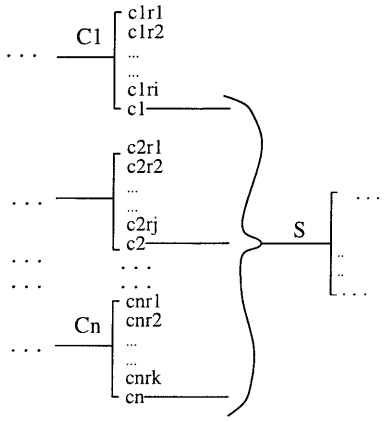


Figure 5.10: A typical conjunctively entered system

respectively, this state of affairs is depicted in figure 5.11. As the figure shows, this is not a permissible form of ATMS justifications as the figure attempts to show. The

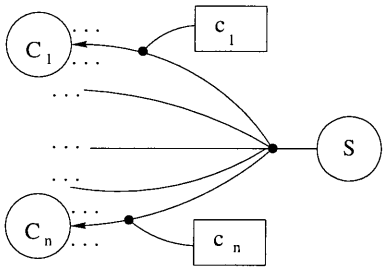


Figure 5.11: An impermissible form of ATMS justifications

permissible form of an ATMS justification is $a_1 \wedge ... \wedge a_i \rightarrow c$ where a *conjunction* of antecedents implies *one* consequent node (and not a conjunction of consequents as shown in the figure). Therefore, the conjunctive entry condition configuration of systemic networks cannot be directly mapped to an ATMS dependency network representation.

In subsequent sections (from Section 5.7 onwards) we look into ways of dealing with the conjunctively entered systems. But before that, let us convince ourselves with more

examples involving parts of real systemic grammars networks (not involving conjunctively entered systems for obvious reasons).

Example 1

In this example, we apply our mapping strategy to some linguistic and more realistic system networks. Consider the clause network of figure 5.12 which has simple, simultaneous and disjunctively entered systems. Its dependency network mapping is shown in figure 5.13. As mentioned earlier, our criteria for correct translation is that labels computed by the ATMS are exactly the selection expressions licensed by the system network.

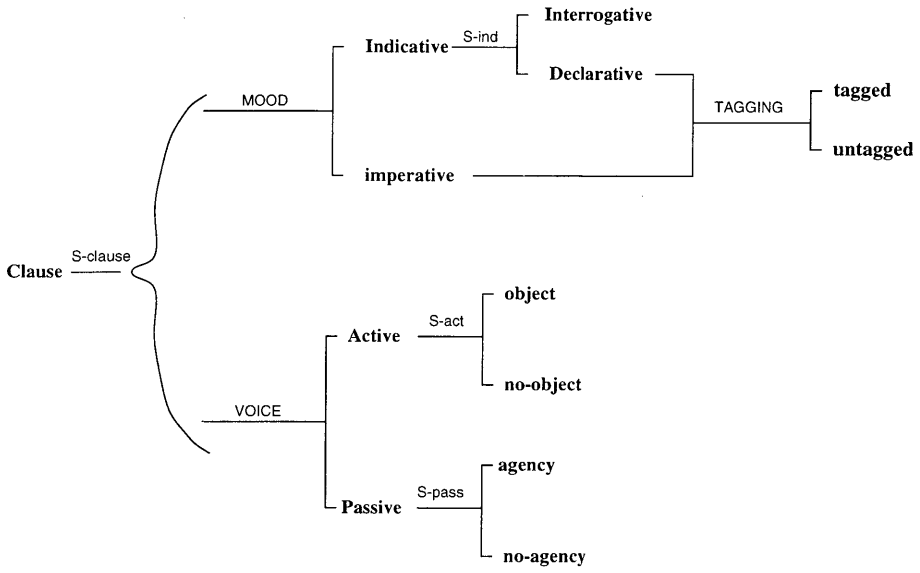


Figure 5.12: A simple clause network

The dependency network of figure 5.13 simply says that the NETWORK node holds if the assumption *clause* is true and the associated system *s-clause* is also true. In turn, the *s-clause* system holds if both the *MOOD* and *VOICE* systems hold simultaneously. The requirement that *MOOD* and *VOICE* must both hold correspond to the fact that they are simultaneous systems in the given grammar fragment. The rest of the network can be interpreted in the same way. Note, however, how the disjunctively entered system *TAGGING* gets mapped. It should hold when either the feature *imperative* or *declarative* is chosen. Finally note the graphical correspondence between both types

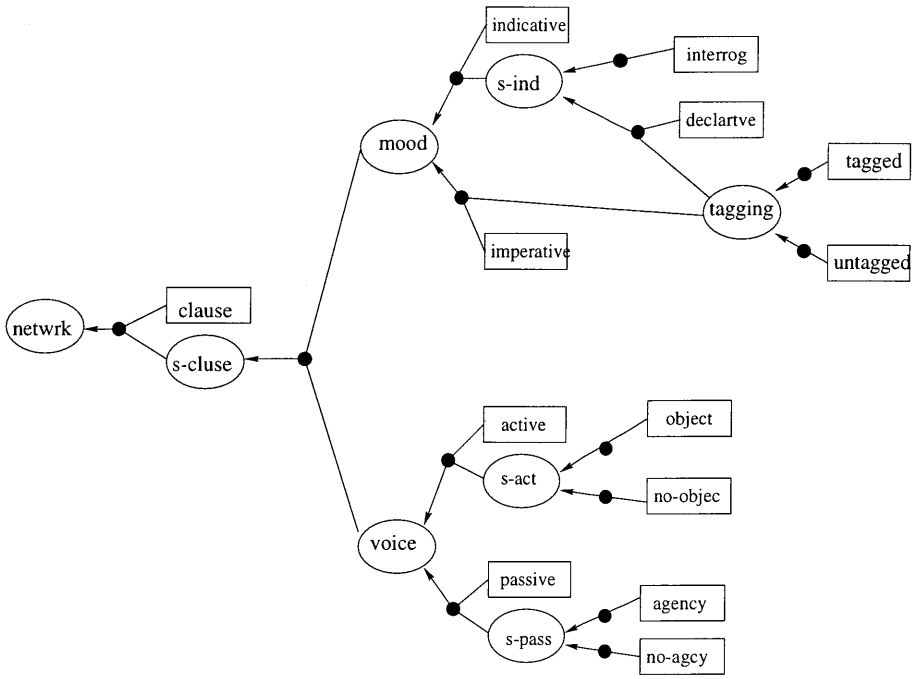


Figure 5.13: The corresponding dependency network of the clause network

of networks.

5.7 Dealing with Conjunctively Entered Systems

The source of difficulty with conjunctively entered systems is that justifications like $S \rightarrow Conj_1 \wedge ... \wedge Conj_i$ do not follow the permissible form of ATMS justifications. In this section, we investigate different ways of re-writing a system network with conjunctive gates using ATMS justifications.

In the previous sections, we noted that the network as a whole is equivalent to a logical formula in DNF: $NETWORK \equiv P_1 \vee P_2 \vee ... \vee P_n$ which can be written as ATMS justification:

$$NETWORK \leftarrow P_1$$

$$...$$

$$NETWORK \leftarrow P_n$$

Each term P_i is a conjunction of features which can also be represented in ATMS justifications (i.e. the antecedents' part of a justification). Here, we investigate this

direction. We first give an algorithm for computing a DNF formula for the whole network in Section 5.7.1. Then we discuss what improvements we can make to the dependency network resulting from applying the justifications implied by the DNF formula directly (Sections 5.7.2 and 5.7.3).

5.7.1 Logical Formula for the Whole Network

Here we present an algorithm that computes a formula in DNF for the whole network. Note that according to our interpretation of system networks (see Section 5.4), a network can be read as follows: start at the root of the network *and* get into all the systems enterable at this point. Then, for each system select any (but only one) of its features *and* get into any enterable systems at this point. This way, a system S with features $f_1, ..., f_n$ can be written as:

$$S \equiv (f_1 \wedge S_1) \vee ... \vee (f_n \wedge S_n)$$

Note that some features might have no systems attached to them while others might have more than one system (i.e. simultaneous systems).

A disjunctively entered system is treated like a simple system (i.e. written like a simple system) except that its name will be mentioned in conjunction with all its entry condition disjuncts (i.e. in the representations of other systems). Therefore, a disjunctively entered system S_d with features $a_1, ..., a_m$ (which are themselves entry conditions to the systems $S_{a_1}, ..., S_{a_m}$ respectively) and entry condition disjuncts $d_1, ..., d_k$ is simply written as:

$$S_d \equiv (a_1 \wedge S_{a_1}) \vee ... \vee (a_m \wedge S_{a_m})$$

The fact that S_d can be entered from any d_i is already taken care of by the representations of the systems of its disjuncts. For example, the system S_i of d_i with sisters $s_1, ..., s_p$ is represented as:

$$S_i \equiv (s_1 \wedge S_{s_1}) \vee ... \vee (d_i \wedge S_d) \vee ... \vee (s_p \wedge S_{s_p})$$

A conjunctively entered system S_c with features $b_1, ..., b_r$ (which themselves are entry conditions to $S_{b_1}, ..., S_{b_r}$ respectively) and conjuncts $c_1, ..., c_q$ is written as:

$$c_1 \wedge ... \wedge c_q \wedge S_c \equiv (b_1 \wedge S_{b_1}) \vee ... \vee (b_r \wedge S_{b_r})$$

To compute the logical formula for the network as a whole we follow this algorithm:

1. starting at the root of the network, find all the systems $S_1, ..., S_n$ enterable at this point.
2. write down the formula $root \wedge S_1 \wedge ... \wedge S_n$
3. start substituting for the systems S_i recursively until no more systems exist in the formula.
4. put the formula in DNF, if not already.
5. for each conjunctively entered system find each disjunct in the DNF that has all the conjuncts of that gated system[2] and append the DNF formula of the gated system to it; compute DNF for the newly created formula.

The final formula is of the form:

$$expr_1 \vee ... \vee expr_n$$

where $expr_1, ..., expr_n$ are all the selection expressions of the network. This way, we get rid of the $antecedents \rightarrow Conj_1 \wedge ... \wedge Conj_i$ implications which are problematic to represent as ATMS justifications.

The resulting formula corresponds to the following ATMS justifications:

$$\begin{aligned} expr_1 &\rightarrow NETWORK \\ expr_2 &\rightarrow NETWORK \\ &... \\ &... \\ expr_n &\rightarrow NETWORK \end{aligned}$$

So what we have done is, in effect, to flatten the system network into this representation. This way, we got rid of the awkward implications (e.g. $antecedents \rightarrow$

[2] Generally speaking, by a gated system we mean either a system with a conjunctive entry condition or a system with a disjunctive entry condition. When it is not clear from the context which type of system we mean, we use, instead, the terms AND-gated system and OR-gated system respectively.

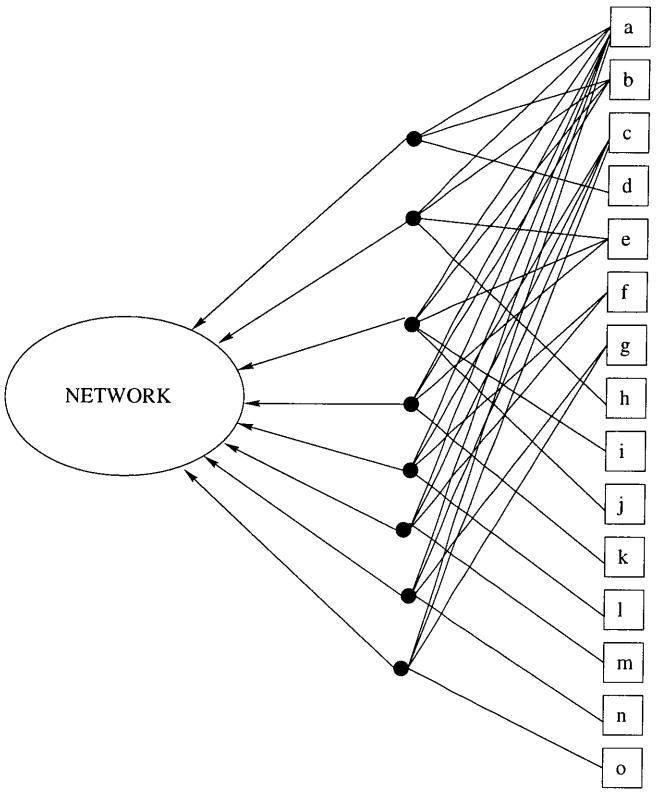


Figure 5.14: A flattened representation of the system network of figure 5.1

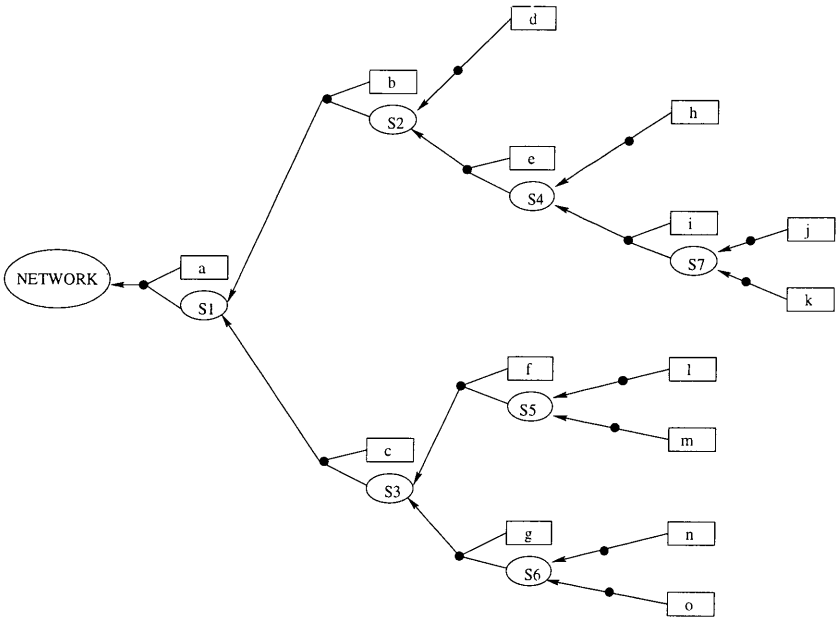


Figure 5.15: A staged dependency network with exactly the same NETWORK label of that of figure 5.14

$Conj_1 \wedge ... \wedge Conj_i$); and now all the implications are representable using the ATMS justifications. The label of the goal node *NETWORK* has all the selection expressions. Figure 5.14 is a flattened dependency network of the system network of figure 5.1. Although the implications of this representation are all valid ATMS justifications, this solution is not appealing because:

- it does not exploit the ATMS's ability of incremental calculation of node labels. Normally, the addition of a justification triggers an incremental re-labelling of involved nodes. For example, in the staged representation of figure 5.15 the labels of the nodes are computed incrementally as the dependency network gets constructed (i.e. after each justification is asserted).

- what we did, in effect, was calculate the label of the goal node ourselves under the guise of computing the DNF formula of the whole network instead of leaving it to the ATMS to find the selection expressions. Again, the network of figure 5.15 does not require the wiring of the nodes in such a way as to reflect the selection expressions of the system network. The label updating algorithms take care of this task. Needless to say, calculating the DNF for real grammars is a lot of work. For example, the WAG (a medium sized grammar compared to the NIGEL grammar) clause network, can give rise to more than 152,000 selection expressions[3]. That is, a DNF formula with this number of disjuncts[4].

5.7.2 Factorising the DNF Formula

The main flaw of the DNF-based solution is that it flattens the whole network and as a result does not utilise the ATMS or the system network incrementality. Obviously, not all systems in a network participate, whether directly or indirectly, in conjunctive gates (e.g. the *animacy* and *case* systems in the pronoun network of figure 5.19). These systems which are uninvolved in any such gates are also called *innocent* systems. It would be nice if there was a way of keeping the uninvolved systems intact. Unfortunately, for large and complicated system networks, this seems to be a difficult task.

[3] This figure is an estimate of the number of selection expressions calculated by hand.

[4] When I tried to compute the DNF for the WAG clause network, SICStus Prolog always complained about insufficient stack memory.

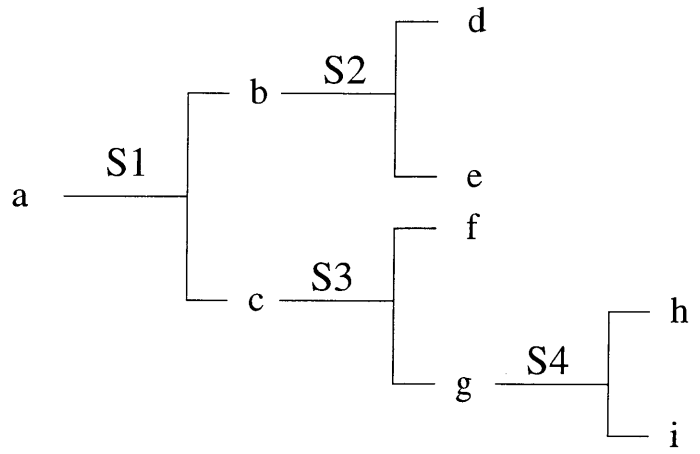


Figure 5.16: A network with simple systems only

One way we have tried to restore most of the original systems is using the idea of factorising the final DNF formula. Note that the process of factorising the DNF formula of a whole network relies on consulting the original system network. Features of systems of the original network are used to factorise the formula and any resultant sub-formula of it. This way the original systems are preserved and so are the choosers which are attached to them. Shuffling around the features of a system without regard to their sister-hood relationships renders the chooser of that system inapplicable. Therefore, the idea of keeping the systems intact motivates much of the design of a translation algorithm since the need to preserve semantically-relevant aspects of the system network is important. By recursively factorising the formula we can recover the *innocent* systems and hence get a staged representation which is very similar to the original network although not exactly.

Example 1

To see what we mean by recursively factorising the DNF formula, let us consider the simple network shown in figure 5.16.

The DNF formula of this network is

$$(a \wedge b \wedge d) \vee (a \wedge b \wedge e) \vee (a \wedge c \wedge f) \vee (a \wedge c \wedge g \wedge h) \vee (a \wedge c \wedge g \wedge i) \quad (5.6)$$

The formula (5.6) is factorised according to the following ordered criteria:

- **Criterion 1:** is there a feature f common to all disjuncts of the formula? If so, then use it to factorise the formula into $f \wedge F$. F now becomes our new formula that needs to be factorised further.

- **Criterion 2:** if not, then is there a system common to all disjuncts of the formula? If so, then use its features $f_1, ..., f_n$ to factorise the formula into sub-formulas such as $(f_1 \wedge F_1) \vee ... \vee (f_n \wedge F_n)$. The whole process now needs to be repeated for each F_i.

- **Criterion 3:** if not, then take a system[5] from the original network and use it to factorise the formula or the remaining part of it. Repeat the whole process for the incurred sub-formulas.

According to the above criteria we get the following result:

$$a \wedge ((b \wedge d) \vee (b \wedge e) \vee (c \wedge f) \vee (c \wedge g \wedge h) \vee (c \wedge g \wedge i)) = a \wedge F_1 \quad (5.7)$$

$$F_1 = (b \wedge (d \vee e)) \vee (c \wedge (f \vee (g \wedge h) \vee (g \wedge i)) = (b \wedge S2) \vee (c \wedge F_2) \quad (5.8)$$

$$F_2 = f \vee (g \wedge (h \vee i)) = f \vee (g \wedge S_4) \quad (5.9)$$

where (5.7) above is factorised using criterion 1; and (5.8) and (5.9) are factorised using criterion 2. This makes $F_2 = S_3$ which means that $F_1 = S_1$. This way, we could completely restore the original staged network from the DNF formula.

We could have factorised the above formula without having to look at the original systems of the network. This process might take a feature from a system and put it underneath another system which in effect changes some of the sister-hood relationships. From the ATMS point of view, this would not hurt; actually, it might help depending on how optimal the outcome of the factorisation is. From the systemic grammar point of view, however, it would not be a good idea since there are usually choosers associated with systems. Shuffling features around invalidates these choosers. Therefore, we need to look at the network in order to restore the original systems of it.

[5] It is not clear however which system to choose. One good heuristic is to pick the systems according to their order in the original network: left to right and top to bottom in the case of simultaneous systems.

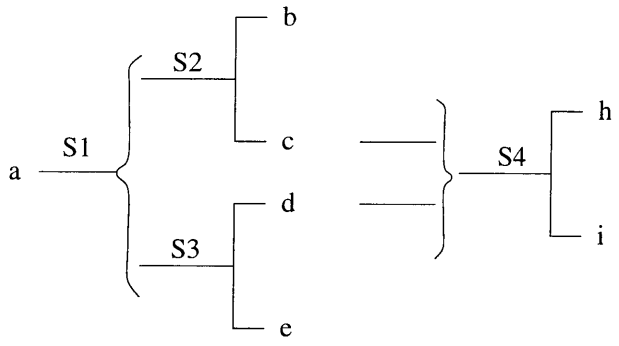


Figure 5.17: A network with a conjunctively entered system

The above example is for illustration purposes only since there is no point of computing the DNF and then factorising it to restore the original systems. However, this step is necessary for networks having conjunctively entered systems. Our idea here is to get rid of the implications of the form $antecedents \rightarrow Conj_1 \wedge ... \wedge Conj_i$ by computing the DNF formula first and then trying to restore those innocent systems; thus ending up with a staged dependency network.

Example 2

The network of figure 5.17, although small, cannot be easily mapped to an equivalent dependency network because of the problematic AND gate preceding S_4. We now show how the idea of factorising the DNF formula helps in getting an equivalent dependency network that is better than the flattened representation. The sequence of factorisation is shown below and its corresponding staged dependency network is shown in figure 5.18. Note that the assumptions d and e in the dotted boxes are exactly the same as the ones of S_3. They are duplicated in the figure for the sake of clarity.

$$(a \wedge b \wedge d) \vee (a \wedge b \wedge e) \vee (a \wedge c \wedge d \wedge h) \vee (a \wedge c \wedge d \wedge i) \vee (a \wedge c \wedge e) \quad (5.10)$$

$$a \wedge ((b \wedge d) \vee (b \wedge e) \vee (c \wedge d \wedge h) \vee (c \wedge d \wedge i) \vee (c \wedge e)) = a \wedge F_1 \quad (5.11)$$

$$F_1 = (b \wedge (d \vee e)) \vee (c \wedge ((d \wedge h) \vee (d \wedge i) \vee e)) = (b \wedge S_3) \vee (c \wedge F_2) \quad (5.12)$$

$$F_2 = (d \wedge (h \vee i)) \vee e = (d \wedge S_4) \vee e \quad (5.13)$$

Note that (5.10) is the DNF formula of the whole network. (5.11) is factorised using criterion 1, (5.12) using criterion 2, and (5.13) using criterion 3.

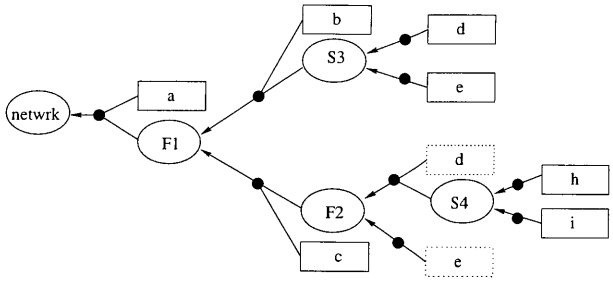


Figure 5.18: A dependency network corresponding to the network of figure 5.17

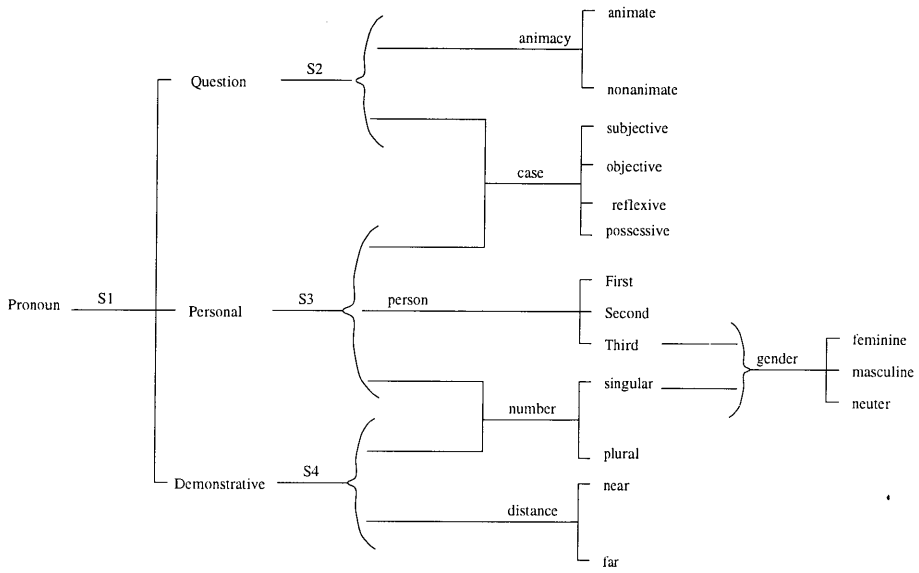


Figure 5.19: The English pronoun network (from [Winograd 83])

Example 3

In this example, we show how the English pronoun network (a famous example having all sorts of system configurations) can be mapped to a dependency network using the idea of finding the DNF formula and then factorising it to restore the unaffected systems. The original pronoun network is shown in figure 5.19 and its corresponding dependency network in figure 5.20. The label of the root of the dependency network of figure 5.20 (i.e. the NETWRK node) is exactly all the legal selection expressions of the original SFG pronoun network.

Note that we can indeed restore all the original systems except for those having features participating in the conjunctive gate. The system *person'* for example differs from the

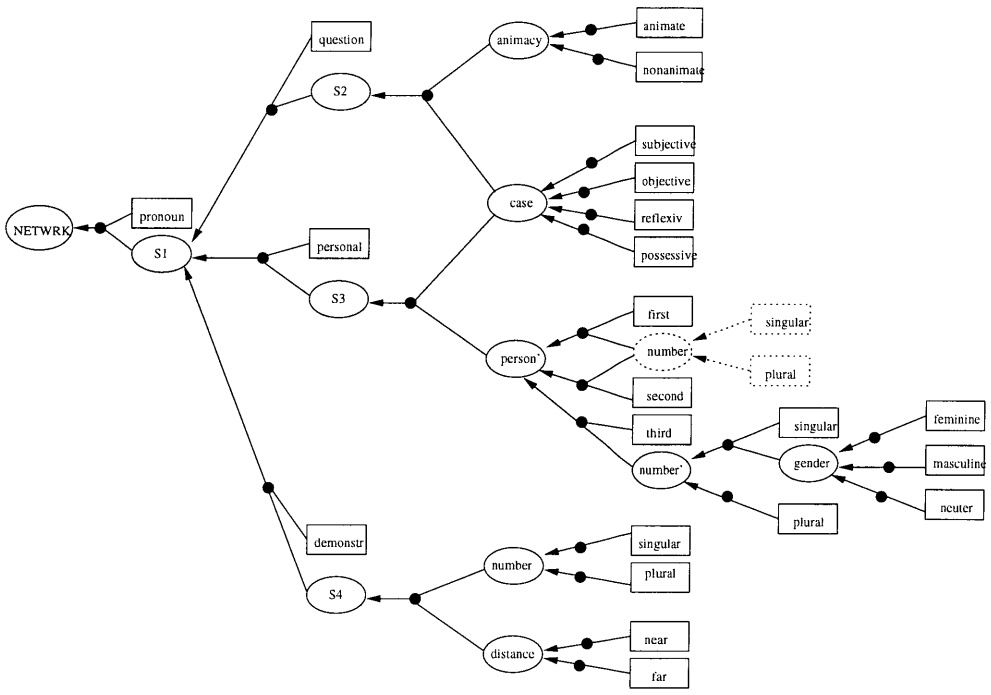


Figure 5.20: The dependency network of the pronoun network

original one in that it now encodes that if *first* or *second* is selected then the original *number* system is brought into perspective. However, if *third* is selected we get into a different version of the *number* system. Within the new system *number'*, if *plural* is chosen, that is fine. However, if *singular* is chosen we get into the AND-gated system *gender* since, by that time, we have satisfied all its entry condition requirements.

5.7.3 Pros and Cons of the Factorisation Algorithm

The main feature of the factorisation algorithm is that it is logically-based. Therefore, it is a theoretically acceptable translation algorithm which succeeds in the following:

1. It solves the problem of the unrepresentable implications caused by the conjunctive gates.

2. It gives a staged dependency network similar in many ways to the original system network except, of course, for those parts that are affected by the conjunctive gates.

3. It is clearly correct and general.

The main disadvantage of the factorisation approach is the need to do expensive intermediate work. As mentioned earlier finding the DNF for a large network is a costly task both in terms of time and memory.

The good news, however, is that this is a one off task. Like the process of writing the systemic grammar itself, its ATMS translation is fixed and could be computed once only. As soon as the staged representation is found the generator can benefit from the main feature of the ATMS: incrementality.

Next, we look into ways of getting a staged dependency network without having to compute the DNF formula for the whole network. That is, we try to find more efficient translation algorithms.

5.8 Creating System Networks without Conjunctive Gates

Despite the fact that the factorisation approach has some disadvantages, it gives us insights into how gated systems should be handled by the translation algorithm. For example, note that the conjunctively gated system is hidden behind all its conjuncts. This ensures that the AND-gated system is brought into perspective only after all the necessary conditions of it (i.e. its conjunctive entry condition) are satisfied. This goes well with the intuitive interpretation of such gated systems. The idea then is to reach this result without having first to compute the DNF formula and then dismantle most of its parts by factorising it. In effect, we try to answer the question: is it possible to redraw a staged system network in such a way as to get rid of all the conjunctive gates? To answer this question we first look again into the meaning of the conjunctive gate and what implications it has.

5.8.1 Implications of Conjunctive Gates

The conjunctive gate in figure 5.21 means that all $Conj_1$, $Conj_2$, ... , and $Conj_i$ should be chosen in order to get into the gated system S_g. Moreover, from the way system networks are specified, all these conjuncts come from different *system chains*. That is, by starting from any $Conj_x$ and going all the way back to the root of the network, none

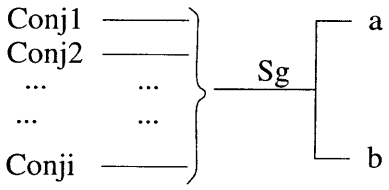


Figure 5.21: A conjunctive gate

of the other conjuncts of that gate can be encountered[6]. Knowing that each conjunct comes from a different system chain, there should be at some earlier level a source of simultaneity parenting all of the system chains. We call this system *the common source of simultaneity.* We will define what is meant by a system chain of a feature and the common source of simultaneity for a set of conjuncts.

Definition: System Chain

A system chain for a feature f of system S_n is a subnetwork consisting of those systems that are encountered by traversing back starting from S_n to some system S_0. A system chain must stop at a source of simultaneity (i.e. a left-facing brace '{'). Thus, S_0 is one of the children of a simultaneous system or the root system. Such a system chain is represented by the sequence $< S_n, S_{n-1}, ..., S_0 >$.

Definition: Common Source of Simultaneity

The common source of simultaneity for the conjuncts $c_1, ..., c_n$ with the corresponding system chains $C_1, .., C_n$ of a gated system S_g is the first common system for all the chains $C_1, .., C_n$.

5.8.2 The meaning of system simultaneity

Figure 5.22 shows two simultaneous systems S_1 and S_2. This means that choices from one system can be made independently of the other; or that both systems can be followed in parallel. It also means that at any choice of S_1, the choices offered by the other system S_2 are also available. This is graphically depicted in figure 5.23 part (1) or more parsimoniously in (2). Note, however, that the parsimonious representation requires, the additional restriction, that a feature of a given system, when selected, is

[6] Because if one conjunct (say $Conj_y$) is in fact on the path then both $Conj_x$ and $Conj_y$ are represented by a single conjunct $Conj_z$.

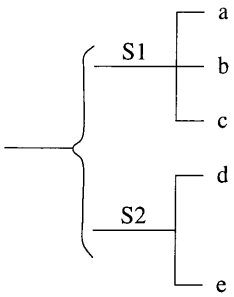


Figure 5.22: A network with two simultaneous systems

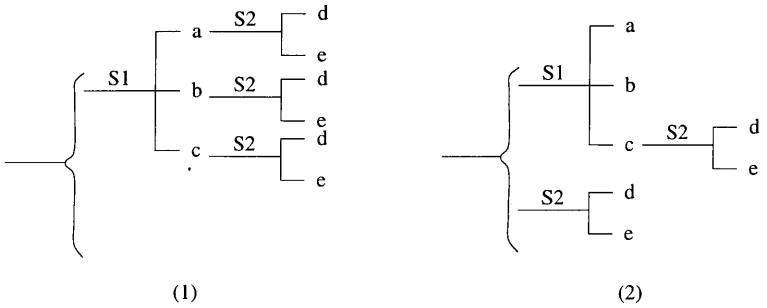


Figure 5.23: System networks equivalent to that of figure 5.22

adhered to wherever that system might appear in the network.

5.8.3 Networks without right-facing braces

In this section, we show how to write system networks without conjunctive gates since such networks can then be directly mapped into the ATMS dependency networks. First we define what we mean by a variant of a system. Then we introduce the basic idea of the translation algorithm. Finally, we give more examples to demonstrate how the algorithm works.

Definition: Variant System

A variant of the system S has the same choice of features but with at least one additional system attached to one or more of its features.

The system S_2' in figure 5.24 (2) is a variant of S_2. If we keep the restriction that if a choice of a feature is made in a system then that choice has to be adhered to in all its variants, then network (1) and (2) are equivalent. Note that network (2) is a parsimonious representation. This way both a_1 and a_2 are allowed to combine with

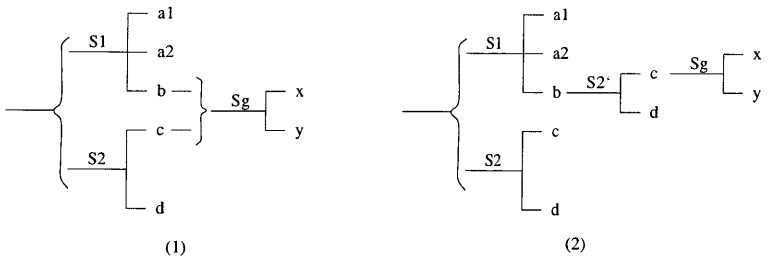


Figure 5.24: A system network and its corresponding braces-free version

S_2 independently. However, b, which is one of the conjuncts of the gate, can combine with a variant of the system S_2 since that could be the first step in satisfying the entry condition requirements. For example, if we choose c of the variant S_2', we get into the gated system S_g since that is all that is needed as an entry condition.

Algorithm for Translation from SFG to ATMS

The idea behind the translation algorithm is to re-arrange the systems of an SFG network so as to get rid of conjunctive entry conditions. Conjunctive gates cannot be directly mapped to corresponding ATMS forms (see Section 5.6.4 for a discussion of why conjunctive gates are awkward to represent). The algorithm attempts to simulate the factorisation of the DNF approach without having to calculate the DNF formula for the whole network first and then restore the original systems. Retaining the original system configuration of the network is essential since each system has a chooser attached to it. In order for these choosers to function correctly the algorithm must preserve those semantic aspects of the original network.

Algorithm Idea

For each AND gate with conjuncts $Conj_1$, $Conj_2$, ..., $Conj_i$ acting as an entry condition for the system S_g, do the following – starting with the first conjunct $Conj_1$. That is, make $Current\_Conj = Conj_1$.

1. at the current conjunct $Current\_Conj$, attach a copy of the system chain containing the next conjunct[7] $Next\_Conj$.

[7] The next conjunct is not necessarily the next in order. One good approach is first to order the

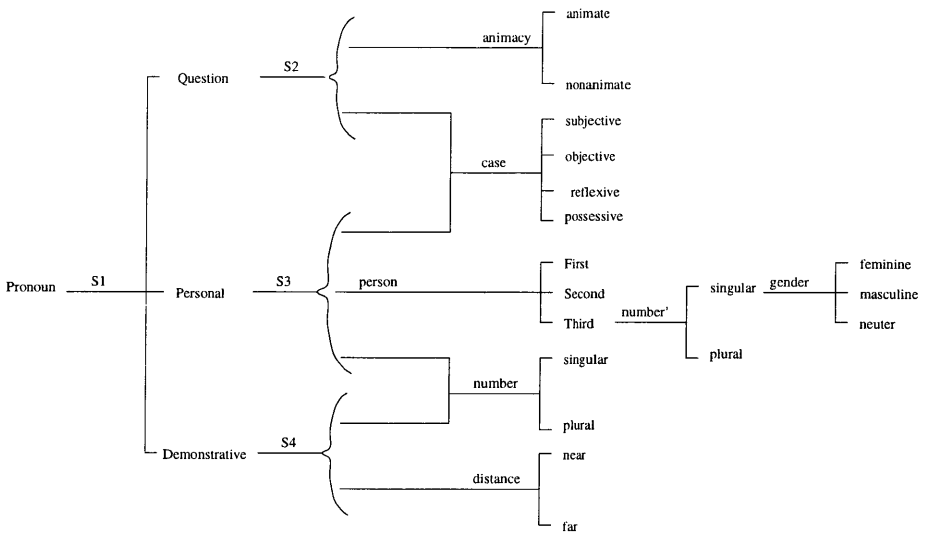


Figure 5.25: A braces-free version of the English pronoun network

2. if $Next\_Conj$ (in the newly created copy) is not the last conjunct of the entry condition, make it the current conjunct (i.e. $Current\_Conj = Next\_Conj$) and repeat step 1.

3. else if $Next\_Conj$ is the last conjunct of the gate, then simply attach to it the gated system itself S_g.

Because whenever a copy of a system is made, another system (or a copy of a system) is connected to one of its features, these copies become variants of systems in the same sense of the above definition. In effect, this algorithm attaches system chains deeper and deeper, starting from the first conjunct to the last one whereby the gated system itself is connected. The intuition behind the algorithm is to hide the gated system behind all the conjuncts of its entry condition, and, in order to get to it, one has to satisfy this condition (i.e. select all the necessary features). This cascading effect can be seen in the networks of figures 5.24, 5.25 and 5.27.

conjuncts according the size of the system chain they are part of. It is always a better strategy to attach smaller system chains to larger ones. This reduces the number of variant systems created.

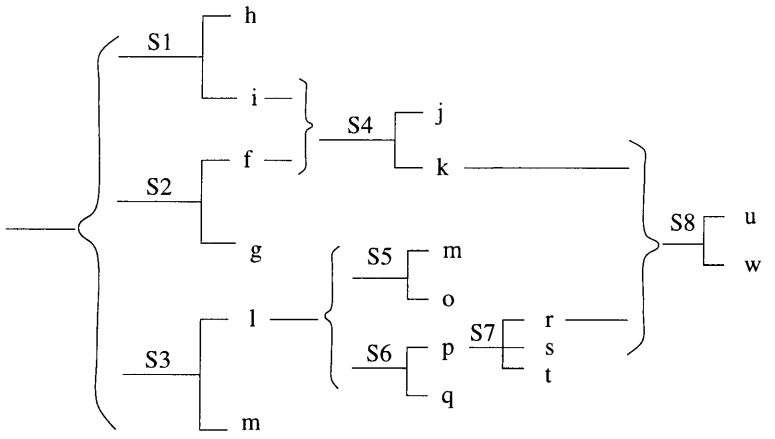


Figure 5.26: A system network with nested AND gates

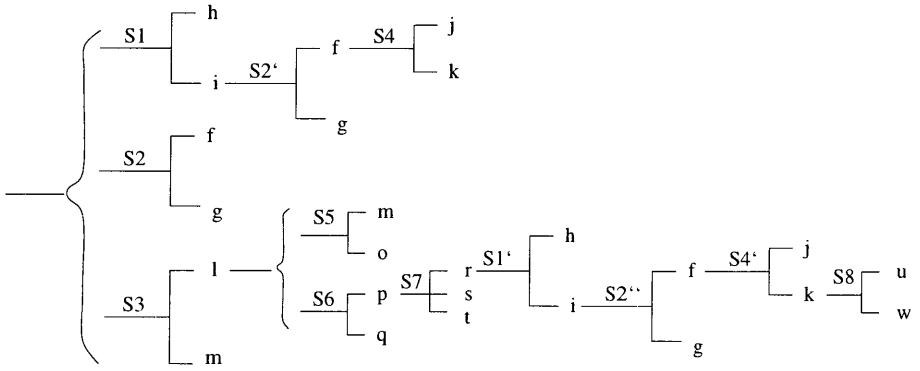


Figure 5.27: A braces-free version of the network of figure 5.26

More Examples

The network of figure 5.25 is a braces-free version of the English pronoun network of figure 5.19. We also give a hypothetical network that has nested conjunctive gates in figure 5.26 and its braces-free version in figure 5.27.

5.9 Compilation of Complete Grammatical Resources

So far, our discussion of the translation process from SFG to ATMS has focussed on the translation of the bare system networks (i.e. different configurations of systems and their features only). A complete Systemic Functional Grammar, however, includes, additionally, realisation statements which are attached to the features of the systems. Therefore, a complete translation algorithm must involve these realisation statements

in addition to the basic system-feature configuration. In the following, we discuss what type of realisation statements there are in systemic grammars and how we can map their effect into some form of ATMS representation.

Mapping of the Realisation Rules

In general, there are three types of realisation operations: structure building, ordering, and classification. The systemic grammar of WAG, which we use in our implementation, uses the *insert* and *conflate* realisation statements as its structure building operations. WAG uses *order* and *partition* for the ordering operations and, for the classification operations, it uses *preselect.*

Therefore, there are three issues with regard to the mapping of realisation statements into ATMS form, namely:

- what function bundles there are (i.e. insertion issues)
- what network rank will be used to realise a given function bundle (i.e. classification issues)
- how to retain the ordering constraints on these function bundles (i.e. ordering issues)

Functions are inserted at different points in a system network. Different network paths (or selection expressions) conflate functions into various function bundles or constituents. Moreover, the preselection operations on a given function bundle determine the rank of that bundle and hence the system network that should be traversed as a realisation for it. Certain paths require that the function bundles be linearised in a particular order. All of this is compiled by the translation algorithm and stored in what we call here a network snapshot. The details of a network snapshot are discussed next. We also show what parts of a snapshot are used to answer the three main issues of mapping the realisation statements into ATMS forms.

System Network Snapshots

The outcome of the compilation process is stored in system network snapshots. By a system network snapshot we mean the complete product of the translation process, not only the systems and their features translation as was shown earlier on in this chapter. Because the re-translation of a system network yields exactly the same ATMS representation, the translator SNAC pre-compiles the system networks once and saves snapshots of them. This way, our grammatical resources become these compiled ATMS dependency networks. The ATMS-based generator no longer uses the conventional systemic grammar network since the static linguistic information is now put in a new representation, namely: the ATMS dependency networks.

The purpose of a network snapshot is to tell the generation stage modules the details of the mapping (e.g. what are the systems, features, function bundles, ... etc.). However, the general mapping strategy of a particular SFG representation into a specific ATMS form is hard-wired into the architecture. Here, we show what we need to store in a snapshot of a network. A snapshot of a network is a structure having the following slots:

- rank of network
- choosers specification
- features
- systems
- function bundle details (for each)
  - function bundle name
  - conflated functions
  - preselection operations of bundle
- system justifications
- order justifications

The *rank* slot specifies the rank of the underlying systemic network. This slot can be filled by either clause or group rank. The *choosers* slot specifies which chooser is related to any given system. The *features* slot simply lists the features of the network. The next slot (the *systems'* slot) lists the systems of a network. In ATMS terms, features will be represented by assumption nodes and systems by derived nodes. Similarly, a function bundle is represented by an ATMS derived node. The *function bundles slot* has three fields: the function bundle name, the conflated functions forming this bundle, and the accumulation of the preselection operations of the bundle. The function bundle name is a convenient way of referring to particular bundle. For example, we will refer to function bundles in a snapshot using the names `f(1)`, `f(2)`, ... etc. The preselection information will be used when it is time to expand that particular function bundle.

The *system justifications'* slot forms the ATMS dependency network. It relates features and function bundles to systems. While the preceding slots specify what node types there are in a network, this slot specifies how these nodes are connected to each other. It also specifies where each function bundle is inserted (i.e. to which feature a function bundle is attached). Finally, the *order justifications* slot specifies what order restrictions there are in the network and what features imply these restrictions. For a complete explanation of the format of a network snapshot refer to appendix A.

To have an idea about the compilation of a system network into a snapshot, consider the simplified clause network of figure 5.28. Its snapshot compilation is given in figure 5.29. During generation, the information stored in this snapshot is used to instantiate an ATMS dependency network representing the underlying systemic clause network. Figure 5.30 represents a possible dependency network instantiation of the simplified clause network. Note how function bundles and order realisation rules are incorporated into the basic ATMS dependency network. In turn, each of these function bundle nodes will have its own supporting assumption nodes during generation time. Note also how the ordering rules are supported by the assumption features. In the stylistic mode of generation (see Chapter 7 for a detailed discussion), these will interact in an involved manner with the surface stylistic requirements to filter out any source of stylistically inept utterances.

This being said, it is clear now what slots of the snapshot answer the three realisation

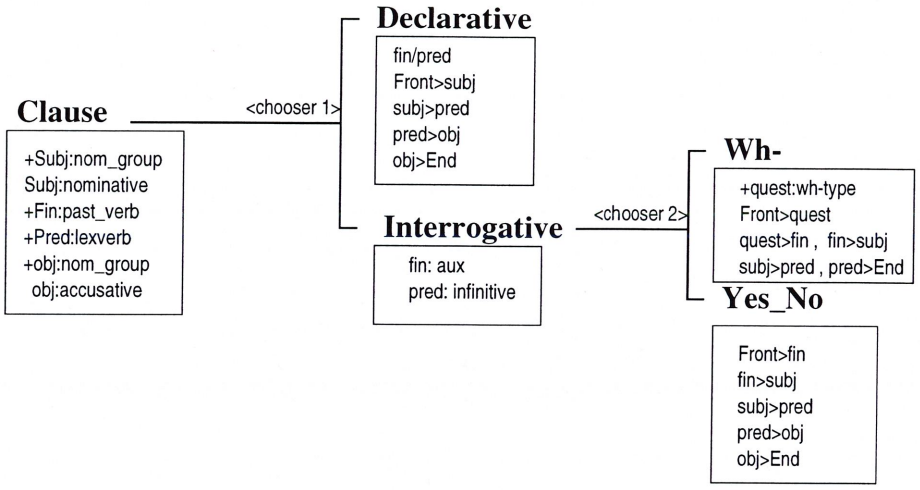


Figure 5.28: A simplified clause network

issues mentioned above. The function bundle slot tackles the classification issues, the system justification slot tackles the insertion issues since function bundles are attached to their proper place in the network via these justifications. Lastly, the ordering issues are taken care of by the order justifications slot.

5.10 Summary and Outlook

In this chapter, we have given a translation algorithm which takes systemic grammars and produces equivalent ATMS dependency networks. We first presented the logical specifications of both the system networks and the ATMS dependency networks. We then showed how the different systemic configurations can be mapped to ATMS representations. We then explained why the conjunctive entry condition cannot be easily represented in the ATMS form. The rest of the chapter was then devoted to solving that problem. We discussed different techniques to handle the awkward conjunctive gate, namely: computing a formula for the whole network in DNF, computing a DNF formula and then factorising it to get a staged representation, and simulating the effect of factorising a DNF formula. Our generation algorithm, which is based on the last technique, was then presented. It re-arranges the systems involved in conjunctive gates without having to compute the DNF formula first and then factorising it. Finally, we discussed how a complete SFG, including realisation statements, is compiled into

```
snapshot(clause, [
       %--choosers of the network
       [[c(1),[_,c(2)]]],

       %--features of the network
       [clause,declarative,interrogative,wh,yes_no],

       %--systems of the network
       [s(1),s(2),s(3)],

       %--function bundle details
        [[f(1),[[subj],[nom_group,nominative]]],
         [f(2),[[obj],[nom_group,accusative]]],
         [f(3),[[pred],[lexverb,infinitive]]],
         [f(4),[[fin,pred],[lexverb,past]]],
         [f(5),[[fin],[aux,past]]],
         [f(6),[[quest],[wh_type]]]
        ],

       %--main (system) justifications
       [[[clause,s(2),f(1),f(2)],s(1)],
        [[declarative,f(4)],s(2)],
        [[interrogative,s(3),f(3),f(5)],s(2)],
        [[wh,f(6)],s(3)],
        [[yes_no],s(3)]
       ],

       %--ordering nodes and justs.
       [ [[declarative],order(front,subj)],
         [[declarative],order(subj,pred)],
         [[declarative],order(pred,obj)],
         [[declarative],order(obj,end)],
         [[wh],order(front,quest)],
         [[wh],order(quest,fin)],
         [[wh],order(fin,subj)],
         [[wh],order(subj,pred)],
         [[wh],order(pred,end)],
         [[yes_no],order(front,fin)],
         [[yes_no],order(fin,subj)],
         [[yes_no],order(subj,pred)],
         [[yes_no],order(pred,obj)],
         [[yes_no],order(obj,end)]
       ] ]))
```

Figure 5.29: A snapshot compilation of the simplified clause network

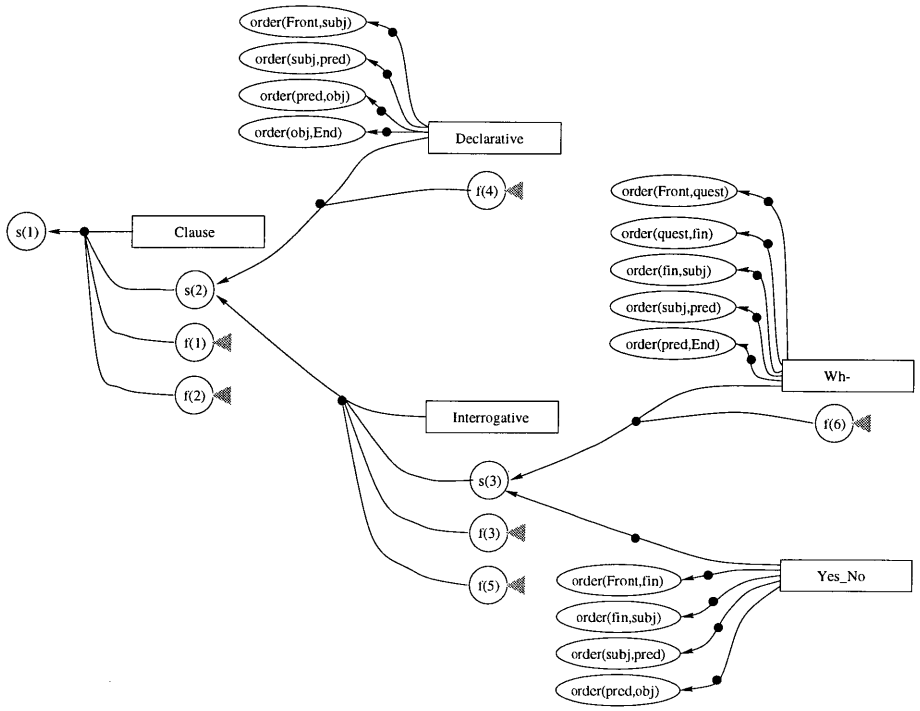


Figure 5.30: A possible instantiation of the simplified clause network

ATMS forms.

In the next chapter, we introduce our ATMS-based architecture for NLG. We show how the outcome of the compilation process is used to generate natural language utterances. Because what we are aiming for in this project is the generation of sentences that have certain surface stylistic requirements, we will then show in a subsequent chapter how the ATMS-based generator can accommodate the specification of these requirements using the same formalism.

Chapter 6

ATMS-Based NLG

In this chapter we present our ATMS-based NLG architecture. We show how the systemic grammars and semantic input are presented to the generation system which then uses them to generate text relying heavily on the ATMS as its search mechanism.

6.1 Introduction

In the preceding chapter, we specified an algorithm for translating system networks to ATMS dependency networks. We now introduce the generation procedure that uses these translated networks. We first introduce the generation algorithm in general showing the main steps involved. We then discuss the individual steps in more detail. We then relate all the parts to each other and show the big picture by giving an overview of the ATMS-based architecture and the different knowledge sources of the system. Towards the end, we give a complete example showing how the conceptual input gets mapped to a natural language text.

The aim of this chapter is to show how the ATMS is used to generate natural language texts. It paves the way for the next chapter which is concerned with the stylistics of the surface form: representation of stylistic requirements and generation of utterances satisfying these requirements.

6.2 ATMS-Based Generation

Our ATMS-based architecture has two main stages: translation and generation. During translation, systemic grammar networks are transformed into ATMS dependency networks. These networks will then be used by the generation module to generate natural language utterances.

Recall that, in general, a typical systemic generator starts with a clause rank network. After responding to all chooser enquiries, it ends up with a selection expression and a corresponding collection of realisation statements. Each of the function bundles resulting from the execution of the realisation statements is in turn realised by traversing its rank network in exactly the same way. Only when a function bundle is of word rank is it considered completely realised.

In spirit, our generation strategy is similar to that of other systemic generators in that it starts with the clause rank and uses a stack to hold the unexpanded constituents. However, in our approach, we are not only interested in one single selection expression but many at the same time. Consequently, the constituents pushed onto the stack at each step are not those of a single path through the system network but of multiple paths.

At a glance, the generation stage goes through the labelled steps of the algorithm shown in figure 6.1. We have labelled the steps for convenience of reference and we will explain each one in detail in the following sections. Note that a snapshot of a network (which we discuss in Section 5.9) simply means a compiled or translated systemic grammar network.

The items on the stack are simply function bundle names (as discussed in Section 5.9). The generation algorithm fetches the function bundle which a name represents and the accumulation of the preselection operations for that bundle from the corresponding network snapshot. The preselection information helps in determining the rank of the network that should be traversed next as a realisation for this unit. For example, suppose that the item popped off the stack is f_1, which happens to be the name of the function bundle $\{subject, actor\}$ having the accumulated preselection operations $\{nom\_group, nominative\}$. These are features of the group network. Hence, the

```
1 PROCEDURE generate()
2 begin
3    start with the snapshot of the clause network;
4    tailor the ATMS dependency network of the current unit;          [TAILORING]
5    assert the circumscribed dependency network;   [NETWORK INSTANTIATION]
6    repeat (for each function bundle)
7    begin
8       do concept-function association;      [CONCEPT-FUNCTION ASSOCIATION]
9       push the unit onto stack;
10   end
11   pop the next element off the stack;
12   if stack is empty then interface-exp-triangles();  [INTERFACING TRIANGLES]
13   elseif current unit is of word rank then
14   begin
15      call lexicalise(current-unit);          [FUNCTION BUNDLE LEXICALISATION]
16      goto step 11;
17   end;
18   else get the network snapshot of the current unit and goto step 4;
19 end;
```

Figure 6.1: The generation algorithm

network to traverse in conjunction with the realisation of this function bundle is the GROUP network.

6.2.1 Tailoring System Network Snapshots

Before we embark on the mechanics of the tailoring process, we discuss the notion of choice in our architecture. We argue that regardless of the choice mechanism applied, the notion of choice in our system remains different from that of previous systemic generators. Their aim was to come up with a single selection expression; ours is to pursue all the selection expressions which are licensed by the conceptual input.

Nature of our Choice Process

Although there are many potential selection expressions in a given system network, traditional systemic generators are interested in only one of them. Choice in such systems is deterministic, and the generation process boils down to determining a selection expression.

Different systems follow different paradigms in determining a selection expression from a system nework. For example, PENMAN-like systems use the chooser-inquiry formalism which was discussed in Section 3.5.2. Such systems assume that their choosers can always choose among alternatives in any generation task. In reality, this is not always the case. There will be situations where a chooser cannot make an informed decision as all the choices at a given point seem equally good [Bateman 97b, O'Donnell 94].

Patten's SLANG generator applies a totally different mechanism to determine a selection expression and hence generate a surface form. In SLANG, the underlying meaning is also represented using system networks. Features from the semantic-level network can preselect[1] features anywhere in the syntactic-level networks. SLANG then uses a production system to efficiently determine a selection expression and execute the relevant realisation statements. Even in this paradigm, there can be situations where a preselected feature in a system network may result in more than one selection expression. Figure 6.2 shows an example where preselecting one feature in a network results in more than one legal path from the root of the network to that feature.

So, regardless of the mechanism applied in the choice process, there will be situations where the conceptual input licenses more than one selection expression. From an NLG perspective, this means that the conceptual input can be realised by more than one surface form. Needless to say, each of these surface forms will have its own stylistic characteristics. In such cases, we would like to pursue all selection expressions until we have good reasons to abandon one path or another based on our surface stylistic requirements. Consequently, our choosers are not obliged to choose one particular feature when they cannot make informed decisions.

[1] This is called *inter-stratal preselection.* See Section 3.5.1 for a discussion of Patten's preselection-based approach.

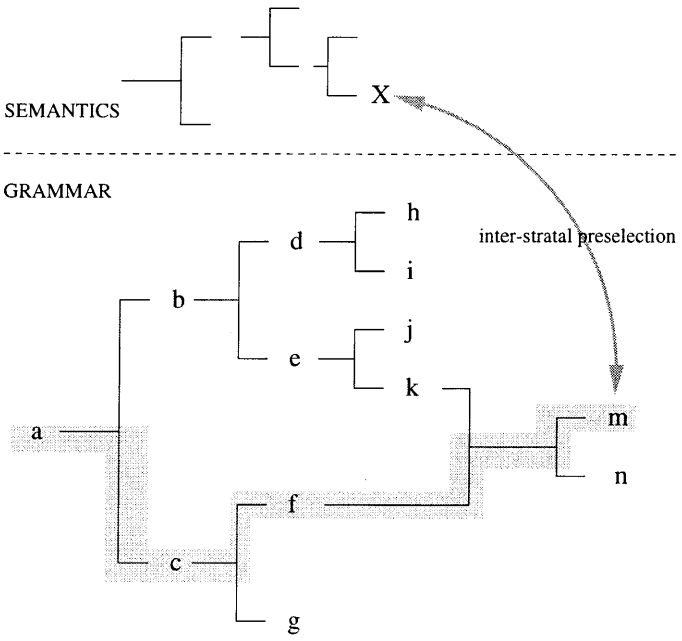


Figure 6.2: An inter-stratal preselection resulting in multiple selection expressions

In our ATMS-based architecture, we adopt a chooser-inquiry approach. We could have used another choice strategy (e.g. a preselection-based approach), but the bottom line remains the same: a non-deterministic notion of choice which can result in more than just one selection expression for a given semantic input. The aim is to generate — among the many selection expressions — only those that satisfy the surface stylistic requirements.

The Tailoring Process

In traversing a complete systemic network, although some choosers will be unable to make informed choices, others will likely be able to do so. Therefore, the complete network needs to be 'tailored' so that only the parts of interest are available. The tailoring algorithm first fires the choosers of the original systemic network as specified in the choosers' slot of the snapshot. With the current unit correctly associated with the semantic piece[2], the choosers enquiries are responded to. Note, however, that our choosers are not obliged to choose one alternative each time they are triggered when they cannot make informed decisions. Because we are trying to avoid

[2] Refer to Section 6.2.3 for a discussion on concept-function association.

committing ourselves to paths that may lead to surface stylistic faults further down the line, a chooser of a system chooses one particular feature only when the conceptual input does not permit the other features to be selected. For example, the chooser of the circumstantial-adjunct system selects *no-circum-adj* only when no circumstantial information is provided by the conceptual input; otherwise both paths (*circum-adj, no-circum-adj*) are kept. Also, if the ACTOR slot is missing from the input then only the passive voice is selected, otherwise both active and passive are equally good (so far at least).

So, whenever a chooser cannot make an informed decision, all children of the particular system are kept; and when it can make a decision only the selected feature is kept. This state of affairs results in three kinds of features: *selected*, *unselected* and *undecided upon*. A *selected* feature of a system renders all its sisters *unselected* features. *Undecided upon* features are all children of a system from which no particular feature is selected. Only the unselected features are of no interest to the generator. Both *selected* and *undecided upon* are relevant and need to be pursued further.

The tailoring algorithm then takes these lists (of selected and undecided upon) and produces a circumscribed version of the dependency network where the irrelevant features, systems, function bundles, and justifications (both system and order justifications) are cut out. What we will have then is a reduced version of the original system network which is still a system network albeit a smaller one. We call this smaller network (e.g. network of figure 5.2) a *tailored* system network and the process of removing the irrelevant parts *tailoring* the system network.

6.2.2 Creating Instances of a Network

Function bundles popped from the top of the stack may be of any rank: clause, group, or word. A word rank unit is realised by an applicable lexeme fetched from the lexicon. A clause or group rank unit is realised by traversing a corresponding system network (i.e. a clause or group rank system network). For different function bundles we will have different traversal outcomes (i.e. various sets of selection expressions). We need some way of recording the traversal of a system network for a given unit. We call this way the expansion or realisation triangle. From an ATMS viewpoint, a word

rank unit is realised by attaching to it an applicable word. A clause or group rank unit is realised by attaching to it an expansion triangle. An *expansion triangle* is an instantiated tailored snapshot of a network that is attached to a function bundle. We sometimes refer to such triangles as the *solution* or *realisation triangles.*

To show what a realisation triangle looks like suppose that we are to realise the constituent `f(1)` which refers to the function bundle *subject* as depicted in figure 5.29. Suppose further that the tailoring process yielded the dependency network[3] shown by part (1) of figure 6.3. Part (1) of the figure represents what we call a realisation triangle. In the remainder of the thesis, a realisation triangle will simply be represented as shown in part (2) of the figure whereby the triangle's features and function bundle names are placed to the right of the triangle (cf. figure 6.4). The nodes in both triangles are connected in exactly the same way. We call the leftmost node (i.e. s(1) or the indexed version of it) the *head* of the triangle. To assert the fact that this triangle is realising the function bundle `f(1)`, the head has to be connected, at some point, to the function bundle node via the justification $s(1) \rightarrow f(1)$.

Note that there might be more than one function bundle of the same rank in one sentence. For example, "the enemy attacked City-X from the air" has three functions of the group rank: the common group '*the enemy*', the proper group '*City-X*' and the prepositional phrase '*from the air*'. Although the expansion triangles of these functions are all of the group rank and hence will be using the same rank network (i.e. group network), they realise different semantic pieces of the input. The selected features might be different and so are their resulting function bundles and lexical items attached to them. Each time a function bundle is to be realised, a new instance of its rank network is created. It is instantiated in such a way as to keep its features, systems, function bundles, and order nodes separate from similar ones of other instantiations.

Creating an instance of a network means asserting an indexed tailored version of that network: its features, function bundles, and order nodes. As mentioned earlier, features are mapped to ATMS assumptions; function bundles, systems, and order restrictions are mapped to derived nodes. Actually, they are mapped to indexed assumptions and derived nodes by the network instance creator. We use the convention `n(x,i)`

[3] Refer to figure 6.7 for the group network upon which the tailoring process is based.

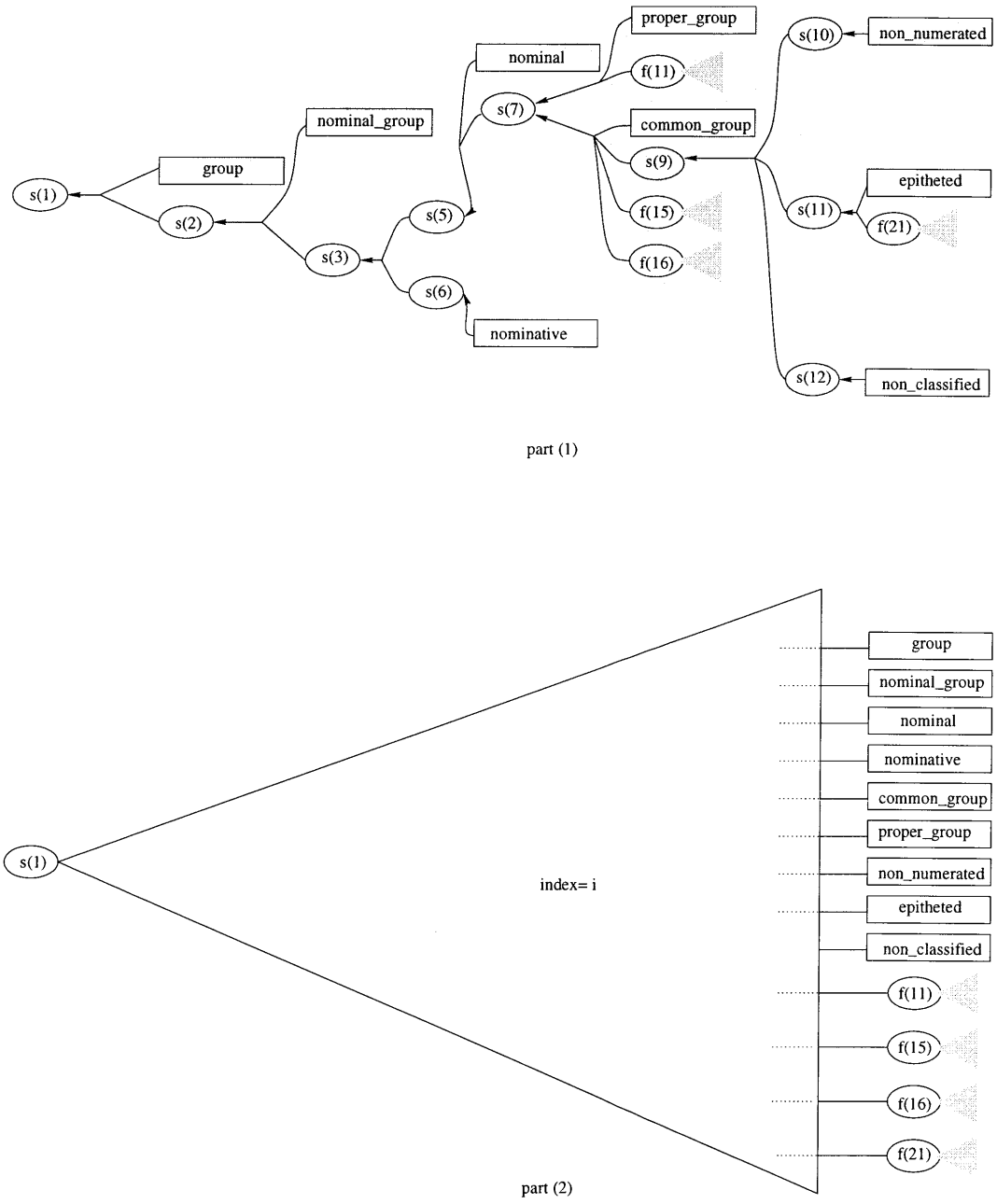


Figure 6.3: A solution triangle for a tailored network

to name the nodes of a network instantiation; where `x` stands for any snapshot node name and the integer `i` for the instantiation index. For example, the `active` node will be referred to as `n(active,0)`, the function bundle `f(8)` as `n(f(8),0)` and the order node `order(subj,pred)` as `n(order(subj,pred),0)` assuming that the instantiation index is 0.

6.2.3 Concept-Function Association

As discussed in Section 3.5, there are different ways of implementing the semantics-syntax interface. The chooser-inquiry paradigm is a common approach which is used in the PENMAN generation system. In our ATMS-based generation architecture we also take a chooser-inquiry approach. Here we present a particular aspect of the approach: the association of grammatical functions to entities of the conceptual input.

For each system there is an associated chooser. The chooser represents the semantic content of the system as it determines the circumstances under which each choice is appropriate [Mann & Matthiessen 83]. To function sensibly, the chooser must ask questions about particular entities of the conceptual input. For example, to choose between singular and plural, the chooser of that system must be referring to a particular entity in order to know whether it is unitary or multiple.

In the PENMAN generator, choosers have access to the Function Association Table (FAT) [Mann & Matthiessen 83] which keeps a record of the associations between the grammatical functions and the corresponding semantic entities. Additionally, one of the tasks of the choosers is to make such an association and put it in the Function Association Table for other choosers to use in answering inquiries. For example, to realise the grammatical function THING which is associated to say *bicycle-x* (i.e. a particular semantic entity), the nominal-group network is traversed. Now, upon entering the Possessiveness system, for example, the chooser of that system must associate the grammatical function DETERMINER with the conceptual entity representing the possessor (say *Tom-y*).

In our system, the remaining function bundles in the tailored network are associated with parts of the semantic input representation before they are pushed onto the stack.

This association is kept in a table similar to PENMAN's FAT. Entries in that table relate function bundles to pointers to particular semantic entity representations. For example, the function bundle *finite/predicate* is associated with the process part of the semantic structure. Similarly, the function bundle *subject/actor* is associated with the agent of the process. Function bundles for which there are no mapping rules are pushed onto the stack without any semantic-syntactic association. These functions are purely syntactic such as the *agency_marker* function which is a preposition and the *mod* function which is a modal verb.

6.2.4 Lexicalisation of Function Bundles

Function bundles popped off the stack are realised by expansion triangles (as shown earlier) if they are of the clause or group rank. If a function bundle is of the word rank then the syntactic restrictions implied by the preselection operations and the denotational restrictions implied by the semantic unit associated with this function bundle are collectively used to fetch the applicable lexemes from the lexicon. Function bundles that are purely syntactic have no semantic association in the FAT and hence only these syntactic restrictions are used to fetch the appropriate lexemes.

It is worth mentioning that lexicalisation or attaching all the applicable lexemes to a word rank function bundle does not necessarily mean that these will be the final lexical choices that will appear in the generated surface forms. Similarly, not all the syntactic choices offered by any tailored network will necessarily show up eventually, since some may not withstand the stylistic scrutiny. This being said, we differentiate between lexicalisation and lexical choice, with lexical choice being taken to mean the process of deciding which words from the lexicon best describe a concept. Therefore the lexicalisation process simply attaches the result of an initial coarse lexical choice process to a function bundle. Still, *stylistic lexical choice* is different in that it is seen from a final-product point of view. Lexicalising a function bundle does not mean that all the lexical choices will appear in the final surface forms. Some choices will not withstand the stylistic constraints. The surviving lexemes represent the result of the stylistic lexical choice operation.

It is this high interaction between the syntactic and lexical choices that makes the

SFL formalism appealing for generation tasks that value surface stylistic properties. Fortunately, the ATMS architecture can seamlessly deal with this interaction using the same language of nodes and justifications.

To show how lexicalisation of constituents is carried out, consider the function bundle *finite/pred.* Semantics-syntax association relates this function bundle with the *process* part of the semantic structure. The preselections incurred by the selection expression (clause,non-modal,past-clause,active,...), for example, gives the set $\{lex\_verb, past\_verb\}$. If the process is KILL-CONCEPT for example, then the union of the syntactic and denotational restrictions is as follows:

$syntactic\_restrictions \cup denotational\_restrictions = lexeme\_selectional\_restrictions,$

or

$\{lex\_verb, past\_verb\} \cup \{\text{KILL\_CONCEPT, NUMBER\_LARGE}\} =$

$\{\text{KILL\_CONCEPT,NUMBER\_LARGE}, lex\_verb, past\_verb\}.$

Depending on the coverage of the lexicon, the lexicaliser could fetch the following lexemes: eliminated, annihilated, extinguished, eradicated, ... etc. The lexemes are then attached to the function bundle by means of ATMS justifications such as:

$$\begin{aligned} lex(eliminated, F_{finite/pred}) &\rightarrow F_{finite/pred} \\ lex(annihilated, F_{finite/pred}) &\rightarrow F_{finite/pred} \\ &\ldots \\ &\ldots \end{aligned}$$

Purely syntactic function bundles have empty sets of denotational restrictions. For instance, the function bundle *agency_marker* is an example of a purely syntactic constituent having the preselection set $\{preposition, by\_prep\}$. This results in the function word *by* being fetched from the lexicon and attached to the function bundle *agency_marker* by the justification $lex(by, F_{agency\_marker}) \rightarrow F_{agency\_marker}$.

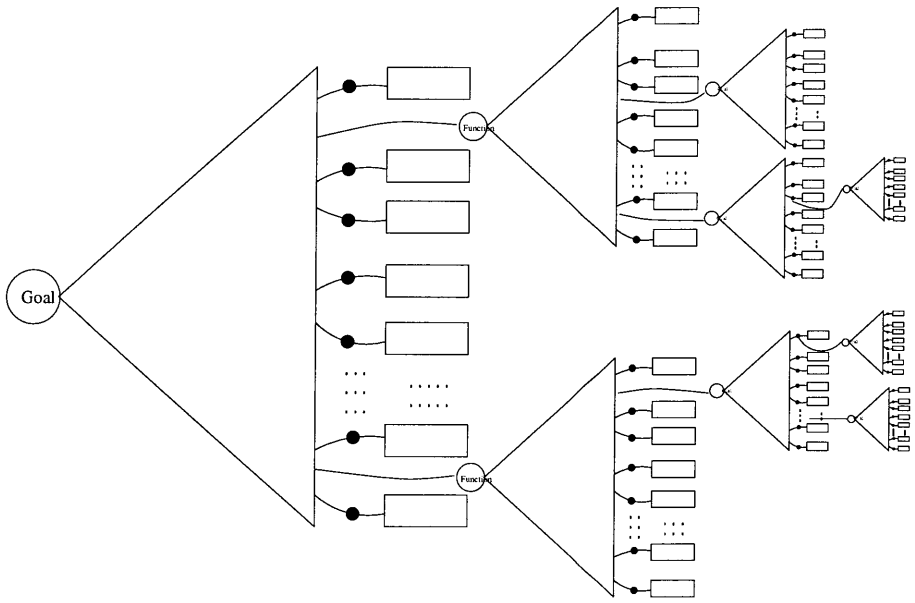


Figure 6.4: Interfacing of realisation triangles

6.2.5 Interfacing Expansion Triangles

The realisation of a unit results in what we call an expansion triangle. In turn, each of this triangle's function bundles is to be realised by its own expansion triangle. After its creation, an expansion triangle is isolated and it has to be connected to the function bundle it is realising in order for it to contribute to the overall generation process. The interfacing process involves connecting a triangle head to the function bundle being realised. This is depicted graphically in figure 6.4.

To improve the efficiency of the system, we delay connecting the triangle head to the function bundle it is realising until all of its function bundles are realised and connected to their own triangles. For this purpose, we have another stack called the delayed-goal-nodes stack which holds the triangle head information. When all the function bundles are realised, we start interfacing the solution triangles from lower levels all the way to the top most solution triangle of the first clause ranked unit. Once a triangle head (say the node *head*) is connected to the function bundle it is realising (i.e. the node $function$) via the justification $head \rightarrow function$, the label of $function$ is computed and its contribution to the overall dependency network is propagated to other nodes for which $function$ is an antecedent.

Interfacing the solution triangles in this way minimises the number of propagation operations. If a lower level triangle is connected to its parent by means of a justification then every update activity in this triangle or one of its children requires the propagation of that update across all levels, up to the topmost goal node. This is because the head node of that triangle appears as an antecedent in other justifications. We avoid all of this unnecessary work by connecting the realisation triangles from lower to higher levels, up to the topmost (i.e. in a bottom-up fashion). Only a fully realised function bundle is connected to its parents; which itself is not connected to its parent until all of its constituents are also realised.

6.3 The Overall System Architecture

Having introduced the generation algorithm and discussed the steps involved in detail, it is time now to put things into perspective and relate the different knowledge sources to each other. The schematic of figure 6.5 provides a high level overview of the ATMS-based generator.

The grammatical resource is a collection of pre-compiled systemic networks put in the form of ATMS dependency networks. As mentioned in the previous chapter, SNAC translates systemic grammar networks into a representation that the generator (STylistics-Aware GEnerator, STAGE) can reason with.

The conceptual input is a structure representing what should be generated by the system. To use the SFL terminology, this is a micro-semantic representation for a single sentence in terms of ideational, interactional, and textual meaning [O'Donnell 94]. Figure 6.6 gives an example of an ideational part of a conceptual input. It represents a material process and information about other roles such as the ACTOR, ACTEE, INSTRUMENT and some circumstantial details (TIME, LOCATION, MANNER of process, ... etc.). Any particular conceptual input can have more or less information depending on the case. Note that we do not regard the semantic representation per se as an important part of our work in this thesis. It only gives a way of driving the generation process and providing a basis for choosers to work correctly.

The output of the system is a set of, what we call here, potential sentences. A potential

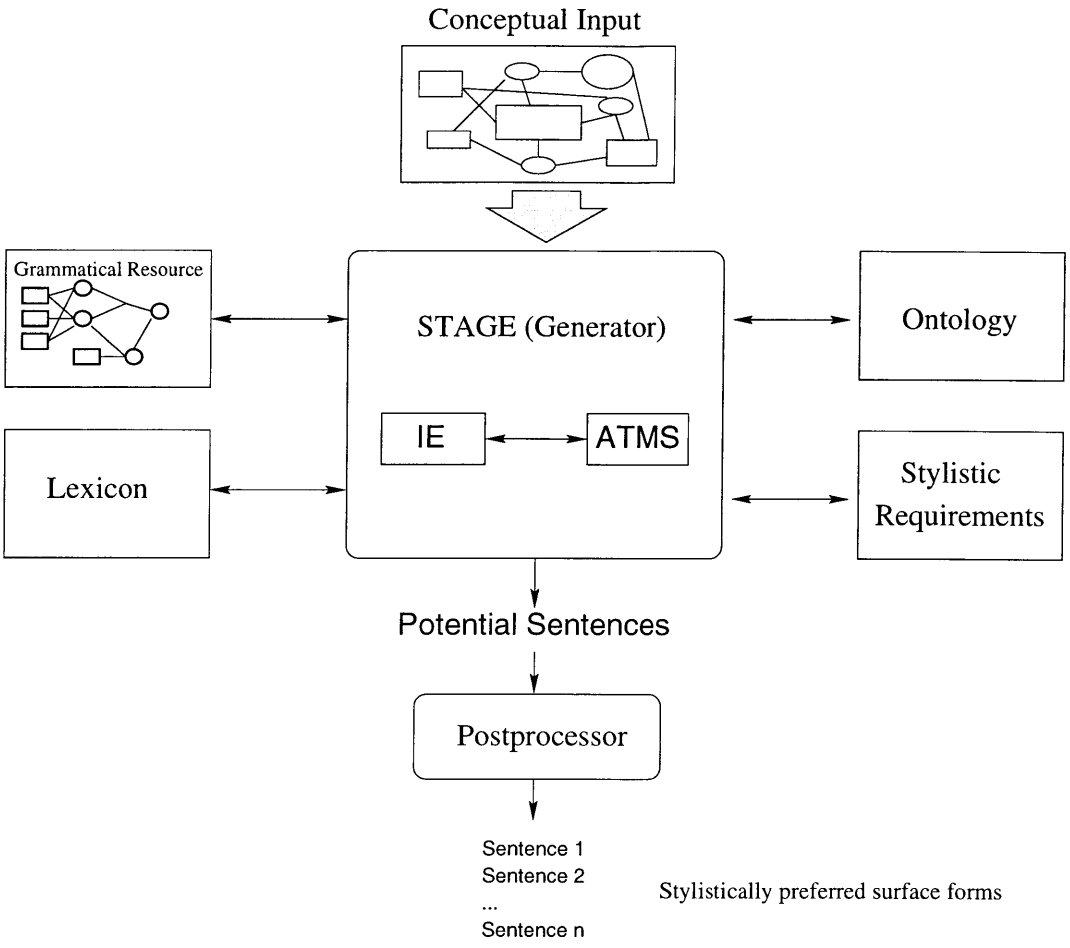


Figure 6.5: An overview of the ATMS-based NLG system

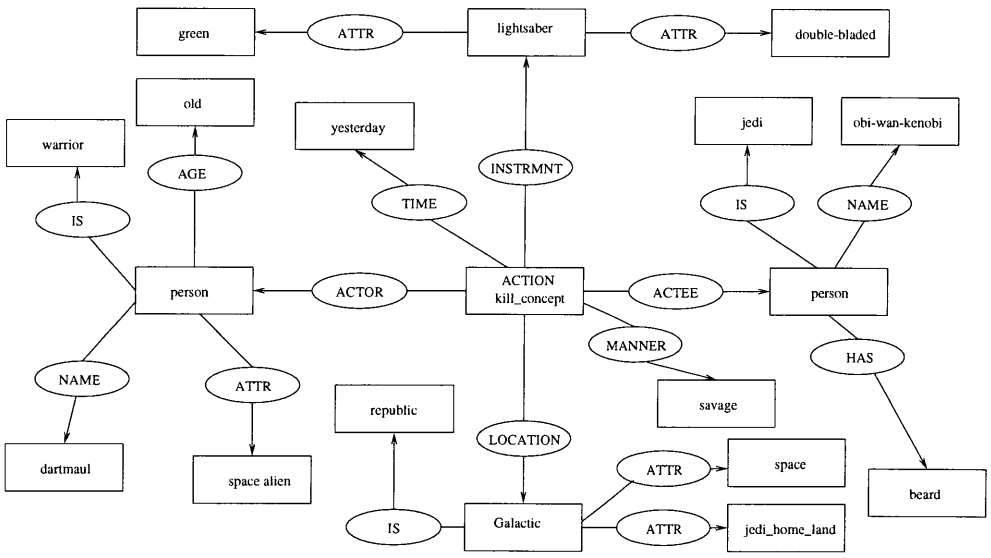


Figure 6.6: An example semantic input representation

sentence is a selection expression and lexical choices for the function bundles incurred by it. As a matter of fact, this is all that is needed to generate the surface form. The post processor then executes the order realisation statements of each selection expression to linearise the lexical choices in syntactically complete utterances.

The lexicon contains, in addition to the usual denotational and syntactic constraints, stylistic properties for each lexical entry such as the sound pattern, root, ... etc.

The stylistic knowledge source specifies what characteristics the generated text should or should not have. The task of taking enough measures to ensure the generation of such texts is STAGE's responsibility. If no stylistic preferences are given, then the process is just plain generation, which gives all the possible paraphrases. In the next chapter, we discuss in detail the internals of the stylistic knowledge source and how the generator uses this knowledge in order to avoid generating utterances that violate the surface stylistic constraints.

6.4 Plain vs. Stylistics-aware Generation

Allowing the ATMS to pursue all the available lexical and syntactic choices is called *plain* generation since no surface stylistic requirements are allowed to constrain the generation process, other than those imposed by the language rules themselves. However, we are very much interested in stylistics-aware generation. In this case, the required stylistic properties are presented to the ATMS in the form of justifications. We assume that it is possible to formalise these requirements in the form of ATMS justifications.

In the next chapter, we will discuss what we mean by the surface stylistic requirements and show how STAGE generates utterances that satisfy such requirements. Meanwhile, we will content ourselves with a complete example showing the workings of the generation system in the plain mode.

6.5 Complete Example

Here, we give an extended example that shows the various stages of the whole process. To keep the example within limits, we will consider only a fragment of the WAG

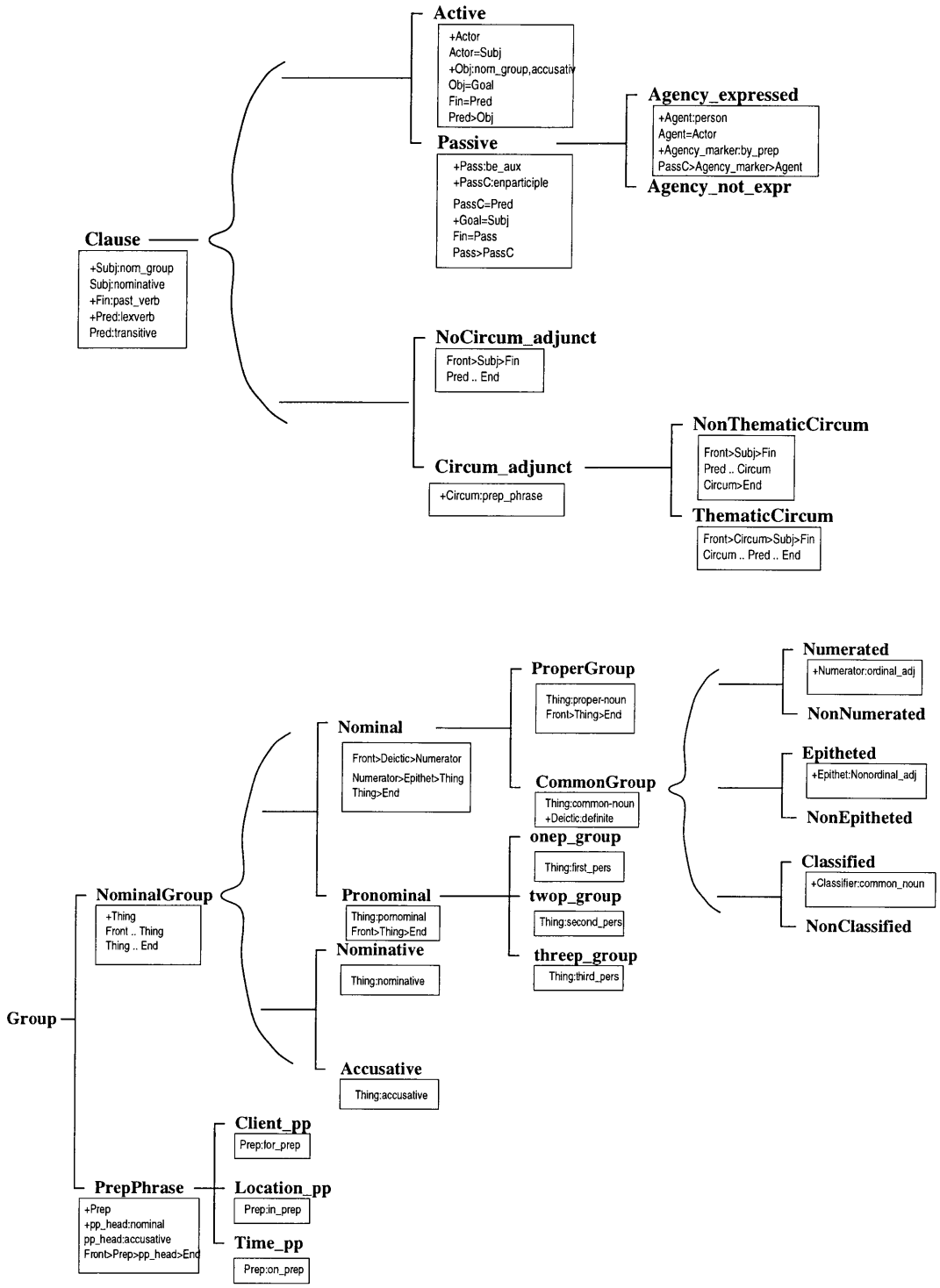


Figure 6.7: Clause and Group network fragments

systemic grammar (see figure 6.7). We will show how the networks of this grammar get translated into ATMS dependency networks. We will give the compiled snapshots of the translated networks. We will then show how the generation algorithm of figure 6.1 maps a micro-semantic representation, similar to the one in figure 6.6, to natural language sentences.

6.5.1 Compilation of the Grammar

The compilation of the grammatical resources is not part of the generation stage. We discuss it here to give a concrete example of how SNAC transforms a systemic grammar into network snapshots which are then used by the generation stage modules. For the grammar shown in figure 6.7, SNAC produces the network snapshots depicted in figure 6.8 and 6.9 for the CLAUSE and GROUP networks respectively.

The first three slots shown act as a declaration section. It tells the module which will create instances of these networks that the features are to be mapped to ATMS assumptions. It also tells the instance creator that both the systems and function bundles are to be mapped to derived nodes. The dependency network itself is constructed by means of the main system justifications of the slot that follows (i.e. the fourth slot). The order nodes and justifications are asserted based on the information provided by the ordering-nodes-and-justs slot. Order nodes are ATMS derived nodes of the form $order(fun_1, fun_2)$ meaning that fun_1 immediately precedes fun_2. The justifications of the ordering slot specify which combination of features imply a particular ordering restriction. For example, the justifications $active \wedge thematic\_circum \rightarrow order(front, circum)$ and $active \wedge thematic\_circum \rightarrow order(circum, subj)$ mean that an active sentence with a thematic circumstance imposes a restriction that the *circum* function comes at the front of an utterance followed by the *subj* function.

From the clause snapshot specification, we can construct the dependency network of figure 6.10. A dependency network for the group network can be constructed in the same way. Note that in the dependency network of figure 6.10, function bundle names — represented by derived nodes (e.g. `f(1)` and `f(3)`) — are not supported by any assumptions. This is fine during compilation. During generation, however, each of these function bundles will have its own realisation triangle. This fact is indicated by

```
snapshot(clause, [
        ...
        ...
        %--features of the network
        [clause,active,passive,agency_expr,agency_not_expr,
         circum_adjunct,non_thematic_circum,thematic_circum,
         nocircum_adjunct],

        %--systems of the network
        [s(1),s(2),s(3),s(4),s(5),s(6)],

        %--function bundle details
         [[f(1),[[subject,actor],[nom_group,nominative]]],
          [f(2),[[subject,goal],[nom_group,nominative]]],
          [f(3),[[finite,pred],[lexverb,past_verb,transitive]]],
          [f(4),[[finite,pass],[be_aux,past_verb]]],
          [f(5),[[pred,passc],[lexverb,transitive,enparticiple]]],
          [f(6),[[object,goal],[nom_group,accusative]]],
          [f(7),[[agency_marker],[by_prep]]],
          [f(8),[[agent,actor],[nom_group,accusative]]],
          [f(9),[[circum],[prep_phrase]]]
         ],

        %--main (system) justifications
        [[[clause,s(2)],s(1)],
         [[s(3),s(4)],s(2)],
         [[active,f(1),f(3),f(6)],s(3)],
         [[passive,s(5),f(2),f(4),f(5)],s(3)],
         [[agency_expr,f(7),f(8)],s(5)],
         [[agency_not_expr],s(5)],
         [[circum_adjunct,s(6),f(9)],s(4)],
         [[non_thematic_circum],s(6)],
         [[thematic_circum],s(6)],
         [[nocircum_adjunct],s(4)]
        ],

        %--ordering nodes and justs
        [ [[active],order(pred,object)],
          [[passive],order(pass,passc)],
          ...
          ...
        ] ]))
```

Figure 6.8: A snapshot of the clause network of figure 6.7

```
snapshot(group, [
        ...
        %--features of the network
        [group,nominal_group,prep_phrase,nominal,pronominal,
         nominative,accusative,proper_group,common_group,
         onep_group,twop_group,threep_group,
         numerated,non_numerated,epitheted,nonepitheted,
         classified,nonclassified,client_pp,location_pp,time_pp],
        %--systems of the network
        [s(1),s(2),s(3),s(4),s(5),s(6),s(7),s(8),s(9),s(10),s(11),s(12)],
        %--function bundle details
         [[f(11),[[thing],[proper_noun]]],
          [f(12),[[thing],[pronominal,first_pers]]],
          [f(13),[[thing],[pronominal,second_pers]]],
          [f(14),[[thing],[pronominal,third_pers]]],
          [f(15),[[thing],[common_noun]]],
          [f(16),[[deictic],[definite]]],
          [f(17),[[prep],[client_prep]]],
          [f(18),[[prep],[location_prep]]],
          [f(19),[[prep],[time_prep]]],
          [f(20),[[numerator],[ordinal_adjective]]],
          [f(21),[[epithet],[nonordinal_adjective]]],
          [f(22),[[classifier],[common_noun]]],
          [f(23),[[pp_head],[nominal,accusative]]] ],
        %--main (system) justifications
        [[[group,s(2)],s(1)],
         [[nominal_group,s(3)],s(2)],
         [[prep_phrase,s(4),f(23)],s(2)],
         [[s(5),s(6)],s(3)],
         [[nominal,s(7)],s(5)],
         [[pronominal,s(8)],s(5)],
         [[proper_group,f(11)],s(7)],
         [[common_group,s(9),f(15),f(16)],s(7)],
         [[onep_group,f(12)],s(8)],
         [[twop_group,f(13)],s(8)],
         [[threep_group,f(14)],s(8)],
         [[s(10),s(11),s(12)],s(9)],
         [[numerated,f(20)],s(10)],
         [[non_numerated],s(10)],
         [[epitheted,f(21)],s(11)],
         [[nonepitheted],s(11)],
         [[classified,f(22)],s(12)],
         [[nonclassified],s(12)],
         [[client_pp,f(17)],s(4)],
         [[location_pp,f(18)],s(4)],
         [[time_pp,f(19)],s(4)]],
        %--ordering nodes and justs
        [ ...
          [[common_group],order(front,deictic)],
          [[common_group,numerated],order(deictic,numerative)],
          ...
          ...
          [[classified],order(classifier,thing)] ]]))
```

Figure 6.9: A snapshot of the group network of figure 6.7

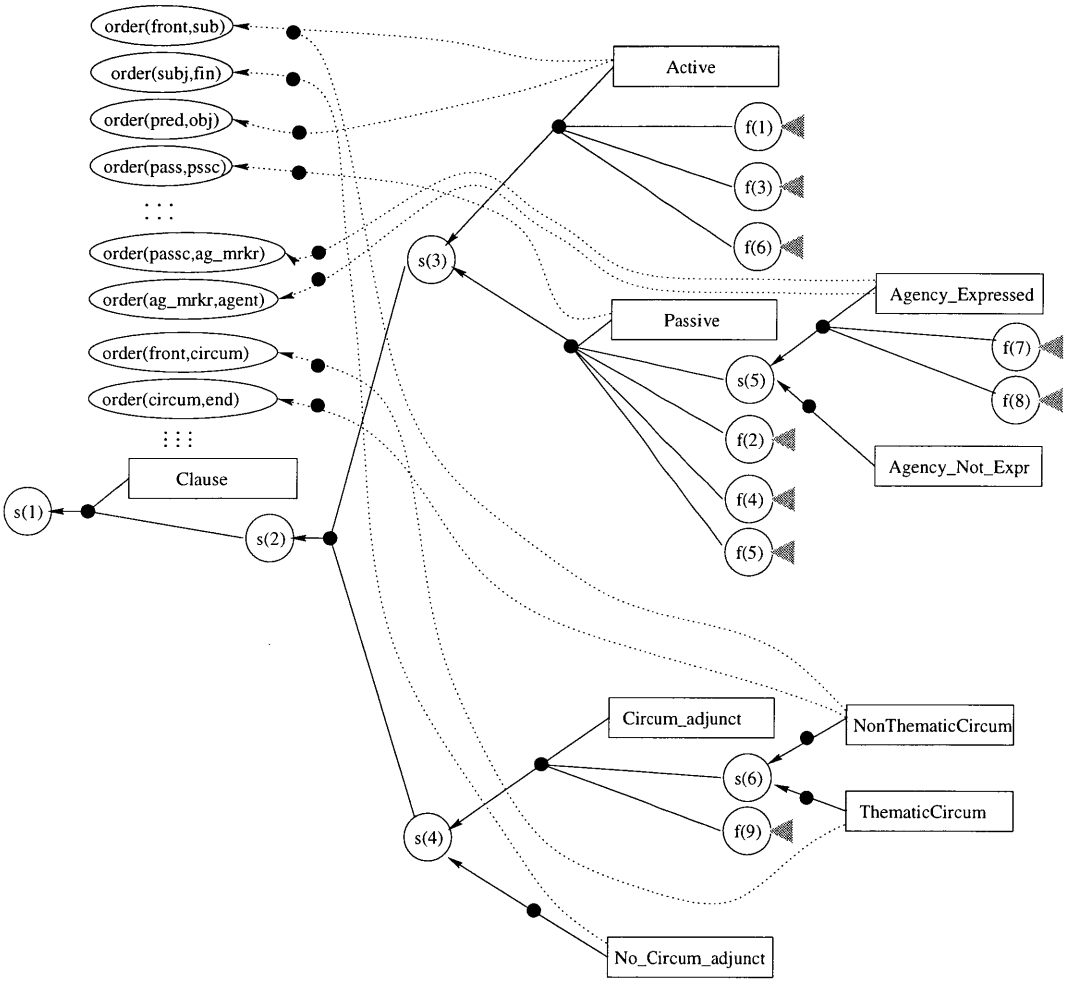


Figure 6.10: The dependency network constructed from the clause snapshot

a small grey triangle attached to function bundle nodes. Meanwhile, each function bundle needs to be associated with semantic entities and pushed on the stack for later expansion. Later on and in place of these grey triangles, each function bundle will have either a triangle of its own or a set of synonymous lexical items depending on its rank.

Note also how the order nodes are related to the assumption nodes (i.e. the features). In the plain mode of generation, they are used at the very end to order the lexical items in linearised surface forms. In the stylistics-aware mode, these order nodes and justifications play an important role in deciding what selection expressions and/or lexical items survive the stylistic requirements. They interact in a complex way with the user-specified surface stylistic constraints, as will be discussed in detail in the next chapter.

6.5.2 The Generation Procedure

Suppose that we have a micro-semantic input which is more or less similar to the conceptual representation of figure 6.6. The first step is to associate the clause as a whole with this representation. This information is recorded in a function association table similar to that of PENMAN. The clause unit is pushed on the stack and the **generate()** procedure is invoked. Now, we show the different stages of the generation algorithm.

1. **Get the top element of the stack and determine its rank**[4]**.** Initially, the top of the stack is the topmost unit which is of the clause rank.

2. **Tailor its corresponding network.** Here, we fetch the clause network snapshot and start firing its choosers. The choosers are relaxed in the sense that a chooser is not obliged to choose a single feature from a system when it cannot make an informed choice. Based on the choosers' responses, we remove the irrelevant parts and we get a tailored network.

   For example, for the simple clause network that we are considering, the semantic input tolerates choosing active or passive. Therefore, the associated chooser will

[4] The preselection operations associated with a function bundle are used to determine the rank of that constituent. For example, a function bundle with an associated preselection operation that preselects a feature from the group network means that this function bundle is of the group rank.

not choose one of them. Instead, it will keep both of them as undecided upon features. Also, since the micro-semantic representation has some circumstantial information, all the paths of the CIRCUMSTANCE system will be maintained.

3. **Instantiate a dependency network as a realisation for the current unit.** The tailored network is indexed in order to make it unique from other possible instantiations of the same network. We get the same dependency network but with each of its nodes indexed. As mentioned earlier, we use the convention `n(x,i)` to name the nodes of a network instantiation; where `x` stands for any snapshot node name and the integer `i` for the instantiation index. For example, the `active` node will be referred to as `n(active,0)`, the function bundle `f(8)` as `n(f(8),0)` and the order node `order(subj,pred)` as `n(order(subj,pred),0)` assuming that the instantiation index is 0. So an indexed version of the tailored network is asserted (i.e. all its remaining features, function bundles, and order nodes).

4. **Associate the Function bundles with Concepts and push them onto the stack.** The PENMAN way of doing the association is via the choosers. To do this association in our implementation, we use a table which associates a function with a symbolic label in the conceptual representation. For example, in the function bundle `subject/agent`, the function *agent*, since it is biased towards the semantics side of the bundle, is used to associate that function bundle to the ACTOR part of the conceptual representation. Similarly, function bundles containing `predicate` or `process` are associated with the ACTION of the micro-semantic representation and so on. After the association between the function bundles and the underlying semantic entities is done, those bundles are then pushed onto the stack. To proceed with the example, suppose that the top item of the stack is `f(3)` and the item below it is `f(1)`.

5. **Recursively, generate each constituent on the stack until the stack is empty.** Each unit on the stack needs to be realised in a similar manner to that which has just been shown. We show, next, the realisation of the function bundle nodes `f(3)` and `f(1)`, which represent the bundles `finite/pred` and `subj/actor` respectively.

- **Realisation of `f(3)`:** Now, the next element popped off the stack is `f(3)` (instantiated as `n(f(3),0)`) which is of the word rank (refer to figure 6.8). Such a function bundle is realised by fetching the applicable words from the lexicon and attaching them to it. To lexicalise this function bundle, we take the union of both the grammatical constraints of the function bundle and the conceptual constraints of its corresponding entity in the FAT-like table. This results in the following collection of selectional constraints: {`lex-verb, past-verb,transitive, ... KILL-PROCESS, MANNER-RUTHLESS`} which fetches the following lexemes from our small lexicon *massacred, annihilated, exterminated.*

- **Realisation of `f(1)`:** Since `f(1)` (instantiated as `n(f(1),0)`) is not of the word rank, it will need to have its own realisation triangle, which is achieved by determining its rank, getting the corresponding network snapshot, and moving control to step 4 according to the algorithm of figure 6.1. This starts the tailoring process for the group network snapshot. Assume that the choosers' responses were: (Nominal-Group), (Nominal), (Nominative), (Proper-Group, Common-Group), (Numerated, NonNumerated), (Epitheted, NonEpitheted), (Classified, NonClassified). Note that tuples of more than one feature indicate that the choosers of these systems were unable to make informed choices. Based on the choosers' responses, the group network is tailored, instantiated, and its remaining function bundles (i.e. `f(11),f(15),f(16), f(20),f(21),f(22)`) are associated with conceptual entities and also pushed onto the stack.

6. **Interface the expansion triangles.** When all the function bundles are realised completely, this being indicated by an empty stack, we start interfacing the solution triangles from lower to higher levels in the way shown in figure 6.4. As the triangles are interfaced, the ATMS computes the labels for the involved nodes incrementally. In the end, the label of the goal node (i.e. the leftmost triangle head) contains all the solutions. These are then linearised based on the order nodes associated with each selection expression.

Figure 6.11 gives a sample of the sentences that can be generated in the plain mode. Remember that no surface stylistic restrictions are specified in this mode. So, the output is mainly determined by the semantic input, lexicon, and syntactic rules. For the input we provided, we get a large number of sentences with different syntactic structures and lexical variations.

```
Dartmaul annihilated the Jedis.
The last ruthless warrior exterminated the Jedis on Friday.
On Friday, the last ruthless warrior exterminated the Jedis.
...
...
The black bearded Jedis were massacred.
The black bearded Jedis were massacred by Dartmaul.
The black bearded Jedis were massacred on Friday.
On Friday, the Jedis were annihilated.
The Jedis were annihilated by the warrior on Friday.
On Friday, the Jedis were annihilated by the last ruthless warrior.
...
...
```

Figure 6.11: Sample sentences with different lexical and syntactic choices

6.6 Summary and Outlook

We have shown how the translated systemic grammars are used under the ATMS-based architecture to generate natural language utterances. The paraphrases generated are only constrained by the conceptual input. That is, no surface stylistic requirements are considered as yet. Although this is not the way we intend to use the generator, it shows the basic workings of the generation procedure: systemic grammars are represented in ATMS dependency networks and these are then used to realise the different functional units of language.

The number of paraphrases generated in this way is large. Both the available syntactic structures and the applicable lexemes for each word rank constituent affect the number of paraphrases generated. This is where the surface restrictions come into play. They constrain which lexeme can neighbour which and in what syntactic structures. This chapter paves the way for the next one which deals with stylistics-aware generation.

Chapter 7

Stylistics-Aware Generation

In this chapter we demonstrate how the ATMS-based architecture can be used for stylistics-aware generation. After re-stating what is meant by surface stylistic constraints (SSC), we discuss how we can tackle them within the notion of functional constituency. We design a framework for user-defined stylistic requirements which can be used to parameterise the generation process described in the previous chapter. We finally demonstrate the specification of some sets of surface stylistic requirements, namely: word adjacency constraints, poetry metre specification and text size limitations. For each set of SSC, we show how the specifications work within the framework.

As NLG technology matures, more and more applications that require computer-human interaction will incorporate text generation components of some sort. This means that generators will have to produce not only understandable but also stylistically appealing texts. These applications may require the generation program to produce text with certain rhyme, alliteration or even poetic aspects. Such tasks may require redoing syntactic and lexical choices under constraints from different levels.

Style is generally defined as the choice between the various ways of expressing the same message. [DiMarco & Hirst 93] in their important work on style in computational linguistics, give the following definition of style:

> Style is created through subtle variation, seemingly minor modulations of exactly what is said, the words used to say it, and the syntactic constructions employed, but the resulting effect on communication can be striking. [DiMarco & Hirst 93]

They identify four parameters that determine the stylistic feel of an utterance: lexical, syntactic, thematic and semantic aspects. Roughly, we can divide these parameters into deep (e.g. thematic and semantic) and surface parameters (e.g. lexical and syntactic). This work is about surface realisation of stylistically preferred sentence-size utterances and, obviously, what concerns us the most are the lexical and syntactic parameters of style.

[Nicolov 99] identifies two issues in generating stylistically preferred paraphrases: first, what does it mean for a realisation to be better than another one? Secondly, how can we incorporate the notion of betterness in the process? This work tackles the second issue, as the object of this project is not to come up with the best stylistic rules but to show that different stylistic requirements can, on the one hand, be expressed using the ATMS representation language and, on the other hand, be accounted for within the proposed architecture. It will be left to the user or the application to specify what is considered stylistically good or bad. As a matter of fact, what is considered a stylistic problem that should be avoided in one application might be a requirement in another one. That is why we only provide tools for the specification of stylistic requirements. The internals, or the meaning of the stylistic nodes and justifications, are left for the user's own judgement.

7.1 The Other Dimension of the Process

So far, we have been considering one dimension of the generation process: that of producing grammatical utterances. The other dimension is to generate, in addition to the grammaticality requirement, only those utterances that possess certain stylistic characteristics. What we are after is the surface form: how it looks and sounds, and whether or not it complies with the surface stylistic requirements. Both lexical choices and syntactic structures shape the final utterances. The lexical choices obviously spec-

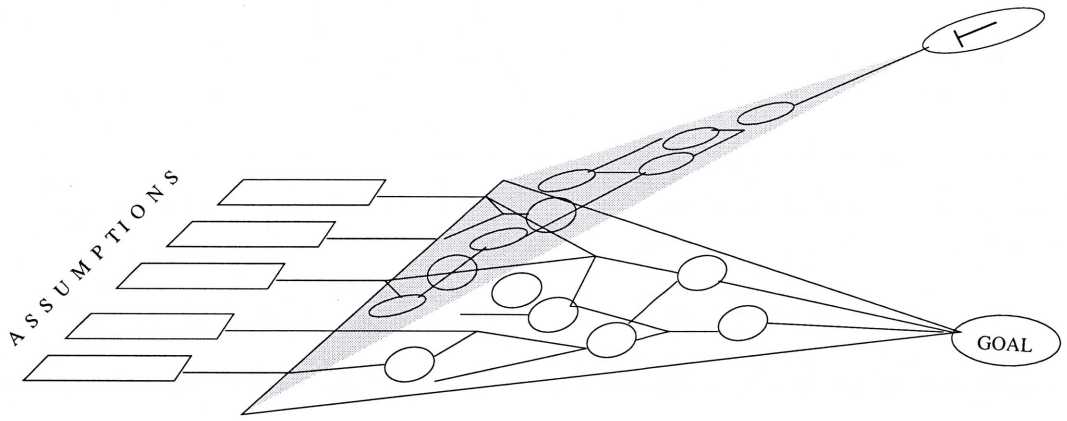


Figure 7.1: The stylistic dimension of the generation process

ify what words will be available in the generated utterance. The syntactic choices order these building blocks in a linearised string of text. For most generation tasks the process ends here. However, for applications that have specific surface stylistic requirements generators must ensure that these choices do not conflict with the stylistic requirements.

Figure 7.1 shows how SSC can be used, in conjunction with the usual realisation triangles, to filter out any source of surface stylistic faults. The lower dependency network is a realisation triangle (as introduced in Section 6.2.2). The upper dependency network will be called a stylistic triangle. The similarity between the two triangles is simply that they are both ATMS networks with distinctive head nodes forming a triangular configuration. The realisation triangle preserves the dependency relations between its nodes (e.g. systems, features) as specified by the underlying systemic grammar networks; whereas the stylistic triangle maintains dependency between a different set of derived nodes (e.g. realisation nodes and their implications) as specified by the given surface stylistic requirements. Both triangles in the figure are supported by the same set of assumptions. This way, both the pure syntactic realisation triangle and the stylistic triangle together determine the surviving sets of assumptions and hence the final problem-free utterances. Both triangles work hand in hand to generate the final utterances. If a node in the stylistic triangle is found to be contradictory — according to the given stylistic requirements — then the ATMS notes that all the sets of assumptions labelling it lead to a contradiction (or surface stylistic problem in

our case). The ATMS then removes these contradictory sets, which form the seeds for possible surface problems, and all their supersets from the labels of the remaining nodes, including the nodes of the lower (syntactic) triangle. This restricts the number of generated sentences — which can be read off the GOAL node label — to only those free of surface problems.

An example of work that uses a similar approach is that of [Power 00]. In his constraint satisfaction text planner, Power designed a procedure that can generate all text structures and, to limit the number of solutions, he applied further constraints in order to eliminate solutions that are *stylistically* deficient. However, the problem addressed in that work is different. Power focuses on generating preferred text structures (TSs) which realise a given rhetorical structure (RS), as there can be a huge number of them, some of which may be stylistically better than others. The kinds of (deep) stylistic problems he is trying to avoid are: whether the nucleus of a particular rhetorical relation (e.g. the background relation) precedes its satellite in a TS, whether a sentence contains more than one text-clause, and whether a TS forms a paragraph with only one sentence. These are micro-planning issues that do not involve surface problems due to unfortunate syntactic and/or lexical choices made.

This brings up the possibility of modelling the problem of stylistics-aware generation as a Constraint Satisfaction Problem (CSP). A CSP is a problem composed of a set of variables, a domain for each variable, and a set of constraints that restricts the values the variables can simultaneously have [Poole *et al.* 98]. A solution to a CSP is an assignment to all variables such that no constraint is violated. Generally speaking, the problem at hand can be viewed as a CSP since it is a multidimensional selection problem. Each selection point can be represented by a variable. At each choice point, the available options would then represent the domain for each variable. The set of constraints would consist of the syntactic constraints of the underlying lexico-grammatical resource and the surface stylistic requirements imposed the given application. Following the CSP alternative usually involves answering the difficult question of how many variables we need to formulate our problem as a CSP [Power 00]. In generating from systemic grammar networks, we do not know in advance how many choice points we will hit. It all depends on the path we follow in a given situation.

Although other search strategies might have been adopted, we opted for the ATMS search mechanism to be the core of our NLG architecture because of the similarity between the representation of the grammatical formalism (i.e. system networks) and the ATMS representation (i.e. dependency networks). We exploit the logical connection between the two representations. We translate the first representation into the second and let the ATMS do the job it is particularly designed for (i.e. reasoning about dependency networks).

7.2 Surface Stylistic Constraints

Here we re-state what we mean by surface stylistic constraints (SSC). SSC are those stylistic requirements that are known beforehand but cannot be tested until after the utterance or (in some lucky cases) a proper linearised part of it has been generated. During the linguistic realisation process, different lexical and syntactic choices are made which impose certain interdependent constraints on the surface form. Consequently, the final utterance exhibits certain stylistic features. If this surface form does not comply with the application's stylistic preferences, it is considered unacceptable although it might be a perfect utterance otherwise. Examples of SSC might be:

1. **Inter-lexical Constraints:** Due to unfortunate lexical choices the utterance might be awkward or ambiguous when words are linearised, although each word is good in isolation. For example, the French pronouns *le, la* cannot precede words starting with *e*. When that happens, both are abbreviated to *l'*. Now, if we want to generate (in French) an unambiguous utterance, the choice between the feminine pronoun *la* and *Sarah* depends on the next word [Reiter & Dale 00]. Although simple, this example shows that there are cases where generators cannot make a final decision on lexical choice until after the surface form has been linearised and its words inflected. Moreover, the phenomenon of collocations is another example of inter-lexical constraints as will be discussed in Section 7.9.1.

2. **Text Size Constraint:** Some lexical choices result in longer utterances because of the way in which each word packages information. The cumulative effect of such verbose choices can be longer texts. However, the exact length of text is

not known until after the text is generated and only then can it be compared to the size limit it is allowed to occupy. In the STOP project, Reiter discusses how even things like punctuation, inflection, and font type can play a role in keeping the text within the allowed limit [Reiter 00]. Also, [Bouayad-Agha *et al.* 91] in their patient information leaflets generation project, regard text length and lexical choice, for example, as matters of style which "require rewording of the text". These constraints are imposed by the user (the patients in this case): the "patients might object that the sentences are too long, or that technical words [...] are used instead of familiar ones".

3. **Poetic Constraints:** In poetic writings, issues like metre, rhyme, and alliteration all restrict the words an NLG system can select in relation to what has already been chosen. Applications that require this kind of text are starting to appear in the NLG literature such as poetry [Manurung *et al.* 00], punning riddles [Binsted & Ritchie 94], and story generation [Bailey 99, Binsted & Ritchie 96].

Current systemic generation algorithms are prone to surface stylistic problems because they need to make decisions at different choice points before the surface form or part of it has been built. When there is not enough information to make a decision, current generators resort to one of two strategies: selection of a default or selection of a random alternative. [Knight & Hatzivassiloglou 95] show that neither strategy guarantees problem-free surface forms, as "the default choices frequently are not the optimal ones" and the alternative of randomised decisions entails "the risk of producing some non-fluent expressions".

The idea of our solution is to discourage both types of choices (i.e. default and randomised) whenever there is not enough information available, and to encourage the user to specify his stylistic preferences so that they eventually play a role in the choice process. This way, sources of surface problems get filtered out and only problem-free utterances get generated.

7.3 Hard vs. Soft Stylistic Constraints

The decision to incorporate surface stylistic constraints in the overall generation process motivates the basic architecture of our generation system. Another equally important decision is how to treat these stylistic requirements. Are they absolute hard constraints or graded soft constraints (i.e. mere preferences)?

Hard constraints can either be satisfied or not. This makes them easy to test for. However, once they result in more than one solution, there is no way one can use them to rank the solutions or weigh them against each other. Soft constraints, on the other hand, can be satisfied to different degrees and hence can be weighed against each other. The problem with soft constraints is that they are not easy to formulate and evaluate. Another difficulty with them is that they can conflict with one another. For example, the preference to have a brief and concise text on the one hand, and to sound "overly bookish" on the other may not be attainable at the same time [Stede 96b]. Although soft constraints can, in theory, be used to rank the solutions, ranking natural language texts is not an easy task. It is usually considered a post-processing task that may require the help of a human being.

In this project, we chose to have stylistic requirements being hard constraints. In a way, this concords with the fact that the particular ATMS implementation we use in this project can only handle true-or-false type of constraints. Another reason for our stance is that, as a general observation, surface stylistic constraints (e.g. poetry metre) are more to the side of hard constraints; and deep stylistic constraints (e.g. text formality) are better represented by soft graded preferences. For example, a sentence rhythmic sequence either complies with a given poetry metre or not. However, with respect to deep stylistic constraints, one can talk about the degree of formality or floridity of a text. In such cases, if we cannot have a totally formal text we can *maximise* the degree of formality of the text. If more than one surface form are generated, then we can weigh them against each other and rank them in a descending order of formality. Obviously, deep stylistic requirements — which are beyond the scope of this thesis — are better tackled using soft constraints.

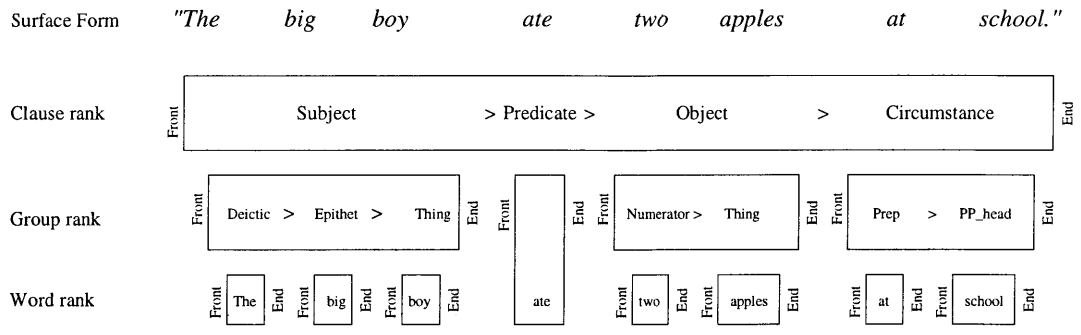


Figure 7.2: The functional constituency of a surface form

7.4 How to Capture Surface Stylistic Properties

Lexical and syntactic choices are highly interdependent. It is not until we lexicalise a function bundle that we start to know what other lexemes might come before or after it. Therefore, the lexicalisation process is an important step in our stylistics-aware architecture. Moreover, a function bundle that has just been lexicalised might be only one of many function bundles realising a higher level unit which itself is a function bundle surrounded by others realising yet another syntactic unit.

Therefore, it is necessary that the lexicalisation operation is followed by another process that propagates the lexeme stylistic consequences to higher levels and across functional boundaries. During the propagation process, order rules encountered at each level play an important role in revealing what might be a source of surface problems. They also govern what stylistic properties can be passed from the current function bundle to its parent unit. We show next the mechanisms the SFL formalism offers and which we can exploit to tackle the problem of surface stylistic constraints on lexical and syntactic choices. This is best understood within the notion of rank and constituency, since surface requirements must be met both within and across constituents.

7.5 Using Functional Constituency

As discussed in Section 3.3.3, a constituent is part of a structure that has other constituents as parts. A constituent might be atomic also, e.g. a word can be a constituent of a phrase. In general, a constituent of the clause rank consists of group rank con-

stituents which tend to consist of word rank constituents. This is depicted in figure 7.2. Note that although the lexemes form the final utterances, they cannot be accessed directly at the surface level. They are encapsulated within layers of constituency. Any ordering operations, for example, must refer to the functions of that level such as *object* > *circumstance*, where $fun_1 > fun_2$ means that fun_1 immediately precedes fun_2, and not the lexemes realising them such as *"two apples"* > *"at school"*.

In the following, we present the basic notions within the SFL formalism, that would enable us to control how the surface form (i.e. lexical and syntactic choice) complies with the surface stylistic requirements. In the discussion below, we draw upon the following hypothesis:

> The stylistic properties of a constituent of the clause or group rank are determined by its sub-constituents' stylistic properties; and the stylistic properties of a constituent of the word rank is determined by the stylistic properties of the lexeme realising it.

7.5.1 Function Bundle Lexicalisation

A function bundle of the word rank is fully realised by an appropriate lexeme. In our ATMS-based architecture, *lexicalisation*[1] means the process of attaching a lexeme or a group of (synonymous) lexemes to a function bundle by means of justifications.

Lexicalisation is an important step in the stylistics-aware mode of generation. It is here that the just-realised function bundle gets its possible stylistic properties. The lexicalisation operation represents the seed for the propagation of the lexeme stylistic properties across different ranks up until the topmost clause rank since a function bundle of the word rank inherits the stylistic implications of the lexeme realising it.

[1] We differentiate between lexicalisation and lexical choice, which is taken to mean the process of deciding which words from the lexicon best describe a concept. Therefore, the lexicalisation process simply attaches the result of an initial coarse lexical choice process (as described in Section 6.2.4) to a function bundle. Nevertheless, *stylistic lexical choice* is different in that it is seen from a final-product point of view. Lexicalising a function bundle does not mean that all the lexical choices will appear in the final surface forms. Some choices will not withstand the stylistic constraints. The surviving lexemes represent the result of the stylistic lexical choice operation.

For example, suppose that lexicon entries have the following form:

$$lexicon(lexeme, property_1, property_2, property_3, ...)$$

where $Property_1$ might stand for the root of the lexeme, $property_2$ for the rhythm of it, and $property_3$ for the length of the lexical item. Now, if we are interested in any particular property, we assert a justification which is, more or less, similar to the following after lexicalising the function bundle Fun:

$$lex(X, Fun) \rightarrow has\_property(Fun, Property_i, Value)$$

where X's entry in the lexicon looks like: $lexicon(X, ..., Value, ...)$. This justification says that the lexeme X realising the function bundle Fun implies that the function bundle itself has $Value$ for the property $Property_i$, which we are interested in. For example, suppose that the function bundle to realise is *Pred* and that the initial lexical choice process has picked two lexemes for this bundle with the following lexical entries:

$$lexicon(destroy, dst, 1010, 1)$$
$$lexicon(annihilate, anhlt, 111100, 1).$$

Now, if we are interested in the rhythm property of lexemes, then *Pred* inherits these properties from the lexemes realising it. This is done by virtue of the following justifications:

$$lex(destroy, Pred) \rightarrow has\_property(Pred, rhythm, 1010)$$
$$lex(annihilate, Pred) \rightarrow has\_property(Pred, rhythm, 111100)$$

7.5.2 Border Meta-Functions

The meta functions $Front/End$ determine the border of a function. Within the realisation triangle of a unit, the order rule $order(Front, Fun_1)$ specifies that Fun_1 is the first constituent of the current unit. Similarly, the order rule $order(Fun_n, End)$ specifies that the Fun_n is the last function. Border meta functions are important in passing the properties of a bordering constituent to its parent. For example, suppose that we are interested in first and last letters of constituents. As shown in figure 7.2, *Deictic* is the first function of *Subject* (denoted by $order(Front, Deictic)$), and *Thing*

is at the end of *Subject* (denoted by $order(Thing, End)$). The lexicalisation process for *Deictic* and *Thing* should have already resulted in the following nodes:

$$has\_property(Deictic, first\_letter, t)$$

$$has\_property(Deictic, last\_letter, e)$$

$$has\_property(Thing, first\_letter, b)$$

$$has\_property(Thing, last\_letter, y)$$

To pass this border information to a higher level, we use these nodes along with meta function order rules as follows:

$$order(Front, Deictic) \wedge has\_property(Deictic, first\_letter, t) \rightarrow$$

$$has\_property(Subject, first\_letter, t)$$

$$order(Thing, End) \wedge has\_property(Thing, last\_letter, y) \rightarrow$$

$$has\_property(Subject, last\_letter, y)$$

This way, not only intra-function stylistic constraints within a constituent are maintained, but also inter-function constraints between the parent and its sisters. Border meta functions also play a role, together with the order rules, in determining what constitutes a complete sequence of functions, as we will see below. Complete sequences help in passing the collective stylistic properties as a single property to the parent constituent.

7.5.3 Order Rules

There are two kinds of order rules: *order* and *partition*. What concerns us here is the immediate adjacency *order* rule as it imposes direct restrictions on word order and hence the final lexical choice. The realisation rule $order(Fun_l, Fun_r)$ specifies that the left function Fun_l is adjacent to the right function Fun_r. The special cases: $Fun_l = Front$ and $Fun_r = End$ were discussed in the previous section.

Immediate adjacency rules are useful for ensuring that the surface stylistic requirements are maintained within a given constituent. For example, if it is required that no two

consecutive words share a particular property (say the root property) then this stylistic requirement can be maintained with the help of the adjacency rule $order(Fun_1, Fun_2)$ as follows:

$$order(Fun_1, Fun_2) \wedge has\_property(end(Fun_1), root, R) \wedge has\_property(front(Fun_2), root, R) \rightarrow \perp$$

7.5.4 Complete Sequences

The border functions and a group of order rules determine what complete sequences there are within any realisation triangle. These have a sequence of the form: $order(Front, Fun_1, Fun_2, ..., Fun_{n-1}, Fun_n, End)$. A complete sequence is important in passing the compositional stylistic properties of its function bundle to their parent. For example, the order rules: *order(Front,Deictic), order(Deictic,Epithet), order(Epithet,Thing), order(Thing,End)* form a complete sequence of say the *Subject.* Therefore the collective properties of the realisation *the ruthless warrior* can now be passed to the *Subject* function bundle. To give a concrete example, suppose that the rhythms of *the*, *ruthless*, and *warrior* are '1', '1010', and '1010' respectively and that the *'the ruthless warrior'* is indeed a complete realisation of the function bundle *Subj.* Therefore, at a higher level, the rhythm of *Subj* as a whole becomes '110101010'.

7.5.5 Constituency Level

A constituent is realised by what we call a realisation triangle. The topmost realisation triangle represents the sentence to be generated. Each of its constituents has its own realisation triangle which in turn might have lower level realisation triangles. For the sake of the stylistic requirement specification, we differentiate between the lower level triangles and the topmost triangle. As we will see later on in Section 7.9, we will be referring to the topmost triangle using level indexing. The topmost level will be indicated by $level = 0$ and any lower level by $level \neq 0$.

7.6 Situation-Action Framework

Since different applications have different stylistic requirements, we provide a framework that allows an application to specify what its requirements are. The ATMS-based generator then ensures that these requirements are satisfied throughout the generation process. The result is a surface form (or a set of surface forms) that meets the application's stylistic needs.

The idea of the situation-action framework is to fire some actions in certain situations. A situation is a combination of different circumstances that may arise at different points during generation. For example, a circumstance might be the existence of an ATMS node that matches a certain pattern or the execution of an NLG operation. The lexicalisation operation, for instance, can represent a situation on its own. One might like to execute some actions in relation to this situation.

An action, in this context, is usually one or more justifications that need to be asserted in response to a particular situation. It can also be the mere introduction of a constraint or definition of a parameter. It is the action part's responsibility to send any sources of surface problems to the false node as soon as they arise (cf. figure 7.1). The intuition behind the approach is summarised in figure 7.3. At different points in the process, we make assumptions and build on them. As soon as we discover the formation of a relevant situation, the necessary action (i.e. assertion of justifications) for that situation is carried out. Note that situation-related nodes are shaded gray in the figure. An action might send some node(s) to the false node or it might propagate a stylistic property to higher levels. Eventually, it is up to the ATMS — with its ability to keep track of causes and effects — to find the surviving utterances that have passed the stylistic scrutiny. Note that — as the figure attempts to show — at a given stage, a whole situation might be formed. Alternatively, that stage might only contribute to the formation of a situation by providing one or more of its circumstances.

Situations at the lexicalisation stage are important as their actions specify the node patterns the user is interested in through the rest of the steps. It is also here that we specify which property of the lexical item we are interested in (i.e. root, rhythm, rhyme, length, ... etc.).

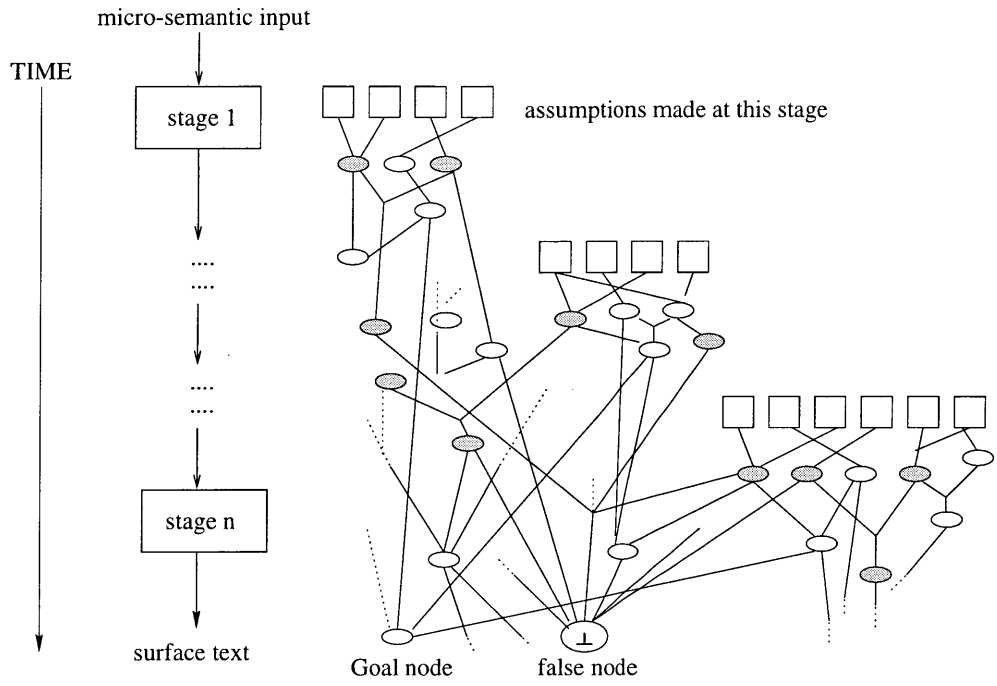


Figure 7.3: The process of promoting and demoting surface stylistic properties

7.7 Situation-Action Specification (SAS)

Here, we formally define what NLG operations there are under the ATMS-based architecture and hence what a situation description can be. We use the term Situation-Action Specification (SAS) to refer to both the situation description part and the action part.

> **Definition:** Generation — in the work of this thesis — means the realisation of a unit or function bundle. The realisation of a unit is considered complete when all its sub-units (or function bundles) have been realised. A function bundle is fully realised by attaching to it a set of lexemes if it is of the word rank. This is known as the *lexicalisation* operation. If a function bundle is not of the word rank then it is realised by attaching to it a realisation triangle. We call this operation *expansion.*

- **Lexicalisation SAS:** A SAS is either of lexicalisation or expansion type. A lexicalisation SAS applies immediately after attaching a lexeme to a function bundle. The situation part of a lexicalisation SAS entry is usually a Prolog

goal which fetches one of the lexeme's properties from the lexicon. The action part of a lexicalisation SAS asserts the implications or properties of the lexeme realising the current function bundle. The action part is one or more ATMS justifications that enable the function bundle to inherit some properties from its realising lexeme. This is what we denoted in Section 7.5.1 by the justification:

$$lex(X, Fun) \rightarrow has\_property(Fun, Property_i, Value)$$

The antecedent $lex(X, Fun)$ is an assumption node representing the lexeme X realising the function bundle Fun. The consequent (represented here by the generic name *has_property*), which is an ATMS derived node, passes a property of the lexeme to the function bundle being lexicalised. Note, however, that the consequent *has_property*(...) can be replaced by any pattern that makes sense under the particular set of surface stylistic requirements (cf. tables in Sections 7.9.1, 7.9.2, and 7.9.3). It is this pattern that will form the basic circumstance — along with order rules — for expansion situations, as we will see next.

- **Expansion SAS:** An expansion SAS applies immediately after attaching a complete realisation triangle to a clause or group rank function bundle. The situation description part of such a SAS entry is a set ATMS derived node patterns (i.e. *order* and *has_property*) and some Prolog goals that must succeed in order for that situation to hold and fire its actions. The action part is justifications that either *promote* or *demote* properties of this level's function bundles. By demoting a property we mean sending the situation circumstances (i.e. conjunction of nodes) to the false node. Alternatively, the action justifications might promote or pass the properties of this level's function bundles to a higher level function bundle (i.e. their parent).

Our convention in representing SAS entries will denote Prolog goals using typewriter-like fonts and ATMS node patterns using italics. Additionally, assumption nodes will be enclosed in rectangles to distinguish them from the rest of ATMS nodes.

7.8 Accommodating the Stylistics-Aware Mode

Here we show how the generation algorithm of the previous chapter (see figure 6.1) can easily accommodate the user specified surface stylistic requirements. The idea is simply to check the applicable situations immediately after the complete realisation of a constituent (i.e. after lexicalisation or expansion operations). The actions of these situations are then carried out according to the stylistic knowledge source specifications. The actions are generally justifications that promote or demote certain stylistic features. Note that a function bundle or constituent is fully realised when all of its immediate constituents are fully realised and that a constituent of word rank is fully realised by an applicable lexeme.

This way, the original algorithm simply checks the situation-action specification upon the completion of constituent realisation; and if there are any applicable situations, then their actions are executed. If no situations are specified in the stylistic knowledge source or if there is not an applicable situation, then the algorithm carries on as it would normally do in the plain mode of generation.

7.9 Examples of SSCs

Having presented our idea of situation-action framework for the specification of stylistic requirements, we now give examples of surface requirements that we might want the generator to satisfy. The purpose of these examples is two fold – first they show how different surface stylistic requirements can be formulated using that framework, and second, they demonstrate the flexibility of the ATMS-based architecture in accommodating some SSC independently of the micro-semantic input or the lexico-grammatical resource.

In the first example (Section 7.9.1), we formulate the requirement that an utterance may not have similar words close to each other, so that we can have interesting utter-

ances with diverse lexical choices. In the second example (Section 7.9.2), we have stricter surface requirements. The generated sentences should follow a prescribed rhythmic sequence such as in the case of poetry metres. In the last example (Section 7.9.3), we formulate a rather more realistic stylistic requirement. It is that of imposing a size limit on the generated sentences. As discussed earlier, we are not suggesting that these are the ultimate stylistic requirements one could aim for; they are only intended to demonstrate the ability of the framework to express stylistic requirements and the flexibility of the ATMS-based architecture in accommodating these requirements.

To proceed with the examples below suppose that an entry in the lexicon looks like this:

```
lexicon(lexeme, property1, property2, property3, ...)
```

Assume that $property_1 = root$, $property_2 = rhythm$, and $property_3 = i$ where i is the length of the lexical item. Note that an entry like 'United Kingdom', for example, is of length 2 although it is represented by one lexical entry in the lexicon.

7.9.1 Word Adjacency Constraints

Two words are collocations if they have a tendency to occur next (or close) to each other in texts. According to [Matthews 97], a collocational constraint is any restriction on the collocability[2] of a word with another. However, if the existence of a word prohibits or minimises the chances of another word then these two words are anti-collocations. For example, although *strong* and *powerful* are synonyms, *strong* and *tea* are collocations in 'strong tea' whereas *powerful* and *tea* are anti-collocations in 'powerful tea'.

Our word adjacency stylistic requirement is a kind of collocational constraint. We assume that each word has a property of some kind (e.g. root of the word) and

[2] This term is used in [Matthews 97] and we take it to mean the ability of a word to collocate.

that two adjacent words having the same (or similar) roots are to be avoided in an utterance. This would make sense if we were generating Arabic sentences since in Arabic — as is the case with other Semitic languages — most words are generated from a limited number of roots [Al-Jabri 97]. Although this is a simple requirement, current systemic generators cannot test for this until after the surface form has been generated because the realisation rules are only collected during system traversal and it is not until the very end that they are executed. Only then can we tell if we have violated the collocational constraints. Next, we show how these requirements are specified within the situation-action framework and we give some examples of generated sentences.

i. Situation-Action Specification

The following table represents the SASs for the word adjacency requirements. It has two types of entries: lexicalisation and expansion entries. In the table, **lexicalisation(X,Fun)** means a function bundle *Fun* being lexicalised by the lexeme *X*; and **expansion(P)** means the expansion of the function bundle *P*.

The **lexicalisation** SAS entries are consulted immediately after lexicalising a word rank function bundle. They define the pattern that we would like to introduce (i.e. *has_property* type of node) so that it can be used by the situation description parts of the **expansion** SAS entries. The **expansion** situations, consist of a collection of ATMS node pattern and Prolog goals. These Prolog goals are utility functions (e.g. `parent(...)`, `submetre(...)`, `children(...)`, etc). The job of these utilities is to perform a certain task or check that a certain condition holds.

Note that ATMS nodes cannot have variables in them as shown in the table below. In fact, these situations and actions contain patterns of ATMS nodes. During generation time, all variables in these patterns will be instantiated and the whole situation description will be referring to particular ATMS nodes (i.e. either derived or assumption nodes).

	SITUATION		ACTION
1	**lexicalisation(X,Fun):**		
	`lexicon(X,R,_,_,...)`	$\Longrightarrow$	$\boxed{lex(X, Fun)} \rightarrow root(front(Fun), R)$
			$\boxed{lex(X, Fun)} \rightarrow root(end(Fun), R)$
2	**expansion(P):**		
	$order(Fun_l, Fun_r)$,		$order(Fun_l, Fun_r) \wedge$
	$root(end(Fun_l), R)$,	$\Longrightarrow$	$root(end(Fun_l), R) \wedge$
	$root(front(Fun_r), R)$,		$root(front(Fun_r), R) \rightarrow \perp$
	`children([Funl,Funr], P)`		
3	**expansion(P):**		
	$order(Front, Fun)$,		
	$root(front(Fun), R)$,		$order(Front, Fun) \wedge$
	`parent(Fun, P)`,	$\Longrightarrow$	$root(front(Fun), R) \rightarrow$
	`Level`$\neq$ `0`		$root(front(P), R)$
4	**expansion(P):**		
	$order(Fun, End)$,		
	$root(end(Fun), R)$,		$order(Fun, End) \wedge$
	`parent(Fun, P)`,	$\Longrightarrow$	$root(end(Fun), R) \rightarrow$
	`Level`$\neq$`0`		$root(end(P), R)$

As shown in the table above, the situation part of entry 1 applies upon lexicalising a function bundle Fun with a lexeme X. The situation holds when the Prolog goal succeeds in fetching the lexeme's root property from the lexicon. The action to be taken in this case is to assert two justifications passing the root property of the lexeme to the function bundle it is realising. The justifications simply state that the root of the *front* and the *end* of a unit being lexicalised inherits the root of the lexeme. The rest of the table SAS entries are of expansion type. Situation 2 captures any consecutive lexemes, within the expansion triangle of a function bundle P, sharing the same root and sends them to the false node. This is a surface problem that this specification is trying to avoid in this generation task. Situation 3 and 4 have to do with passing the root information from one level to another; each working on either side of a function bundle.

ii. Generation Examples

To appreciate the stylistics-aware mode of the generator, we show below the sentences that are generated when the above constraints are applied to the generation example

shown in appendix B. For the sake of this example, we added artificial root properties in the lexicon for testing purposes (see appendix A, Section A.3). Two consecutive words share the same property if the second letters are both vowels or both non-vowels. From the possible 9450 sentences, only those that do not have adjacent words sharing the same property are generated. Examples of such sentences are:

```
the first space warrior annihilated jitrax.
the merciless space warrior annihilated jitrax.
the ruthless space warrior annihilated jitrax.
dartmaul annihilated jitrax.
the warrior annihilated jitrax.
```

In another run, we relaxed the conditions to see what other sentences could be generated. We made the exception that the root of *was* is neither vowel nor non-vowel; it falls into a third category. The following sentences were generated with new lexical and syntactic combinations.

```
jitrax was destroyed.
the submarine was destroyed.
jitrax was annihilated.
the submarine was annihilated.
jitrax was annihilated by the warrior.
jitrax was annihilated by the ruthless space warrior.
jitrax was annihilated by the merciless space warrior.
jitrax was annihilated by the first space warrior.
the submarine was annihilated by the warrior.
the submarine was annihilated by the ruthless space warrior.
the submarine was annihilated by the merciless space warrior.
the submarine was annihilated by the first space warrior.
the first space warrior annihilated jitrax.
```

```
the merciless space warrior annihilated jitrax.

the ruthless space warrior annihilated jitrax.

dartmaul annihilated jitrax.

the warrior annihilated jitrax.
```

7.9.2 Poetry Metre Constraints

The regular occurrence of certain properties such as metre and rhyme is one main feature of classic poetry. These are obviously surface constraints that impose restrictions on syntactic structures and lexical choices. [Manurung *et al.* 00] use strings of `w`'s and `s`'s to represent syllables with weak and strong stresses respectively. For example, a limerick can be represented as:

```
w,s,w,w,s,w,w,s(a)
w,s,w,w,s,w,w,s(a)
w,s,w,w,s(b)
w,s,w,w,s(b)
w,s,w,w,s,w,w,s(a)
```

The `(a)`'s and `(b)`'s at the end of each line represent the rhyme scheme. We use a similar approach (i.e. a string of `0`'s and `1`'s) to represent the rhythmic sequence of a surface form. Note that our aim here is only to show how these poetic constraints are specified in our situation-action framework. First, we give the situation-action specifications and then some examples of generated sentences that respect these constraints.

i. Situation-Action Specification

The following table represents the SASs for the poetry metre requirements. The **lexicalisation** and **expansion** entries are also shown.

	SITUATION		ACTION
1	**lexicalisation(X,Fun):**		
	`lexicon(X,_,Sound,_,...)`	$\Longrightarrow$	$\boxed{lex(X, Fun)} \rightarrow$ $rhythm(Fun, Sound)$
2	**expansion(P):**		
	$order(Fun_x, Fun_y)$, $rhythm(Fun_x, S_x)$, $rhythm(Fun_y, S_y)$, `¬ submetre([Sx,Sy], Metre),` `children([Funx,Funy], P)`	$\Longrightarrow$	$order(Fun_x, Fun_y) \wedge$ $rhythm(Fun_x, S_x) \wedge$ $rhythm(Fun_y, S_y) \rightarrow \perp$
3	**expansion(P):**		
	$order(Order)$, `Order=[Funx,...,Funy],` `complete_order(Order),` $rhythm(Fun_x, S_x), ...,$ $rhythm(Fun_y, S_y)$, `submetre([Sx,...,Sy], Metre),` `children([Funx,...,Funy], P),` `Level` $\neq$ `0`	$\Longrightarrow$	$order(Order) \wedge$ $rhythm(Fun_x, S_x) \wedge$ $... \wedge$ $rhythm(Fun_y, S_y) \rightarrow$ $rhythm(P, S_x \cdot ... \cdot S_y)$
4	**expansion(P):**		
	$order(Order)$, `Order=[Funx,...,Funy],` `complete_order(Order),` $rhythm(Fun_x, S_x), ...,$ $rhythm(Fun_y, S_y)$, `¬ submetre([Sx,...,Sy], Metre),` `children([Funx,...,Funy], P)`	$\Longrightarrow$	$order(Order) \wedge$ $rhythm(Fun_x, S_x) \wedge$ $... \wedge$ $rhythm(Fun_y, S_y) \rightarrow \perp$

Situation 1 specifies what action is necessary upon lexicalising a function bundle Fun with a lexeme X. The action simply states that a function bundle realised by a lexeme inherits the lexeme's rhythmic sequence. Note that a lexeme with a rhythm that cannot occur anywhere in the metre is not considered among the lexical choices in the first place. Situation 2 starts capturing adjacent function bundle rhythms. If any adjacent function bundles, at any level, form a rhythmic sequence that is not a substring of the metre, it is sent to the false node. Situation 3 passes the concatenated rhythmic arrangement of a complete sequence to the parent of that sequence, if it is still a possible sub-metre. Situation 4 sends any complete rhythmic arrangement that is not a substring of the metre to false.

ii. Generation Examples

We show below the sentences that are generated when the above poetry metre constraints are applied to the generation example shown in appendix B. The exact metre rhythm is: '1001001111010110110010010100' which is also shown in the Prolog specification of the above situations in appendix C.2. From the possible 9450 sentences only one sentence was found to follow the target metre which is shown below:

```
dartmaul annihilated the submarine on friday.
```

7.9.3 Text Size Constraints

Some applications have size constraints on the generated text. For some applications, the overall generated text must fit within certain limits without regard to the length of individual sentences. Other applications might have restrictions on the length of individual sentences for whatever reason (e.g. readability/comprehension). They might additionally impose length constraints on the level of embedding or the length of certain constituents such as noun phrases for the same reasons [Lin 96]. But the size constraint remains simple: the generated span of text should be of size (or length) of no more than a given number. We show next how this restriction can be specified using the situation-action framework. We then give examples of generated sentences that comply with the size constraint. In these examples, we measure the size of a sentence by the number of words it contains.

i. Situation-Action Specification

The following table represents the SASs for the sentence size requirements. The **lexicalisation** and **expansion** entries are also shown.

	SITUATION		ACTION
1	**lexicalisation(X,Fun):**		
	`lexicon(X,_,_,Length,...)`	$\Longrightarrow$	$\boxed{lex(X, Fun)} \rightarrow$ $size(Fun, Length)$
2	**expansion(P):**		
	$order(Fun_x, Fun_y)$, $size(Fun_x, L_x)$, $size(Fun_y, L_y)$, `Lx+Ly > MaxSize,` `children([Funx,Funy], P)`	$\Longrightarrow$	$order(Fun_x, Fun_y) \wedge$ $size(Fun_x, L_x) \wedge$ $size(Fun_y, L_y) \rightarrow \perp$
3	**expansion(P):**		
	$order(Order)$, `Order=[Funx,...,Funy],` `complete_order(Order),` $size(Fun_x, L_x), ...,$ $size(Fun_y, L_y)$, `sum([Lx,...,Ly], Sum)` $\leq$ `MaxSize` `children([Funx,...,Funy], P),` `Level` $\neq$ `0`	$\Longrightarrow$	$order(Order) \wedge$ $size(Fun_x, L_x) \wedge$ $... \wedge$ $size(Fun_y, L_y) \rightarrow$ $size(P, Sum)$
4	**expansion(P):**		
	$order(Order)$, `Order=[Funx,...,Funy],` `complete_order(Order),` $size(Fun_x, L_x), ...,$ $size(Fun_y, L_y)$, `sum([Lx,...,Ly],Sum)> MaxSize,` `children([Funx,...,Funy], P)`	$\Longrightarrow$	$order(Order) \wedge$ $size(Fun_x, L_x) \wedge$ $... \wedge$ $size(Fun_y, L_y) \rightarrow \perp$

Situation 1 asserts the length implication of a lexeme X which realises a function bundle Fun. A lexeme of size $Length$ realising a function bundle Fun implies that the size of that function bundle is $Length$ also. Note that we assume here that the length of a lexical item is not necessarily one (e.g. the proper noun 'United Kingdom' is of length 2). Situation 2 sends any two function bundles with a total size exceeding the limit to the false node. Situation 3 passes the size of a complete sequence that is still within the size constraint to the parent of that sequence. Situation 4 takes care of any sequences of function bundles that exceed the limit. It sends that combination of circumstances to the false node.

ii. Generation Examples

We show below the sentences that are generated when the above size constraints are applied to the generation example shown in appendix B. The size limit is set to four, as shown in the Prolog specification of the above situations in appendix C.3. From the possible 9450 sentences, only those sentences that are within the size constraints are generated. These are:

```
jitrax was destroyed.
the submarine was destroyed.
jitrax was annihilated.
the submarine was annihilated.
dartmaul destroyed jitrax.
dartmaul annihilated jitrax.
the warrior destroyed jitrax.
the warrior annihilated jitrax.
dartmaul destroyed the submarine.
dartmaul annihilated the submarine.
```

7.10 How the Situation-Action Approach Works

Having presented our situation-action framework for the specification of SASs along with three example sets of SSC, we now show how the approach works hand in hand with the usual ATMS nodes representing systems, features, function bundles and ordering restrictions. Here, we give a detailed example that explains how the system works in the stylistics-aware mode. It also shows when each type of SAS entries is activated and what effects their actions have on the overall generation process.

To proceed with the example, suppose that the SSC we are considering are those of collocational constraints between adjacent words (as given in the table of Section 7.9.1). Also, assume that the lexicon entries for the words we will use are:

```
lexicon(the,t,...)
lexicon(big,b,...)
lexicon(fat,a,...)
lexicon(boy,b,...)
lexicon(lad,a,...)
lexicon(had,a,...)
lexicon(lunch,l,...)
```

where the root category of a lexeme is represented by a single letter (e.g. `t, a, b,`

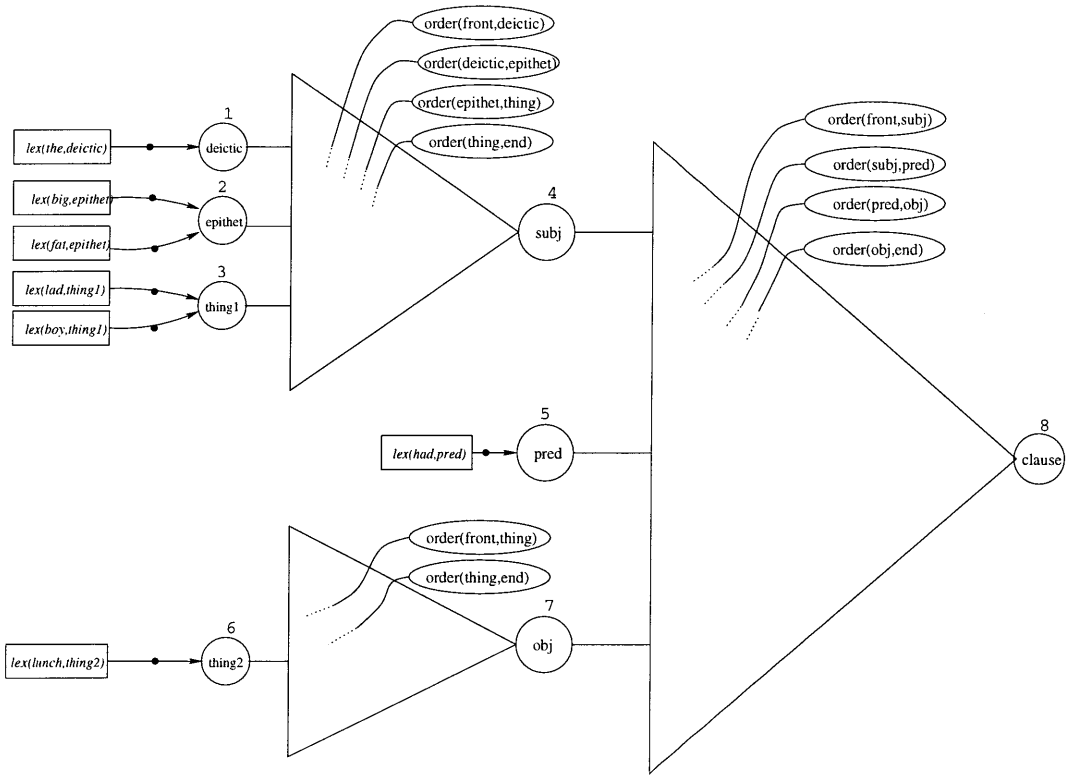


Figure 7.4: Plain-mode interfacing of realisation triangles

`1, ...`). Assume further that if no stylistic constraints were considered then the plain mode generation process would result in the realisation triangles shown in figure 7.4.

To keep the diagrams of the figure simple, we do not show the features and how they are connected to other nodes within a realisation triangle. We show, however, the order rules of each triangle as they play an important role in combination with the SAS specifications. The function bundle nodes are numbered to depict the order in which each bundle is considered fully realised. As discussed earlier, a fully realised function bundle activates applicable SAS entries: either of lexicalisation type if its of the word rank or else of expansion type.

In the following, we show the different steps of realising each function bundle and what situation applies and what actions take place.

1. **Deictic Realisation:** The *deictic* function bundle is of the word rank. It is considered fully realised by attaching to it the applicable lexemes. This is done

via the following justification:

$$lex(the, deictic) \rightarrow deictic$$

At this point, SAS entry 1 (i.e. **lexicalisation(the,deictic)**) becomes applicable resulting in the assertion of the following justifications:

$$lex(the, deictic) \rightarrow root(front(deictic), t)$$
$$lex(the, deictic) \rightarrow root(end(deictic), t)$$

2. **Epithet Realisation:** The *epithet* function bundle is of the word rank also. It is considered fully realised by attaching to it the applicable lexemes. This is done via the following justification:

$$lex(big, epithet) \rightarrow epithet$$

At this point, SAS entry 1 (i.e. **lexicalisation(big,epithet)**) becomes applicable resulting in the assertion of the following justifications:

$$lex(big, epithet) \rightarrow root(front(epithet), b)$$
$$lex(big, epithet) \rightarrow root(end(epithet), b)$$

The second lexeme attachment results in the following sequence of justifications:

$$lex(fat, epithet) \rightarrow epithet$$
$$lex(fat, epithet) \rightarrow root(front(epithet), a)$$
$$lex(fat, epithet) \rightarrow root(end(epithet), a)$$

3. **Thing1 Realisation:** The subject's *thing* function bundle is of the word rank. It is realised by attaching to it some lexemes. For the first applicable lexeme *lad*, we get the following sequence of justifications:

$$lex(lad, thing1) \rightarrow thing1$$
$$lex(lad, thing1) \rightarrow root(front(thing1), a)$$
$$lex(lad, thing1) \rightarrow root(end(thing1), a)$$

For the second applicable lexeme *boy*, the following justification assertions take place:

$$lex(boy, thing1) \rightarrow thing1$$
$$lex(boy, thing1) \rightarrow root(front(thing1), b)$$
$$lex(boy, thing1) \rightarrow root(end(thing1), b)$$

4. **Subj Realisation:** Having realised its sub-constituents, the function bundle *subj* is now considered fully realised (or expanded). Since *subj* is of the group rank, the **expansion(subj)** SAS entries are checked for the applicable situations. When a situation applies, its corresponding actions are carried out. Entry 2 applies and its action part asserts the following justification:

$$order(epithet, thing1) \wedge root(end(epithet), a) \wedge root(front(thing1), a) \rightarrow \perp$$

The effect of this implication is to prohibit the words *fat* and *lad* to be adjacent to each other. Entry 2 also applies for another set of instantiations. Its action part asserts the following justification:

$$order(epithet, thing1) \wedge root(end(epithet), b) \wedge root(front(thing1), b) \rightarrow \perp$$

This justification's effect is to ban *big* and *boy* to occur next to each other in the surface form.

The situation of entry 3 applies asserting the justification:

$$order(front, deictic) \wedge root(front(deictic), t) \rightarrow root(front(subj), t)$$

which makes the function bundle *subj* inherit the root t as the root of its front.

Entry 4 applies for two different instantiations asserting the following justifications:

$$order(thing1) \wedge root(end(thing1), a) \rightarrow root(end(subj), a)$$
$$order(thing1) \wedge root(end(thing1), b) \rightarrow root(end(subj), b)$$

5. **Pred Realisation:** The function bundle *pred* is of the word rank. So, only the lexicalisation SAS entries are consulted at this point to see if they apply. This

step results in the assertion of the following justifications:

$$lex(had, pred) \rightarrow pred$$
$$lex(had, pred) \rightarrow root(front(pred), a)$$
$$lex(had, pred) \rightarrow root(end(pred), a)$$

6. **Thing2 Realisation:** *thing2* is of the word rank. The SAS lexicalisation entry results in the following justifications being asserted:

$$lex(lunch, thing2) \rightarrow thing2$$
$$lex(lunch, thing2) \rightarrow root(front(thing2), l)$$
$$lex(lunch, thing2) \rightarrow root(end(thing2), l)$$

7. **Obj Realisation:** The function bundle *obj* is of the group rank. This signals the completion an expansion operation. Therefore, the **expansion(obj)** SAS entries are checked for the applicable situations. Only SAS entries 3 and 4 apply here resulting in the following instantiations:

$$order(front, thing2) \wedge root(front(thing2), l) \rightarrow root(front(obj), l)$$
$$order(thing2, end) \wedge root(end(thing2), l) \rightarrow root(end(obj), l)$$

8. **Clause Realisation:** The function bundle *clause* is of the clause rank. This signals the completion of an expansion operation. Note, however, that the level of this unit is the top most (indicated by level=0). Therefore, SAS entries 3 and 4 do not apply. The situation of entry 2 holds resulting in the instantiations shown by following justification:

$$order(subj, pred) \wedge root(end(subj), a) \wedge root(front(pred), a) \rightarrow \perp$$

The effect of this justification is to ban from the surface form any *subj* constituent ending with the root *a* (e.g. *lad*) to occur immediately before a *pred* constituent starting with the same root (e.g. *had*).

Figure 7.5 shows the interfaced realisation triangles along with the surface stylistic constraints imposed by the assertion of SAS's action justifications. As a result, the ATMS-based generator only produces the surface form:

```
the fat boy had lunch
```

The other three possible surface forms:

```
the big*boy had lunch
the big lad*had lunch
the fat*lad*had lunch
```

would not survive the stylistic constraints. Each (sub)realisation would have been abandoned at some stage during the generation process. The asterisks indicate the places where two consecutive words share the same root according to our lexicon.

7.11 Summary and Outlook

In this chapter, we have presented our idea of a situation-action framework in which the user can specify his own stylistic requirements. Also, we have demonstrated how various sets of surface stylistic requirements can be specified using the situation-action framework. This shows that the ATMS-based generation architecture introduced in the previous chapter can be used easily in a stylistics-aware mode. In the next chapter, we discuss the implementation details of our prototype generation system that embodies our ideas discussed in this thesis. We also evaluate the two main stages of the system implementation: the translation and generation stages.

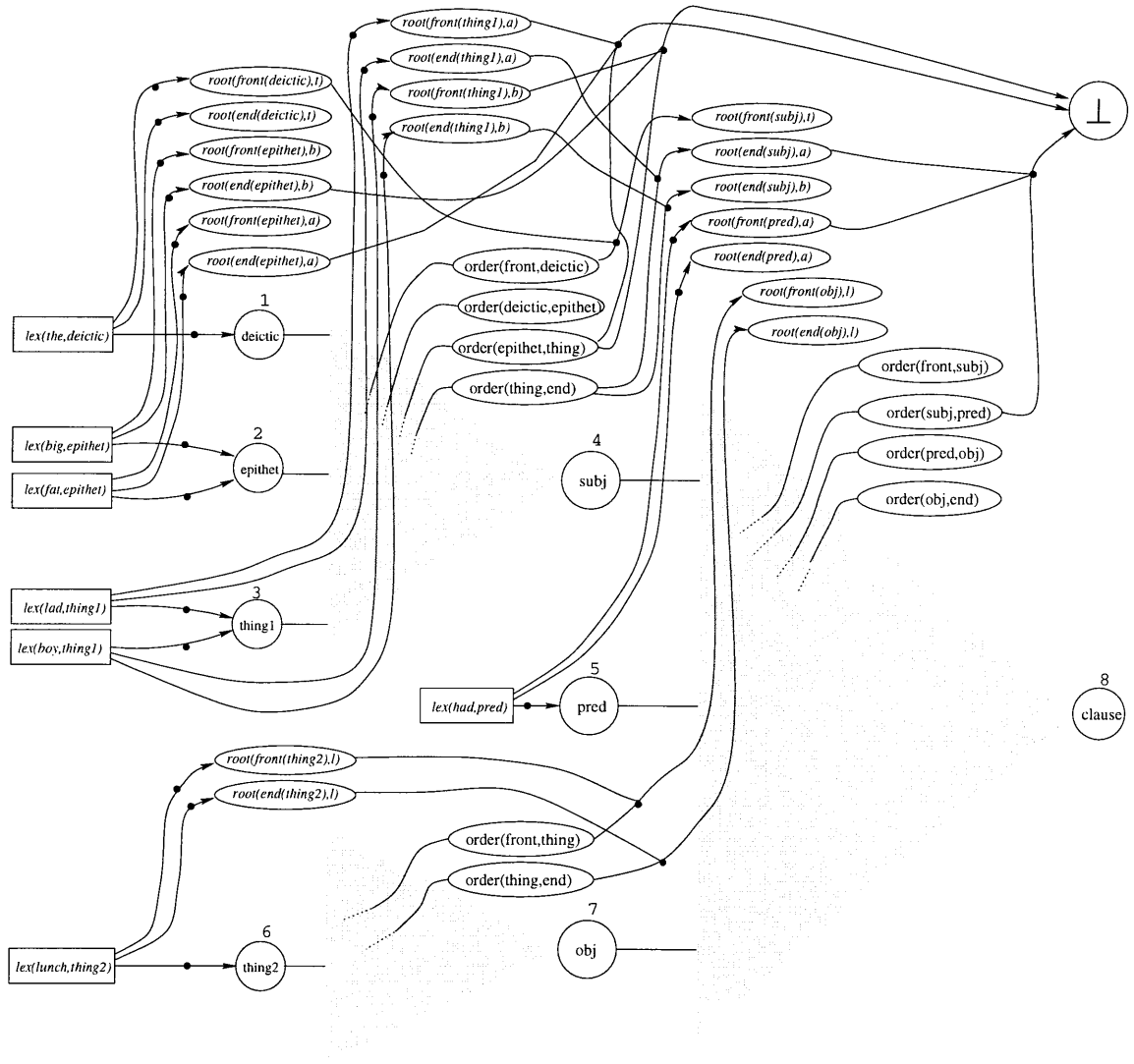


Figure 7.5: Interfacing of realisation triangles in the stylistics-aware mode

Chapter 8

System Implementation and Evaluation

In this chapter we present the implementation details of the prototype generation system that demonstrates the ideas introduced in the previous three chapters. Also, we evaluate the system's two main phases: translation and generation. We evaluate the translation stage in terms of the grammar size and the outcome of the translation process. Then, different modes of generation are compared and the effect of the particular set of stylistic constraints on the performance of the generator is discussed. The chapter ends with a discussion of the system's merits and demerits.

8.1 Implementation Notes

The schematic of figure 8.1 which was also shown in Chapter 6 provides an overview of the ATMS-based generation system. A prototype generator embodying the different modules and knowledge sources shown in the schematic was implemented in SICStus Prolog. From an input-output point of view, the system takes a conceptual representation and generates natural language sentences that have certain surface stylistic requirements. In Chapter 6 and 7, we already discussed how the new generation architecture works and we presented the nature of the knowledge sources involved in the process. In the following, we give the implementation details to bridge the gap between the architectural modularisation and the actual implementation.

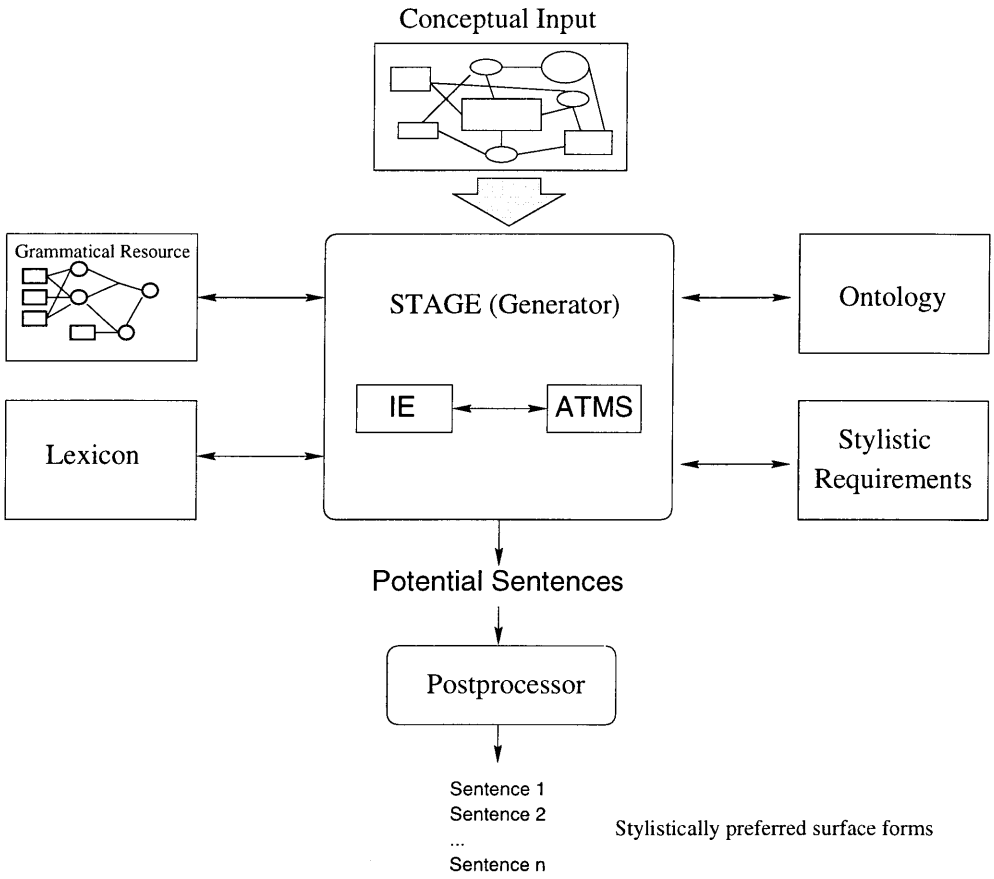


Figure 8.1: An overview of the ATMS-based NLG system

8.1.1 SNAC Input/Output

The **S**ystem **N**etwork to **A**TMS **C**onverter (SNAC) takes systemic grammar networks and transforms them into network snapshots which form the basis for ATMS dependency networks (see Section 6.5.1 for examples of system networks and network snapshots). The systemic grammar we use is WAG's dialog grammar. This is the complete grammar resource of the WAG system. In Section 8.2.1, below, when we evaluate the translation phase, we discuss the size of the grammar and the outcome of the translation process. WAG is written in Lisp and so are its linguistic resources. Because our system is implemented in Prolog, we have rewritten the WAG's grammar in a Prology representation. For example, the following Prolog construct represents the GENDER system which we call here `genderSYS`.

```
system_def( entry_cond(third & singular),genderSYS,
            [[feminine,[]],
             [masculine,[]],
             [neuter,[]]
            ]).
```

It has the entry condition *third* $\wedge$ *singular*. The system has three features `feminine`, `masculine` and `neuter`. Each of these features may have attached to it a list of realisation statements which happen to be empty. Examples of realisation statements can be found in the grammar fragments given in Appendix A. The realisation statements are represented as a list of single realisation operations. The realisation operations we consider in our grammar are those found in WAG's: *insert, preselect, lexify, order* and *partition*. As an example of realisation lists, the *clause* feature may have the following realisation statements:

```
[ [insert,subject],
  [insert,finite],
  [insert, pred],
  [insert, punct],
  [preselect, subject,nominal_group],
  [preselect, subject,nominative],
  [preselect, pred, lexverb],
  [lexify, punct, period],
  [order, front, subject],
  [order, subject, finite],
  [order, punct, end],
  [partition, pred, punct] ]
```

SNAC produces a network snapshot for each system network. Network snapshots were discussed in Section 6.5.1 and the exact format of a snapshot is given in Appendix A.1.

8.1.2 STAGE Input/Output

The generation component of the system takes as an input a micro-semantic representation such as the one shown in Appendix B.1 and generates surface forms that follow the stylistic constraints specified in the situation-action repository. STAGE depends heavily on the ATMS component in generating the required utterances. It only decides what bits of the problem are to be communicated to the ATMS. The computation of the surviving selection expressions and lexical choices are delegated for the ATMS to

handle. In the end, STAGE reads off the label of goal node that represents the top-most clause unit and passes it to a post-processor which prints out the actual sentences. Next, we present the ATMS implementation we use in our generation system.

8.1.3 The ATMS Component

STAGE uses an off-the-shelf ATMS implementation that is not biased towards NLG applications. The ATMS implementation[1] we used in our system is written in Prolog. We did not intend to extend or modify the ATMS in any way. Rather, the objective was to show how an existing neutral ATMS implementation can be used as a basis for a new NLG architecture that is sensitive to SSC.

The ATMS component is a typical implementation providing the following services:

- **Node Creation:** Assumption, premise and derived nodes can be created using `newassump(Node)`, `newfact(Node)` and `newderived(Node)` respectively.
- **Assertion of justifications:** This operation is made possible via the predicate `newjust(LHS,RHS)` where `LHS` is a list of antecedents and `RHS` is a consequent node. Contradiction justifications can be formulated either by replacing `RHS` with `false` in `newjust(LHS,RHS)` or directly by `newcontradiction(LHS)`.
- **Explanations:** An explanation for a given node (i.e. the label of that node) can be extracted through `explain(Node,Exp)`, or printed out via `prlabel(Node)`.
- **Queries:** The user can check whether a node (of any type) exists by using `node(X)`. Alternatively, he can check a specific node type such as `assump(Node)`, `fact(Node)` or `derived(Node)`.

Because the ATMS will spend most of the time performing set operations, the specific ATMS we are using represents sets as bit vectors to improve on the set operations.

[1] Chris Mellish kindly provided the ATMS implementation I use in the prototype generator.

8.2 System Evaluation

Before getting into the details of the empirical evaluation of our implementation, it is worth mentioning, at the beginning, that the implementation embodies a new generation architecture that succeeds in satisfying the main objective set forth at the start of this thesis. The objective was to account for the effect of surface stylistic constraints on earlier linguistic decisions such as syntactic and lexical choice in one unified generation architecture. We showed in the previous chapter how SSC can seamlessly be incorporated into the generation process and how they play a role in deciding what the final lexical and syntactic choices are. The ability to follow more than one option through the system network and to fold in constraints that go straight across the structures produced is in itself an interesting research tool, especially for some applications where efficiency is not such an issue. Although the main objective has been achieved, we must nevertheless still evaluate the main stages of our implementation: compilation and generation. We do this so that we get a clearer idea about the efficiency of the system and how different factors influence the behaviour of the system. This being clarified, the implemented generation system can be viewed as a general test-bed for experimenting with different sets of stylistic preferences. For example, it can be used to experiment with the automatic generation of controlled languages, something that we have not investigated in this thesis because it is beyond the scope of this work.

8.2.1 Translation Phase

The solution path we followed in this project required that systemic grammar networks be transformed into dependency networks; a representation that the ATMS can reason with. The implementation of an automatic procedure for the translation of general systemic grammars into the ATMS representation has been carried out. SNAC takes advantage of the fact that the re-translation of a system network yields exactly the same ATMS representation. Consequently, it pre-compiles the system networks once and saves snapshots of them. At generation time, the semantic input is used to pick only those parts of the network that are relevant. The generator cuts away those superfluous branches along with the corresponding function bundles based on the semantic input.

SNAC was tested on an existing systemic grammar (WAG's complete grammatical resources, called the Dialog grammar) that was written without any ATMS dependency networks in mind. However, these grammars are written in Lisp and we had to put them, without change, into our Prolog format of system networks (see Section 8.1.1). The Dialog grammar has 158 systems and 340 features. This grammar is not as comprehensive as the NIGEL grammar (which has around 500 systems and 1500 features), but it is more comprehensive than the difference in systems and features suggests[2]. Appendix A shows all the systems of the WAG grammar on which SNAC was tested. As is clear from the appendix, some of the syntactic structures that can be generated by this grammar include:

1. active and passive clauses
2. infinitive and -ing forms
3. declarative, interrogative, and imperative
4. transitive, intransitive, modal and other auxiliary verbs
5. positive and negative clauses
6. wh- constructs such as wh-subject, wh-object, wh-circumstances
7. different verb tenses
8. thematisation of circumstances
9. primary and secondary circumstance constructs
10. different if-complex constructs
11. prepositional phrases
12. adjectival groups
13. nominalisations, etc.

[2] According to personal communication with the writer of the WAG grammar, Mick O'Donnell: the NIGEL grammar makes a lot of distinctions which do not necessarily reflect syntactic differences but rather lexical differences; WAG systems cover only grammatical distinctions.

The WAG clause network alone has 75 systems and 170 features. Of the 75 systems, 32 have complex entry conditions. SNAC gives a conjunction-free[3] network of 138 systems. The number of features stays exactly the same as assumptions are not allowed to duplicate in the ATMS. The more AND gates there are in a network, the more variant systems we have to create. Also, as the number of conjuncts in a single gate increases, the number of variant systems increases accordingly.

The WAG group network is smaller than the clause network having 36 systems and 76 features. Of these 36 systems, 12 have complex entry conditions. SNAC has produced a network of 48 systems that is free of conjunctive gates. The translation task is quite fast as there is no need to compute the DNF formula for the whole network.

The number N of systems involving complex conjunctive entry conditions and the average number of conjuncts to these gated systems give a good estimate of the size of the resultant dependency network. For the clause network, the average number Avg of conjuncts is 3.125 and for the group network it is 2.33. The additional variant systems we need to create will be $Avg - 1$ per gate; making the expected size of the translation of a network of size S equal to $S + ((Avg - 1) \times N)$. According to this estimate, the clause network can be expected to be of size $75 + (2.125 \times 32) = 143$ and the group network of size $36 + (1.33 \times 12) = 52$.

The remaining 47 systems and 94 features belong to the network of the word rank. However, SNAC does not make use of these word networks since it realises function bundles of the word rank by attaching to them words that are directly fetched from the lexicon. These word rank networks are relevant for the sentence analysis side of the WAG system as it uses the same grammar for sentence generation and analysis.

8.2.2 Generation Phase: Empirical Results

The idea of using the ATMS for processing systemic grammars is new and we need to know what factors affect the performance of the system. At first, we can think of many factors that may influence generation time, these might be:

[3] By conjunction-free networks we mean networks without explicit AND gates. See Chapter 5 for a discussion of why we need to have such networks, and how the translation algorithm accounts for their effect in the ATMS dependency networks.

- the degree of under-specificity of the input.
- the number of open paths incurred by the relaxed choosers.
- the generosity of the lexicalisation process, or the number of available/allowed synonyms for a give concept.
- the size of the underlying networks.
- the size and complexity of the string to be generated.

Of course some of these factors might be related to others in some way. For example, an under-specified semantic input might decrease the choosers appetite to select particular features resulting in many open paths to be explored; hence expanding the search space.

In the following, we take a closer look at these factors from an ATMS standpoint since what really affects the ATMS performance is the number of assumptions. Therefore, we will revisit these factors and discuss how they influence the creation of assumption nodes. Next, we analyse the plain mode of the generation process and then we turn our attention to the other mode (i.e. stylistics-aware generation) to study how the system performs under different sets of surface stylistic requirements. In the following discussion, we will presume that the semantic input is the same as that of the complete example of Appendix B.

Plain Generation Mode

Although the ATMS-based generation system can deterministically generate a single sentence for a given semantic input just like classical systemic generators, the way it usually works is to have its choosers keep many alternative paths active for further discovery, resulting in many paraphrases for the same input. Table 8.1 shows the cpu time needed by the ATMS-based generator to deterministically generate one sentence[4] on the one hand and nondeterministically many sentences (9450 for the particular example we tried) on the other hand.

[4] The time shown in the table is the average for generating sentences of different lengths and structures. The sentences were selected in such a way as to give a range of generation times. Six syntactic structures at the clause level that give rise to sentences ranging from lengths 3 to 8 were selected pretty much like those shown in Table 8.5.

	Deterministic	Nondeterministic
Number of Sentences	1	9450
Generation Time	0.770 secs	125.16 mins
Time per Sentence	0.770 secs	0.794 secs

Table 8.1: Deterministic and nondeterministic runs of the the ATMS-based system

A relevant question at this point is: how does the deterministic behaviour of the ATMS-based generator compare to traditional systemic generators? To answer this question, we designed an experiment where the ATMS component was stripped from the generator. The generator still works in the same environment (i.e. SicStus Prolog). Also, it still consults the same lexicon and grammatical networks. Like classical systemic generators, it ends up with one selection expression and produces one surface form. The aim is to see how much overhead the ATMS is putting on our generation system. Table 8.2 compares the generation time for our generator if it worked deterministically with another version of it where it was stripped from the ATMS component. The table shows that the ATMS is putting around 45% overhead on the system. The times shown in the table are the average for generating the same set of sentences. Next, we focus on the ATMS-based generation architecture. We investigate the factors responsible for the system performance and the extent to which they are responsible.

Deterministic ATMS-based	Classical Systemic
0.770 sec.	0.422 sec.

Table 8.2: Generation times for ATMS-based and traditional generators

The ATMS plays a central role in our generation architecture and its internal operations can have great influence on the overall performance of the generation system. The question then becomes: what does affect the performance of a given ATMS implementation? The obvious answer is the number of assumptions, as they determine the size of the search space. Next, we discuss the previous first-impression factors with respect to the number of assumptions they give rise to.

From a solution-construction viewpoint, the number of assumptions in a solution triangle depends on the number of features and function bundles which have survived the tailoring process. Although a function bundle is not directly represented by an

assumption, it is either realised by lexical items (and hence assumptions) if it is of the word rank, or else by a fresh realisation triangle which incurs more assumptions to represent both its features and function bundles. The number of remaining features in a tailored network is influenced by the tailoring process (i.e. the choosers' responses). The bigger the part of the original network that remains, the more features and function bundles there are in a unit's realisation triangle. The number of lexical items attached to a word rank constituent depends on the richness of the lexicon. That is, how many synonymous words there are for a given concept. The more lexical items attached to a function bundle, the more assumptions are needed to represent them.

Having said that, we now briefly discuss the initial impressions of what might influence the generator's performance.

1. **Semantic input:** The size of the semantic input representation, in terms of the process roles and circumstantial details, influences the number of syntactic functions that might appear in the surface form to represent these semantic entities. For example, a process representation with no information about the time or place of a process is realised by shorter utterances with less functions, since, for example, the *prep* and *pp-head* functions representing the *circumstance* function of the clause level will be missing from the surface form.

   Also, the level of specificity of the input affects the number of assumptions. The more specific the input is, the more choosers are able to choose one feature among alternatives; resulting in a smaller number of features and function bundles. The extreme case happens when the input is so specific that only the features forming one selection expression survive tailoring.

2. **Network size:** The initial size of the network is not influential. What matters is the size of the remaining part after the tailoring process, as all features and function bundles of this tailored network are to be represented by assumptions or realisation triangles respectively. We assume that the tailoring process is fast and therefore its overhead can be ignored.

3. **Choosers behaviour:** The more a chooser opts not to choose one particular feature among alternatives, the more open paths there are to explore and hence

the more features and function bundles to realise in turn. To simplify things, we implemented the relaxed version of choosers in such a way as to keep all alternatives whenever the semantic representation allows it. However, there will always be cases where choosers can make informed decisions.

4. **Lexicon size:** The lexicaliser attaches to a function bundle all the applicable lexemes (i.e. synonymous ones). All of these lexical choices are initially considered potential realisations for that given function bundle and hence made into ATMS assumptions. So, it is not the actual size of the lexicon that affects the performance but how rich it is in terms of synonymous entries per concept.

5. **Output String Length:** According to the SFL theory, a word in the surface form is a realisation of an underlying functional constituent. Longer sentences have more words and they are more likely to have more open class words each of which may be replaced by one of the its synonyms. Also, if a function bundle is not of word rank, then it needs a realisation triangle of its own which means more assumptions to represent the features which have survived tailoring. Therefore, the size and complexity of the generated string does affect the generation time.

To investigate what factors affect performance and to what extent, we designed some experiments that enable us to vary a specific factor while fixing others. We use the same input of the examples given in Appendix B.1. Without any surface stylistic requirements, that input results in 9450 sentences using the usual relaxed choosers and the lexicon of Appendix A.1. We show next how each factor affects performance.

Degree of Nondeterminism: As we have discussed above, the degree of nondeterminism might be affected by under-specified input or relaxed choosers. The result is larger fragments of networks to explore. Table 8.1 compares the generation times of nondeterministic plain mode and the time it takes when the system generates deterministically. To have the system work deterministically we disable the current choosers and manually drive the choice process so that one feature at each choice point is selected. Deterministic manual choice is achieved by having a predetermined list of features that must be selected by the relevant choosers. This list represents a valid selection expression. Also, the lexicaliser is restricted to one lexeme per function bun-

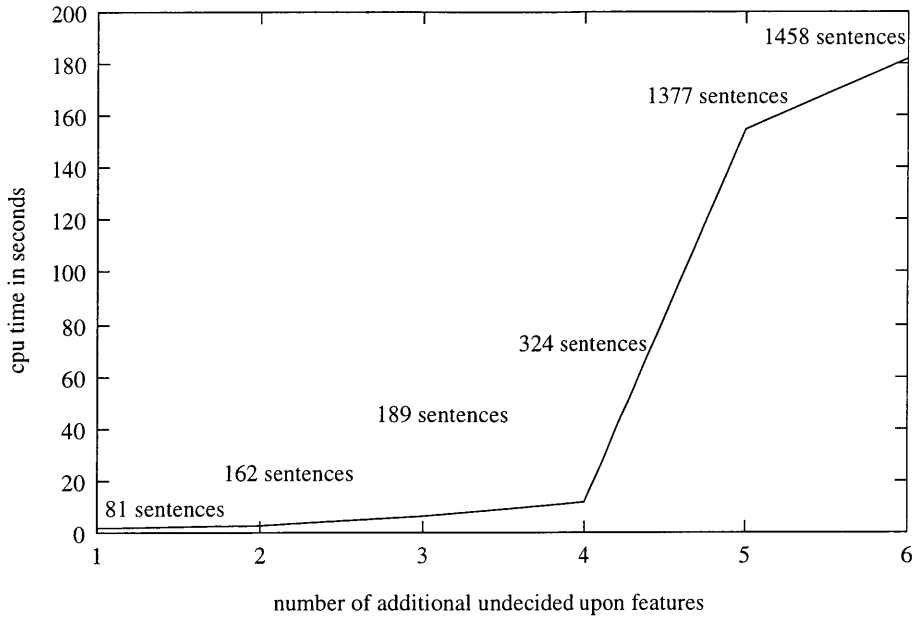


Figure 8.2: Effect of nondeterminism on generation time

dle. The deterministic time is the average of different runs involving different selection expressions. Note that the time per sentence for the nondeterministic case is also 0.794 seconds (i.e. 125.16 mins/9450=0.794 seconds per sentence) which is not very different from the time it takes the system to deterministically generate one sentence.

Although the time per sentence has not changed substantially, the search space has and one would still have to wait for longer to get many surface forms at once. Figure 8.2 shows how the total generation time increases as larger and larger search spaces need to be explored. In this figure, the number of lexemes per function bundle is fixed (3 lexemes per function bundle in this particular experiment). The increase in search space is due to increasing network sizes. In this experiment, we started with a deterministic manual choice process which resulted in a tailored network of certain size (measured here by the number of features). To study the effect of nondeterminism on generation time, we gradually increased the number of *undecided upon* features[5], hence the size of the remaining part of the network. Note that the number of lexemes is kept fixed at 3 per function bundle. Table 8.3 reveals more details in addition to the increase in

[5] As discussed in Section 6.2.1, there are three kinds of features from the chooser's point of view: selected, unselected and undecided. *Undecided upon* features are all the children of a system that its corresponding chooser opts not to choose.

the generation time shown in the plot of figure 8.2. It gives an idea of the size of the network in terms of the number of derived and assumption nodes. It also counts the number of the primitive union operation. We count this operation because it reflects the amount of work done by the ATMS. Any activity which requires that a node label is updated triggers a series of union operations. As discussed in Section 4.5, the union operation is at the heart of both Algorithm 1 and 2 which perform local and global label updates respectively.

	one	two	three	four	five	six
Number of Sentences	81	162	189	324	1377	1458
Generation Time (sec)	1.750	3.310	6.370	12.390	152.550	189.900
Time per Sentence (sec)	0.021	0.020	0.034	0.038	0.111	0.130
Union Operations	578	764	1328	1614	4837	5432
Derived Nodes	73	78	121	127	86	133
Assumption Nodes	51	52	87	87	74	109
Label Sizes Sum	357	529	722	993	3622	3921

Table 8.3: Effect of increasing the number of features

Lexicon Richness: To see the effect of the number of lexemes attached to function bundles, we carried out two experiments: one for deterministic and another for non-deterministic generation. By deterministic generation we mean a choice process that ends up with one selection expression (i.e. all choosers choose). In nondeterministic generation, some choosers do not choose. Hence, the choice process returns multiple selection expressions (or equivalently, multiple syntactic structures). In each experiment, we increase the lexicalisation magnitude which we take here as an indication to the richness of the lexicon. We start with one lexeme per open-class function[6] and increase the number up to six lexemes per function bundle. We only do this for the sake of experimentation, as in a real situation different function bundles will have different numbers of synonyms available in the lexicon.

In both cases, Table 8.4 shows that as the lexicalisation process gets richer, more solutions are generated and a longer time is taken by the ATMS to explore the search space. In the deterministic case, for example, the number of solutions is n^i where n stands for the lexicalisation limit and i is the number of open-class functions in the

[6] That is, function bundles realised by open-class words.

		one lexeme	two lexemes	three lexemes	four lexemes	five lexemes	six lexemes
Determ.	Time (sec)	0.840	1.070	1.890	7.290	37.580	145.320
	Time/sent.	0.840	0.067	0.023	0.028	0.060	0.112
	Sentences	1	16	81	256	625	1296
	Unions	194	315	578	1115	2106	3779
	Deriveds	73	73	73	73	73	73
	Assumps.	43	47	51	55	59	63
	Labels	114	164	276	516	974	1764
Nondet.	Time (sec)	2.110	7.340	124.120	1508.390	10974.230	58389.810
	Time/sent.	0.176	0.038	0.115	0.392	1.045	2.414
	Sentences	12	192	1080	3840	10500	24192
	Unions	500	1500	5151	15281	38394	84390
	Deriveds	123	123	123	123	123	123
	Assumps.	82	91	100	109	118	127
	Labels	240	565	1752	5067	12664	27825

Table 8.4: Lexicalisation richness effect on search space size

surface form.

Sentence Length Effect: To study the effect of the sentence length on the performance of the ATMS-based generator, we designed an experiment whose results are summarised in Table 8.5. In this experiment, we fixed both the degree of nondeterminism (fixed to 2 undecided upon features, which explains why two sentences — and not one — are generated for all lengths when only one lexeme per function bundle is allowed) and the number of lexemes per function bundle (in the upper part of the table fixed to 1, and in the lower part to 3). Having fixed these two parameters, we then started varying the number of function bundles from 3 (the shortest sentence the grammar can generate) up to 8. Table 8.5 shows that the length of sentences (i.e. the number of function bundles in each surface form) does affect the performance of the system. Although the search space remained fixed in the case of one lexeme, having longer sentences requires more time to process the extra function bundles. It also shows that when an increase in the number of functions in a sentence is combined with increases in the richness of the lexicon and nondeterminism degree, the search space expands significantly.

Lexeme per Function		length 3	length 4	length 5	length 6	length 7	length 8
One	Time (sec.)	1.080	1.070	1.600	2.180	2.280	2.350
	Time/sent.	0.540	0.535	0.800	1.090	1.140	1.175
	Sentences	2	2	2	2	2	2
	Unions	281	283	377	439	445	451
	Deriveds	97	101	129	152	156	160
	Assumps.	59	61	81	94	96	98
	Labels	156	162	210	246	252	258
Many	Time (sec.)	1.910	5.990	37.860	200.290	1786.610	15501.980
	Time/sent.	0.035	0.037	0.078	0.137	0.408	1.181
	Sentences	54	162	486	1458	4374	13122
	Unions	772	1460	3454	6231	16969	48605
	Deriveds	97	101	129	152	156	160
	Assumps.	71	77	101	116	122	128
	Labels	330	612	1382	2410	6460	18304

Table 8.5: Sentence Length effect on search space size

Stylistics-aware Mode

It is necessary to determine how well the stylistics-aware generation mode described in the previous chapter improves performance and limits the results to only those utterances that are problem-free, since this is the way we intend to use our ATMS-based generator.

In our ATMS-based generation architecture, choosers are relaxed so that they may opt not to choose a single alternative resulting in non-deterministic generation. For a given semantic input, many selection expressions might arise due to the relaxed nature of choosers and many solutions or surface forms arise as a result. This has the effect of creating a much larger search space than with current systemic generators which are designed to deterministically produce the surface implications of one selection expression. To cope with this complexity, the system uses the surface stylistic specifications to prefer (partial) realisations over others. [Nicolov 99] identifies two issues in generating preferred paraphrases: (1) what does it mean for a realisation to be better than another one, and (2) how to incorporate the notion of betterness in the process. To this end, our implementation provides the means for the second issue and leaves it to the user to define what characteristics the better utterance is required to have.

Next, we analyse the system performance with respect to the three different sets of

stylistic requirements.

	No Constraints	Adjacency Constraints	Metre Constraints	Size Constraints
Number of Sentences	9450	16	1	10
Time (minutes)	125.16	1.222	0.621	13.294
Union operations	27209	7565	4602	24572
Derived Nodes	324	423	387	391
Assumption Nodes	242	242	242	242
Label sizes Sum	10638	1014	1012	4652
Improv. in gen. Time	—	99.02%	99.50%	89.38%

Table 8.6: Improvement in generation time for different sets of SSC

Table 8.6 shows the system's performance in the stylistics-aware mode under the three sets of surface stylistic requirements. It shows a significant improvement over the generate and test approach, had we chosen to generate the solutions in the plain mode and then checked which sentences conform to the surface stylistic requirements. This supports our initial expectation that the ATMS-based generator is best appreciated when used in the stylistics-aware mode.

It is worth noting here that although the number of sentences generated under size constraints (which is 4 in this particular case) is less than that in the case of word adjacency constraints, the time taken is much more. This is because most of the pruning is done at the top level under the size constraints. The earlier that (sub)realisations are excluded from the search space the better.

	Size One	Size Two	Size Three	Size Four
Number of Sentences	0	0	4	10
Time (minutes)	0.325	0.358	1.051	13.294

Table 8.7: System performance for different sentence size limits

Table 8.7 supports this explanation. In this experiment we tried different size values starting from 1 to 4. The table shows that for the length constraints 1 and 2 the system performs very well although it is still discovering the same search space for that of generating 9450 sentences. However, no sentences were found to withstand these constraints. This is because, according to our grammar, the shortest sentence is

of length 3 (e.g. *Dartmaul destroyed Jitrax* and *Jitrax was destroyed*). Once the size limit is increased to 3 the time rises to more than one minute and for size 4 it jumps to 13.294 minutes . This is because we only have one *universal* size constraint which applies to the surface string as well as to its sub-constituents. In effect, all possible function bundles of size 3 or less are generated without noticing that a constituent of length 3 cannot be part of a sentence of length 3 along with at least two other constituents. This suggests that we should probably have two size constraints, one for the overall surface string and another for function bundles at lower levels. We conducted several experiments to see how the system would perform under such new constraints. Table 8.8 summarises the results. In these experiments, we had two size constraints: a surface size limit, and a function bundle (or sub-constituent) size limit.

		Sentence size 3	**Sentence size 4**	**Sentence size 5**	**Sentence size 6**	**Sentence size 7**
Constituent Size 1	Time (mins)	0.303	0.300	0.298	0.300	0.298
	Sentences	4	4	6	6	6
Constituent Size 2	Time (mins)	0.465	0.453	0.433	0.407	0.393
	Sentences	4	16	34	66	94

Table 8.8: Performance results of having two size limits

Although the results shown in Table 8.8 are concerned with the size stylistic requirement, they nevertheless reveal several points:

- the system gives better results with stylistic constraints that apply to partial realisations as well as to complete surface realisations. This is shown in both the case of word adjacency constraints and poetry metre.
- for cases where the above is not possible, blending constraint types can improve performance such as those done with the size constraints when we had two length limits: one for the overall surface form and one for the functional constituents of that surface form.
- the flexibility of the situation-action framework is also shown. We could change our stylistic specifications easily to accommodate the new requirement of having another sub-constituent size constraint in addition to the universal constraint.

This suggests that one can probably use the empirical evaluation exercise as a feedback mechanism to fine-tune the initial stylistic specifications.

8.3 Discussion

Although measuring cpu times is sometimes unreliable as it can be affected by other system dependent factors such as garbage collection it nevertheless does give a rough idea of the speed of the generation process for different sentences.

Since the stylistics-aware mode is the normal way we intend to use the system, what matters is the performance of the system under this mode and we should not be put off by the generation time taken in the plain mode. The latter mode represents the generate and test possibility which was considerably outperformed by the stylistics-aware mode (see the tables 8.6 and 8.8 for the exact figures).

The results show that for generation without stylistics, our system can be competitive to deterministic SFG generators when the process is less likely to succeed the first time. The figures of Table 8.2 show that the ATMS overhead will start to pay off after the first failure. The system's inability to succeed from the first time can be due to either syntactic (i.e. there is no construct to express the meaning), or lexical (i.e. no appropriate word in the lexicon) gaps. It may also be due to ill-formed or under-specified semantic input.

We emphasise here that the ATMS we are currently using is written in an interpreted high level language. Only a few attempts were made to optimise efficiency (e.g. bit vector representation of sets). The number of candidate solutions increases exponentially as both the number of functions in a sentence and the number of lexemes per function increase. For instance, for 50-word sentence and a lexicalisation magnitude of 10, the number of solutions is roughly 10^{50}. In practice, the current ATMS is inconceivable to be able to deal with such a complexity. Things even get worse when there are many 50-word syntactic structures. Although we have not tried it, a serious ATMS implementation may be able to handle such a problem or a reduced version of it. [deKleer 86] claims that the ATMS (an optimised one, I would imagine) is designed in such a way that even when the number of assumptions is very large (say $n = 1000$),

it is still practical to use. For a sentence with 50 functions and 10 lexemes per function bundle, roughly 500 assumptions are needed (plus, of course, other assumptions to represent other bits of the network).

To this end, if the current system is to be used in the non-deterministic plain mode, measures must be taken to confine the scale-up behaviour within a predefined processing time threshold. It is the combinations of open paths and lexicalisation richness that greatly expands the search space. This behaviour is expected from an ATMS-based system since more nondeterminism and richer lexicons mean that there are more assumptions to represent them. However, to keep the scaling up effect within the given ATMS computational capacity, the generator can limit the magnitude of the lexicalisation process if the semantic input has already resulted in a high degree of nondeterminism. Alternatively, it can lift the restrictions on lexicalisation if the non-determinism degree is low. This way, the system will always explore a search space that it can handle. Faster ATMS implementations can have their own threshold values.

For deterministic generation, the results show that the ATMS overhead is around 45% (cf. Table 8.2). This suggests that with faster ATMSs, our system might well be comparable to traditional deterministic generation. However, tackling the problem of non-deterministic generation without stylistic constraints (or heuristics of some sort) is not a difficult task for the ATMS only; it is an inherently complex problem regardless of how one attempts to optimise the implementation. The results show that using the ATMS to solve the already intractable problem does not make it any worse. On the contrary, the ATMS might well help in certain generation tasks as was shown by the stylistics-aware mode. For example, incorporating SSC greatly improved the non-deterministic generation time (cf. Table 8.6).

Although the generation time per sentence is small in the stylistics-aware mode, the ATMS-based generator pursues all solutions in parallel and it, eventually, produces all the surviving sentences at once. This suggests that, for discovering large search spaces, the ATMS-based generator be used for real-time generation only in the deterministic mode. The non-deterministic mode (usually incurring larger search spaces) is more appropriate for non-interactive applications (e.g. leaflet preparation systems) that have clear surface stylistic requirements so that they can be used to cut a great deal

of the search space.

In summary, our evaluation of the current prototype system shows that the new ATMS-based architecture is promising and extensible. With optimised ATMS components, the system can be practical for real-time deterministic (or low-degree non-deterministic) generation. Because of the inherent complexity of non-determinism, the fully non-deterministic mode (i.e. relaxed choosers, and generous lexicaliser) of the system is more appropriate for applications that have strict surface stylistic requirements.

8.4 Limitations

> In this section we discuss the limitations of our work. While we certainly cannot provide a complete list of all the things this work is unable to address, we provide a list of the limitations which we can use as a lead-in to our discussion on future work in Section 9.2 of the next chapter.

The goal of this work was to account for the effect of surface stylistic constraints on earlier linguistic decisions, such as syntactic and lexical choice, in one unified generation architecture. To this end, we were successful in designing a new architecture that relies heavily on the ATMS to generate natural language utterances that have certain surface stylistic requirements.

Having said that, we now list the following limitations of the work in this thesis:

- Using an ATMS to generate text from systemic grammars is a new idea and one primary objective was to see how far we can take the approach, and study what advantages this architecture brings to NLG. In order to get a prototype implemented, we made certain simplifying assumptions at the beginning. For example, the semantic input was intended only to drive the choice process and hence was not based on a structured ontology or any notion of Upper Model.
- Another limitation of our system — as is the case with any ATMS-based problem solver — is that it will only be as fast as the underlying ATMS implementation. This very point may also be seen as an advantage since our system can then

benefit from any improvements in ATMS implementation (e.g. parallel ATMS). It is not known, however, how our system's efficiency will improve as faster ATMS implementations are used. This could be a possible direction for future research.

- Although we evaluated some aspects of the system and characterised how different factors influence generation time, the system needs to be evaluated in a more realistic setting with real size lexicons and grammars. The controlled languages domain may be a plausible choice to test the system in as will be discussed in the future work section of the next chapter.

- As far as the surface stylistic constraints are concerned, we only deal with hard constraints here (or as we call them surface stylistic requirements). There is no such notion as soft constraints or preferences. This is due to the fact that the ATMS we are using is a standard one: assumptions (nodes in general) are either true or false.

- Because — in general — traditional systemic generators do not use the ordering rules until the very end when the lexical items need to be linearised in a surface form, the grammar that we used did not pay much attention to such rules. Partition rules are used as a shorthand for several immediate adjacency rules. However, for our stylistics-aware generator the order rules play an important role in the whole process. That is why we had to manually expand the partition rules into sets of order rules. Also, we had to make explicit the border meta functions which are in some cases missed out in the WAG grammar.

- Currently our generation system does not do any morphology or post-processing. This means that more lexicon entries are needed, as no inflection is performed. However, this is helpful to our surface stylistic needs, since immediately after a word is attached to a function bundle, its stylistic properties are obtainable from the the very same lexicon entry without the need to do any morphology processing.

- Due to the nature of the ATMS' processing, one has to wait and then all the surviving solutions are produced at once. This does not allow an interactive mode of generation where the complete sentences are presented to the user as soon as

they are constructed. The user may be satisfied with some of the sentences generated early on and the whole process can then come to an end. This way (i.e. the interactive mode) allows for acceptable (but not necessarily optimal) solutions to be generated.

8.5 Summary and Outlook

In this chapter, we have presented the implementation details of the ATMS-based generation system: the translator SNAC, the generator STAGE and the particular ATMS implementation we used in this project. We also evaluated the two phases of the generation process by empirically evaluating the system's performance. Several experiments were designed to study the factors that influence generation time. Generation results under the three sets of surface stylistic requirements were then compared and their implications discussed. These results show that folding in surface stylistic constraints, that go straight across the structures produced, significantly improves on the generate-and-test approach. The chapter ended with a discussion of the limitations of the prototype generator.

The next chapter closes the thesis by emphasising the contributions made and speculating on the ways in which this work can be extended in the future.

Chapter 9

Conclusions

This last chapter closes the thesis by revisiting the contributions made and set forth in the introductory chapter. It emphasises the contributions by linking them to where they have been made in the thesis. It also discusses possible extensions to our work.

9.1 Contributions of this Thesis

The main objective of this work was to account for surface stylistic constraints (SSC) — as defined in this thesis — in the generation process. Because of the high interaction between SSC and other linguistic decisions such as syntactic and lexical choice, this goal was viewed from a broader perspective: the system architecture perspective. To this end, a new generation architecture, that allows later constraints to influence earlier decisions, was developed.

Along the way, from the initial objective to the final outcome, this thesis has made several contributions. In the following we highlight these contributions and point out where they have been addressed in the thesis.

- **Identifying the Limitations of Current Systems or Relaxing The Notion of Choice:** Because of the high interaction between SSC and syntactic and lexical choice, we opted for the SFL approach. The SFL formalism intertwines the lexical and syntactic knowledge in one formalism known as the lexicogrammar (Section 2.7). However, current systemic generators have their own

limitations as far as SSC are concerned. In Section 3.8 and 3.9, we concluded that the rigid deterministic nature of current implementations of choosers is to blame for this deficiency, as any uninformed choice has its effect on the stylistic appearance of the surface form. In Chapter 6, we redefined the task of a chooser. Consequently, a chooser is now relaxed in that it does not have to choose if it cannot make an informed decision (Section 6.2.1). It is now the generation algorithm's responsibility to pursue these open paths in an efficient manner.

- **Establishing the Logical Relation between System Networks and Dependency Networks:** In Chapter 4, we argued that the ATMS is a plausible search technique for handling the now relaxed nature of choosers since the ATMS is designed to work in multiple contexts simultaneously. Taking work that interprets system networks in logic as a starting point (Section 5.4), we took the idea further (Section 5.5). We devoted the first part of Chapter 5 to establishing the relation between system networks and ATMS dependency networks. The aim was to represent SFG as ATMS dependency networks.

- **Design of a Translation Algorithm from Systemic Grammars to Equivalent ATMS Representations:** Based on the logical relation between system networks and ATMS dependency networks (established in the first part of Chapter 5), we designed (in the second part of the chapter) an algorithm for the automatic translation of systemic grammars to equivalent ATMS representations. The translation algorithm takes conventional systemic grammars (i.e. system networks and realisation statements) and maps them to an equivalent ATMS representation. The translation algorithm can be used in applications other than text generation since there is no reason which makes the system networks framework specific to linguistic applications. System networks simply specify how combinations of features may imply or be inconsistent with other combinations. However, applying the translation algorithm to other contexts is beyond the scope of this thesis.

- **Developing an ATMS-based Generation Architecture:** Systemic networks are put into what we called network snapshots (Section 5.9). A network snapshot is the result of the translation of a complete systemic grammar network

(i.e. system features and realisation statements). In Sections 6.2 and 6.3, we presented the generation algorithm and showed how it uses network snapshots to generate natural language utterances (Section 6.5). The main thrust of the architecture is in its tolerance. It tolerates relaxed choosers which in some cases may choose not to do their job. It makes up for their laziness by pursuing multiple paths simultaneously.

- **Developing a Framework for the Specification of SSC:** Since each application will have its own surface stylistic requirements, we provided a situation-action framework for the specification of SSC (Section 7.3). We also showed how different sets of surface stylistic requirements can be encoded within the confines of the framework provided (Section 7.9). In Section 7.8, we showed that the ATMS-based architecture can seamlessly accommodate the surface stylistic specifications by switching to a stylistics-aware mode of generation.

- **Implementation and Evaluation of an ATMS-based Generator:** In Chapter 8, we introduced the implementation notes of our generation system: the STylistics-Aware GEenerator (STAGE). STAGE is a prototype ATMS-based generation system embodying the ideas presented in this thesis such as:

  - relaxed notion of choice,
  - translation of system network to ATMS dependency networks,
  - an ATMS-based generation architecture,
  - incorporation of the effect of SSC on syntactic and lexical choice through the stylistics-aware mode of generation,
  - facilitating the specification of different surface stylistic requirements independently of the micro-semantic input, through a situation-action framework.

  The implemented generation system can be viewed as a general test-bed for experimenting with different stylistic requirements since the ability to follow more than one option through the system network and to fold in constraints that go straight across the structures produced can be a helpful research tool.

In the second part of Chapter 8, we evaluated the two stages: translation and generation. The translation stage was evaluated in terms of the size of the grammar used and the size of the result of the translation process (Section 8.2.1). Then, in evaluating the generation stage, we identified the factors that affect the system performance and compared the system behaviour with respect to the three sets of surface stylistic requirements (Section 8.2.2). Finally, we discussed the limitations of our generation system in Section 8.4.

9.2 Future Directions

The avenues for further research range form minor algorithm tuning to broad areas for exploration. We can divide the directions for future work into the following categories:

- **The Generation Tasks Involved:** Currently, we assume that a semantic input is chunked in such a way as to give a sentence-sized output. Hence, the generation tasks involved are those at the tactical level. We believe, however, that linguistic constraints can sometimes affect what one can say and that "language can feed back to the planning component [...] because a particular language construction forces [one] to include additional semantic material" [Nicolov 99] or probably miss some out. Having said that, further work can incorporate other stages of the generation process such as content determination. This is very much motivated by the viewpoint that the SFL formalism can actually be thought of as a general knowledge representation language and not just as a resource for syntactic structuring [O'Donnell 94].

  [Patten 86] for example uses system networks to represent semantics. The same idea of relaxed choosers can be applied to systems at this level and there will be concepts that can be represented by more than one path. Each of these possibilities will then be realised differently at the lexico-grammatical level.

  Moreover, involving the strategic planning stages into the generation process would allow more flexibility in tackling lexical choice, a central issue in this work. This would allow semantic entities to be packed into lexemes in different ways. [Elhadad *et al.* 97] stretch the lexical chooser functionality to cover some

planning tasks, so that more than one semantic entity is fused in one linguistic unit, which is then realised by one word.

- **Stylistics-related Issues:** The sets of surface stylistic requirements we used are meant to demonstrate the functionality of the approach. They are by no means the ultimate stylistic needs one can aim for. Although the generator seemed to have done well under these requirements, it will be interesting to see how well it performs under stylistic requirements that are driven by real applications. For example, lexical collocational constraints can be extracted from machine-readable corpora using statistical approaches. Other lexical properties can come from real size lexicons.

  One possibility as a source of real stylistic requirements is in the area of controlled language specifications. We have already experimented with length constraints on sentences and constituents within sentences. However, further constraints that have to do with allowed grammatical structures need to be tested. The objective would be to test whether our approach represents a practical tool that can be used in controlled language generation.

  The system can be extended to deal with multiple sentence generation. This way, we will have a better environment in which to test the stylistic sensitivity of the system (e.g. rhyme constraints in poetic writings and how several sentences are involved in satisfying the surface constraints).

  The situation-action framework interface can be enhanced so that it becomes more user friendly. Instead of specifying the surface requirements in Prolog, the user can do so using a high level representation mechanism.

  Finally, the inclusion of earlier generation tasks brings with it the possibility of incorporating deep stylistic constraints. For example, conciseness constraints can be achieved more easily if planned for early on in the generation process. A relevant issue here is how to represent deeper stylistic requirements and whether the situation-action framework, developed to cater for surface constraints, will be of any use at this level.

- **ATMS-related Extensions:** We embarked on this research guided solely by the intuition that the ATMS' ability to keep track of dependencies, between new

information and earlier assumptions, should be beneficially applicable to NLG. However, we had no details of how this could be accomplished. So, we picked a simple plain ATMS implementation and started exploring possibilities. We ended up with an architecture in which the ATMS plays a central role. This means that the generator efficiency is very much influenced by the ATMS component. It would be revealing, however, to plug in different ATMS implementations. Plugging in different ATMS implementations (e.g. parallel, probabilistic, ... etc.) would enable us to test the effect of the particular ATMS on the generation speed in conjunction with other factors. For example, [Haenni 98] extends the ATMS to allow its True-or-False-only assumptions to take different degrees of certainty ranging from 0 to 1. These would allow the representation of soft stylistic constraints. This way, any construct or sub-realisation that falls below a certain threshold value will be sent the false node. Consequently, we will be talking about stylistic preferences as opposed to crisp stylistic requirements only.

- **SNAC-related Extensions:** SNAC translates SFG networks to ATMS dependency networks. Meanwhile, it assumes the systemic networks to be in a Prolog notation that we defined. Also, on the output side of SNAC, it prepares the dependency networks so that they can readily be used by the particular ATMS we are using. Because system networks are written in different formats and using different conventions, it would be helpful to have a way of defining a given convention to SNAC, so that it can then translate any systemic grammar no matter what notation it uses. Also, to facilitate the idea of plugging in different ATMS implementations, the same can be done to the other side of the process (i.e. the output of SNAC). The goal is to have SNAC generate output in the correct format for any ATMS implementation to consume.

- **Linguistic Resources Issues:** One possible direction for future work in this regard is to extend the coverage of the grammar we are currently using. Another possibility is to use another grammar altogether – one with a broader linguistic coverage.

9.3 Concluding Remarks

In this thesis, we have brought together a linguistic theory and an AI search technique and put them into a new NLG system architecture. However, our main objective was not the development of a "new" architecture as much as the incorporation of the surface stylistic constraints into the overall generation process. In particular, the proposed architecture accounts for the interaction between syntactic, lexical, and surface constraints.

We believe that although our motivation to design the new architecture was the need to incorporate surface stylistic constraints in the generation process, the solution presented in this thesis can also be used to answer the general question: what happens when a generator makes many decisions during generation and then new circumstances/constraints arise suggesting that it has made — at some unknown point — the wrong choices? We hope that the essence of our approach remains useful to other generation tasks especially those at the strategic planning level; and to other domains using the system network formalism as their representation language.

From an NLG standpoint, this work offers a new ATMS-based generation architecture to experiment with; while — from the ATMS viewpoint — it can be seen as a new application using the ATMS which shows the versatility of the ATMS as a search mechanism.

Bibliography

[Adorni & Zock 96] Giovanni Adorni and Michael Zock, editors. *Trends in Natural Language Generation: an Artificial Intelligence Perspective.* Springer-Verlag, Germany, 1996.

[Al-Jabri 97] Saad K. Al-Jabri. *Generating Arabic Words from Semantic Descriptions.* Unpublished PhD thesis, University of Edinburgh, 1997.

[Bailey 99] Paul Bailey. Searching for storiness: Story-generation from a reader's perspective. In *Proceedings of the 1999 AAAI Fall Symposium Series: Narrative Intelligence (AAAI-99)*, 1999.

[Bateman 97a] John A. Bateman. Enabling technology for multilingual natural language generation: the KPML development environment. *Journal of Natural Language Engineering*, 3(1):15–55, 1997.

[Bateman 97b] John A. Bateman. Sentence generation and systemic grammar: an introduction. In *Iwanami Lecture Series: Language Sciences*, volume 8. Iwanami Shoten Publishers, Tokyo, Japan, 1997.

[Bateman *et al.* 90] John A. Bateman, Robert T. Kasper, Johanna D. Moore, and Richard A. Whitney. A general organization of knowledge for natural language processing: The Penman upper model. Unpublished technical report, USC Information Sciences Institute, 1990.

[Berry 75] H. Margaret Berry. *Introduction to Systemic Linguistics*, volume 1: Structures and Systems. B. T. Batsford Ltd., London, 1975.

[Berry 77] H. Margaret Berry. *Introduction to Systemic Linguistics*, volume 2: Levels and Links. B. T. Batsford Ltd., London, 1977.

[Binsted & Ritchie 94] Kim Binsted and Graeme Ritchie. An implemented model of punning riddles. In *Proceedings of the 12th National Conference on Artificial Intelligence. Volume*

1, pages 633–638, Menlo Park, CA, USA, July 31–August 4 1994. AAAI Press.

[Binsted & Ritchie 96] Kim Binsted and Graeme Ritchie. Speculations on story puns. In J. Hulstijn and A. Nijholt, editors, *Proceedings of International Workshop on Computational Humour (TWLT 12)*, pages 151–159, University of Twente, Enschede, Netherlands, September 1996.

[Binsted *et al.* 95] K. Binsted, A. Cawsey, and R. Jones. Generating personalised patient information using the medical record. *Lecture Notes in Computer Science*, 934:29–41, 1995.

[Bouayad-Agha *et al.* 91] N. Bouayad-Agha, D. Scott, and R. Power. Integrating content and style in documents: a case study of patient information leaflets. *Information Design Journal*, 9(2-3):161–176, 1991.

[Bourbeau *et al.* 90] Laurent Bourbeau, Denis Carcagno, E. Goldberg, Richard Kittredge, and Alain Polguère. Bilingual generation of weather forecasts in an operational environment. In *Proceedings of the 13th International Conference on Computational Linguistics (COLING-90)*, volume 1, pages 90–92, Helsinki, 1990.

[Brew 91] Chris Brew. Systemic classification and its efficiency. *Computational Linguistics*, 17(4):375–408, December 1991.

[Busemann 93] Stephan Busemann. A holistic view of lexical choice. In Helmut Horacek and Michael Zock, editors, *New concepts in Natural Language Generation: Planning, Realization, and Systems*, pages 302–308. Pinter Publishers, New York, 1993.

[Cahill & Reape 99] L. Cahill and M. Reape. Component tasks in applied nlg systems. Technical Report ITRI-99-05, RAGS project, 1999.

[Cahill *et al.* 99] L. Cahill, C. Doran, R. Evans, C. Mellish, D. Paiva, M. Reape, D. Scott, and N. Tipper. In search of a reference architectures for nlg systems. In *Proceedings of the 7th European Workshop on Natural Language Generation*, Toulouse, France, 1999.

[Calder 99] Jo Calder. The horn subset of systemic networks. In *Proceedings of the 6th Meeting on Mathematics of Language, MOL6*, University of Central Florida, Orlando, Florida, USA, July 23-25, 1999. to appear.

[COLING-88 88] *Proceedings of the 12th International Conference on Computational Linguistics (COLING-88)*, Budapest, August 22-27, 1988.

[Dale *et al.* 92] Robert Dale, Eduard H. Hovy, Dietmar Rösner, and Oliviero Stock. *Aspects of Automated Natural Language Generation.* Lecture Notes in Artificial Intelligence, 587. Springer-Verlag, Berlin, April 1992.

[De Smedt *et al.* 96] Koenraad De Smedt, Helmut Horacek, and Michael Zock. Architectures for natural language generation: Problems and perspectives. In Adorni and Zock [Adorni & Zock 96], pages 17–46.

[deKleer & Williams 86] Johan de Kleer and Brian Williams. Reasoning about multiple faults. In Tom Kehler and Stan Rosenschein, editors, *Proceedings of the 5th National Conference on Artificial Intelligence. Volume 1*, pages 132–139, Los Altos, CA, USA, August 1986. Morgan Kaufmann.

[deKleer 86] Johan de Kleer. An assumption-based TMS. *Artificial Intelligence*, 28:127–162, 1986.

[DiMarco & Hirst 93] Chrysanne DiMarco and Graeme Hirst. A computational theory of goal-directed style in syntax. *Computational Linguistics*, 19(3):451–499, 1993.

[DiMarco & Stede 93] Chrysanne DiMarco and Manfred Stede. The semantic and stylistic differentiation of synonyms and near-synonyms. In *Working notes of the AAAI Spring Symposium on Building Lexicons for Machine Translation*, Stanford University, March 1993.

[Dixon & deKleer 89] M. Dixon and J. de Kleer. Massively parallel assumption-based truth maintenance. In M. Reinfrank, J. de Kleer, M. L. Ginsberg, and E. Sandewall, editors, *Non-monotonic reasoning*, pages 131–142. Berlin: Springer-Verlag, 1989. Lecture Notes in Artificial Intelligence, 346.

[Doyle 79] Jon Doyle. A truth maintenance system. *Artificial Intelligence*, 12(3):231–272, 1979.

[Edmonds 95] Philip Edmonds. Lexical knowledge for natural language generation. PhD qualification paper. Technical report, Department of Computer Science, University of Toront, Canada, 1995.

[Edmonds 97] Philip Edmonds. Choosing the word most typical in context using a lexical co-occurrence network. In *Proceedings of the 35th Annual Meeting of the Association*

for Computational Linguistics (ACL/EACL 97), pages 507–509, Madrid, 1997.

[Edmonds 99] Philip Edmonds. *Semantic Representations of Near-Synonyms for Automatic Lexical Choice.* Unpublished PhD thesis, Department of Computer Science, University of Toronto, 1999.

[Elhadad 92] Michael Elhadad. *Using argumentation to control lexical choice: A functional unification-based approach.* Unpublished PhD thesis, Computer Science Department, Columbia University, 1992.

[Elhadad *et al.* 97] Michael Elhadad, Kathleen McKeown, and Jacques Robin. Floating constraints in lexical choice. *Computational Linguistics*, 23(2), 1997.

[Evelyne & Pierette 94] V. Evelyne and B. Pierette. Semantic lexicons: The cornerstone for lexical choice in natural language generation, 1994.

[Forbus & de Kleer 93] Kenneth D. Forbus and Johan de Kleer. *Building Problem Solvers.* MIT Press, Cambridge, Massachusetts, 1993.

[Forbus 87] Kenneth D. Forbus. The qualitative process engine: A study in assumption-based truth maintenance. In *Qualitative Reasoning Workshop Abstracts.* Qualitative Reasoning Group, University of Illinois at Urbana-Champaign, 1987.

[Ginsberg 87] Matthew L. Ginsberg. *Readings in Nonmonotonic Reasoning*, chapter 1, pages 1–23. Morgan Kaufmann, Los Altos, CA, 1987.

[Goldman 75] Neil M. Goldman. Conceptual generation. In Roger C. Schank and Christopher K. Riesbeck, editors, *Conceptual Information Processing.* American Elsevier, New York, NY, 1975.

[Granville 84] Robert Granville. Controlling lexical substitution in computer text generation. In *Proceedings of the Tenth International Conference on Computational Linguistics (COLING-84) and the 22nd Annual Meeting of the ACL*, pages 381–384, Stanford University, Stanford, CA, July 2-6, 1984.

[Haenni 98] R. Haenni. Modelling uncertainty with propositional assumption-based systems. *Lecture Notes in Computer Science*, 1455:446–??, 1998.

[Halliday & Hasan 76] Michael A. K. Halliday and R. Hasan. *Cohesion in English.* Longman, London, 1976.

[Halliday 73] Michael A. K. Halliday. *Explorations in the Functions of Language.* Edward Arnold, London, 1973.

[Haruno *et al.* 93] Masahiko Haruno, Makoto Nagao, and Yasuharu Den. Bidirectional chart generation of natural language texts. In *Proceedings of the 11th National Conference on Artificial Intelligence (AAAI-93)*, pages 350–356, Washington, DC, July 11-15, 1993.

[Haruno *et al.* 96] Masahiko Haruno, Yasuharu Den, and Yuji Matsumoto. A chart-based semantic head driven generation algorithm. In Giovanni Adorni and Michael Zock, editors, *Proceedings of the 4th European Workshop on Natural Language Generation*, volume 1036 of *LNAI*, pages 300–313, Berlin, April 1996. Springer Verlag.

[Hasan 87] Ruqaiya Hasan. The grammarian's dream: lexis as most delicate grammar. In M. A. K. Halliday and R. Fawcett, editors, *New developments in systemic linguistics: theory and description.* Pinter, London, 1987.

[Henschel 97] Renate Henschel. Compiling systemic grammar into feature logic systems. In Suresh Manandhar, editor, *Proceedings of CLNLP.* Springer, 1997.

[Hovy 88] Eduard H. Hovy. *Generating Natural Language under Pragmatic Constraints.* Lawrence Erlbaum Associates, Hillsdale, NJ, 1988. Based on PhD thesis, Yale University.

[Hovy 96] E. Hovy. Language generation. In *Survey of the State of the Art in Human Language Technology.* `http://cslu.cse.ogi.edu/HLTsurvey/`, 1996. A hardcopy of this document is available from Cambridge University Press ISBN 0-521-59277-1.

[Hudson 71] Richard A. Hudson. *English Complex Sentences: An Introduction to Systemic Grammar.* North-Holland, Amsterdam, 1971.

[Inui *et al.* 92] Kentaro Inui, Takenobu Tokunaga, and Hozumi Tanaka. Text revision: A model and its implementation. In *Aspects of Automated Natural Language Generation* [Dale *et al.* 92], pages 215–230.

[Iordanskaja *et al.* 88] Lidija Iordanskaja, Richard Kittredge, and Alain Polguère. Implementing a meaning-text model for language generation. In COLING-88 [COLING-88 88].

[Iordanskaja *et al.* 91] Lidija Iordanskaja, Richard Kittredge, and Alain Polguère. Lexical selection and paraphrase in a meaning-text generation model. In Cécile L. Paris, William R. Swartout, and William C. Mann, editors, *Natural Language Generation in Artificial Intelligence and Computational Linguistics*, pages 293–312. Kluwer Academic Publishers, Boston, 1991.

[Jacobs 87] Paul S. Jacobs. Knowledge-intensive natural language generation. *Artificial Intelligence*, 33(3):325–378, November 1987.

[Jones & Millington 88] John Jones and Mark Millington. Modelling Unix users with an ATMS: Some preliminary findings. In Barbara M. Smith and Gerald Kelleher, editors, *Reason Maintenance Systems and Their Applications*, pages 134–154. Ellis Horwood, Ltd., Chichester, 1988.

[Kantrowitz & Bates 92] Mark Kantrowitz and Joseph Bates. Integrated natural language generation systems. In *Aspects of Automated Natural Language Generation*, Lecture Notes in Artificial Intelligence, 587, pages 13–28. Springer-Verlag, Berlin, April 1992.

[Kasper & O'Donnell 90] R. Kasper and M. O'Donnell. Representing the nigel grammar and semantics in loom. Technical report, USC Information Science Institute, Marina del Rey, CA, 1990.

[Kay 96] Martin Kay. Chart generation. In *Proceedings of the 34th Annual Meeting of the ACL*, pages 200–204, University of California, Santa Cruz, CA, June 24-27, 1996.

[Knight & Hatzivassiloglou 95] K. Knight and V. Hatzivassiloglou. Two-level, many-paths generation. In *Proceedings of the 33rd Annual Meeting of the ACL*, pages 252–260, MIT, Cambridge, Massachusetts, June 26-30, 1995.

[Kohlas *et al.* 98] J. Kohlas, R. Haenni, and N. Lehmann. Assumption-based reasoning and probabilistic argumentation systems. Accepted for Publication in the final DRUMS handbook, 1998.

[Lamma & Mello 93] E. Lamma and P. Mello. A rationalisation of the ATMS in terms of partial evaluation. In Kung-Kiu Lau and Tim Clement, editors, *Proceedings of the International Workshop on Logic Program Synthesis and Transformation*, Workshops in Computing, pages 118–131, London, July 2–3 1993. Springer Verlag.

[Laskey & Lehner 88] Kathryn B. Laskey and Paul E. Lehner. Belief maintenance: An integrated approach to uncertainty management. In Tom M. Smith, Reid G.; Mitchell, editor, *Proceedings of the 7th National Conference on Artificial Intelligence*, pages 210–214, St. Paul, MN, August 1988. Morgan Kaufmann.

[Lin 96] Dekang Lin. On the structural complexity of natural language sentences. In *Proceedings of the 16th International Conference on Computational Linguistics (COLING-96)*, pages 729–733, Copenhagen, Denmark, 1996.

[Mann & Matthiessen 83] William C. Mann and Christian Matthiessen. NIGEL: A systemic grammar for text generation. Technical Report ISI-RR-83-105, USC Information Sciences Institute, Marina Del Rey, CA, 1983.

[Mann 82] William C. Mann. The anatomy of a systemic choice. Technical Report ISI/RS-82-104, USC Information Science Institute, Marina del Rey, CA, 1982.

[Mann 83a] William C. Mann. A linguistic overview of the nigel text generation grammar. Technical Report ISI/RS-83-9, USC Information Science Institute, Marina del Rey, CA, 1983.

[Mann 83b] William C. Mann. An overview of the Penman text generation system. Technical Report ISI/RS-83-114, USC Information Science Institute, Marina del Rey, CA, 1983.

[Manurung *et al.* 00] H. Manurung, G. Ritchie, and H. Thompson. A flexible integrated architecture for generating poetic texts. In *Proceedings of the Fourth Symposium on Natural Language Processing (SNLP 2000)*, pages 123–132, Chiang Mai, Thailand, 10-12 May 2000.

[Matthews 97] Peter Matthews. *The Concise Oxford Dictionary of Linguistics.* Oxford University Press, Oxford, UK, 1997.

[Matthiessen 87] Christian Matthiessen. Semantics for a systemic grammar: the chooser and inquiry framework. Technical Report RS-87-189, USC Information Sciences Institute, Marina Del Rey, CA, 1987.

[McKeown *et al.* 93] K. McKeown, J. Robin, and M. Tanenblatt. Tailoring lexical choice to the user's vocabulary in multimedia explanation generation. In *Proceedings of the*

31st Annual Meeting of the Association for Computational Linguistics (ACL), pages 226–234, Columbus, OH, 1993.

[Mellish 88] Chris Mellish. Implementing systemic classification by unification. *Computational Linguistics*, 14(1):40–51, Winter 1988.

[Mellish 95] Chris Mellish. Natural language generation and technical documentation. *Saudi Computer Journal*, 1(1):3–20, November 1995.

[Mel'čuk 88] Igor Mel'čuk. *Dependency Syntax: Theory and Practice.* State University of New York Press, Albany, NY, 1988.

[Morgue & Chehire 91] Geneviéve Morgue and Thomas Chehire. Efficiency of production systems when coupled with an assumption based truth maintenance system. In Kathleen Dean, Thomas L.; McKeown, editor, *Proceedings of the 9th National Conference on Artificial Intelligence*, pages 268–274. MIT Press, July 1991.

[Nicolov 99] Nicolas N. Nicolov. *Approximate text generation from non-hierarchical representations in a declarative framework.* Unpublished PhD thesis, University of Edinburgh, 1999.

[Nirenburg & Nirenburg 88] Sergei Nirenburg and Irene Nirenburg. A framework for lexical selection in natural language generation. In COLING-88 [COLING-88 88], pages 471–475.

[Nogier & Zock 91] J. Nogier and Michael Zock. Lexical choice as pattern matching. In T. Nagle, J. Nagle, L. Gerholz, and P. Elklund, editors, *Current directions in conceptual structures research.* Springer-Verlag, New York, NY, 1991.

[O'Donnell 94] Michael O'Donnell. *Sentence Analysis and Generation – a Systemic Perspective.* Unpublished PhD thesis, Department of Linguistics, University of Sydney, 1994.

[O'Donnell 97] Michael O'Donnell. Variable-length on-line document generation. In *Proceedings of the 6th European Workshop on Natural Language Generation*, Gerhard-Mercator University, Duisburg, Germany, 1997.

[O'Donnell *et al.* 00] Michael O'Donnell, Alistair Knott, Jon Oberlander, and Chris Mellish. Optimising text quality in generation from relational databases. In *Proceedings of*

the International Natural Language Generation Conference, INLG'2000, Mitzpe Ramon, Israel, June 2-16 2000.

[Paiva 99] Daniel Paiva. Invistigating nlg architectures taking style into consideration, 1999.

[Patten & Ritchie 86] Terry Patten and Graeme D. Ritchie. A formal model of systemic grammar. DAI Research Paper 290, Department of Artificial Intelligence, University of Edinburgh, 1986.

[Patten 86] Terry Patten. *Interpreting Systemic Grammar as a Computational Representation: A Problem Solving Approach to Text Generation.* Unpublished PhD thesis, Edinburgh University, Department of Artificial Intelligence, 1986.

[Patten 88] Terry Patten. *Systemic text generation as problem solving.* Cambridge University Press, New York, 1988. Based on PhD Thesis.

[Poole *et al.* 98] David Poole, Alan Mackworth, and Randy Goebel. *Computational Intelligence: A Logical Approach.* Oxford University Press, Oxford, 1998.

[Power 00] R. Power. Planning texts by constraint satisfaction. In *Proceedings of the 18th International Conference on Computational Linguistics (COLING-2000)*, pages 642–648, Saarbrucken, Germany, 2000.

[Provan 88] Gregory M. Provan. A complexity analysis of assumption-based truth maintenance systems. In Barbara M. Smith and Gerald Kelleher, editors, *Reason Maintenance Systems and Their Applications*, pages 98–113. Ellis Horwood, Ltd., Chichester, 1988.

[Pustejovsky & Nirenburg 87] James D. Pustejovsky and Sergei Nirenburg. Lexical selection in the process of language generation. In *Proceedings of the 25th Annual Meeting of the ACL*, pages 201–206, Stanford University, Stanford, CA, July 6-9, 1987.

[Reiter & Dale 00] Ehud Reiter and Robert Dale. *Building Natural Language Generation Systems.* Cambridge University Press, Cambridge, UK, 2000.

[Reiter & deKleer 87] R. Reiter and J. de Kleer. Foundations of assumption-based truth maintenance systems. In *Proceedings of AAAI-87*, pages 183–188, Seattle, WA, 1987.

[Reiter 91] Ehud Reiter. A new model of lexical choice for nouns. *Computational Intelligence*, 7(4):240–251, November 1991. Also appears as DAI Research Paper 547, Department of Artificial Intelligence, University of Edinburgh.

[Reiter 94] Ehud Reiter. Has a consensus on NL generation appeared, and is it psycholinguistically plausible? In *Proceedings of the Fifth International Natural Language Generation Workshop*, pages 163–170, 1994.

[Reiter 00] Ehud Reiter. Pipelines and size constraints. *Computational Linguistics*, 26(2):251–259, 2000.

[Robin 90] Jacques Robin. Lexical choice in natural language generation. Technical Report CUCS-040-90, Department of Computer Science, Columbia University, New York, 1990.

[Robin 94] Jacques Robin. *Revision-Based Generation of Natural Language Summaries Providing Historical Background: Corpus-Based Analysis, Design, Implementation and Evaluation.* Unpublished PhD thesis, Computer Science Department, Columbia University, 1994.

[Rösner & Stede 94] Dietmar Rösner and Manfred Stede. Generating multilingual documents from a knowledge base: the TECHDOC project. In *Proceedings of the 15th International Conference on Computational Linguistics (COLING-94)*, volume 1, pages 339–346, Kyoto, Japan, 1994.

[Rubinoff 92] Robert Rubinoff. Integrating text planning and linguistic choice. In *Aspects of Automated Natural Language Generation* [Dale *et al.* 92], pages 45–56.

[Rubinoff 00] Robert Rubinoff. Integrating text planning and linguistic choice without abandoning modularity: The igen generator. *Computational Linguistics*, 26(2):107–138, June 2000.

[Russell & Norvig 95] Stuart Russell and Peter Norvig. *Artifical Intelligence: A Modern Approach,*. Prentice-Hall, Englewood Cliffs, NJ, ISBN 0-13-103805-2, 912 pp., 1995, 1995.

[Shemtov 98] H. Shemtov. A method for preserving ambiguities in chart generation. In *Proceedings of Tabulation in Parsing and Deduction (TAPD'98)*, pages 36–43, Paris (FRANCE), April 1998.

[Shoham 87] Yoav Shoham. A semantical approach to nonmonotonic logics. In Matthew L. Ginsberg, editor, *Readings*

in Nonmonotonic Reasoning, pages 227–250. Morgan Kaufmann, Los Altos, California, 1987.

[Shoham 94] Yoav Shoham. *Artifical Intelligence: Techniques in Prolog.* Morgan-Kauffman, 1994.

[Smith & Kelleher 88] Barbara M. Smith and Gerald Kelleher. A brief introduction to reason maintenance systems. In Barbara M. Smith and Gerald Kelleher, editors, *Reason Maintenance Systems and Their Applications*, pages 4–20. Ellis Horwood, Ltd., Chichester, 1988.

[Smith 88] Barbara M. Smith. Forward checking, the ATMS and search reduction. In Barbara M. Smith and Gerald Kelleher, editors, *Reason Maintenance Systems and Their Applications*, pages 115–168. Ellis Horwood, Ltd., Chichester, 1988.

[Sondheimer *et al.* 89] Norman K. Sondheimer, Susanna Cumming, and Robert Albano. How to realize a concept: Lexical selection and the conceptual network in text generation. Technical Report RS-89-248, USC Information Sciences Institute, 1989.

[Stede 93] Manfred Stede. Lexical choice criteria in language generation. In *Proceedings of the Sixth European Meeting of the ACL*, Utrecht, 1993.

[Stede 96a] Manfred Stede. Lexical options in multilingual generation from a knowledge base. In Adorni and Zock [Adorni & Zock 96], pages 222–237.

[Stede 96b] Manfred Stede. *Lexical Semantics and Knowledge Representation in Multilingual Sentence Generation.* Unpublished PhD thesis, University of Toronto, 1996.

[Teich 95] Elke Teich. *A proposal for dependency in Systemic Functional Grammar: metasemiosis in Computational Systemic Functional Linguistics.* Unpublished PhD thesis, University of the Saarland, Saarbrücken, Germany, 1995.

[Teich 99] Elke Teich. *Systemic functional grammar in natural language generation: linguistic description and computational representation.* Cassell, London, 1999.

[Thompson 77] Henry S. Thompson. Strategy and tactics: A model for language production. In W. A. Beach, S. E. Fox, and S. Philosoph, editors, *Papers from the 13th Regional Meeting of the Chicago Linguistics Society*, pages 651–668, Chicago, IL, April 14-16, 1977.

[Tung *et al.* 88] Yu-Wen Tung, C. Matthiessen, and N. Sondheimer. On parallelism and the penman natural language generation system. Technical Report ISI/RS-88-195, USC Information Science Institute, Marina del Rey, CA, 1988.

[Winograd 83] Terry Winograd. *Language as a Cognitive Process: Syntax*, volume I. Addison-Wesley, Reading, MA, 1983.

Appendix A

The Lexico-grammatical Resources

A.1 The Systemic Grammar Networks

Here we present the complete WAG grammar that SNAC translated into ATMS dependency networks. Note that because we had to enrich the ordering rules manually, we only used fragments of these networks in generation.

A.1.1 Clause Network

```
system_def( entry_cond(grammatical_unit), g_unit_type,
            [clause,
             group,
             word,
             quotation]).

%--- complexity
system_def( entry_cond(clause), clause_complexity,
        [clause_simplex,
         clause_complex]).

%--- fullness
system_def( entry_cond(clause_simplex), clause_type,
        [full,
         fragment,
         minor]).

system_def( entry_cond(fragment), fragment_type,
        [nominal_fragment,
         polar_fragment]).

%--- finiteness
system_def( entry_cond(full), dependence,
        [finite,
         nonfinite]).
```

```
%--- nonfinite type
system_def( entry_cond(nonfinite), nonfinite_sys,
        [infinitive_clause,
         ing_form]).

%--- VOICE: active/passive
system_def( entry_cond(nonrelational_clause), voice,
        [active,
         passive]).

%--- passive_type
system_def( entry_cond(passive), passive_type,
        [medio_passive,
         recipio_passive,
         addressee_passive,
         indirect_agent_passive]).

%--- mood
system_def( entry_cond(finite),  mood,
        [indicative,
         imperative]).

system_def( entry_cond(indicative & subject_resolved), subj_fin_sys,
        [subj_fin,
         fin_subj]).

system_def( entry_cond(indicative & subj_fin & nonwh_subject &
                    (nonwh_object \/ object_not_inserted) &
                    (nonwh_primary_circumstance \/ no_primary_circumstance)),
         declarative_gate,
        [declarative]).

system_def( entry_cond(finite & (wh_subject \/ wh_object \/
            wh_primary_circumstance \/ fin_subj)), interrogative_gate,
        [interrogative]).

%---- minor_fragment_mood
system_def( entry_cond(ec((minor \/ fragment) & sentential)), minor_fragment_mood,
        [interrogative_fragment_or_minor,
         declarative_fragment_or_minor]).

%--- AUXILARIES:
%--- modality
system_def( entry_cond(indicative), modality,
        [modal_or_future,
         not_modal_or_future]).

system_def( entry_cond(ec(not_modal_or_future & no_do_insert)), nomodalordo_gate,
        [not_modal_or_do]).

%--- tense
system_def( entry_cond(not_modal_or_future), clause_tense,
        [past_clause,
         present_clause]).

%--- perfective
system_def( entry_cond(full), perfect,
        [perfective,
         nonperfective]).
```

```
%--- progressive
system_def( entry_cond(full), progressive,
        [progressive,
         nonprogressive]).

%--- do_insertion
system_def( entry_cond(tense_pred & not_modal_or_future), noaux_system,
        [no_auxilaries]).

system_def( entry_cond(negative_imperative \/ declarative_do \/
                       (no_auxilaries & nonrelational_clause &
                       (negative_clause \/ fin_subj))), do_insert_gate,
        [do_insert]).

system_def( entry_cond(no_auxilaries & subj_fin & positive_clause), subj_fin_do_insert,
        [declarative_do,
         no_declarative_do]).

system_def( entry_cond(ec(positive_imperative \/ (indicative &
                     (no_declarative_do \/ modal_or_future \/
                      passive \/ perfective \/ progressive \/
                      relational_clause))), no_do_insert_gate,
        [no_do_insert]).

%--- auxillary agreement
system_def( entry_cond(progressive & active), progpred,
        [prog_pred]).

system_def( entry_cond(progressive & passive), prog_pass,
        [prog_pass]).

system_def( entry_cond(perfective & progressive), perf_prog,
        [perf_prog]).

system_def( entry_cond(nonperfective & progressive), tense_prog,
        [tense_prog]).

system_def( entry_cond(perfective & nonprogressive & active), perf_pred,
        [perf_pred]).

system_def( entry_cond(perfective & nonprogressive & passive), perf_pass,
        [perf_pass]).

system_def( entry_cond(nonperfective & nonprogressive & passive), tense_pass,
        [tense_pass]).

system_def( entry_cond(nonperfective & nonprogressive &
                       (active \/ relational_clause)), tense_pred,
        [tense_pred]).

%--- subject/finite agreement
system_def( entry_cond(not_modal_or_future & subject_resolved), subject_person,
        [first_person_subject,
         second_person_subject,
         third_person_subject]).

system_def( entry_cond(not_modal_or_future & subject_resolved), subject_number,
        [singsubj,
```

```
         plursubj]).

%--- clause polarity
system_def( entry_cond(finite), clause_polarity,
        [positive_clause,
         negative_clause]).

system_def( entry_cond(negative_clause), clause_negation_type,
        [negation_in_finite,
         negation_separate]).

system_def( entry_cond(modal_or_future & negation_in_finite), neg_mod_or-future_gate,
        [negative_modal_clause]).

system_def( entry_cond(not_modal_or_future & negation_in_finite),
         neg_notmod_or-future_gate,
        [negative_not_modal_or_future_clause]).

system_def( entry_cond(imperative & positive_clause), positive_imperative_gate,
        [positive_imperative]).

system_def( entry_cond(imperative & negative_clause), negative-imperative-gate,
        [negative_imperative]).

%--- process type
system_def( entry_cond(full), process_type1,
        [relational_clause,
         nonrelational_clause]).

system_def( entry_cond(nonrelational_clause), nonrelational_clause_type,
        [doing_clause,
         projecting_clause]).

system_def( entry_cond(relational_clause), relational_clause_type,
        [identifying_clause,
         attributive_clause,
         possessive_clause]).

system_def( entry_cond(attributive_clause), projective_attribution_sys,
        [projecting_attributive_clause,
         nonprojecting_attributive_clause]).

system_def( entry_cond([or,nonprojecting_attributive_clause,
        projecting_attributive_clause,possessive_clause,identifying_clause]),
        relational_object_resolution_gate,
        [relational_object_resolved]).

system_def( entry_cond([or,nonprojecting_attributive_clause,
                projecting_attributive_clause,
                possessive_clause,
                identifying_clause,
                active,
                medio_passive,
                recipio_passive,
                addressee_passive,
                indirect_agent_passive]), subject_resolved_gate,
        [subject_resolved]).

%--- subject insertion & type
```

```
system_def( entry_cond(finite & indicative & subject_resolved), subj_whness,
        [wh_subject,
         nonwh_subject]).

system_def( entry_cond(nonwh_subject),
        [nominal_subject,
         clausal_subject]).

%--- object insertion & type
system_def( entry_cond(nonrelational_clause & subject_resolved), subj_filler_type,
        [nonrelational_object_inserted,
         object_not_inserted]).

system_def( entry_cond([or,nonrelational_object_inserted,relational_object_resolved]),
        nonrelational_object_insertion,
        [object_inserted]).

%--- object whness
system_def( entry_cond(object_inserted), object_insertion,
        [wh_object,
         nonwh_object]).

%--- object form
system_def( entry_cond(nonwh_object & nonrelational_clause), obj_filler_type,
        [nominal_object,
         quoted_object,
         clausal_object]).

%--- agency insertion/form
system_def( entry_cond(passive), agency_insertion,
        [explicit_agency,
         implicit_agency]).

%--- secondary agency __ john told mary to eat chocolate
system_def( entry_cond(nonrelational_object_inserted & active), act_ind_agency,
        [active_indirect_agent,
         no_active_indirect_agent]).

%--- indir_agent whness
system_def( entry_cond(active_indirect_agent), indir_agency_whness,
        [wh_indir_agent,
         nonwh_indir_agent]).

%--- indir_agent gate
system_def( entry_cond([or, active_indirect_agent, indirect_agent_passive]),
        indirect_agen_gate,
        [indirect_agent]).

%--- recipiency/addressee expression
system_def( entry_cond(medio_passive \/ (active &
                    (object_not_inserted \/ no_active_indirect_agent))), recipiency_expr,
        [recipiency_expressed,
         recipiency_not_expressed]).

system_def( entry_cond(recipiency_expressed), recipiency_foregrounding,
        [forground_recipient,
         background_recipient]).

%--- circumstances
```

```
system_def( entry_cond(full), beneficiary_choice,
        [beneficial,
         nonbeneficial]).

system_def( entry_cond(full), primary_circ_sys,
        [primary_circumstance,
         no_primary_circumstance]).

system_def( entry_cond(primary_circumstance), thematic_primary_circ_sys,
        [nonthematic_primary_circumstance,
         thematic_primary_circumstance]).

system_def( entry_cond(primary_circumstance), nonwh_primary_circ_type,
        [clausal_primary_circumstance,
         prepositional_primary_circumstance]).

%--- circ_1 wh_ness
system_def( entry_cond(prepositional_primary_circumstance), wh_primary_circ_sys,
        [wh_primary_circumstance,
         nonwh_primary_circumstance]).

system_def( entry_cond(nonwh_primary_circumstance & thematic_primary_circumstance),
        thematic_nonwh_circ_sys,
        [thematic_nonwh_circumstance]).

%--- secondary circumstance
system_def( entry_cond(primary_circumstance), secondary_circ_sys,
        [secondary_circumstance,
         no_secondary_circumstance]).

system_def( entry_cond(full), clause_adverbial_sys,
        [adverbial_modifier,
         no_adverbial_modifier]).

%--- clause complex
system_def( entry_cond(clause_complex), complex_type,
        [paratactic_clause_complex,
         hypotactic_clause_complex]).

system_def( entry_cond(hypotactic_clause_complex), hypotactic_type,
        [if_complex]).

system_def( entry_cond(if_complex), if_complex_ordering,
        [if_then,
         then_if]).

system_def( entry_cond(clause_complex), clause_complex_mood,
        [interrogative_complex,
         declarative_complex,
         imperative_complex]).

%--- SOME GENERAL CLAUSAL SYSTEMS:
%--- sententiality _ is the unit punctuated?
system_def( entry_cond([or,clause_complex,finite,fragment,minor]), sententiality,
        [non_sentential,
         sentential]).

system_def( entry_cond(sentential & (declarative_complex \/
            declarative \/ declarative_fragment_or_minor)), decl_sent_gate,
```

```
        [declarative_sentence]).

system_def( entry_cond(sentential & (imperative_complex \/ imperative)), imp_sent_gate,
        [imperative_sentence]).

system_def( entry_cond(sentential & (interrogative_complex \/
            interrogative \/interrogative_fragment_or_minor)), inter_sent_gate,
        [interrogative_sentence]).

%--- clause conjunction
system_def( entry_cond(non_sentential), clause_conjunction,
        [conjuncted,
         non_conjuncted]).

system_def( entry_cond(conjuncted), conjunction_type,
        [if_conjunct,
         then_conjunct,
         that_conjunct,
         experiential_conjunct]).
```

A.1.2 Group Network

```
system_def( entry_cond(group), group_type,
        [nominal_group,
         prep_phrase,
         adjectival_group]).

system_def( entry_cond(nominal_group), nom_group_moderation,
        [moderated,
         non_moderated]).

system_def( entry_cond(nominal_group), nom_group_type,
        [nominal,
         pronominal]).

system_def( entry_cond(nominal_group), nominal_number,
        [sing_group,
         plur_group,
         substance_group]).

system_def( entry_cond(substance_group & nominal), substance_nominal_gate,
        [substance_nominal]).

system_def( entry_cond(nonqualified & (sing_group & nondeixis \/
proper_group)),
        classifying_group,
        [classifying_group]).

system_def( entry_cond(pronominal \/ proper_group \/ deixis \/
                       (common_group & (plur_group \/ substance_group))),
       participant_group,
      [participant_group]).

%--- common vs proper
system_def( entry_cond(nominal), nominal_type,
        [common_group,
         proper_group]).
```

```
system_def( entry_cond(common_group), classification,
        [classified,
         nonclassified]).

system_def( entry_cond(common_group), description,
        [epitheted,
         nonepitheted]).

system_def( entry_cond(common_group), numeration,
        [numerated,
         non_numerated]).

%--- deixis
system_def( entry_cond(common_group), deixis,
        [deixis,
         nondeixis]).

system_def( entry_cond(deixis), deixis_definiteness,
        [definite_deixis,
         indefinite_deixis]).

system_def( entry_cond(definite_deixis), deixis_possessiveness,
        [nonpossesive_deixis,
         possesive_deixis]).

system_def( entry_cond(nonpossesive_deixis), nonpossesive_deixis_type,
        [proximal_deixis,
         nonproximal_deixis]).

system_def( entry_cond((plur_group \/ substance_group) &
                       (nonpossesive_deixis \/ indefinite_deixis)), plur_deix,
         [plur_deixis]).

system_def( entry_cond(sing_group & (nonpossesive_deixis \/
indefinite_deixis)),
        sing_deix,
        [sing_deixis]).

%--- case
system_def( entry_cond(nominal_group), nominal_case,
        [nominative,
         accusative,
         genitive]).

system_def( entry_cond(pronominal &  nominative), pronom_nominal_group,
        [nom_pronom_group]).

system_def( entry_cond(pronominal & accusative), pronom_accusative_group,
        [acc_pronom_group]).

system_def( entry_cond(pronominal & genitive), gen_pronom_group,
        [gen_pronom_group]).

system_def( entry_cond(nominal & genitive), gentive_nominal_group,
        [genitive_nominal_group]).

%--- pronom_group definiteness
system_def( entry_cond(pronominal), pronm_group_definiteness,
```

```
        [definite_pronom_group,
         indefinite_pronom_group]).

system_def( entry_cond(definite_pronom_group), pronoun_group_whness,
        [nonwh_pronom_group,
         wh_pronom_group]).

system_def( entry_cond(nonwh_pronom_group), interactancy,
        [interactant_group,
         noninteractant_group]).

system_def( entry_cond(wh_pronom_group & threep_group & sing_group), wh_group_gate,
        [wh_group]).

system_def( entry_cond([or, wh_group, noninteractant_group]), pronom_group_type,
        [spatial_group,
         temporal_group,
         nonhuman_group,
         human_group]).

system_def( entry_cond(nominal_group), nom_group_qualified,
        [qualified,
         nonqualified]).

system_def( entry_cond(qualified), qualification_type,
        [pp_qualified,
         nonfin_qualified]).

system_def( entry_cond(pronominal), pronom_group_person,
        [onep_group,
         twop_group,
         threep_group]).

%--- prep phrases
system_def( entry_cond(prep_phrase), prep_phrase_type,
        [client_pp,
         recipient_pp,
         location_pp,
         origin_pp,
         possessor_pp,
         accompaniment_pp,
         destination_pp,
         matter_pp,
         duration_pp,
         agent_pp,
         materials_pp,
         other_prep_phrase]).

system_def( entry_cond([or, location_pp, destination_pp, duration_pp]), prep_ellipsis,
        [prep_present,
         prep_ellipsed]).

system_def( entry_cond(prep_ellipsed), prep_phrase_pronominalisation,
        [full_pp,
        pronom_pp]).

system_def( entry_cond(prep_phrase), prep_phrase_whness,
        [wh_pp,
         nonwh_pp]).
```

```
%--- adjectival groups
system_def( entry_cond(adjectival_group), projecting_adjectival_group,
        [projecting_adjectival_group,
         nonprojecting_adjectival_group ]).
```

A.2 SNAC Translations of the Grammar

SNAC translates systemic grammar networks (written in the notation of Section A.1) into network snapshots. A network snapshot forms the basis for the instantiation process of a tailored network. It tells the system how an ATMS dependency network is to be constructed and what should be an assumption, derived node, or a justification connecting different type of nodes. A snapshot also contains other information that helps the generator as well such as the accumulated preselection operations for a given function bundle. In the following, we explain the format of a network snapshot and how each part of it is used by the generation system. Then we give (in Sections A.2.2 and A.2.3) excerpts from the snapshots of the systemic grammar networks given in Section A.1.

A.2.1 The Format of a Network Snapshot

SNAC compiles conventional systemic grammars into what we call network snapshots. They are to be used by STAGE during generation. STAGE expects a snapshot to be in a certain format so that it can construct dependency networks correctly. A snapshot has the following format:

$snapshot(\langle rank\rangle, \langle choosers\rangle, \langle features\rangle, \langle systems\rangle, \langle bundles\rangle, \langle system\_justs\rangle, \langle order\_justs\rangle)$.

Next we explain the structure of each slot in a snapshot and how it helps the generator proceed in the process.

- $\langle rank\rangle$
  This slot can have one of two values: *clause* or *group*. It helps the system pick the right network to start with in response to the realisation of a new function just popped off the stack. As we will see below, the information provided by the preselection operations of a function bundle is used to determine the rank of the function. Once this bit is determined, the rank slot is used to pick the right network.

- $\langle choosers\rangle$
  This slot holds information about what choosers there are in a network and how they are related. The actual Prolog procedural code is kept separate though.

- $\langle features\rangle$
  This is a list of features. Initially, it contains all features in a network. After

tailoring, however, it contains only those features that have not been cut out from the network. During the instantiation of a network, these features will be represented by ATMS assumptions. This is necessary since these features will be referred to in the justifications slot (as will be shown below) and the ATMS requires a node to be created before it is used in any justification.

- $\langle systems \rangle$
  This is a list of system names that are also used in the network justifications. Therefore they need to be created first. That is why we have this list. Systems are represented by ATMS derived nodes. Note that initially this list contains all the systems of a network. During tailoring, the irrelevant systems are removed from this list.

- $\langle bundles \rangle$
  The generator also uses this list to create ATMS derived nodes. However, this slot contains other bits of information that help in the correct realisation of network's functions. Actually, the slot is a list of lists each containing the following information: $[function\_name, \langle conflated\_functions \rangle, \langle preselection\_operations \rangle]$ where $function\_name$ is a shorthand for naming the function bundle. The next item: $\langle conflated\_functions \rangle$ is a list of all conflated functions and which form this bundle such as the function bundle `[finite,pass]`. $\langle preselection\_operations \rangle$ is also a list of preselection operations for this function bundle. For example, the bundle `[finite,pass]` may have the preselection operations `[be-aux,past-verb]`. Initially, this slot contains all the function bundles in a network. During tailoring, some of the function bundles are removed based on the choosers behaviour.

- $\langle system\_justs \rangle$
  This is simply a list of justifications. Each justification is represented by a list of antecedents and one consequent as is the case with all ATMS justifications. For example, the list `[[active,f(1),s(2)],s(1)]` reprsents the justification $active \wedge f(1) \wedge s(2) \rightarrow s(1)$. The assertion of these justifications forms the dependency network. Since the system cannot differentiate between node types in a justification, the previous slots ($\langle features \rangle$, $\langle systems \rangle$, and $\langle bundles \rangle$) are used to create the correct node types (i.e. features are mapped to assumptions and both systems and functions to derived nodes). Initially, this list contains all the justifications of a network. During tailoring, some may be removed based on the responses of the choosers.

- $\langle order\_justs \rangle$
  This also is a list of justifications from a set of antecedents to a consequent. The antecedents are features and the consequent is an order node. So during network instantiation, the generator creates a derived node (i.e. the order node if not already created) and then asserts the justification itself. For example, the list `[[agency_expressed,non_thematic_circum],order(agent,circum)]` represents the justification

$$agency\_expressed \wedge non\_thematic\_circum \rightarrow order(agent, circum)$$

A.2.2 The Clause Network Snapshot

The following representation is a fragment of the clause network snapshot. It shows the different slots explained above.

```
snapshot(clause, [
        ...
        ...
        %--features of the network
        [clause,active,passive,agency_expr,agency_not_expr,
         circum_adjunct,non_thematic_circum,thematic_circum,
         nocircum_adjunct],

        %--systems of the network
        [s(1),s(2),s(3),s(4),s(5),s(6)],

        %--function bundle details
         [[f(1),[[subject,actor],[nom_group,nominative]]],
          [f(2),[[subject,goal],[nom_group,nominative]]],
          [f(3),[[finite,pred],[lexverb,past_verb,transitive]]],
          [f(4),[[finite,pass],[be_aux,past_verb]]],
          [f(5),[[pred,passc],[lexverb,transitive,enparticiple]]],
          [f(6),[[object,goal],[nom_group,accusative]]],
          [f(7),[[agency_marker],[by_prep]]],
          [f(8),[[agent,actor],[nom_group,accusative]]],
          [f(9),[[circum],[prep_phrase]]]
         ],

        %--main (system) justifications
        [[[clause,s(2)],s(1)],
         [[s(3),s(4)],s(2)],
         [[active,f(1),f(3),f(6)],s(3)],
         [[passive,s(5),f(2),f(4),f(5)],s(3)],
         [[agency_expr,f(7),f(8)],s(5)],
         [[agency_not_expr],s(5)],
         [[circum_adjunct,s(6),f(9)],s(4)],
         [[non_thematic_circum],s(6)],
         [[thematic_circum],s(6)],
         [[nocircum_adjunct],s(4)]
        ],

        %--ordering nodes and justs
        [ [[active],order(pred,object)],
          [[passive],order(pass,passc)],
          ...
          ...
        ] ]))
```

A.2.3 The Group Network Snapshot

The following representation is a fragment of the group network snapshot. It shows the different slots explained above.

```
snapshot(group, [
        ...
        %--features of the network
        [group,nominal_group,prep_phrase,nominal,pronominal,
         nominative,accusative,proper_group,common_group,
         onep_group,twop_group,threep_group,
         numerated,non_numerated,epitheted,nonepitheted,
         classified,nonclassified,client_pp,location_pp,time_pp],
        %--systems of the network
        [s(1),s(2),s(3),s(4),s(5),s(6),s(7),s(8),s(9),s(10),s(11),s(12)],
        %--function bundle details
         [[f(11),[[thing],[proper_noun]]],
          [f(12),[[thing],[pronominal,first_pers]]],
          [f(13),[[thing],[pronominal,second_pers]]],
          [f(14),[[thing],[pronominal,third_pers]]],
          [f(15),[[thing],[common_noun]]],
          [f(16),[[deictic],[definite]]],
          [f(17),[[prep],[client_prep]]],
          [f(18),[[prep],[location_prep]]],
          [f(19),[[prep],[time_prep]]],
          [f(20),[[numerator],[ordinal_adjective]]],
          [f(21),[[epithet],[nonordinal_adjective]]],
          [f(22),[[classifier],[common_noun]]],
          [f(23),[[pp_head],[nominal,accusative]]] ],
        %--main (system) justifications
        [[[group,s(2)],s(1)],
         [[nominal_group,s(3)],s(2)],
         [[prep_phrase,s(4),f(23)],s(2)],
         [[s(5),s(6)],s(3)],
         [[nominal,s(7)],s(5)],
         [[pronominal,s(8)],s(5)],
         [[proper_group,f(11)],s(7)],
         [[common_group,s(9),f(15),f(16)],s(7)],
         [[onep_group,f(12)],s(8)],
         [[twop_group,f(13)],s(8)],
         [[threep_group,f(14)],s(8)],
         [[s(10),s(11),s(12)],s(9)],
         [[numerated,f(20)],s(10)],
         [[non_numerated],s(10)],
         [[epitheted,f(21)],s(11)],
         [[nonepitheted],s(11)],
         [[classified,f(22)],s(12)],
         [[nonclassified],s(12)],
         [[client_pp,f(17)],s(4)],
         [[location_pp,f(18)],s(4)],
         [[time_pp,f(19)],s(4)]],
        %--ordering nodes and justs
        [ ...
          [[common_group],order(front,deictic)],
          [[common_group,numerated],order(deictic,numerative)],
          ...
          ...
          [[classified],order(classifier,thing)] ]]))
```

A.3 The Lexicon

Here we give examples of the related lexicon entries. As mentioned in Section 7.5.1 an entry in the lexicon might look like the following:

$$lexicon(lexeme, property_1, property_2, property_3, ...)$$

As typical lexical entry, consider the following Prolog representation for the lexeme *annihilate*.

```
lexicon([annihilate,anhlt,[1,1,1,1,0]],[destroy_proc,infinitive,lexverb,
        trans_verb]).
```

It consists of two lists. The first list has the lexeme itself, its root property, and its sound pattern (e.g. `[1,1,1,1,0]`). The length of lexical items is assumed here to be one although it can be any number. The second list contains the selectional restrictions of the lexeme: both denotational and grammatical restrictions (e.g. `[destroy_proc, sing_verb, third_person, past_verb, lexverb,trans_verb]`).

```
lexicon([will,o,[1,0]],[modal_aux]).
lexicon([be,o,[1,0]],[infinitive,be_aux]).
lexicon([was,o,[1,0]],[sing_verb,third_person,past_verb,be_aux]).
lexicon([is,x,[1,0]],[sing_verb,third_person,present_verb,be_aux]).
lexicon([destroyed,o,[1,0,0,1,0,0]],[destroy_proc,sing_verb,third_person,
        past_verb,lexverb,trans_verb]).
lexicon([annihilated,x,[1,1,1,1,0,1,0]],[destroy_proc,sing_verb,third_person,
        past_verb,lexverb,trans_verb]).
lexicon([destroyed,o,[1,0,0,1,0,0]],[destroy_proc,lexverb,trans_verb,
        enparticiple]).
lexicon([annihilated,x,[1,1,1,1,0,1,0]],[destroy_proc,lexverb,trans_verb,
        enparticiple]).
lexicon([destroys,o,[1,0,0,1,0,0]],[destroy_proc,sing_verb,third_person,
        present_verb,lexverb,trans_verb]).
lexicon([annihilates,x,[1,1,1,1,0,0]],[destroy_proc,sing_verb,third_person,
        present_verb,lexverb,trans_verb]).
lexicon([destroy,o,[1,0,0,1,0]],[destroy_proc,infinitive,lexverb,
        trans_verb]).
lexicon([annihilate,x,[1,1,1,1,0]],[destroy_proc,infinitive,lexverb,
        trans_verb]).
lexicon([ruthlessly,o,[1,0,0,1,0,1,0,0]],[ruthless_con,moderating_adverb]).
lexicon([mercilessly,o,[1,0,1,1,0,1,0,0]],[ruthless_con,moderating_adverb]).
lexicon([peacefully,o,[1,0,0,1,1,0,0]],[peaceful_con,moderating_adverb]).
lexicon([nonviolently,o,[1,0,1,1,1,0,0,1,0,0]],[peaceful_con,
        moderating_adverb]).
lexicon([first,o,[1,0,0,0]],[first,ordinal_adjective]).
lexicon([second,o,[1,1,0,0]],[second,ordinal_adjective]).
lexicon([last,o,[1,0,0,0]],[last,ordinal_adjective]).
lexicon([ruthless,o,[1,0,0,1,0]],[ruthless_con,nonordinal_adjective]).
lexicon([merciless,o,[1,0,1,1,0]],[ruthless_con,nonordinal_adjective]).
lexicon([working,o,[1,0,1,0,0]],[working,nonordinal_adjective]).
lexicon([active,x,[1,0,1,0]],[working,nonordinal_adjective]).
lexicon([functioning,o,[1,0,0,1,1,0,0]],[working,nonordinal_adjective]).
lexicon([lustrous,o,[1,0,0,1,0]],[shiny_con,nonordinal_adjective]).
lexicon([shiny,x,[1,0,1,0,0]],[shiny_con,nonordinal_adjective]).
```

```
lexicon([beautiful,o,[0,1,0,1,1,0]],[beautiful_con,nonordinal_adjective]).
lexicon([gorgeous,o,[1,0,1,0]],[beautiful_con,nonordinal_adjective]).
lexicon([warrior,o,[1,1,1,0]],[warrior,common_noun,singular_noun]).
lexicon([fighter,o,[1,0,1,0]],[warrior,common_noun,singular_noun]).
lexicon([city,o,[1,1,0]],[city,common_noun,singular_noun]).
lexicon([day,o,[1,0]],[day,common_noun,singular_noun]).
lexicon([week,o,[1,0,0]],[week,common_noun]).
lexicon([peace,o,[1,0,0]],[peace,common_noun]).
lexicon([rescue,o,[1,0,0,1]],[rescue,common_noun]).
lexicon([space,wl,[0,1,0]],[space,common_noun]).
lexicon([sky,x,[0,1,0]],[space,common_noun]).
lexicon([submarine,o,[1,0,1,1,0,0]],[submarine,common_noun,
        singular_noun]).
lexicon([subaquatic,o,[1,0,1,0,1,1,0]],[submarine,common_noun,
        singular_noun]).
...
lexicon([dartmaul,o,[1,0,0,1,0,0]],[dart_maul,proper_noun]).
lexicon([jitrax,o,[1,0,1,0]],[jitrax,proper_noun]).
lexicon([edinburgh,x,[1,1,0,1,1,0]],[edinburgh,proper_noun]).
lexicon([friday,x,[0,1,0,1,0,0]],[friday,proper_noun]).
...
lexicon([by,o,[1,0]],[by_prep]).
lexicon([for,o,[1,0]],[preposition,for_prep]).
lexicon([to,o,[1]],[preposition,destination_prep]).
lexicon([in,o,[1,0]],[preposition,location_prep]).
lexicon([from,x,[0,1,0]],[preposition,origin_prep]).
lexicon([with,o,[1,0]],[preposition,accompaniment_prep]).
lexicon([of,x,[1,0]],[preposition,matter_prep]).
lexicon([by,o,[1,0]],[preposition,agent_prep]).
lexicon([on,x,[1,0]],[preposition,on_prep]).
...
lexicon([i,x,[1,0]],[nominative_pronoun,speaker_pronoun]).
lexicon([me,o,[1,0]],[accusative_pronoun,speaker_pronoun]).
lexicon([you,o,[1,0]],[nominative_pronoun,listener_pronoun]).
lexicon([you,o,[1,0]],[accusative_pronoun,listener_pronoun]).
lexicon([he,o,[1,0]],[nominative_pronoun,noninteractant_pronoun]).
lexicon([him,o,[1,0]],[accusative_pronoun,noninteractant_pronoun]).
lexicon([the,x,[1]],[singular,definite_determiner]).
lexicon([a,x,[1]],[singular,indefinite_determiner]).
lexicon([the,x,[1]],[plural,definite_determiner]).
...
```

Appendix B

Generation Examples

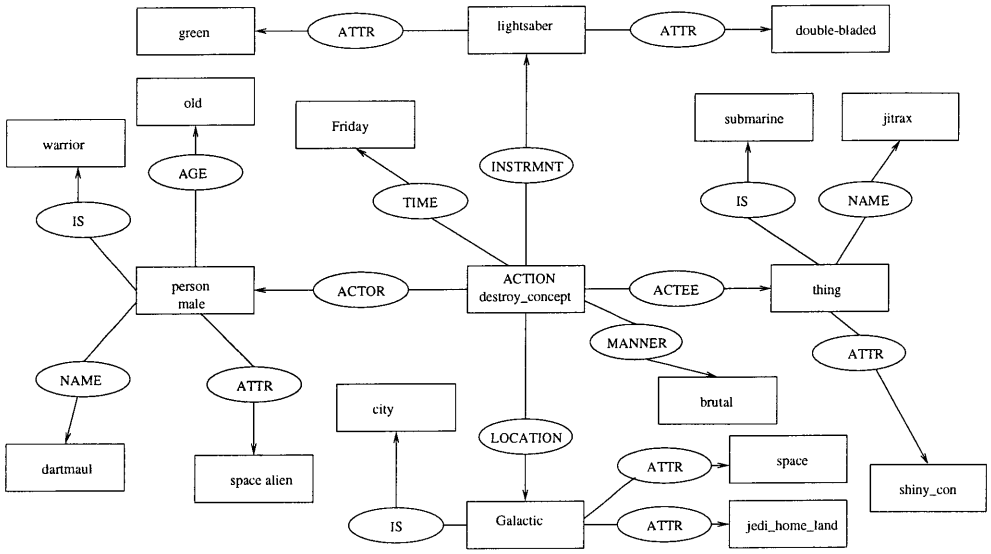
This appendix contains a trace of a complete generation example. It shows the steps that the system would go through. First we give the micro-semantic input in Section B.1. Then the sequence of activities carried out. Finally, we give sample sentences of the output. Note that the stylistic requirements assumed here are those for poetry metre as specified in Appendix C.2.

B.1 Micro-Semantic Input

The conceptual input is a structure representing the content that should be generated by the system. The input is a micro-semantic representation for a single sentence in terms of ideational, interactional, and textual meaning. Below, we show the micro-semantic input in two forms: as a diagram and as Prolog encoding. The conceptual input represents a material process and information about other roles such as the ACTOR, ACTEE, INSTRUMENT and some circumstantial details (TIME, LOCATION, MANNER of process, ... etc.). Note that we do not regard the semantic representation per se as an important part of our work in this thesis. It only gives a way of driving the generation process and providing a basis for choosers to work correctly.

```
sem(p(1),[process(p(2)),
    actor(p(3)),
    actee(p(4)),
    time(-),                    % + future, - past, or 0 present
    place(p(6)),
    manner(p(7)),
    date(p(8)),
    instrument(p(9)),
    polarity(+),                % + or -
    communic_goal(propose),     % initiate or ask
    voice(none)]).              % any voice preference?


sem(p(2),[proc_type(material),
    is(destroy_proc)]).
```




```
sem(p(3),[cname(dart_maul),
    is(male),
    attrib(none),
    category(warrior),
    order(first),
    description(ruthless_con),    %ie ruthless_concept
    class(space),
    origin(skytrax)]).


sem(p(4),[cname(jitrax),
    is(thing),
    attrib(none),
    category(submarine),
    order(none),
    description(shiny_con),
    class(rescue),
    origin(none)]).


sem(p(6),[cname(edinburgh),
    is(thing),
    attrib(none),
    category(city),
    order(second),
    description(beautiful_con),
    class(peace),
    origin(earth)]).


sem(p(7),[cname(brutal_con),      % manner semantic piece
    degree(high)]).               % {low,medium,high}


sem(p(8),[cname(friday),
    is(thing),
```

```
    attrib(none),
    category(day),
    order(last),
    description(working),
    class(week),
    origin(none)]).


sem(p(9),[cname(sharpenx),
    is(thing),
    attrib(none),
    category(lightsaber),
    order(only),
    description(double_sided),
    class(encounter),
    origin(none)]).
```

B.2 Example Choosers

Given the above systems and semantic input, we now show what some of the choosers do. We intend to show the relaxed behaviour of our choosers. As we will see next, not all choosers do not choose; some choosers can take informed decisions. It all depends on the semantic input and whether or not there are preferences specified by the user (e.g. preference of active voice over passive).

B.2.1 Clause Network Choosers

Here, we give some of the clause network choosers. Note that these choosers are encoded in Prolog. They choose with a specific function in mind. They have access to the syntactic function and the semantic entity it represents. The information of the concept-function association is stored in a FAT-like table.

- **active/passive chooser:** The strategy of this chooser is first to check whether there is any voice preference specified in the input. If there is one, then it chooses accordingly. Otherwise, if no ACTEE information is provided then the voice must be active. However, if the ACTOR part is missing, the voice must be passive. If none of the above applies the chooser concludes that it cannot make an informed decision as either way is possible. To avoid any unfavoured stylistic consequence of a single choice, it keeps both paths open (i.e. it does not choose). According to the above input, this chooser does not choose a particular feature.

- **modal/nonmodal chooser:** This chooser can always choose. It examines the time specification of the process. If time is in the future, then it chooses modal. Otherwise it chooses nonmodal. A subsequent chooser (which we do not show here), chooses between *past* and *present* if this chooser chose *nonmodal*. According to the above input, this chooser chooses *nonmodal*, and the related chooser chooses *past*.

- **circum-adjunct/no-circum-adj chooser:** If the input does not contain circumstantial information about the process (such as where and when exactly it took place), this chooser chooses the second feature *no-circum-adj*. If it chooses or keeps *circum-adjunct* a set of related choosers are triggered which we do not show here such as (secondary-circum/no-secondary-circum), (thematic-primary-circum/nonthematic-primary-circum), ... etc. They make further choices based on the input specification.
- **adverbial-modifier/nonadverbial-modifier chooser:** This chooser examines the process part of the input. If the process does not contain information about the manner of the action, it chooses nonadverbial-modifier; otherwise, it keeps both.

B.2.2 Group Network Choosers

Below are examples of the group network choosers.

- **nominal-group/prep-phrase/adjectival-group chooser:** This is an example of choosers that can always choose. The preselection operations related to the function to be realised dictate what feature to be selected.
- **proper-group/common-group chooser:** Depending on the details of the semantic entities, this chooser either chooses *common-group* if no proper name is specified or keeps both alternatives (i.e. does not choose).
- **count/mass chooser:** This chooser can always choose as a semantic entity is either count or mass. The same applies to choosers of other systems such as the singular/plural system and definite/indefinite system.
- **epitheted/nonepitheted chooser:** The strategy followed by this chooser (and other similar choosers, e.g. numerated-nonnumerated and classified-nonclassified) is to choose nonepitheted if the semantic input does not contain information which can be used as epithet for Thing; otherwise, both features of system are kept.

B.3 Generation Trace

We give here an example trace of the generator in the stylistics-aware mode. Because of space constraints, we cut those repetitive parts of the output. Where this is done, we indicate this by "`... etc.`".

```
| ?- gen_sem4.

### Read off network snapshots ...
### Read off translated system networks ...
    (system networks can also be translated on the fly)

### Associate the semantic input with the head of
```

```
    the rightmost clause-ranked expansion triangle
### Get the semantic input

### Associate the semantic input with the head of
    the rightmost clause-ranked expansion triangle

<<************************************************
<<************ BEGIN: constituent expansion ***********
... Fire choosers associated with systems of this netowrk
    to help circumscribe the superflous systems
     ... choose among: [modal,nonmodal]
         selected feature: nonmodal

     ... choose among: [present_clause,past_clause]
         selected feature: past_clause

     ... choose among: [active,passive]
         no choice made, all features kept

     ... choose among: [agency_expr,agency_not_expr]
         no choice made, all features kept

     ... choose among: [circum_adjunct,nocircum_adjunct]
         no choice made, all features kept

     ... choose among: [thematic_p_circum,non_thematic_p_circum]
         no choice made, all features kept

     ... choose among: [wh_p_circum,nonwh_p_circum]
         selected feature: nonwh_p_circum

     ... choose among: [secondary_circum,nosecondary_circum]
         no choice made, all features kept

... System network tailoring ... done

... Put the tailored network in the ATMS representation

                              |\
                              | \
                         f(1)|  \
                         f(2)|   \
                         f(4)|    \
                         f(6)|     \
                        f(10)|      \ clause
                        f(11)|      /
                        f(12)|     /
                        f(14)|    /
                        f(15)|   /
                        f(16)|  /
                              | /
                              |/

... For the Function: n(f(1),0)
     with the preselections: [nom_group,not_accusative]
     and representing the bundle: [subject,actor]
     do semantics association (if any), and push on stak
```

```
... For the Function: n(f(2),0)
     with the preselections: [nom_group,not_accusative]
     and representing the bundle: [subject,goal]
     do semantics association (if any), and push on stak


... For the Function: n(f(4),0)
     with the preselections: [sing_verb,third_person,past_verb,be_aux]
     and representing the bundle: [finite,pass]
     do semantics association (if any), and push on stak


... For the Function: n(f(6),0)
     with the preselections: [sing_verb,third_person,past_verb,lexverb,trans_verb]
     and representing the bundle: [finite,pred]
     do semantics association (if any), and push on stak


... For the Function: n(f(10),0)
     with the preselections: [lexverb,trans_verb,enparticiple]
     and representing the bundle: [passc,pred]
     do semantics association (if any), and push on stak


... For the Function: n(f(11),0)
     with the preselections: [nom_group,not_nominative]
     and representing the bundle: [agent,actor]
     do semantics association (if any), and push on stak


... For the Function: n(f(12),0)
     with the preselections: [by_prep]
     and representing the bundle: [agency_marker]
     do semantics association (if any), and push on stak


... For the Function: n(f(14),0)
     with the preselections: [prep_phrase,nonwh_pp]
     and representing the bundle: [circum1]
     do semantics association (if any), and push on stak


... For the Function: n(f(15),0)
     with the preselections: [prep_phrase]
     and representing the bundle: [circum2]
     do semantics association (if any), and push on stak


... For the Function: n(f(16),0)
     with the preselections: [nom_group,not_nominative]
     and representing the bundle: [object,goal]
     do semantics association (if any), and push on stak

... The Constituent has been dealt with and its triangle
    head is pushed onto the stack of delayed goal nodes

************ END: constituent expansion ************>>
**************************************************>>
```

```
### Pop next constituent from stack
    A constituent of GROUP rank: n(f(16),0)
     representing bundle: [object,goal]
     with  preselections: [nom_group,not_nominative]

<<**************************************************
<<************ BEGIN: constituent expansion ***********
... Fire choosers associated with systems of this netowrk
    to help circumscribe the superflous systems
     ... choose among: [nominal_group,prep_phrase,adjectival_group]
         selected feature: nominal_group

     ... choose among: [moderated,non_moderated]
         selected feature: non_moderated

     ... choose among: [qualified,nonqualified]
         selected feature: nonqualified

    ... choose among: [nominal,pronominal]
         selected feature: nominal

     ... choose among: [proper_group,common_group]
         no choice made, all features kept

     ... choose among: [count,mass]
         selected feature: count

     ... choose among: [singular,plural]
         selected feature: singular

     ... choose among: [deixis,nondeixis]
         selected feature: deixis

     ... choose among: [definite_deixis,indefinite_deixis]
         selected feature: definite_deixis

     ... choose among: [numerated,non_numerated]
         selected feature: non_numerated

     ... choose among: [epitheted,nonepitheted]
         no choice made, all features kept

     ... choose among: [classified,nonclassified]
         no choice made, all features kept

     ... choose among: [onep_group,twop_group,threep_group]
         no choice made, all features kept

     ... choose among: [nominative,accusative]
         selected feature: accusative

     ... choose among: [prep_ellipsed,prep_present]
         no choice made, all features kept

     ... choose among: [pronom_pp,full_pp]
         no choice made, all features kept

     ... choose among: [projecting_adjectival_group,nonprojecting_adjectival_group]
         no choice made, all features kept
```

```
... System network tailoring ... done

... Put the tailored network in the ATMS representation

                              |\
                              | \
                        f(35)|  \
                        f(36)|   \
                        f(41)|    ' n(f(16),0)
                        f(42)|   /
                        f(51)|  /
                              | /
                              |/

... For the Function: n(f(35),1)
     with the preselections: [nonordinal_adjective]
     and representing the bundle: [epithet]
     do semantics association (if any), and push on stak

     ... etc.

... The Constituent has been dealt with and its triangle
    head is pushed onto the stack of delayed goal nodes

************ END: constituent expansion ************>>
************************************************>>

### Pop next constituent from stack
    A constituent of WORD rank: n(f(51),1)
     representing bundle: [deictic]
     with  preselections: [singular,definite_determiner]

... fetch words from lexicon
     for the overall restrictions:
     [singular,definite_determiner]
      applicable lexemes: [[the,wl,[1]]]
.... Assert the lexeme stylistic properties, for: the


### Pop next constituent from stack
    A constituent of WORD rank: n(f(42),1)
     representing bundle: [thing]
     with  preselections: [common_noun,singular_noun]

... fetch words from lexicon
     for the overall restrictions:
     [submarine,common_noun,singular_noun]
     applicable lexemes: [[submarine,wl_sub,[1,0,1,1,0,0]]]
.... Assert the lexeme stylistic properties, for: submarine


### Pop next constituent from stack
    A constituent of WORD rank: n(f(41),1)
     representing bundle: [thing]
     with  preselections: [proper_noun]

... fetch words from lexicon
     for the overall restrictions:
```

```
     [jitrax,proper_noun]
     applicable lexemes: [[jitrax,wl,[1,0,1,0]]]
.... Assert the lexeme stylistic properties, for: jitrax


### Pop next constituent from stack
    A constituent of WORD rank: n(f(36),1)
     representing bundle: [classifier]
     with  preselections: [common_noun]

... fetch words from lexicon
     for the overall restrictions:
     [rescue,common_noun]
     applicable lexemes: [[rescue,wl,[1,0,0,1]]]
.... Assert the lexeme stylistic properties, for: rescue


### Pop next constituent from stack
    A constituent of WORD rank: n(f(35),1)
     representing bundle: [epithet]
     with  preselections: [nonordinal_adjective]

... fetch words from lexicon
     for the overall restrictions:
     [shiny_con,nonordinal_adjective]
     applicable lexemes: [[lustrous,wl,[1,0,0,1,0]]]
.... Assert the lexeme stylistic properties, for: lustrous


### Pop next constituent from stack
    A constituent of GROUP rank: n(f(15),0)
     representing bundle: [circum2]
     with  preselections: [prep_phrase]

<<**************************************************
<<************ BEGIN: constituent expansion ***********
... Fire choosers associated with systems of this netowrk
    to help circumscribe the superflous systems
     ... choose among: [nominal_group,prep_phrase,adjectival_group]
         selected feature: prep_phrase

         ... etc.

... System network tailoring ... done

... Put the tailored network in the ATMS representation

                              |\
                              | \
                        f(28)|  \ n(f(15),0)
                        f(37)|  /
                              | /
                              |/

... For the Function: n(f(28),2)
     with the preselections: [preposition,location_prep]
     and representing the bundle: [prep]
     do semantics association (if any), and push on stak
```

```
... For the Function: n(f(37),2)
     with the preselections: [accusative,nom_group]
     and representing the bundle: [pphead]
     do semantics association (if any), and push on stak

... The Constituent has been dealt with and its triangle
    head is pushed onto the stack of delayed goal nodes

************ END: constituent expansion ************>>
*****************************************************>>

### Pop next constituent from stack
    A constituent of GROUP rank: n(f(37),2)
     representing bundle: [pphead]
     with  preselections: [accusative,nom_group]

<<*****************************************************
<<************ BEGIN: constituent expansion ***********
... Fire choosers associated with systems of this netowrk
    to help circumscribe the superflous systems
     ... choose among: [nominal_group,prep_phrase,adjectival_group]
         selected feature: nominal_group

         ... etc.

... Put the tailored network in the ATMS representation

                              |\
                              | \
                         f(34)|  \
                         f(35)|   \
                         f(36)|    \ n(f(37),2)
                         f(41)|    /
                         f(42)|   /
                         f(51)|  /
                              | /
                              |/

... For the Function: n(f(34),3)
     with the preselections: [ordinal_adjective]
     and representing the bundle: [numerative]
     do semantics association (if any), and push on stak

     ... etc.

... The Constituent has been dealt with and its triangle
    head is pushed onto the stack of delayed goal nodes

************ END: constituent expansion ************>>
*****************************************************>>

### Pop next constituent from stack
    A constituent of WORD rank: n(f(51),3)
     representing bundle: [deictic]
     with  preselections: [singular,definite_determiner]

     ... etc.
```

```
### Pop next constituent from stack
    A constituent of GROUP rank: n(f(14),0)
     representing bundle: [circum1]
     with  preselections: [prep_phrase,nonwh_pp]

<<**************************************************
<<************ BEGIN: constituent expansion ***********
... Fire choosers associated with systems of this netowrk
    to help circumscribe the superflous systems
     ... choose among: [nominal_group,prep_phrase,adjectival_group]
         selected feature: prep_phrase

         ... etc.

... Put the tailored network in the ATMS representation

                              |\
                              | \
                        f(37)|  \ n(f(14),0)
                        f(38)|  /
                              | /
                              |/

... For the Function: n(f(37),4)
     with the preselections: [accusative,nom_group]
     and representing the bundle: [pphead]
     do semantics association (if any), and push on stak


... For the Function: n(f(38),4)
     with the preselections: [preposition,on_prep]
     and representing the bundle: [prep]
     do semantics association (if any), and push on stak

... The Constituent has been dealt with and its triangle
    head is pushed onto the stack of delayed goal nodes

************ END: constituent expansion ************>>
**************************************************>>

### Pop next constituent from stack
    A constituent of WORD rank: n(f(38),4)
     representing bundle: [prep]
     with  preselections: [preposition,on_prep]
     ...etc.

### Pop next constituent from stack
    A constituent of GROUP rank: n(f(37),4)
     representing bundle: [pphead]
     with  preselections: [accusative,nom_group]

<<**************************************************
<<************ BEGIN: constituent expansion ***********
... Fire choosers associated with systems of this netowrk
    to help circumscribe the superflous systems

    ...etc.

... Put the tailored network in the ATMS representation
```

```
                              |\
                              | \
                         f(34)|  \
                         f(35)|   \
                         f(36)|    \ n(f(37),4)
                         f(41)|    /
                         f(42)|   /
                         f(51)|  /
                              | /
                              |/
```

... etc.

... The Constituent has been dealt with and its triangle
head is pushed onto the stack of delayed goal nodes

********** END: constituent expansion **********>>
**>>

... etc.

Pop next constituent from stack
A constituent of WORD rank: n(f(35),5)
representing bundle: [epithet]
with preselections: [nonordinal_adjective]

... fetch words from lexicon
for the overall restrictions:
[working,nonordinal_adjective]
applicable lexemes: [[working,wl,[1,0,1,0,0]],
[active,wl,[1,0,1,0]],
[functioning,wl,[1,0,0,1,1,0,0]]]
.... Assert the lexeme stylistic properties, for: working
.... Assert the lexeme stylistic properties, for: active
.... Assert the lexeme stylistic properties, for: functioning

... etc.

... Put the tailored network in the ATMS representation

```
                              |\
                              | \
                         f(34)|  \
                         f(35)|   \
                         f(36)|    \ n(f(11),0)
                         f(41)|    /
                         f(42)|   /
                         f(51)|  /
                              | /
                              |/
```

... For the Function: n(f(34),6)
with the preselections: [ordinal_adjective]
and representing the bundle: [numerative]
do semantics association (if any), and push on stak

... etc.

```
... The Constituent has been dealt with and its triangle
    head is pushed onto the stack of delayed goal nodes

************ END: constituent expansion ************>>
*****************************************************>>

### Pop next constituent from stack
    A constituent of WORD rank: n(f(51),6)
     representing bundle: [deictic]
     with  preselections: [singular,definite_determiner]

     ... etc.

... fetch words from lexicon
     for the overall restrictions:
     [ruthless_con,nonordinal_adjective]
     applicable lexemes: [[ruthless,wl,[1,0,0,1,0]],
                          [merciless,wl,[1,0,1,1,0]]]
.... Assert the lexeme stylistic properties, for: ruthless
.... Assert the lexeme stylistic properties, for: merciless

    ... etc.

<<*****************************************************
<<************ BEGIN: constituent expansion ***********
... Fire choosers associated with systems of this netowrk
    to help circumscribe the superflous systems
     ... choose among: [nominal_group,prep_phrase,adjectival_group]
         selected feature: nominal_group

         ... etc.

... Put the tailored network in the ATMS representation

                              |\
                              | \
                         f(35)|  \
                         f(36)|   \
                         f(41)|    ' n(f(2),0)
                         f(42)|   /
                         f(51)|  /
                              | /
                              |/

... For the Function: n(f(35),7)
     with the preselections: [nonordinal_adjective]
     and representing the bundle: [epithet]
     do semantics association (if any), and push on stak

     ... etc.

... The Constituent has been dealt with and its triangle
    head is pushed onto the stack of delayed goal nodes

************ END: constituent expansion ************>>
*****************************************************>>

   ... etc.
```

```
### Pop next constituent from stack
    A constituent of WORD rank: n(f(35),7)
     representing bundle: [epithet]
     with  preselections: [nonordinal_adjective]

... fetch words from lexicon
     for the overall restrictions:
     [shiny_con,nonordinal_adjective]
     applicable lexemes: [[lustrous,wl,[1,0,0,1,0]]]
.... Assert the lexeme stylistic properties, for: lustrous


### Pop next constituent from stack
    A constituent of GROUP rank: n(f(1),0)
     representing bundle: [subject,actor]
     with  preselections: [nom_group,not_accusative]

<<**************************************************
<<************ BEGIN: constituent expansion ***********
... Fire choosers associated with systems of this netowrk
    to help circumscribe the superflous systems
     ... choose among: [nominal_group,prep_phrase,adjectival_group]
         selected feature: nominal_group

     ... etc.

... Put the tailored network in the ATMS representation

                              |\
                              | \
                         f(34)|  \
                         f(35)|   \
                         f(36)|    \ n(f(1),0)
                         f(41)|    /
                         f(42)|   /
                         f(51)|  /
                              | /
                              |/

     ... etc.

... The Constituent has been dealt with and its triangle
    head is pushed onto the stack of delayed goal nodes

************ END: constituent expansion ************>>
**************************************************>>

    ... etc.

### Pop next constituent from stack
    A constituent of WORD rank: n(f(34),8)
     representing bundle: [numerative]
     with  preselections: [ordinal_adjective]

... fetch words from lexicon
     for the overall restrictions:
     [first,ordinal_adjective]
     applicable lexemes: [[first,wl,[1,0,0,0]]]
.... Assert the lexeme stylistic properties, for: first
```

```
...

### STACK IS EMPTY, TIME  TO INTERFACE TRIANGLES
### IN ORDER TO GENERATE THE SURVIVING SENTENCES

||| POPPED OUT OF THE GOAL STACK |||   n(f(1),0)
||| DO STYLISTIC SIT-ACT FIRST, THEN INTERFACE TO PARENT

situation-action triggered
n(f(35),8) > n(f(42),8) &
pattern(n(f(35),8),[1,0,0,1,0]) &
pattern(n(f(42),8),[1,1,1,0]) --> false


situation-action triggered
n(f(51),8) > n(f(34),8) &
pattern(n(f(51),8),[1]) &
pattern(n(f(34),8),[1,0,0,0]) --> false

 ... etc.

situation-action triggered
Sequence([n(f(51),8),n(f(42),8)]) &
patterns:[pattern(n(f(51),8),[1]),[pattern(n(f(42),8),[1,1,1,0])]] &
 --> pattern(n(f(1),0),[1,1,1,1,0])


situation-action triggered
Sequence([n(f(41),8)]) &
patterns:[pattern(n(f(41),8),[1,0,0,1,0,0]),[]] &
 --> pattern(n(f(1),0),[1,0,0,1,0,0])


||| POPPED OUT OF THE GOAL STACK |||   n(f(2),0)
||| DO STYLISTIC SIT-ACT FIRST, THEN INTERFACE TO PARENT

situation-action triggered
n(f(35),7) > n(f(42),7) &
pattern(n(f(35),7),[1,0,0,1,0]) &
pattern(n(f(42),7),[1,0,1,1,0,0]) --> false

 ... etc.

situation-action triggered
Sequence([n(f(51),7),n(f(42),7)]) &
patterns:[pattern(n(f(51),7),[1]),[pattern(n(f(42),7),[1,0,1,1,0,0])]] &
 --> pattern(n(f(2),0),[1,1,0,1,1,0,0])


situation-action triggered
Sequence([n(f(41),7)]) &
patterns:[pattern(n(f(41),7),[1,0,1,0]),[]] &
 --> pattern(n(f(2),0),[1,0,1,0])


||| POPPED OUT OF THE GOAL STACK |||   n(f(11),0)
||| DO STYLISTIC SIT-ACT FIRST, THEN INTERFACE TO PARENT
```

```
situation-action triggered
n(f(35),6) > n(f(42),6) &
pattern(n(f(35),6),[1,0,1,1,0]) &
pattern(n(f(42),6),[1,1,1,0]) --> false

 ... etc.

situation-action triggered
Sequence([n(f(51),6),n(f(42),6)]) &
patterns:[pattern(n(f(51),6),[1]),[pattern(n(f(42),6),[1,1,1,0])]] &
 --> pattern(n(f(11),0),[1,1,1,1,0])


situation-action triggered
Sequence([n(f(41),6)]) &
patterns:[pattern(n(f(41),6),[1,0,0,1,0,0]),[]] &
 --> pattern(n(f(11),0),[1,0,0,1,0,0])


||| POPPED OUT OF THE GOAL STACK |||   n(f(37),4)
||| DO STYLISTIC SIT-ACT FIRST, THEN INTERFACE TO PARENT

situation-action triggered
n(f(51),5) > n(f(35),5) &
pattern(n(f(51),5),[1]) &
pattern(n(f(35),5),[1,0,1,0,0]) --> false

 ... etc.

situation-action triggered
n(f(34),5) > n(f(35),5) &
pattern(n(f(34),5),[1,0,0,0]) &
pattern(n(f(35),5),[1,0,0,1,1,0,0]) --> false

 ... etc.

situation-action triggered
Sequence([n(f(51),5),n(f(36),5),n(f(42),5)]) &
patterns:[pattern(n(f(51),5),[1]),[pattern(n(f(42),5),[1,0]),
pattern(n(f(36),5),[1,0,0])]] &
 --> pattern(n(f(37),4),[1,1,0,0,1,0])


situation-action triggered
Sequence([n(f(41),5)]) &
patterns:[pattern(n(f(41),5),[0,1,0,1,0,0]),[]] &
 --> pattern(n(f(37),4),[0,1,0,1,0,0])


||| POPPED OUT OF THE GOAL STACK |||   n(f(14),0)
||| DO STYLISTIC SIT-ACT FIRST, THEN INTERFACE TO PARENT

situation-action triggered
Sequence([n(f(38),4),n(f(37),4)]) &
patterns:[pattern(n(f(38),4),[1,0]),[pattern(n(f(37),4),[1,1,0])]] &
 --> pattern(n(f(14),0),[1,0,1,1,0])


situation-action triggered
```

```
Sequence([n(f(38),4),n(f(37),4)]) &
patterns:[pattern(n(f(38),4),[1,0]),[pattern(n(f(37),4),[1,1,0,0,1,0])]] &
 --> pattern(n(f(14),0),[1,0,1,1,0,0,1,0])

||| POPPED OUT OF THE GOAL STACK |||   n(f(37),2)
||| DO STYLISTIC SIT-ACT FIRST, THEN INTERFACE TO PARENT

  ... etc.

situation-action triggered
Sequence([n(f(51),3),n(f(42),3)]) &
patterns:[pattern(n(f(51),3),[1]),[pattern(n(f(42),3),[1,1,0])]] &
 --> pattern(n(f(37),2),[1,1,1,0])


situation-action triggered
Sequence([n(f(51),3),n(f(35),3),n(f(42),3)]) &
patterns:[pattern(n(f(51),3),[1]),[pattern(n(f(42),3),[1,1,0]),
pattern(n(f(35),3),[0,1,0,1,1,0])]] &
 --> pattern(n(f(37),2),[1,0,1,0,1,1,0,1,1,0])

  ...etc.

||| POPPED OUT OF THE GOAL STACK |||   n(f(16),0)
||| DO STYLISTIC SIT-ACT FIRST, THEN INTERFACE TO PARENT

situation-action triggered
n(f(35),1) > n(f(42),1) &
pattern(n(f(35),1),[1,0,0,1,0]) &
pattern(n(f(42),1),[1,0,1,1,0,0]) --> false

  ... etc.

situation-action triggered
Sequence([n(f(51),1),n(f(42),1)]) &
patterns:[pattern(n(f(51),1),[1]),[pattern(n(f(42),1),[1,0,1,1,0,0])]] &
 --> pattern(n(f(16),0),[1,1,0,1,1,0,0])


situation-action triggered
Sequence([n(f(41),1)]) &
patterns:[pattern(n(f(41),1),[1,0,1,0]),[]] &
 --> pattern(n(f(16),0),[1,0,1,0])


||| POPPED OUT OF THE GOAL STACK |||   topnode
||| DO STYLISTIC SIT-ACT FIRST, THEN INTERFACE TO PARENT

situation-action triggered
n(f(12),0) > n(f(11),0) &
pattern(n(f(12),0),[1,0]) &
pattern(n(f(11),0),[1,1,1,1,0]) --> false

  ... etc.

situation-action triggered
n(f(4),0) > n(f(10),0) &
pattern(n(f(4),0),[1,0]) &
pattern(n(f(10),0),[1,0,0,1,0,0]) --> false
```

```
situation-action triggered
n(f(4),0) > n(f(10),0) &
pattern(n(f(4),0),[1,0]) &
pattern(n(f(10),0),[1,1,1,1,0,1,0]) --> false

  ... etc.

situation-action triggered
Sequence([n(f(14),0),n(f(2),0),n(f(4),0),n(f(10),0),n(f(12),0),n(f(11),0)]) &
patterns:[pattern(n(f(14),0),[1,0,1,1,0]),[pattern(n(f(11),0),[1,0,0,1,0,0]),
pattern(n(f(12),0),[1,0]),pattern(n(f(10),0),[1,0,0,1,0,0]),
pattern(n(f(4),0),[1,0]),pattern(n(f(2),0),[1,1,0,1,1,0,0])]] &
 --> false

  ... etc.

situation-action triggered
Sequence([n(f(2),0),n(f(4),0),n(f(10),0),n(f(12),0),n(f(11),0),n(f(14),0)]) &
patterns:[pattern(n(f(2),0),[1,0,1,0]),[pattern(n(f(14),0),[1,0,0,1,0,1,0,0]),
pattern(n(f(11),0),[1,1,1,1,0]),pattern(n(f(12),0),[1,0]),
pattern(n(f(10),0),[1,0,0,1,0,0]),pattern(n(f(4),0),[1,0])]] &
 --> false

  ... etc.

situation-action triggered
Sequence([n(f(1),0),n(f(6),0),n(f(16),0)]) &
patterns:[pattern(n(f(1),0),[1,0,0,1,0,0]),[pattern(n(f(16),0),[1,0,1,0]),
pattern(n(f(6),0),[1,1,1,1,0,1,0])]] &
 --> false

|||| ALL CONSTITUENTS DEALT WITH ...
|||| LABEL OF THE TOPNODE IS

[[lexical(dartmaul,n(f(41),8)),n(nominal_group,8),n(nonqualified,8),n(nominal,8),
n(proper_group,8),n(non_moderated,8),n(nominative,8),lexical(annihilated,n(f(6),0)),
lexical(friday,n(f(41),5)),n(nominal_group,5),n(nonqualified,5),n(nominal,5),
n(proper_group,5),n(non_moderated,5),n(accusative,5),lexical(on,n(f(38),4)),
n(prep_phrase,4),n(other_prep_phrase,4),lexical(submarine,n(f(42),1)),
lexical(the,n(f(51),1)),n(nominal_group,1),n(nonqualified,1),n(nominal,1),
n(common_group,1),n(deixis,1),n(count,1),n(non_moderated,1),n(accusative,1),
n(nonclassified,1),n(nonepitheted,1),n(non_numerated,1),n(singular,1),
n(sing_def_deixis,1),n(definite_deixis,1),n(clause,0),n(nonmodal,0),
n(circum_adjunct,0),n(past_clause,0),n(non_thematic_p_circum,0),n(nonwh_p_circum,0),
n(nosecondary_circum,0),n(fin_pred_conf,0),n(active,0)]]

|||| TIME TO LINEARISE THE SURFACE FORMS ...

 dartmaul annihilated the submarine on friday.
yes
| ?-
```

Appendix C

Situation-Action Specifications

In this appendix we give the Prolog code of the surface stylistic constraints discussed in Section 7.9. At a higher level of abstraction they should be considered equivalent to the specifications given there. Because our objective was to test the incorporation of these constraints in the generation process and not to provide a user-friendly way of writing specifications, we encoded the SSC directly in Prolog. They still remain similar in many ways to the abstract specifications given in chapter 7. We also give the code for the utilities that these situations are using to check for certain circumstances if they hold or not.

C.1 Word Adjacency Constraints

```
sit_act(start,                    %-- definition SAS entry
         %-- no circumstances at the beginning
         [],
         %-- only some definitions
         [assert(lexprop(root))] ).


sit_act(lexicalise(X,F),          %-- lexicalisation SAS entry
        %--situation circumstances
          [lex_property(X,root,R)],
        %--corresponding actions
          [just([X],root(front(F,R))),
           just([X],root(end(F,R)))    ] ).


sit_act(order(x,y),I,_P) :-       %-- expansion SAS entry
    findall(X,(
        %--situation circumstances
        match(complete_seq(Sequence,I)),
        match(root(end(X,R))),   X=n(_,I),
        match(root(front(Y,R))), Y=n(_,I),
        sub_seq([X,Y],Sequence),
        %--corresponding actions
        newjust([complete_seq(Sequence,I),root(end(X,R)),
                     root(front(Y,R))], false)),L),fail.
```

```
sit_act(order(x,y),I,P) :-        %-- expansion SAS entry
        %----second situation:
        \+ I=0,
    findall((Xf,Xe),(
        %--situation circumstances
        match(complete_seq(Sequence,I)),
        match(root(front(Xf,Rf))), Xf=n(_,I),
        match(root(end(Xe,Re))),   Xe=n(_,I),
        firstelement(Xf,Sequence),
        lastelement(Xe,Sequence),
        %--corresponding actions
        just([complete_seq(Sequence,I),root(front(Xf,Rf))],
                    root(front(P,Rf))),

        just([complete_seq(Sequence,I),root(end(Xe,Re))],
                    root(end(P,Re))),
        ),L).
```

C.2 Poetry Metre Constraints

```
sit_act(start,                    %-- definition SAS entry
    %-- no circumstances at the beginning
    [],
    %-- only some definitions
    [assert(lexprop(sound)),
     assert(metre([1,0,0,1,0,0,1,1,1,1,0,1,0,1,1,0,1,1,0,0,1,0,0,1,0,1,0,0]))
    ]).


sit_act(lexicalise(X,F),          %-- lexicalisation SAS entry
        %--situation
          [lex_property(X,sound,S)],
        %--actions
          [ just([X],pattern(F,S)) ] ).


sit_act(order(x,y),I,_P) :-       %-- expansion SAS entry
    findall(X,(
       %--situation circumstances
       match(complete_seq(Sequence,I)),
       match(pattern(X,Px)),   X=n(_,I),
       match(pattern(Y,Py)), Y=n(_,I),
       sub_seq([X,Y],Sequence),
       \+ sub_metre([Px,Py]),
       %--corresponding actions
       newjust([complete_seq(Sequence,I),pattern(X,Px),
                                  pattern(Y,Py)], false)),L),fail.


sit_act(order(x,y),I,P) :-        %-- expansion SAS entry
    \+ I=0,    %ie not the top level
    findall((X,Px,Sequence),(
       %--situation circumstances
       match(complete_seq(Sequence,I)),
```

```
        firstelement(X,Sequence),
        match(pattern(X,Px)),
        complete_pattern(Sequence,pattern(X,Px),Pattern,PickedPatterns),
        sub_metre([Pattern]),
        %--corresponding actions
        just([complete_seq(Sequence,I),pattern(X,Px)|PickedPatterns],
                                        pattern(P,Pattern))),L),fail.


sit_act(order(x,y),I,_P) :-        %-- expansion SAS entry
    I=0,
    findall((X,Px,Sequence),(
        %--situation circumstances
        match(complete_seq(Sequence,I)),
        firstelement(X,Sequence),
        match(pattern(X,Px)),
        complete_pattern(Sequence,pattern(X,Px),Pattern,PickedPatterns),
        metre(Metre),
        \+ Pattern=Metre,
        %--corresponding actions
        just([complete_seq(Sequence,I),pattern(X,Px)|PickedPatterns],
                                                        false)),L).

%%%
%%% UTILITIES USED BY THE POETRY SAS ENTRIES.
%%%

%-----------------------------------------------------------------------
% sub_metre/1 takes a list of smaller patterns, concatenates them
% into one pattern and checks if they can occur anywhere in the metre.
%-----------------------------------------------------------------------
sub_metre(PatternList) :-
        concatenate_patterns(PatternList,[],SinglePattern),
        metre(M),
        possible_pattern(SinglePattern,M).

concatenate_patterns([],L,L).
concatenate_patterns([L|T],SoFar,Final) :-
        append(SoFar,L,SoFar2),
        concatenate_patterns(T,SoFar2,Final).

possible_pattern(Pattern,Metre) :-
        prefix(Pattern,Metre).
possible_pattern(Pattern,[_|RemainingOfMetre]) :-
        possible_pattern(Pattern,RemainingOfMetre).



%-----------------------------------------------------------------------
% complete_pattern/4 takes a complete Sequence of functions
% (e.g. [front, deictic, thing, end]) and returns the resulting
% Pattern and the subpatterns participating in building it.
%-----------------------------------------------------------------------
complete_pattern(Sequence,pattern(X,Px),Pattern,PickedPatterns) :-
        Sequence=[X|Remainder],
        form_pattern(Remainder,Px,Pattern,[],PickedPatterns).

form_pattern([],Patt,Patt,Picked,Picked).
form_pattern([Fun|T],SoFar,FinalPattern,SoFarPicked,FinalPicked) :-
```

```
        match(pattern(Fun,Pfun)),
        append(SoFar,Pfun,SoFar2),
        SoFarPicked2=[pattern(Fun,Pfun)|SoFarPicked],
        form_pattern(T,SoFar2,FinalPattern,SoFarPicked2,FinalPicked).
```

C.3 Text Size Constraints

```
sit_act(start,                    %-- definition SAS entry
        %-- no circumstances at the beginning
          [],
        %-- only some definitions
          [assert(lexprop(size)),
           assert(maxsize(4)) ] ).


sit_act(lexicalise(X,F),          %-- lexicalisation SAS entry
        %--situation
          [lex_property(X,size,S)],
        %--actions
          [ just([X],size(F,S)) ] ).


sit_act(order(x,y),I,_P) :-       %-- expansion SAS entry
    findall(X,(
        %--situation circumstances
        match(complete_seq(Sequence,I)),
        match(size(X,Sx)),   X=n(_,I),
        match(size(Y,Sy)),   Y=n(_,I),
        sub_seq([X,Y],Sequence),
        maxsize(Max),
        Sum is Sx+Sy,
        Sum > Max,
        %--corresponding actions
        newjust([complete_seq(Sequence,I),size(X,Sx),
                                        size(Y,Sy)], false)),L).


sit_act(order(x,y),I,P) :-        %-- expansion SAS entry
    \+ I=0,    %ie not the top level
    findall((X,Sx,Sequence),(
        %--situation circumstances
        match(complete_seq(Sequence,I)),
        firstelement(X,Sequence),
        match(size(X,Sx)),
        maxsize(Max),
        sum_sizes(Sequence,size(X,Sx),Sum,PickedSizes),
        Sum =< Max,
        %--corresponding actions
        just([complete_seq(Sequence,I),size(X,Sx)|PickedSizes],
                                          size(P,Sum))),L).


sit_act(order(x,y),I,_P) :-       %-- expansion SAS entry
    findall((X,Sx,Sequence),(
        %--situation circumstances
        match(complete_seq(Sequence,I)),
```

```
        firstelement(X,Sequence),
        match(size(X,Sx)),
        sum_sizes(Sequence,size(X,Sx),Sum,PickedSizes),
        maxsize(Max),
        Sum > Max,
        %--corresponding actions
        just([complete_seq(Sequence,I),size(X,Sx)|PickedSizes],
                                                        false)),L).

%%%
%%% UTILITIES TEXT SIZE CONSTRAINTS
%%%

%-----------------------------------------------------------------------
% sum_sizes/4 takes a Sequence of functions (e.g. [front, deictic,
% thing, end]) and returns the total Sum of sizes and the size nodes
% contributing to Sum so that they can be used in justification
% antecedents.
%-----------------------------------------------------------------------
sum_sizes(Sequence,size(X,Sx),Sum,PickedSizes) :-
        Sequence=[X|Remainder],
        sum_size(Remainder,Sx,Sum,[],PickedSizes).

sum_size([],Sum,Sum,Picked,Picked).
sum_size([Fun|T],SoFar,FinalSum,SoFarPicked,FinalPicked) :-
        match(size(Fun,Sfun)),
        SoFar2 is SoFar+Sfun,
        SoFarPicked2=[size(Fun,Sfun)|SoFarPicked],
        sum_size(T,SoFar2,FinalSum,SoFarPicked2,FinalPicked).
```