Human Media Interaction Group
Department of Electrical Engineering,
Mathematics and Computer Science
University of Twente, Enschede
The Netherlands

Narration for Virtual Storytelling

Nanda Slabbers
March, 2006

Graduation committee
dr. M. Theune
dr. ir. H. J. A. op den Akker
prof. dr. ir. A. Nijholt

Samenvatting

De Virtual Storyteller is een systeem dat automatisch sprookjes kan genereren en is geïmplementeerd in Java. Het systeem bestaat uit een aantal Characters agents, een World agent en een Plot agent, die gezamenlijk een plot genereren. Dit plot wordt vervolgens doorgegeven aan de Narrator agent, die het plot om moet zetten naar tekst in natuurlijke taal. Eerdere versies van het systeem gebruikten templates voor de natuurlijke taal generatie, wat veelal resulteerde in monotone teksten. Dit document beschrijft het ontwerp en de implementatie van een nieuwe Narrator agent die gebruik maakt van meer geavanceerde technieken op het gebied van natuurlijke taal generatie.

Om de module te ontwikkelen is eerst een literatuurstudie uitgevoerd, die zich richtte op het schrijfproces van mensen, natuurlijke taal generatie en bestaande systemen die automatisch verhalen kunnen genereren. Vervolgens is een aantal met de hand geschreven sprookjes geanalyseerd om erachter te komen waarin dit soort verhalen verschillen van de verhalen zoals ze door de Virtual Storyteller gegenereerd worden.

Op grond van de resultaten van het literatuuronderzoek en de analyse van met de hand geschreven verhalen is de Narrator agent ontworpen. Zoals veel andere natuurlijke taal generatie systemen is de module geïmplementeerd als een pipeline bestaande uit een Document Planner, een Microplanner en een Surface Realizer. De Document Planner is verantwoordelijk voor het selecteren van de relevante informatie uit het plot, het ordenen van deze informatie en het bepalen hoe de verschillende elementen met elkaar verbonden moeten worden. De Microplanner zet vervolgens elk informatie element in een dependency tree om, waarbij eerst een sentence plan gegenereerd wordt en vervolgens de eerste stap van de lexicalizatie uitgevoerd wordt. De Surface Realizer voert eerst syntactische aggregatie uit door verschillende dependency trees te combineren tot één nieuwe boom. Vervolgens genereert de module geschikte referring expressions, waarbij de laatste stap van de lexicalizatie uitgevoerd wordt. Tenslotte worden de dependency trees omgezet in de uiteindelijke tekst in natuurlijke taal, waarbij morphologie en orthografie worden toegepast.

De Narrator agent die in dit document beschreven wordt, is in staat om een plot structuur zoals die door de overige componenten van het systeem gegenereerd wordt, om te zetten in tekst in natuurlijke taal. Op dit moment zijn de gegenereerde teksten nog vrij eenvoudig, maar met slechts een aantal kleine wijzigingen worden de teksten al gevarieerder en daarmee aantrekkelijker voor de lezer. In het laatste hoofdstuk wordt daarom een aantal suggesties gegeven om het systeem te verbeteren.

Abstract

The Virtual Storyteller is an agent-based system implemented in Java which can automatically generate simple fairy-tales. The system consists of several Character agents, a World agent and a Plot agent which together generate a plot. This plot is then passed to the Narrator agent which has to convert the plot into text in natural language. Earlier versions of the system used templates for the natural language generation process which resulted in rather monotonous texts. This document describes the design and implementation of a new Narrator agent which uses more sophisticated natural language generation techniques.

In order to develop the module, first of all a literature study has been carried out which focussed on the human writing process, natural language generation and other existing systems that automatically generate stories. Next a number of human-written fairy-tales have been analyzed in order to find out how human-written stories differ from the stories generated by the template-based Virtual Storyteller.

Using the results of the literature study and the analysis of human-written fairy-tales the Narrator agent has been designed. Like many other natural language generation systems, the module has been implemented as a pipeline consisting of a Document Planner, a Microplanner and a Surface Realizer. The Document Planner is responsible for selecting the relevant facts from the plot, ordering them and deciding how they should be related to each other. The Microplanner then turns each fact into a dependency tree by first generating a sentence plan and then performing the first part of the lexicalization. The Surface Realizer first of all performs syntactic aggregation by combining several dependency trees into one. Then it generates appropriate referring expressions while performing the final part of the lexicalization. Finally the dependency trees are turned into the output text in natural language, with the application of morphology and orthography.

The Narrator agent described in this document is able to convert a plot structure generated by the other components of the system into text in natural language. At this moment the generated output texts are rather simple, but with a few changes the texts become more varied and more attractive for the reader. Therefore the final chapter gives some suggestions for improving the system.

Contents

1 Introduction **11**
- 1.1 The Virtual Storyteller . . . 11
- 1.2 Overview of the Project . . . 12

2 Literature **13**
- 2.1 Human Writing . . . 13
  - 2.1.1 The steps of the writing process . . . 13
  - 2.1.2 Elements of successful narratives . . . 14
  - 2.1.3 Language use in narratives . . . 14
- 2.2 Natural Language Generation . . . 15
  - 2.2.1 Tasks of a natural language generation system . . . 16
  - 2.2.2 Architecture of a natural language generation system . . . 16
- 2.3 Existing Work on Storytelling . . . 16
  - 2.3.1 GESTER . . . 17
  - 2.3.2 Gervas's system . . . 18
  - 2.3.3 StoryBook . . . 20

3 Virtual Storyteller **24**
- 3.1 Architecture of the Virtual Storyteller . . . 24
- 3.2 Surface Realizer . . . 25
  - 3.2.1 Dependency trees . . . 25
  - 3.2.2 Rhetorical relations . . . 26
  - 3.2.3 Tasks of the Surface Realizer . . . 26
- 3.3 Results Earlier Versions . . . 28
  - 3.3.1 Version 1 . . . 29
  - 3.3.2 Version 2 . . . 29

4 Analysis of Human-Written Fairy-Tales **31**
- 4.1 Questions . . . 32
- 4.2 Input and Structure . . . 32
  - 4.2.1 Which information is available, and where? . . . 33
  - 4.2.2 Conclusions input and structure . . . 36
- 4.3 Sentence and Paragraph Structures . . . 36
  - 4.3.1 Used rhetorical relations within sentences . . . 36
  - 4.3.2 Number of clauses combined into one sentence . . . 37
  - 4.3.3 Used rhetorical relations between sentences . . . 38
  - 4.3.4 Paragraph structures . . . 38
  - 4.3.5 New syntactic constructions . . . 39
  - 4.3.6 Conclusions sentence and paragraph structures . . . 40
- 4.4 Referring Expressions . . . 40
  - 4.4.1 'Koning Lijsterbaard' . . . 40
  - 4.4.2 'De Gelaarsde Kat' . . . 41

4.4.3 'De ring van de koningsdochter' . . . 41
4.4.4 'De jongen die lezen en schrijven leerde' . . . 42
4.4.5 Conclusions referring expressions . . . 42
4.5 Rhetorical Structure Theory . . . 42
4.5.1 Example 1: Taken from 'Koning Lijsterbaard' . . . 43
4.5.2 Example 2: Taken from 'De jongen die lezen en schrijven leerde' . . . 43
4.5.3 Conclusions Rhetorical Structure Theory . . . 44
4.6 Conclusions . . . 45

5 Project Goals and Architecture **46**
5.1 Detailed Project Goals . . . 46
5.2 Global Architecture of the Narrator Agent . . . 47
5.3 Inputs and Outputs . . . 48
5.3.1 Inputs and outputs in a standard NLG system . . . 48
5.3.2 Inputs to the Narrator . . . 50
5.3.3 Document plans (output of the Document Planner) . . . 53
5.3.4 Rhetorical dependency graphs (output of the Microplanner) . . . 53
5.3.5 Output of the Narrator . . . 53

6 Document Planner **54**
6.1 Architecture . . . 54
6.2 Initial Document Plan Builder . . . 54
6.2.1 Retrieving all information about the plot elements . . . 54
6.2.2 Converting the fabula into an initial document plan . . . 58
6.2.3 Performing the NLG tasks . . . 64
6.3 Background Information Supplier . . . 66
6.3.1 Adding background information . . . 66
6.3.2 Discourse history . . . 66
6.4 State Transformer . . . 68
6.5 Mood Creator . . . 69
6.6 Branch Remover . . . 70
6.7 Example . . . 73
6.7.1 Initial Document Plan Builder . . . 73
6.7.2 Background Information Supplier . . . 74
6.7.3 State Transformer . . . 75
6.7.4 Mood Creator . . . 75
6.7.5 Branch Remover . . . 75

7 Microplanner **76**
7.1 Architecture . . . 76
7.2 Sentence Plan Generator . . . 77
7.2.1 Actions and events . . . 77
7.2.2 Internal states . . . 77
7.2.3 Perceptions and beliefs . . . 79
7.2.4 Goals . . . 79
7.2.5 Settings . . . 80
7.2.6 Adding modifiers . . . 80
7.3 Lexicalizer . . . 80
7.3.1 Lexicon . . . 80
7.3.2 Word History . . . 81
7.3.3 Lexical Chooser . . . 81
7.4 Example . . . 81
7.4.1 Sentence Plan Generator . . . 81
7.4.2 Lexicalizer . . . 82

8 Surface Realizer **83**
8.1 Architecture . . . 83
8.2 Syntactic Aggregator . . . 84
8.2.1 Relation Transformer . . . 84
8.2.2 Conjunctor . . . 86
8.2.3 Elliptor . . . 87
8.3 Referring Expression Generator . . . 88
8.3.1 Existing algorithms for referring expression generation . . . 88
8.3.2 Algorithm for referring expression generation in the Virtual Storyteller . . . 90
8.3.3 Inference-based referring expression generation . . . 95
8.3.4 Results of the referring expression generation . . . 100
8.4 Surface Form Generator . . . 102
8.4.1 Ordering the nodes in a dependency tree . . . 102
8.4.2 Morphology . . . 102
8.4.3 Orthography . . . 103
8.5 Relative Clauses . . . 103
8.5.1 Types of relative clauses . . . 103
8.5.2 Algorithm for generating relative clauses . . . 104
8.5.3 Example execution of the algorithm . . . 106
8.5.4 Adding a library class . . . 106
8.5.5 Adding punctuation . . . 106
8.6 Example . . . 107
8.6.1 Syntactic Aggregator . . . 107
8.6.2 Referring Expression Generator . . . 108
8.6.3 Surface Form Generator . . . 109

9 Results and Evaluation **110**
9.1 Example 1: Knight Story . . . 110
9.1.1 Using a modelled fabula structure . . . 110
9.1.2 Using modelled document plans . . . 112
9.1.3 Evaluation of the Knight story . . . 113
9.2 Example 2: Story from Chapter 3 . . . 118
9.3 Example 3: Using the Mood Creator and State Transformer . . . 120

10 Conclusions and Future Work **122**
10.1 Conclusions . . . 122
10.2 Suggestions for Future Work . . . 123
10.2.1 Extending the input with more information . . . 123
10.2.2 Extending existing elements of the Narrator . . . 124
10.2.3 Adding new functionalities to the Narrator . . . 125
10.2.4 Suggestions for completely new projects . . . 126

A Human-Written Stories for Analysis **130**
A.1 Koning Lijsterbaard . . . 130
A.1.1 Story . . . 130
A.1.2 Rhetorical relations . . . 131
A.1.3 Referring expressions . . . 132
A.2 De gelaarsde kat . . . 133
A.2.1 Story . . . 133
A.2.2 Rhetorical relations . . . 134
A.2.3 Referring expressions . . . 135
A.3 De ring van de koningsdochter . . . 135
A.3.1 Story . . . 135
A.3.2 Rhetorical relations . . . 136

A.3.3 Referring expressions . . . 137
A.4 De jongen die lezen en schrijven leerde . . . 137
A.4.1 Story . . . 137
A.4.2 Rhetorical relations . . . 138
A.4.3 Referring expressions . . . 139

B The Plop Example **140**
B.1 Fabula Structure . . . 140
B.2 Dependency Trees after Lexicalization . . . 140
B.3 Dependency Trees after Referring Expression Generation . . . 140

C Pronominalization **145**
C.1 Existing Theories and Algorithms . . . 145
C.1.1 Centering Theory . . . 145
C.1.2 Thread-based . . . 146
C.1.3 Local focus . . . 147
C.1.4 Parallelism . . . 147
C.1.5 Plausibility . . . 147
C.1.6 Rhetorical relations . . . 148
C.1.7 Salience-based . . . 149
C.1.8 Probabilistic model . . . 149
C.2 Combination of the Algorithms . . . 150
C.2.1 Advantages of the algorithms . . . 150
C.2.2 The combined algorithm . . . 150

List of Figures

2.1 Architecture of a natural language generation system . . . 17
2.2 Execution of the stages in Gervas' system . . . 19
2.3 Author's architecture . . . 21
2.4 Example of StoryBook's output . . . 23

3.1 Architecture of the Virtual Storyteller, taken from [Swa06] . . . 25
3.2 Example of a dependency tree, taken from [vdBBD+02] . . . 26
3.3 Example story before performing syntactic aggregation . . . 29
3.4 Example story after performing aggregation . . . 30

4.1 Example tree for the beginning of the first story . . . 43
4.2 Example tree for the beginning of the third story . . . 44

5.1 Global architecture of the Narrator agent . . . 47
5.2 Example abstract syntactic structure, taken from [RD00] . . . 49
5.3 Example lexicalized case frames, taken from [RD00] . . . 49
5.4 Fabula structure, taken from [Swa06] . . . 50
5.5 Example of an actual fabula structure, taken from [Swa06] . . . 51
5.6 Examples of plot elements represented in OWL . . . 52
5.7 Example of a document plan . . . 53

6.1 Architecture of the Document Planner . . . 55
6.2 Examples of plot elements . . . 56
6.3 Examples of sub plot elements . . . 57
6.4 Example of a fabula containing a plot element that causes two other plot elements . . . 58
6.5 Example of a fabula containing two plot elements that together cause another plot element . . . 59
6.6 Situations in CreateDocumentPlan(curr, tree, forward) . . . 61
6.7 Example of converting the fabula from figure 6.4 . . . 64
6.8 Example of converting the fabula from figure 6.5 . . . 65
6.9 Example of removing a long branch from a document plan . . . 71
6.10 Algorithm for removing a long branch from a document plan . . . 72
6.11 Example of a fabula structure for the Plop story . . . 73
6.12 Output of the Initial Document Plan Builder for the Plop story . . . 74
6.13 Output of the Background Information Supplier for the Plop story . . . 74

7.1 Architecture of the Microplanner . . . 76
7.2 Templates for each type of plot element . . . 78
7.3 Output of the Sentence Plan Generator for the Plop story . . . 82
7.4 Output of the Lexicalizer for the Plop story . . . 82

8.1 Architecture of the Surface Realizer . . . 83
8.2 Example of a rhetorical dependency graph . . . 85
8.3 Example of a rhetorical dependency graph after combining some trees . . . 85

8.4 Example of generating a relative clause . . . 107
8.5 Output of the Syntactic Aggregator for the Plop story . . . 108
8.6 Output of the Referring Expression Generator for the Plop story . . . 109
8.7 Output of the Surface Form Generator for the Plop story . . . 109

9.1 Modelled fabula structure for the Knight story . . . 110
9.2 Example output without performing aggregation and referring expression generation . . . 111
9.3 Example output after performing aggregation and referring expression generation . . . 111
9.4 Modelled initial document plans for the Knight story . . . 113
9.5 Example output with modelled document plans . . . 113
9.6 Modelled initial document plans for the story from Chapter 3 . . . 118
9.7 Example output of the story from Chapter 3 . . . 119
9.8 Modelled document plans for the example using the Mood Creator and State Transformer . . . 120
9.9 Example output of the example without using the Mood Creator and State Transformer . . . 121
9.10 Example output of the example using the Mood Creator and State Transformer . . . 121

B.1 Example of the fabula structure for the Plop story in OWL representation . . . 141
B.2 Generated dependency tree for the plot element describing the dwarf's name . . . 142
B.3 Generated dependency tree for the plot element Hungry . . . 142
B.4 Generated dependency tree for the plot element Belief . . . 142
B.5 Generated dependency tree for the plot element Goal . . . 142
B.6 Generated dependency tree for the plot element Take apple . . . 143
B.7 Generated dependency tree for the plot element Eat apple . . . 143
B.8 Generated dependency tree for "Hij had honger en dacht dat er een appel in een huis lag." . . . 143
B.9 Generated dependency tree for "Daarom wilde hij de appel eten." . . . 144
B.10 Generated dependency tree for "Nadat kabouter Plop de appel op had gepakt, at hij hem." . . . 144

List of Tables

4.1 The fourth paragraph of 'Koning Lijsterbaard' 33
4.2 The second paragraph of 'De gelaarsde kat' 34
4.3 The second paragraph of 'De ring van de koningsdochter' 35
4.4 The sixth paragraph of 'De jongen die lezen en schrijven leerde' 35

7.1 Examples of lexicalizations of all types of goals 79

8.1 Examples of discourse status and hearer status 95
8.2 Examples of inflection of adjectives 103

A.1 Rhetorical relations used in 'Koning Lijsterbaard' 131
A.2 Referring expressions used in 'Koning Lijsterbaard' 132
A.3 Rhetorical relations used in 'De gelaarsde kat' 134
A.4 Referring expressions used in 'De gelaarsde kat' 135
A.5 Rhetorical relations used in 'De ring van de koningsdochter' 136
A.6 Referring expressions used in 'De ring van de koningsdochter' 137
A.7 Rhetorical relations used in 'De jongen die lezen en schrijven leerde' 138
A.8 Referring expressions used in 'De jongen die lezen en schrijven leerde' 139

Preface

During the course of the study Computer Science I became fascinated by the field of natural language generation. Therefore this was an obvious subject to choose for my graduation project. I made an appointment with Mariët Theune, who was involved in a number of projects dealing with computational linguistics. After she had suggested several projects I came to the conclusion that I found each suggestion interesting, but I was not sure if I would enjoy working on a single one of them for seven months in a row. Since I didn't know which one to choose, I finally told her that I enjoyed an earlier project because it was so 'blij!' ('joyous'). Immediately she answered 'Oh, but then you may like the Virtual Storyteller!' and started to tell me about the project enthusiastically. Soon afterwards I knew this would be the perfect project for me, and now, eight months later, I haven't regretted it once.

During the project I have had assistance from several people, who deserve my gratitude. First of all, I would like to thank Mariët Theune for meeting me regularly. Furthermore I would like to thank her for her helpful comments on the things I wrote, but most of all for her continued enthusiasm which motivated me enormously.

I would also like to thank Rieks op den Akker for helping me design the architecture of the Narrator agent and for his useful remarks on this document. Furthermore I would like to thank Anton Nijholt for his comments on the final version of this document and its structure. I would also like to thank Ivo Swartjes for answering the countless questions I have asked about the fabula structure and Joline Mreijen for her advice on linguistics. Finally I would like to thank my family and friends for their interest in my progress and for supporting me.

Chapter 1

Introduction

For centuries storytelling has been an important form of entertainment. Even before the invention of the script people already told stories to entertain other people, and nowadays this is still the case. Because the fields of artificial intelligence, agent technology and computational linguistics are developing at such a rapid pace, automated story generation has gained much interest over the last few decades.

Two very important aspects of automated story generation are plot generation and language generation. *Plot* generation focusses on the underlying content of the story, such as the actions and motivations the story consists of. *Language* generation, on the other hand, focusses on the way the plot is expressed in a natural language such as the English and Dutch languages. For a story generation system to be effective, both parts should be carefully developed. This because no matter how interesting the plot is, if the natural language generation process has been neglected, the generated stories will never gain the user's interest. Note that this also holds the other way round; if more advanced techniques are used for the natural language generation, but the underlying plots do not contain any depth, the generated texts will not be interesting either.

The Virtual Storyteller is a system that can automatically generate simple stories. In earlier versions of the system the natural language generation was very simple, even though the language generation can be seen as one of the most important parts of the system. Therefore I have designed and implemented a new module for the natural language generation in the system, which I will describe in this document. First I will give a short description of the Virtual Storyteller and its already existing components. Then I will give an overview of the project and mention some of the project's goals.

1.1 The Virtual Storyteller

The Virtual Storyteller is an agent-based system which can generate simple fairy-tales. The system consists of several agents which together create a plot. This plot is then passed to the Narrator agent which takes care of the natural language generation by converting the plot into text.

In earlier versions the architecture of the Narrator agent was very simple; the agent translated the actions generated by the actor agents into fixed sentences using templates. The use of simple templates for natural language generation usually results in rather monotonous texts. Therefore many recent natural language generation systems use a pipelined architecture, consisting of a Document Planner, a Microplanner and a Surface Realizer (see for example Reiter and Dale [RD00]). Since most of these systems are able to generate acceptably coherent texts, we would like to use a similar architecture for the Virtual Storyteller. Hielkema [Hie05] has already implemented a first version of the Surface Realizer, which can translate dependency trees into sentences in natural language and performs syntactic aggregation and referring expression generation. Since the Surface Realizer has already been implemented, the main focus is on the design and implementation of the other two modules. However, in order to generate somewhat more attractive stories I have also made a number of changes and extensions to the Surface Realizer, which are also described in the document.

1.2 Overview of the Project

In order to design the Narrator agent, first of all a literature study has been carried out. In chapter 2 I will point to some relevant results, focussing first of all on literature about human-written stories and language use in such stories. Furthermore natural language generation and the different tasks a natural language generation system has to carry out will be discussed. Finally a number of existing systems that automatically generate stories are mentioned, but most of the existing storytellers focus on plot generation rather than language generation. StoryBook is an exception, so this system will be described somewhat more detailed. Chapter 3 will then describe the Virtual Storyteller, the system in which the Narrator agent will be embedded. This includes the architecture of the system and some examples of stories generated by earlier versions of the system. In order to find out how these generated stories differ from human-written stories I analyzed a number of human-written fairy-tales which is described in chapter 4. This analysis focusses on different aspects of human-written stories, such as paragraph and sentence structures, and referring expressions.

Using the literature and the results of the analysis I will define the project goals more precisely in chapter 5, after which I will describe the global architecture of the Narrator agent. Following Reiter and Dale this architecture is represented as a pipeline consisting of a Document Planner, a Microplanner and a Surface Realizer, and for each component some goals have been defined.

The main task of the Document Planner (see chapter 6) is to convert the plot structure generated by the other agents of the Virtual Storyteller into a document plan, which is a tree structure containing the relevant information from the plot which can be passed to the Microplanner. While converting the plot structure into a document plan the Document Planner is responsible for selecting the relevant plot elements from the input, ordering these plot elements and deciding how they should be related to other plot elements. Furthermore the Document Planner can add background information such as names and properties of characters and locations of objects in order to make the stories more varied and coherent.

The second module in the Narrator agent is the Microplanner (see chapter 7), which is responsible for the first part of the lexicalization process. This module first generates sentence plans by applying templates and then uses a lexicon in order to select Dutch words that can be used in the final text. Furthermore the module can detect word repetition and will try to prevent this by selecting synonyms if possible.

The Surface Realizer (see chapter 8) first of all performs syntactic aggregation by combining single clauses into more complex sentences. Furthermore the module is responsible for the generation of referring expressions while performing the last part of the lexicalization. The original algorithm was very simple, but I have extended the module with a more detailed Referring Expression Generator. This new algorithm generates more varied referring expressions by supporting different types of referring expressions and adding different kinds of adjectives. The final task of the Surface Realizer is generating the surface form, the final output text, and doing this the module has to make sure that the words are inflected correctly and punctuation is added.

Throughout chapters 6 to 8 the functioning of the system is illustrated by an example. This example is very simple, so in chapter 9 I will give some other examples and evaluate them more carefully. The examples show that the resulting texts are more varied and coherent than the texts generated by earlier versions of the system. However, there are still many possibilities for improving the system, so these are described in the final chapter together with some conclusions.

Chapter 2

Literature

This chapter discusses some relevant literature for the natural language generation process in an automated story generation system. In section 2.1 I will give a general introduction into the human writing process, including the elements of a good story and language use in narratives. The Virtual Storyteller generates stories automatically using natural language generation, so section 2.2 will deal with the natural language generation process. Finally I will point to some existing work on storytelling in section 2.3 and give a detailed description of the following systems: GESTER, Gervas's system and StoryBook.

2.1 Human Writing

Our ultimate goal is to generate texts which are comparable to texts written by human authors. Of course we will never reach this goal in a relatively short project, but it is definitely useful to look at some research into the human writing process.

2.1.1 The steps of the writing process

Obviously human writers will not write down the entire text at once; they will write down a draft first and then they will try to improve this draft into the final text. The process of having an idea in your mind and turning this idea into text is called the *writing process*. Hayes and Flower [HF86] focus on the writing of texts in general and they subdivide the writing process into the following three stages: planning, sentence generation and revision. *Planning* is the process of generating ideas and organizing them into a writing plan. In *sentence generation* the writer produces sentences intended to be part of a draft. Finally, *revising* is needed to improve the draft into the final text.

Mohan [Moh00] focusses mainly on the writing process for scientific papers. She also believes that the writing process consists of a number of stages, but she distinguishes six different stages. *Pre-writing* is the process of finding something to say and considering different ways of saying it. Next *drafting* is the first attempt to get ideas on paper and to write a first version. The *conferencing* stage consists of showing this first version of the text to somebody else and discussing it. The next stage is *revision* which means making changes in order to improve the text. Possible types of such changes are adding something, removing something, moving something somewhere else and substituting something for something else. *Editing* is checking if the text is correct, such as checking spelling and punctuation. Finally, *publishing* is putting the written text somewhere public in order to show it to the audience. Since Mohan focusses mainly on the writing process of scientific papers and we are focussing on writing fairy-tales, not all of these stages seem necessary. However, the point is that the human writing process can be subdivided into a number of consecutive stages, independent of the type of text.

2.1.2 Elements of successful narratives

Fairy-tales differ from short stories in the sense that fairy-tales always tell of imaginative and miraculous things. Furthermore the main characters in fairy-tales are often supernatural and the stories always contain a moral in which the good is rewarded and the evil is punished. The stories generated by the Virtual Storyteller are mainly about princesses and villains and do not include a moral, so they are more like short stories. The purpose however is the same for both types of text: to entertain, and to gain and hold the reader's interest.

Kilian [Kil92] set up a number of directives for authors of novels or short stories. These directives do not focus on the different steps of the writing process, but on the text to be generated. Kilian begins with stating the elements of a successful story which can be subdivided into elements in the opening, in the body and in the conclusion of a story and some elements throughout the story.

In the *opening* of a story the main characters and the settings should be introduced. At least one of the characters should be shown under some kind of stress in order to get the reader interested. Furthermore it should be made clear who's the good guy and who's the bad guy, so it should be clear who the reader should get emotionally involved with. This means that even if the hero is morally repugnant, for example because he is a killer, he should have some trait or attitude the reader can admire and identify with. Finally the tone of the story should be set, such as solemn, excited, humorous or tragic.

In the *body* the story should be told in scenes which all consist of a purpose, an obstacle or conflict and a resolution that tells us something new about the characters and their circumstances. Furthermore the story will be best if the characters act based on good motivations. Finally it is best to develop the plot as a series of increasingly serious problems. Doing this the suspense is increased gradually and can be increased even more by making the solution of a problem uncertain.

The *conclusion* of the story should present a climax in which a really serious problem is described or in which different problems are merged together. This part of the story should also find a conclusion to all problems mentioned sofar.

Kilian also gives some suggestions which can be used *throughout* the story. First of all, he believes that nothing in the story should happen at random, even the names of the characters should be selected with the greatest care. Furthermore the style, tone and point of view should be consistent throughout the story. Finally he states that stereotype characters should act according to the expectations of the reader, otherwise this might confuse the reader.

All of these directives mainly focus on plot generation rather than language generation. Since I will focus on language generation these directives do not seem very relevant, but some of them may be useful for the natural language generation module as well. An example is the directive that a story should start with a detailed description of the characters and settings.

2.1.3 Language use in narratives

Language use in narratives differs very much from language use in instructional texts and other types of texts, so Kilian also gives a number of directives for the language used in short stories and fairy-tales. If we combine this list with the directives given in [Onl98], this results in the following directives:

- Sentences should not begin with the words 'There' or 'It' (apart from the very first sentence in a fairy-tale).
- Many action verbs should be used and it is better to use active voice than passive voice.
- It is better to be clear and to use simple words, because the use of many difficult words may lead to confusion and redundancy.
- The prose should be fluent, varied in rhythm, and suitable in tone to the type of story.
- Descriptive language is often used to enhance the story and create images in the reader's mind.
- The text should include many linking words, especially linking words that have to do with time.

- Stories are normally written in past tense, but dialogues between characters may change the tense to present or future.

- The narrative can be written in first person (I, we) or third person (he, she, they), but in short stories it is best to keep the same narrative style throughout the story.

- Showing is better than simply telling.

  This final directive may need some additional explanation. Kilian states that the basic unit of a story is not the sentence or paragraph, but the *scene*. Every scene has a verbal and a nonverbal content: the verbal content is the actual content of the scene as it is told, and the nonverbal content is determined by the way the scene is presented. Using these definitions it is best if the verbal content leads to some nonverbal content. This can be illustrated by the following example: if you want the reader to believe that a character is awkward, it is more believable to *show* this by describing his awkward behavior rather than simply telling that the character is awkward without any evidence.

Strunk [Str18] adds to this list of characteristics of narratives some preferences for texts in general. He believes that paragraphs should be about the same topic and should not be too long. Furthermore he admits that simple sentences (sentences without any conjunctions, such as 'because' or 'but') result in monotonous texts, but he also believes that simply combining two clauses into one sentence over and over again is even worse. He therefore suggests that the sentence length should vary as much as possible. Finally he writes that it is best to place the most important words at the end of the sentence.

The final directive for language use in narratives is that there should be much variety in the sentence beginnings in order to let the text sound more fluently. This can be achieved in the following ways (examples are taken from [Onl98]):

- Participles: "Jumping with joy I ran home to tell mum my good news."

- Adverbs: "Silently the cat crept toward the bird."

- Adjectives: "Brilliant sunlight shone through the window."

- Nouns: "Thunder claps filled the air."

- Adverbial Phrases: "Along the street walked the girl as if she had not a care in the world."

- Conversations/Dialogue: these may be used as an opener. This may be done through a series of short or one-word sentences or as one long complex sentence.

The directives given in this section may be very useful for the language generation in the Virtual Storyteller. I will not focus on all of them, but some of the directives will be referred to in later chapters of the document. Furthermore chapter 9 will point to all of these directives as well and will evaluate the generated stories on these directives.

2.2 Natural Language Generation

In the previous section I described the elements of human writing. The goal of the Virtual Storyteller is to model this writing process automatically using natural language generation. Natural language generation is the subfield of artificial intelligence and computational linguistics that focusses on systems which can produce texts in natural language, such as English or Dutch. The input to the system is some non linguistic representation of information and the system's task is to transform this input into text [RD00].

2.2.1 Tasks of a natural language generation system

In order to transform the non linguistic representation into text a number of tasks have to be accomplished, including:

- Content determination: deciding what information should be included in the output document, so selecting the relevant information from the non linguistic input.
- Document structuring: deciding how chunks of content selected in the previous step should be grouped and how the chunks should be related using rhetorical relations.
- Lexicalization: choosing which specific words will be used to express the content selected by the content determination component.
- Referring expression generation: deciding what expressions should be used to refer to the entities used in the output text.
- Aggregation: deciding how the structures created by the component responsible for content determination and document structuring should be mapped onto linguistic structures such as sentences and paragraphs.
- Linguistic realization: converting abstract representations of sentences into real text, by determining the order of the words in the sentence.
- Structure realization: adding mark-up symbols which can be understood by the document presentation component such that this component will present the text in a correctly formatted form.

2.2.2 Architecture of a natural language generation system

A natural language generation system has to carry out all of the aforementioned tasks. There are many different ways in which this can be done, but Reiter and Dale [RD00] define the task of natural language generation as a pipeline involving three components which together accomplish all of the tasks. First of all, the *Document Planner* determines the content and the structure of the text to be generated. Then the *Microplanner* is responsible for the tasks of lexicalization, referring expression generation and aggregation. Finally, the *Surface Realizer* converts this representation into a sentence (or possibly a number of sentences) in natural language by applying linguistic realization and structure realization. This pipelined architecture is presented schematically in figure 2.1.

The pipeline defined by Reiter and Dale is to some extent comparable to the stages defined by Hayes and Flower [HF86], described in section 2.1.1. The difference is that Hayes and Flower state that the stages are interwoven and that Reiter and Dale see the writing process as a pipeline in which the stages are processed successively. There are two reasons for interweaving the three stages. First of all, the writing task may be performed in parts, such that the writer plans, generates and revises a first paragraph, then plans, generates and revises a second paragraph, and so on. Secondly, the writing process may be applied recursively, for example by discovering the need for an additional paragraph while revising. The reason that Reiter and Dale define the writing process as a pipeline is simply that it is easier to implement, especially if the software is being developed by several people. Many recent natural language generation systems are also represented as a pipeline and since these systems usually produce acceptably coherent and varied texts, we believe a pipelined architecture is satisfactory for the language generation in the Virtual Storyteller.

2.3 Existing Work on Storytelling

In the past decades the number of researchers interested in the automatic generation of stories has increased enormously. Therefore much literature can be found about existing systems which can generate stories, such as Tale-Spin, Universe, Minstrel, Joseph, The OZ Project, NIMIS: Teatrix and StoryBook. However, most of these systems focus on *plot* generation and describe the natural language generation only roughly. This means that most of the existing systems mainly focus on *what* is being told than on how this is expressed in

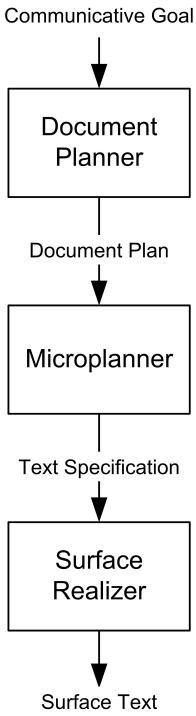

Figure 2.1: Architecture of a natural language generation system

words. An exception is StoryBook; this program only generates different versions in the Red Little Riding Hood domain, so the language generation in particular is very important. This section describes the following existing systems: GESTER, Gervas's system and StoryBook.

2.3.1 GESTER

Pemberton [Pem89] believes that the structure of each type of text can be described by a number of general principles and that each sub-genre can be described by the same principles with the addition of some genre-specific rules. Furthermore she states that a story generation system should have access to five different, but interacting databases: the story structure, the audience, the author, the cultural context and the rules of the sub-genre. In order to show this Pemberton developed GESTER (GEnerating STories from Epic Rules) which is a first step towards generating stories using interacting knowledge bases. She focussed mainly on the knowledge bases of story structure and rules of the sub-genre, but also to some extent on the knowledge base of cultural context.

Pemberton attempted to identify a general model of story structure as well as some additional constraints on the medieval French epic sub-genre. In her study she distinguishes between three different levels of text analysis. First of all, the *story line* is the sequence of actions in the story. *Discourse* contains the same events, but in a format which can be transformed into text. Finally the *textual level* is the text as the user sees it.

Furthermore a story is represented as a complex story which itself consists of one or several simple stories. These stories can be combined in four different ways: *cause*, when the first story causes the second story; *motive*, when a certain action in the first story motivates the second story; *then*, when all events in the second story succeed the events in the first story; and *same actor*, when one or several actors appear in both stories. In [Pem89] you can read how a simple story consists of an initial situation, an active event and a final situation, but this is more about the plot generation rather than the language generation, so I will not describe this in more detail here.

The discourse component focusses to some extent on content determination, for example by deciding when a piece of text can be left unsaid. An example is removing redundancy, such as removing the initial situation of the second story when the story is connected to the first story by means of a cause relation; we can assume the reader has already read the first story, so the cause is known to the reader and does not have to be repeated in the initial situation of the second story.

GESTER uses a number of features to produce an acceptably coherent story. Some of these features focus on plot generation, while others focus on language generation. However, for clarity I will discuss all of these features, which can be subdivided into the following groups:

- *Story features* are used to determine a number of features of the complex story. There are four story features: link, resolution, mode and motive. *Link* specifies how the two simple stories are combined to create the complex story (one of the possibilities mentioned before). *Resolution* determines whether the story will end successfully or not. *Mode* describes the mode of the overall story, such as friendliness or hostility. Finally, *motivation* can be used to represent certain motivations.

- *Role features* make sure that characters which appear in several stories play the same role in each story. There are seven different role features, such as subject, object, opponent or auxiliary.

- *Transformation features* are needed to transform the story line into the discourse. The features that are used are delete and move: *delete* can be used to remove redundant information and *move* can be used to combine elements of the second story with elements of the first story.

The GESTER system generates stories about kings and knights who want to take over regions, all in the Old French sub-genre. Its output consists of a string of terminal elements making up the content of the story, together with a representation of its parse tree. The final result is an acceptably coherent story, although the language use is still rather monotonous; most of the sentences start with 'then' and some of the sentences are very short.

2.3.2 Gervas's system

Gervas et al. [GDAPH05] built a story generation system which focusses on plot generation as well as language generation. For the generation of the actual text templates are used, but they do distinguish between the following six tasks of natural language generation: content determination, document structuring, sentence aggregation, lexicalization, referring expression generation and surface realization (according to Reiter and Dale [RD00]). The disadvantage of using templates for text generation is that the generated texts may become rather monotonous, but by applying aggregation and referring expression generation the texts become somewhat more varied.

Using case based reasoning and input from the user a plot plan is generated which is simply represented as a table with character functions, describing actions carried out by the characters in the story. Next it is the task of the natural language generation module to convert this plot plan into text by processing each character function in the plot plan separately. First of all, the system determines which characters and locations are used in the character function being processed and which roles the different characters play. This information is combined into a *draft* data structure and the text that has already been generated at the time of processing the character function forms the context for the new character function. In order to achieve this, a separate data structure is created which keeps track of the discourse history.

After the draft data structure has been created, the different tasks of the language generation process are applied successively. Figure 2.2 shows an example of the execution of all tasks with the outputs at each stage of the pipeline.

First of all, the draft data structure is passed to the *content determination module*. This module decides what information has already been told using the discourse history in order to avoid telling facts twice. The module also checks whether a character has already been introduced into the context and if the character has *not* been introduced a detailed description is generated.

The result of the content determination module is then passed to the *discourse planning module* which decides the order in which the information should be presented in the final text. In order to achieve this,

Content determination	Discourse planning	Sentence aggregation
character(ch1, princess) character(ch3, lioness) location(l1, forest) role(ch3, villain) attribute(ch3, hungry) attribute(ch3, fierce) action(ch3, ch1, devour)	character(ch3, lioness), attribute(ch3, hungry) character(ch3, lioness), attribute(ch3, fierce) character(ch3, lioness), character(ch1, princess), action(ch3, ch1, devour)	character(ch3, lioness), attribute(ch3, hungry), attribute(ch3, fierce) character(ch3, lioness), character(ch1, princess), action(ch3, ch1, devour)

Referring expression generation	Lexicalization	Surface realization
character(ch3, lioness), ref(ch3, def), attribute(ch3, hungry), attribute(ch3, fierce) character(ch3, pron), character(ch1, princess), ref(ch1, def), action(ch3, ch1, devour)	“lioness” “the” “hungry” “fierce” L(x) + “ was ” + L(y) + “ and ” + L(z) “she” “princess” “the” L(x) + “ devoured ” + L(y)	“The lioness was hungry and fierce.” “She devoured the princess.”

Figure 2.2: Execution of the stages in Gervas' system

a character function first describes the characters, next the location, then the necessary objects and finally the actions.

The *sentence aggregation module* combines all elements which can be aggregated in order to create a more fluent text. The module checks if there are sentences which have the same object and combines these objects if possible. An example is 'The lioness was hungry and fierce' instead of two separate sentences (see the example in figure 2.2).

The *referring expression generator* creates referring expressions which are sufficient to identify and are not exactly the same as those used in the immediately preceding sentence. At this moment noun phrases are replaced by pronouns, but only if the noun phrase appeared in the *same* character function and if no confusion can arise. Additionally, the module decides whether definite or indefinite determiners should be used. Like many other systems this module selects an indefinite determiner if the corresponding referent has not been mentioned before and selects a definite determiner in the other cases.

The *lexicalization module* selects the words to describe the concepts used in the character function. Static objects (such characters, locations and objects) are mapped to specific lexical terms. For the generation of verbs, templates are used corresponding to the structures of the sentences to be built. Finally linguistic features (such as pronouns and determiners) are put in the correct form. Since the system does not make use of a grammar, the module has to use an ontology which makes sure that the generated text satisfies all grammatical requirements, such as number and gender agreement.

Finally, the *Surface Realizer* converts the output of the lexicalization module into sentences in natural language using templates. Additionally, it carries out a basic orthographic transformation by capitalizing the first letter of the sentence and adding punctuation.

At the time of writing the system has not been fully implemented yet, so the generated stories are still to be evaluated.

2.3.3 StoryBook

Callaway (see [Cal00] and [CL01]) believed that the level of writing quality of automatically generated stories was still rather low in the early nineties. The main reason for this was that most of the story generation systems focussed on plot design rather than language generation. Secondly, the natural language generation at that time was simply not complex and varied enough. Finally most of the existing NLG systems focussed on *explanation* generation rather than *narrative* generation which are two completely different areas. Therefore Callaway built the Author architecture which is embedded in the StoryBook system, a system that can produce multi-page stories in the Little Red Riding Hood fairy-tale domain.

Functionalities of Callaway's Author system

Callaway defines narrative prose as a temporally-ordered sequence of sentences consisting of story settings, events, descriptions, characters, locations and properties that form a coherent text about things which can happen in a fictional world. Furthermore, he believes a computational model of narrative should be able to produce texts which are comparable in quality to human-produced stories.

Human-produced narratives often use many different points of view and the choice of point of view has effects on the author as well as on the reader. Callaway distinguishes between three different aspects of point of view:

- *Grammatical point of view* determines whether a particular character is referred to as 'I', 'you' or 'he'.
- *Character perspective* represents a character's outlook on the events and descriptions that occur in the narrative.
- *Narratorial point of view* is the exact characterization of the narrator role. This includes person (the choice of first, second or third person), embodiment (whether the narrator is embodied or not, or even an actual character in the story), omniscience (whether the narrator is omniscient or limited in his knowledge), reliability (whether the narrator can be trusted to relay information correctly in accordance with his desires) and diegesis/mimesis (whether the information is communicated via telling or showing).

StoryBook also focusses on generating dialogues and distinguishes four different kinds of character interactions. Furthermore Callaway describes the difficulties that can arise when generating dialogues, such as the generation of incomplete sentences and punctuation. At this moment the Virtual Storyteller does not support dialogues yet, but once this has been added to the system Callaway might have some useful suggestions.

There are a large number of types of discourse markers supported by the Author architecture when expressing the relation between sentences. *Temporal* discourse markers indicate the temporal difference between the two sentences (such as 'while', 'after' and 'before'). *Causal* discourse markers indicate a causal dependency between the sentences (such as 'because' and 'despite'). *Enumerative* discourse markers enumerate utterances and are the most frequently used discourse markers (such as 'first', 'next' and 'and then'). Finally, *reinforcing* discourse markers emphasize some part of the sentence (such as 'also', 'besides' and 'as well'). At the time of Callaway's research there were no models that could account for multiple markers, such as 'and then', but StoryBook includes a model which can generate this kind of discourse markers as well.

Terminology used in Callaway's Author system

Many researchers in narratology state that narratives consist of the *fabula*, or collection of facts and knowledge about a narrative world, and the *suzjet*, or the presented order of the events. The Author architecture and StoryBook also adopt this view, but call the suzjet the *narrative stream*. The fabula and narrative stream are separable in the sense that one fabula can result in different stories by varying the order in which the events are told.

When a human author writes a story, he or she uses a large amount of background information. For a computational model of narrative prose generation we also need this background knowledge which Callaway

calls the *story ontology*. The ontology contains simple facts such as "Trees have green leaves" and concepts of generic characters, generic events and generic objects. The fabula then consists of specific instances of these generic characters, events and objects. Finally the plot is a subset of the fabula; a particular ordered set of events in the fabula.

Architecture of Callaway's Author system

In figure 2.3 the architecture of the Author system is shown, represented as a standard pipeline. The pipeline consists of the following four components: the narrative organizer, the sentence planner, the revision component and the surface realizer. All of these components will be described in the remainder of this section. Right before the pipeline is entered the *narrative planner* generates the fabula and the narrative stream. Since the system only focusses on the Little Red Riding Hood domain this component is rather simple.

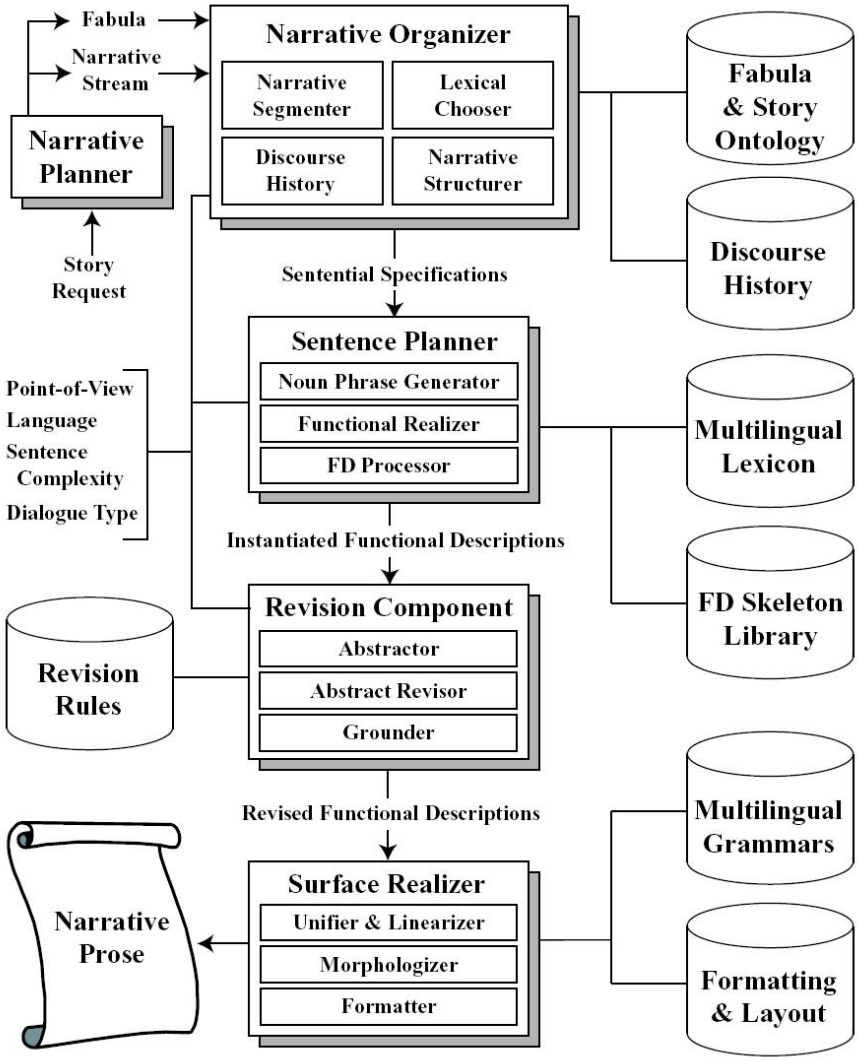

Figure 2.3: Author's architecture

The fabula and the narrative stream are then passed to the first component in the pipeline, the *narrative organizer* whose task is to convert the flat, linear narrative stream into a hierarchical narrative structure.

First of all the *narrative segmenter* segments the long narrative stream into groups of *narrative stream primitives* which are comparable to paragraphs in real texts. Callaway distinguishes four reasons for inserting paragraph boundaries: scene boundaries (such as a new location, time or character), dialogue boundaries, topic boundaries and size boundaries.

Narrative stream primitives are subdivided into three categories: delimiting primitives, base primitives and modifying primitives. *Delimiting primitives* are used to create the narrative context; they establish scenes, introduce characters and narrators, and describe the author's overall intent with respect to the audience. *Base primitives* provide most of the raw content used for creating sentences in the story, such as actions carried out by the characters, and properties of characters. Finally *modifying primitives* present information that specifically modifies the content of a prior base primitive, and they also include reasons for carrying out actions and character's purposes. Using these primitives a narrative stream usually consists of a large number of delimiting primitives that define the narrative's context, followed by base primitives mixed with modifying primitives that make up the body of the narrative.

Then the *discourse history* is used to decide whether pronouns can be used to refer to the entities and whether an entity should have a definite or an indefinite article. In contrast to most existing systems definite articles may be used even if the corresponding noun has not been used before. First of all definite articles are used with some mass nouns, for example in the sentence "Once upon a time there lived a woodman in a cottage, on *the* borders of a great forest". In this case the forest has not been introduced yet (so the borders have neither), but the system decides that 'borders' should get a definite article like a human author would do. Secondly, definite articles are used with aliases. An example of this is: "Once upon a time there was a princess. *The* girl lived in a castle." In this case the noun 'girl' has not been used before, but because it refers to the princess, an entity which has been introduced into the context, a definite article is generated. In order to decide if a pronoun can be used the discourse history keeps the following properties for each concept instance in the story:

- Last-Usage: Whether or not its most recent use was in a fully lexicalized form.
- Frequency: How many times a concept instance has been used.
- Recency: How many distinct concept instances have been used since its last use.
- Distance: The number of paragraph breaks since its last occurrence.

The *lexical chooser* decides which words should be used and checks for repetition of words. It can detect repetition in noun phrases, repetition in verb phrases and repetition in thematic role ordering. If the module detects repetition in noun or verb phrases it chooses a word with a similar meaning if possible. If the module detects repetition in thematic role ordering it can for example convert a sentence such as "Little Red Riding Hood gave her mother cookies" into "Grandmother received cookies from Little Red Riding Hood". Finally the *narrative structurer* is responsible for converting the groups of narrative primitives into a sequence of specifications suitable for the sentence planner.

The second module in the pipeline is the *sentence planner* and this module has to plan the roles (either semantic or syntactic) that each element plays in a particular sentence. The output of this module is then a sequence of functional descriptions. The module's first component is the *noun phrase generator* which creates functional descriptions for each of the noun phrases. It must be able to produce different types of relative clauses, adjectival phrases, prepositional phrases etc. Next the *functional realizer* assigns thematic roles to each argument of the functional descriptions. Finally the *functional description processor* makes sure that the generated functional descriptions do indeed create grammatically correct sentences.

The third module in the pipeline is the *revision component* which receives a paragraph-sized group of sentences from the sentence planner represented as an ordered set of functional descriptions and converts them into revised functional descriptions. First the *abstractor* transforms the initial ground level narrative plan (received from the sentence planner) into an abstract narrative plan which contains only the most important lexical, syntactic and semantic features needed for revision. The removed features are stored away for future use. Then the *abstract revisor* iteratively applies revision operators to the abstract narrative plan and evaluates the resulting narrative plans with respect to quantitative discourse, style and length constraints. Since abstract narrative plans lack most syntactic details and lexical items, they cannot be

translated into text directly. Therefore the *grounder* reconstructs ground level narrative plans by integrating the syntactic and semantic information that was stored away by the abstractor into the abstract narrative plans.

The final module of the pipeline is the *surface realizer* which has to fulfill five major functions. First the *unifier and linearizer* component is responsible for three of these functions. First, it adds closed-class and default lexical items, such as articles and prepositions which are not lexically specified in the functional descriptions. Then it ensures the grammaticality of the utterances and rejects functional descriptions that represent ungrammatical sentences. Finally the component orders the lexical items using a grammar specifying possible word orders in sentences. The *morphologizer* adjusts lexical items for morphology; stems are changed based on gender, number and other features. Finally the *formatter* adds punctuation and other formatting directives.

The architecture described in this section is embedded in the StoryBook system which generates two to three page stories in the Little Red Riding Hood domain. The resulting stories have been evaluated and it turns out that the stories' quality is comparable to the quality of a story written by a human author. In figure 2.4 an example of the first page of a story is shown, represented as a HTML-page including text and pictures.

Figure 2.4: Example of StoryBook's output

Chapter 3

Virtual Storyteller

The Virtual Storyteller is a system which can automatically generate stories, developed at the University of Twente, the Netherlands. At this moment the system creates simple fairy-tales about princesses and villains, but in the future more complicated stories may be possible.

The goal of this project is to transform a plot generated by the Virtual Storyteller into natural language. Therefore it is useful to look at the architecture of the existing system first, which is done in section 3.1. Furthermore Hielkema [Hie05] has designed and implemented a Surface Realizer which is part of the language generation process. This module has not been integrated in the Virtual Storyteller yet, but it can be used by the component responsible for the natural language generation, so the Surface Realizer is the main focus of section 3.2. Finally section 3.3 gives some examples of stories generated by earlier versions of the Virtual Storyteller.

3.1 Architecture of the Virtual Storyteller

The Virtual Storyteller is an agent-based system implemented in Java (see [Faa02] and [Ren04] for earlier versions of the system, and [Swa06] and [Uij06] for the newest version of the system). Figure 3.1 shows the system's global architecture, taken from [Swa06]. You can see that the architecture consists of one or several Character agents, a World agent, a Plot agent and a Narrator agent.

The *Character agents* are the real actors in the story. It is important that the characters are believable, so they include a complex emotional model. Using this model the agent can define goals for himself and the agent can reason in order to come up with plans to fulfill his goals.

The *World agent* keeps a current model of the Story World (the world in which the story develops). Furthermore the World agent processes the actions and events caused by the other agents and decides how the world should be updated after the actions have been carried out.

The *Plot agent* is responsible for the actual generation of the plot. The World agent tells the Plot agent if there are any world changes and the Plot agent can respond with actions and events. Therefore it communicates with the Character agents as well; it passes the perceptions to the characters and it passes the actions carried out by the characters to the World agent. Using all this information the Plot agent also builds the fabula structure (see section 5.3.2).

The Character agents, the World agent and the Plot agent thus generate the plot (the fabula structure) together which is then passed to the *Narrator agent*. This agent then has to convert this fabula structure into sentences in natural language. In the future it will also add mark-up symbols which can be used by a Presentation agent when converting the text into speech, but this agent has not been included in the architecture yet.

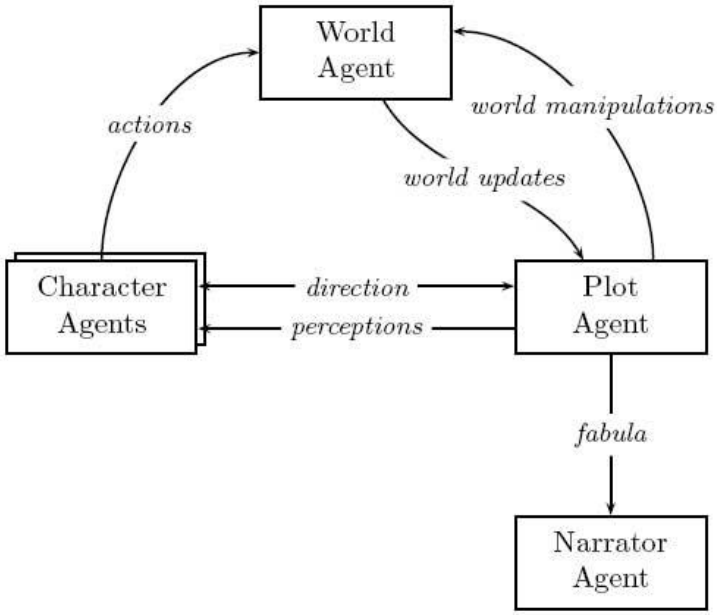

Figure 3.1: Architecture of the Virtual Storyteller, taken from [Swa06]

3.2 Surface Realizer

In earlier versions the Narrator agent used templates for the language generation. These templates received an action as input and translated this action into a standard sentence, which resulted in rather monotonous texts. In order to create more varied texts Hielkema [Hie05] has designed and implemented a Surface Realizer for the Virtual Storyteller. The input to the Surface Realizer is a Rhetorical Dependency Graph, which consists of dependency trees connected by rhetorical relations.

3.2.1 Dependency trees

Dependency trees represent the grammatical relations that hold in and between constituents, and specify the lexemes needed to translate the dependency tree into real text [vdBBD+02]. Nodes in the tree structure thus have relations to other nodes and may be labelled with lexemes. An example of a dependency tree for the sentence "Kim wil weten of Anne komt." ("Kim wants to know whether Anne is coming.") is given in figure 3.2, taken from [vdBBD+02]. You can see that each node in the tree has a relation to its parent node (the top most word in the boxes). Furthermore leaves have a part-of-speech tag (such as noun and verb) and a lexeme which is the actual word in the sentence to be generated. Other nodes only have a relation to their parent nodes and a grammatical category (such as inf, ssub or cp).

The Surface Realizer in the Virtual Storyteller uses dependency trees for a number of different reasons. First of all, the word order is not expressed in dependency trees which make them easy to manipulate. Secondly, dependency trees include dependency relations for each node which is useful for performing ellipsis (described later in this section). An example of ellipsis is subject deletion and this can easily be performed by deleting the node labelled 'subject'.

[HMR+03] gives a detailed explanation of all possible dependency labels using examples. Currently, not all of these labels are supported by the Surface Realizer, but this may be extended if desirable.

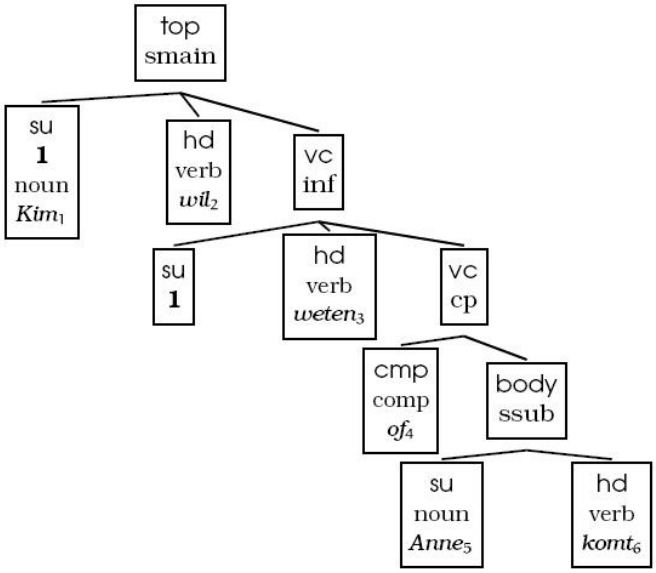

Figure 3.2: Example of a dependency tree, taken from [vdBBD+02]

3.2.2 Rhetorical relations

The Rhetorical Dependency Graph also contains the rhetorical relations (or discourse relations) which hold between dependency trees or groups of dependency trees. Rhetorical Structure Theory [MT87] states that a text is coherent if for every part of the text there is some plausible reason for its presence. This also means that all dependency trees in the rhetorical dependency graph should be connected to other dependency trees by rhetorical relations in order to get a single tree structure.

Rhetorical Structure Theory furthermore claims that there are about 25 different rhetorical relationships. For the Virtual Storyteller it is not necessary to distinguish between all of these relations, so we can simply use a subset of the relations. Rhetorical relations hold between text spans which are either atomic sentences or larger text spans (or in our case single dependency trees or groups of dependency trees already related by rhetorical relations). Most RST discourse relations are binary and the two portions of text between which the relations holds, are called the nucleus and the satellite. The *nucleus* is the piece of text which cannot be left unspecified; the *satellite* tells something about the nucleus and is far less important. If a relation does not have a particular span of text which is more central than the other one (both pieces of text are equally important), the relation is called *multi-nuclear*.

A rhetorical dependency graph thus consists of dependency trees connected by rhetorical relations. All leaves in such a graph are dependency trees and all internal nodes are rhetorical relations, which specify the relations that hold between the node's direct child nodes. A rhetorical relation node always has exactly one or two children. If a rhetorical relation has one node, this node is called the nucleus; if the rhetorical relation has two children, the left one is called the satellite and the other one is called the nucleus.

3.2.3 Tasks of the Surface Realizer

Using such a Rhetorical Dependency Graph in which dependency trees are connected by rhetorical relations, the Surface Realizer is able to execute the following four tasks: syntactic aggregation, referring expression generation, surface form generation (generating the final text in natural language) and adding orthography.

First of all *syntactic aggregation* is applied, in which two dependency trees are combined in order to get a more complex tree. For each rhetorical relation in the graph the module decides how the relation should be represented in the final text. Firstly, this can be done by combining the two child nodes (the dependency

trees) into one by adding a conjunction (such as 'but' or 'because'). Furthermore it can generate two separate trees and add an adjunct (such as 'however' or 'therefore') to one of the trees that represents the rhetorical relation explicitly. Finally a relation can be represented implicitly by doing nothing at all; in this case the aggregation module simply returns two separate dependency trees.

The rhetorical relations supported by the Surface Realizer are Additive, Cause, Contrast, Purpose and Temporal. The Additive relation is somewhat like an enumeration of two elements and the other relations are standard RST relations [MT87]. Most of these relations also have subrelations in order to create a greater variety of sentences. Examples of all (sub)relations are as follows (note that the examples use only one cue word but other cue words may have been stored as well):

- Additive
  - additive: De ridder sloeg de prinses *en* zij ging naar het bos.
    (The knight hit the princess and she went to the forest.)
  - additive-moreover: De ridder sloeg de prinses. *Bovendien* zij ging naar het bos.
    (The knight hit the princess. Moreover she went to the forest.)
- Cause
  - cause-voluntary-first: *Omdat* de ridder de prinses sloeg, ging zij naar het bos.
    (Because the knight hit the princess, she went to the forest.)
  - cause-voluntary-last: De ridder sloeg de prinses, *dus* ging zij naar het bos.
    (The knight hit the princess, so she went to the forest.)
  - cause-involuntary-first: *Doordat* de ridder de prinses sloeg, ging zij naar het bos.
    (Because the knight hit the princess, she went to the forest.)
  - cause-involuntary-last: De ridder sloeg de prinses, *zodat* zij naar het bos ging.
    (The knight hit the princess, so that she went to the forest.)
- Contrast
  - contrast: De ridder sloeg de prinses, *maar* zij ging naar het bos.
    (The knight hit the princess, but she went to the forest.)
  - contrast-causal: *Hoewel* de ridder de prinses sloeg, ging zij naar het bos.
    (Though the knight hit the princess, she went to the forest.)
- Purpose
  - purpose: *Om* de prinses te slaan, ging de ridder naar het bos.
    (In order to hit the princess, the knight went to the forest.)
- Temporal
  - temp-before-sequence: De ridder sloeg de prinses, *voordat* zij naar het bos ging.
    (The knight hit the princess, before she went to the forest.)
  - temp-after-sequence: *Nadat* de ridder de prinses geslagen had, ging zij naar het bos.
    (After the knight had hit the princess, she went to the forest.)
  - temp-before-gap: De ridder sloeg de prinses. *Ooit* ging zij naar het bos.
    (The knight went to the forest. One day she went to the forest.)
  - temp-after-gap: De ridder sloeg de prinses. *Later* zou zij naar het bos gaan.
    (The knight hit the princess. Later she would go to the forest.)
  - temp-during: *Terwijl* de ridder de prinses sloeg, ging zij naar het bos.
    (While the knight was hitting the princess, she went to the forest.)

- temp-atlast: De ridder sloeg de prinses. *Eindelijk* ging zij naar het bos.
  (The knight hit the princess. At last she went to the forest.)
- temp-when-soonas: *Zodra* de ridder de prinses sloeg, ging zij naar het bos.
  (As soon as the knight hit the princess, she went to the forest.)
- temp-suddenly: De ridder sloeg de prinses. *Plotseling* ging zij naar het bos.
  (The knight hit the princess. Suddenly she went to the forest.)
- temp-suddenly: De ridder sloeg de prinses. *Uiteindelijk* ging zij naar het bos.
  (The knight hit the princess. Finally she went to the forest.)
- temp-when: *Als* de ridder de prinses slaat, gaat zij naar het bos.
  (When the knight hits the princess, she goes to the forest.)
- temp-once: Er was *eens* een prinses.
  (Once upon a time there was a princess.)

After combining two dependency trees, the resulting tree may contain a certain node more than once. If the node has the same grammatical role in both parts of the tree, ellipsis may be applied; the node can be removed from one part of the tree, possibly after making some corrections. The system supports the following kinds of ellipsis:

- Conjunction Reduction: Diana betrad de woestijn en zag Brutus.
  (Diana entered the desert and saw Brutus.)
- Right Node Raising: Diana betrad en Brutus verliet de woestijn.
  (Diana entered and Brutus left the desert.)
- Gapping: Diana verliet de woestijn en Brutus het bos.
  (Diana left the desert and Brutus the forest.)
- Stripping: Diana verliet de woestijn en Brutus ook.
  (Diana left the desert and Brutus too.)
- Coordinating one constituent: Diana en Brutus betraden de woestijn.
  (Diana and Brutus entered the desert.)

The next component in the Surface Realizer is responsible for the *referring expression generation* and decides whether a description or a pronoun should be used to refer to a certain entity. Adding pronouns is a useful addition to create a less monotonous text, but before a pronoun can be selected it should be checked if no confusion can arise. The current Referring Expression Generator uses a very simple algorithm for pronominalization; it simply checks whether the entity was also mentioned in the previous utterance and whether there are no intervening entities with the same gender.

Next the created dependency trees are converted into *surface forms*. The module uses a grammar which specifies the exact word order in the sentence. The dependency trees hold the constituents' grammatical roles (such as subject and object) and using the grammar these tree structures can be converted into sentences in natural language.

Finally *orthography* is applied by adding punctuation and capitalizing the first letter of the sentence. Adding punctuation is performed by placing a period at the end of each sentence and adding a comma before a conjunction (except when this conjunction is the cue word 'en' ('and')).

3.3 Results Earlier Versions

In this section I will give examples of the results of earlier versions of the Virtual Storyteller. First I will show an example story generated by the system implemented by Rensen [Ren04] and then I will show the same story after performing syntactic aggregation using the Surface Realizer implemented by Hielkema [Hie05].

3.3.1 Version 1

S. Rensen mainly focussed on plot generation rather than language generation. Therefore the language generation in the system is done using templates and the resulted texts are rather monotonous. An example story generated by this version of the Virtual Storyteller is shown in figure 3.3.

Er was eens een prinses.
Ze heette Amalia.
Ze bevond zich in Het kleine bos.
Er was eens een schurk.
Zijn naam was Brutus.
De schurk bevond zich in het moeras.
Er ligt een Zwaard in De bergen.
Er ligt een Zwaard in Het grote bos.
Amalia loopt naar de woestijn.
Brutus loopt naar de woestijn.
Amalia ervaart angst ten opzichte van Brutus vanwege de volgende actie:
Amalia ziet Brutus
Amalia gaat het kasteel binnen.
Brutus gaat het kasteel binnen.
Amalia ervaart angst ten opzichte van Brutus vanwege de volgende actie:
Amalia ziet Brutus
Amalia slaat de mens.
Brutus schopt de mens.
Amalia schreeuwt.
Brutus slaat de mens.
Amalia schreeuwt.
Brutus pakt de mens op.
Amalia schreeuwt.
Brutus ervaart hoop ten opzichte van Amalia vanwege de volgende actie:
Brutus pakt de mens op.
Brutus neemt in het kasteel de mens gevangen.
en de mensen spraken jaren later nog over deze treurnis

Figure 3.3: Example story before performing syntactic aggregation

You can see that the sentences are very short, that each sentence starts with the subject and that the entire name is used each time a character is being referred to (apart from the references in the second and third sentences). Obviously, Rensen succeeded in building a system which can generate interesting plots, but the language generation can be improved considerably.

3.3.2 Version 2

As explained in the previous section Hielkema has built a Surface Realizer that performs syntactic aggregation. This module has not been integrated in the Virtual Storyteller, but if this would have been done the system would generate the story like the example shown in figure 3.4.

You can see that this story is much better than the other story and that it includes a greater variety of sentence structures. Many clauses have been combined using different rhetorical relations which makes the text less monotonous and more fluent. Furthermore the referring expression generator makes sure that pronouns are used when no confusion can arise about which entity is being referred to.

Despite of these improvements there are still many things to note. First of all the adjectives have not been inflected. Furthermore some clauses have been embedded which makes the sentences more complicated than necessary. An example is the sentence “Hij sloeg Amalia, want, doordat Amalia het zwaard op pakte,

werd hij bang." ("He hit Amalia, because, because/when Amalia picked up the sword, he got scared."). If this sentence would have been told as "Hij sloeg Amalia, want hij werd bang, doordat zij het zwaard oppakte." ("He hit Amalia, because he got scared, because/when she picked up the sword."), the sentence would be easier to understand. As already mentioned the referring expression generator uses a very simple algorithm to decide whether to use a noun phrase or a pronoun. This results in a number of noun phrases in cases in which it is better to use a pronoun. Take for example the same sentence "Hij sloeg Amalia, want, doordat Amalia het zwaard op pakte, werd hij bang." ("He hit Amalia, because, because Amalia picked up the sword, he got scared."). In this story Amalia is the only female character, so a pronoun can easily be used to refer to the princess, especially in a sentence in which she is referred to twice such as this sentence. Finally the text does not include much detail, such as adjectives and adverbs. Among other things I will focus on these problems during this project.

Er was eens een prinses.
De prinses heette Amalia en bevond zich in het klein bos.
Er was eens een schurk.
De schurk heette Brutus en bevond zich in het moeras.
Er lag een zwaard in het groot bos en ook in de bergen.
Brutus ging, terwijl Amalia naar de woestijn ging, erheen.
Doordat zij Brutus zag, was zij bang.
Brutus ging het kasteel binnen, want zij ging het kasteel binnen.
Omdat zij Brutus zag, was zij bang.
Amalia sloeg hem.
Brutus schopte Amalia en Amalia schreeuwde.
Doordat hij Amalia sloeg, schreeuwde Amalia.
Zij schreeuwde, maar hij was hoopvol, omdat hij Amalia op pakte.
Hij nam haar gevangen in het kasteel.
Nog jaren spraken de mensen over deze treurnis.

Figure 3.4: Example story after performing aggregation

Chapter 4

Analysis of Human-Written Fairy-Tales

In the last section of the previous chapter we looked at some examples of stories generated by earlier versions of the Virtual Storyteller. In this chapter I will analyze human-written fairy-tales in order to find out how human-written texts differ from the automatically generated texts. First of all I will examine the story structure and distinguish between information that is directly available in the input (in the fabula structure and other sources of information such as the Character Information and Story World modules), and information that should be added by the Narrator agent (such as adjectives and sentences to create a mood). Furthermore I will look at sentence and paragraph structures, because the original Surface Realizer simply received three dependency trees at most and tried to combine them into one dependency tree. Since we will have to deal with longer texts, it may be useful to look at sentences and paragraphs in real texts as well. Finally I will analyze the referring expressions used in the fairy-tales in order to find out when it is best to use a noun phrase and when to use a pronoun. As explained in the previous chapter the original referring expression generator uses a very simple algorithm, so using the results of this analysis a more complicated algorithm may be designed.

Section 4.1 lists the questions that will be answered during the analysis. The next sections (section 4.2 to section 4.5) give the answers to the questions and the chapter ends with a conclusion in section 4.6. Appendix A shows the four stories used for the analysis: 'Koning Lijsterbaard', 'De Gelaarsde Kat', 'De ring van de koningsdochter' and 'De jongen die lezen en schrijven leerde'. The first two stories are taken from a website [VH] and the other two stories are taken from the book "Nieuwe Sprookjes van de Lage Landen" [dJS74]. I picked those stories first of all because the characters do not talk very much in the stories. This is important because the characters in the Virtual Storyteller will not be able to talk at all in the near future. Furthermore the first two stories have a clear structure, there are only a few characters and the first story contains a character with a name. The other two stories have a less clear structure (they contain many short paragraphs), but there are some interesting sentences in the stories. Note that the characters in the third story talk rather much at the end of the story, so I analyzed this story only up to and including the seventh paragraph.

Throughout the chapter I will use the word 'sentence' to refer to a complete sentence, the word 'clause' to refer to the shorter sentences one sentence consists of (main clauses as well as subordinate clauses) and the word 'phrase' to refer to the smallest set of words which belong together. The sentence "De prinses schreeuwde, omdat ze bang was." ("The princess screamed, because she was scared.") therefore consists of the clauses "De prinses schreeuwde." ("The princess screamed.") and "omdat ze bang was" ("because she was scared"), and the first clause consists of the phrases 'de prinses' ('the princess') and 'schreeuwde' ('screamed').

4.1 Questions

In this chapter I will try to answer the following questions:

- Input and structure (see section 4.2):
  - What information would probably be present in the input (the graph of connected story elements), if such a story were to be generated by the Virtual Storyteller?
  - What information can be retrieved from other sources of information?

    Examples of such sources of information are the knowledge base containing information about the characters and the settings module.
  - What information should be added by the Narrator agent?

    For example information that does not change the meaning of the story, but is added in order to create a mood. This information includes all information that cannot be categorized as one of the other types of information.
- Sentence and paragraph structures (see section 4.3):
  - How are the propositions combined into a sentence? (Which rhetorical relations are used?)
  - How many propositions are combined into a sentence?
  - Are the consecutive sentences related using cue words?
  - When do paragraph breaks occur?
  - Are there any syntactic constructions that cannot be used by the Virtual Storyteller yet?
  - Do more complex sentences always contain a specific *kind* of relation?
- Referring expressions (see section 4.4):
  - How often (and when) are noun phrases used to refer to an entity?
  - How often (and when) are pronouns used?
  - How often (and when) are the actual names used (if available)?
  - How much variation is there between the different types of referring expressions?
  - Can pronouns be used if the entity has not been mentioned before in the current paragraph?

    We predict that the first reference in a paragraph is always done using a definite description, but this assumption will be checked in the analysis.

Furthermore a small part of the story will be described using the RST notation (see section 4.5) and section 4.6 will summarize the conclusions of the analysis.

4.2 Input and Structure

In order to find out what kind of information will be available in the input, which information will be available in other sources of information and which information should be added by the Narrator, I will analyze one paragraph for each story. Doing this I assume that the stories are to be generated by the Virtual Storyteller. First of all this analysis will show whether it is possible to describe a story by categorizing each proposition as either an action, an event, a perception, a goal, a state, a setting or as 'others'. Secondly, it may give some directions when to add background information.

4.2.1 Which information is available, and where?

In order to find out which information is available and where, I will categorize the clauses into one of the following categories: actions, events, perceptions, goals, internal states, settings and others. The first five of these are available in the input fabula structure. Furthermore the settings are available in the Character Information module and in the Story World module. The Character Information is the module that contains all information about the characters, such as names and properties of characters, but also relations to other characters. The Story World module is the module that contains all information about the world in which the story develops, such as locations of objects. Finally, the clauses marked 'others' are not available in the input and should therefore either be added by the Narrator agent or they should be left out. Additionally some *phrases* (or parts of phrases) in the sentences should be added by the Narrator, but some of these may be available in other sources of information, such as some properties in the Character Information.

Koning Lijsterbaard

In this story the fourth paragraph has been selected for analysis. Table 4.1 shows all sentences that appear in this paragraph and for each sentence it shows the clauses it consists of. Furthermore the type of each clause is given in the right column of the table.

Sent	**Clause**	**Type**
1	Op een dag gaf de koning een groot diner met bal na.	action
2	Voor al het personeel in de keuken was het een drukke dag.	event
3	Ook de prinses moest de hele dag en avond hard werken, maar zij kon wel allerlei lekkere hapjes onder haar schort verzamelen om thuis met haar man een heerlijk maaltje te kunnen eten.	action action goal
4	Tijdens het dansen moest zij in de balzaal hapjes rondbrengen voor de gasten.	action (or two simultaneous actions)
5	De koning, en natuurlijk was dit niemand anders dan onze koning Lijsterbaard, kwam naar haar toe en vroeg haar ten dans.	action action
6	De prinses wilde weigeren, maar de koning had haar hand al gegrepen en leidde haar naar het midden van de dansvloer.	failed action action (reason for failure) action
7	Daar gebeurde iets vreselijks: haar schort scheurde en alle etensrestjes die zij had verzameld, vielen om haar heen op de grond.	others event event
8	Alle gasten begonnen hard te lachen; het was ook zo'n raar gezicht.	action others
9	De prinses had zich nog nooit zo ellendig gevoeld.	state

Table 4.1: The fourth paragraph of 'Koning Lijsterbaard'

First of all I chose to categorize the first two clauses of the third sentence as actions, even though this may sound somewhat strange. If the sentence is changed into "De prinses werkte de hele dag en avond." or into "De koning verplichtte de prinses heel de dag en avond te werken.", the sentences are actions and therefore I put them into this category.

Furthermore the final sentence describes the internal state of feeling miserable. This could have been expressed using the following sentence: "Ze voelde zich ellendig.". However, describing the internal state using the sentence "De prinses had zich nog nooit zo ellendig gevoeld." shows that the princess felt extremely miserable, so this construction leads to more variety.

Adjuncts like 'op een dag' ('one day') in the first sentence are to be added by the Narrator agent. Adjectives such as 'groot' ('big') in the first sentence and 'lekkere' ('tasty') and 'heerlijk' ('delicious') in the third sentence should also be added by the Narrator. Properties of characters can usually be retrieved from the Character Information module, but properties of other nouns (such as the property 'big' for the 'diner') can probably not be retrieved. Therefore the Narrator agent will have to be extended with a list of possible properties for a number of nouns.

Furthermore the clause 'en natuurlijk was dit niemand anders dan koning Lijsterbaard' in the fifth sentence would not have been available in the input. The input probably contained the action with the agens Lijsterbaard, so the Narrator agent (or actually the Referring Expression Generator) should generate the entire sentence.

Finally the clauses "Daar gebeurde iets vreselijks:" (in the seventh sentence) and "Het was ook zo'n raar gezicht." (in the eighth sentence) would not be available in the input. The first sentence prepares the reader for the next sentence and the second one motivates why everyone laughed. These sentences could also be left out, because they do not have an important meaning. On the other hand, their function is to create a mood, so adding such sentences would be useful.

De Gelaarsde Kat

The second paragraph of this story contains mostly actions, but the paragraph shows which information can be added by the Narrator. Furthermore this example shows that some sentences can be described differently (such as an action that is described as an adjunct). Table 4.2 shows a classification for each clause that appears in the paragraph.

Sent	**Clause**	**Type**
1	Deze kat was een bijzonder dier; zodra zij de molen hadden verlaten, begon zij te spreken: "..."	setting action action
2	De verbaasde jongen besloot om te doen wat de kat zei.	action
3	Van zijn laatste geld kocht hij een rode cape, een hoed en een paar laarzen voor haar.	action
4	Intussen had de kat een konijn gevangen en, uitgedost in haar nieuwe kleren, bracht zij dat naar het kasteel van de koning: "..." zei ze.	action action action
5	Een paar dagen later bracht ze een mooie fazant naar het kasteel en weer zei ze: "...".	action action

Table 4.2: The second paragraph of 'De gelaarsde kat'

It looks like the input would only consist of actions, but the second sentence would probably be based on a state *and* an action. The action that the cat started speaking leads to an internal response (state) of being surprised. This state can then be combined with the next action (the boy's decision) and turned into an adjective.

The fourth sentence shows that also actions can be told differently. The input could consist of the following three actions: the cat caught a rabbit, the cat changed her clothes and the cat brought the rabbit to the castle. The second action is then described as an adjunct and thereby it adds more variety to the story.

De ring van de koningsdochter

In this story the second paragraph has been selected, and the classification for each clause in the paragraph is shown in table 4.3.

Sent	Clause	Type
1	Als het mooi weer was, ging die koningsdochter graag spelevaren op zee, en op een keer verloor ze daar haar ring.	setting setting event
2	Wat was ze bedroefd!	state
3	De koning liet overal bekendmaken dat zijn dochter haar ring in de zee had verloren en (hij liet bekendmaken) dat degene die hem vond, een flinke som geld zou krijgen.	action action

Table 4.3: The second paragraph of 'De ring van de koningsdochter'

The first two clauses describe the fact that the princess liked to play outside when the weather was good, which is available in the settings. It may be difficult to include the condition, but the fact that she liked to play outside can be stored in the settings.

One day she loses the ring and this leads to the internal state of being sad. This could have been told like "Ze was bedroefd", but the sentence "Wat was ze bedroefd!" shows that she was extremely sad, and using such a construction more variety is added.

De jongen die lezen en schrijven leerde

In the final story the sixth paragraph has been chosen for analysis, and table 4.1 gives the classification for each clause in the paragraph.

Sent	Clause	Type
1	Maar hij vond deze kunst zo mooi, dat hij zich kort daarna in een paard veranderde.	state action
2	Dit werd gekocht door de heer die de jongen het lezen en schrijven had geleerd.	action
3	Hij had het dier nog niet lang op stal, of hij merkte wel dat het niemand anders was dan zijn vroegere leerling.	action perception
4	Daarover was hij zeer boos, want hij wilde niet dat iemand buiten hem de kunst zou kennen.	state setting

Table 4.4: The sixth paragraph of 'De jongen die lezen en schrijven leerde'

This example first uses a 'zo ..., dat' construction showing that the boy liked the art very much. The second sentence contains a relative clause, which should be added by the Referring Expression Generator. The input would probably contain the action that the 'master' bought the horse and then the Referring Expression Generator should add the information that he is the master who taught the boy to read and write. Finally the Referring Expression Generator should transform these facts into the referring expression 'de heer die de jongen het lezen en schrijven had geleerd'.

The third sentence describes two plot elements (an action and a perception) which occur almost simultaneously and therefore the clauses can be combined using this construction.

4.2.2 Conclusions input and structure

The analysis described in this section provides a better understanding of which information will be available from the start and which information has to be included by the Narrator agent. Plot elements, such as actions, events, goals, states and perceptions are present in the input (represented as a fabula structure, explained in more detail in the next chapter). The settings are also available in the Character Information and Story World modules, but the Narrator should decide when to add this information. Finally the Character Information module can also be used to add adjectives to the generated texts.

All other information will not be available in the system, so this information should be added by the Narrator or will be left out. In order to create some adjectives for entities that do not appear in the Character Information, the Narrator may keep a list with possible properties for a number of nouns. An example is the adjective 'groot' ('big') for the noun 'diner' ('dinner') in the first story. Adding this adjective does not change the whole story, but it does make the story somewhat less monotonous. Doing this, we assume that the input contains information *if the meaning is important*. For example, if it is important for the rest of the story that the diner was tasty, then this will be available in the input, otherwise the Narrator may add adjectives itself. This also means that the Narrator should not add any adjectives if some information is specified in the input.

Furthermore the analysis shows that certain plot elements can be described in different ways. Examples are states which can be described as an adjective in a noun phrase (such as 'de verbaasde jongen' ('the surprised boy')) and actions which can be described as adjuncts (such as 'uitgedost in haar nieuwe kleren' ('dressed up in her new clothes')). In order to convert an action into an adjunct additional information is necessary, so for the moment I will only focus on telling states differently.

The examples in this section also show that there are a number of syntactic constructions that cannot be used yet, but these are described in section 4.3.5. Finally the examples use only a few facts from the settings, but it seems that facts from the settings should be included right *before* a new entity is introduced. The stories in the appendix affirm this assumption; the stories usually begin with some facts from the settings and include the remaining facts when a new character is introduced.

4.3 Sentence and Paragraph Structures

The second section of each appendix shows for each paragraph how many sentences are used and for each sentence how many clauses it contains. Furthermore it shows which rhetorical relations are used in order to combine different clauses into one sentence. Note that limiting relative clauses and clausal subjects and objects are considered parts of the main clause rather than separate clauses (such as in Rhetorical Structure Theory [MT87]). This means that a sentence such as "De jongen vergat nooit dat hij zijn rijkdom aan de kat te danken had" will be seen as one sentence instead of two clauses.

Non-limiting relative clauses often give additional information about an entity used in the main clause and therefore RST connects those clauses to the main sentence using an elaboration relation. Since I want to distinguish between an elaboration in which the second sentence elaborates on the whole sentence and an elaboration in which the second sentence tells something more about an *entity* mentioned in the first sentence, I will call the latter an elaboration2 relation. This kind of relation can then be expressed using a non-limiting clause.

4.3.1 Used rhetorical relations within sentences

The appendix shows which rhetorical relations are used in order to combine two clauses into a sentence. The table shows that each sentence can be categorized into one of the following categories:

Additive An additive relation simply enumerates two facts which are not related by any other relation. This is not an official RST relation, but can be used to combine two clauses after which ellipsis can be performed.

Cause In a cause relation one of the clauses represents the cause and the other one represents the consequence. This relation includes the volitional cause, volitional result, non-volitional cause and non-volitional result relations used in RST.

Purpose The purpose relation is similar to a cause relation, but in this relation something is done *in order* to reach a desirable state in the future.

Contrast This relation combines two clauses which describe two contradicting facts.

Temporal This relation combines two clauses which are connected using a temporal relation; this means that the clauses happen at the same time or consecutively.

Condition In the condition relation one clause represents the condition and the other clause represents the situation which will occur *if* the condition is true.

Elaboration In the elaboration the second clause tells something more about the first clause.

Elaboration2 This relation is a special kind of the elaboration relation; the second clause does tell more about the first clause, but only about an entity used in the clause.

As described in section 3.2 the first five of these relations are already supported by the Surface Realizer, but the other relations cannot be used yet. First of all, the condition relation has not been implemented yet, but the temporal relation supports the cue word 'als' ('if'). The algorithm to generate a condition relation would be the same as generating a temporal relation with the cue word 'als' ('if'), so this (or something similar) can be used when generating a condition relation. On the other hand, in the near future a condition will not be available in the input, so for the moment we can ignore this kind of relation.

The second relation not supported by the Surface Realizer is the elaboration relation. In some cases this relation can also be described using a different relation; in other cases the sentences can also be told separately. Take for example the sentence "Alle gasten begonnen hard te lachen; het was ook zo'n raar gezicht" (in the fourth paragraph of the first story). This can also be told using a cause relation like "Alle gasten begonnen hard te lachen, want het was zo'n raar gezicht", or separately like "Alle gasten begonnen te lachen. Het was ook zo'n raar gezicht." If we first try to express the relation using a different relation, most of the elaboration relations will have been removed. The remaining elaboration relations can then simply be told separately. This way the sentences will still be connected by an elaboration relation, but this relation will not be expressed explicitly. This means that there is no real need to implement a new relation in the Surface Realizer.

Elaboration2 relations are used extensively, so adding these to the system would definitely be useful and they can be expressed using relative clauses. The stories show that relative clauses appear in many different forms. First of all they can modify any phrase in the sentence (subject, object, indirect object). Secondly, the used relative pronoun can fulfill different grammatical roles, such as subject and object, or it can be combined with a preposition. Furthermore one can distinguish between limiting and non-limiting relative clauses and finally some relative clauses appear with a relative pronoun with implicit embedded antecedent, such as "De jongen besloot te doen *wat* de kat zei" ("The boy decided to do *what* the cat had said").

4.3.2 Number of clauses combined into one sentence

Currently the Surface Realizer is able to combine three clauses into one sentence. Analysis of existing stories shows that some sentences are more complicated. The first story contains four sentences consisting of four clauses and one clause containing eight clauses. The other stories contain much shorter sentences and the maximum is four clauses.

If we look at the more complex sentences it turns out that each of them (except one) contains either a (semi)colon or the cue word 'en' ('and'). A (semi)colon is often used with the elaboration relation and since the clauses in such a relation will be expressed *separately*, most of the sentences containing a (semi)colon are simplified automatically. Furthermore it shows that the cue word 'en' ('and') does make the sentence more complicated, but far less than other cue words do. This may be due to the fact that the cue word 'en' ('and') combines two main clauses instead of a main clause and a subordinate clause. Compare for example

the following two sentences “De ridder zette de prinses op zijn paard, nadat hij haar op had gepakt” and “De ridder pakte de prinses op en zette haar op zijn paard”. The first sentence of this example contains a main clause and a subordinate clause and the second sentence consists of two main clauses. As a result the second sentence is much easier to understand. Finally a sentence containing a relative clause is also only somewhat more difficult to read than a single clause, but this is probably caused by the length of the relative clause; relative clauses usually are rather short in comparison to other clauses.

Moreover all of the sentences which contain more than three clauses can easily be described using simpler sentences. Take for example the sentence which consists of eight clauses (from the third paragraph of the first story): “Toen liet de muzikantkoning haar potten bakken om te verkopen, maar toen zij met haar waren op de markt stond, kwam er een ruiter die dwars over haar potten heen reed en alles brak (die ruiter was natuurlijk koning Lijsterbaard die zich had vermomd).” This can also be told like this: “Toen liet de muzikantkoning haar potten bakken om te verkopen. Toen zij echter met haar waren op de markt stond, kwam er een ruiter die dwars over haar potten heen reed en alles brak. Deze ruiter was natuurlijk koning Lijsterbaard die zich had vermomd.” This example still contains one sentence containing four clauses, but this sentence can also be described as “Toen zij echter met haar waren op de markt stond, kwam er een ruiter. Hij reed dwars over haar potten heen en brak alles.” This final example may be somewhat worse, but it is still acceptable.

The stories thus show that a maximum of three clauses is acceptable, even though four clauses may be better in some cases. On the other hand using a maximum of four may also lead to overly complicated sentences, so for the moment I will use a maximum of three. Another possibility is to use a maximum of three clauses by default, and to use a maximum of four clauses if the sentence contains a relative clause or the cue word ‘en’ (‘and’).

In section 2.1.3 we mentioned that Strunk [Str18] suggests that sentence lengths should vary as much as possible. The tables in the appendix seems to confirm this suggestion; sentences consisting of one clause appear approximately as frequently as sentences consisting of two or three clauses.

4.3.3 Used rhetorical relations between sentences

As described before the different clauses within a sentence are related by rhetorical relations, but in a coherent text the consecutive *sentences* are also related. In the example stories in the appendix this is definitely the case, but in the first three stories most of these relations are not specified explicitly. The final story, on the other hand, shows some examples in which the consecutive sentences are related explicitly. These are marked by underlining the cue words which relate the sentences (take for example ‘echter’ in the second sentence which relates the sentence to the first one using a contrast relation).

Expressing some rhetorical relations between sentences explicitly can make the stories more coherent. On the other hand, if the Narrator will add many of those relations, this may result in a text in which the same cue words are used frequently. Since many clauses are simply connected by causal or temporal relations, the cue words ‘toen’ (‘then’) and ‘daarom’ (‘so’) would be used too often. Therefore we have to make sure the Narrator will not express too many relations explicitly.

4.3.4 Paragraph structures

The second section of the appendix also shows how many paragraphs the story consists of and how many sentences each paragraph contains. The first two stories have the clearest structure; they only have five or six paragraphs. Each paragraph contains about six sentences, with a minimum of four and a maximum of nine sentences. The other two stories contain more and shorter paragraphs; the paragraphs in the third story contain at most five sentences, and the paragraphs in the final story range from one sentence to four sentences. This does not mean that we can simply add a paragraph break after a fixed number of sentences, because this heavily depends on the content, but it does show that it is best to create paragraphs which are comparable in size.

The first story has the clearest structure and each paragraph has a different *function*. In the first paragraph the initial situation is described; a conceited princess rejects everyone who wants to marry her. In the second paragraph there is a change of this situation; the king decides to change her behavior and picks a

husband for the princess. The third paragraph describes the new situation; king Lijsterbaard is teasing the princess and she is suffering. The fourth paragraph describes the climax; the princess makes a fool of herself in front of everyone and she feels miserable. This leads to the moral of the story in the final paragraph; the princess regrets her earlier behavior and the story concludes with a happy ending.

This way the paragraph breaks can be added based on the contents of the paragraphs. This may be rather hard to do automatically, but the story also shows that there are easier ways to decide whether to add a paragraph break or not. First of all, in the first paragraph the central character is the princess and in the second paragraph the central character is her father. This means that a paragraph break may appear when the story moves from one *character* to another. Secondly, the second paragraph is situated in the castle and the third paragraph is situated in and around her new home, which shows that a paragraph break may also appear when there is a switch of *location*. Finally, the fourth paragraph starts with 'op een dag' to show that the paragraph happens some time later than the preceding paragraph, so a *time gap* may lead to a paragraph break as well.

The other stories show similar results; all paragraph breaks can be categorized as either a switch of character, a switch of time or a switch of location. This information can therefore be used to determine whether to add a paragraph break or not.

4.3.5 New syntactic constructions

Most syntactic constructions used in the stories can be used in our system, but the following constructions may be interesting extensions:

- Relative clauses: Er was eens een prinses, die heel verwaand was. (Taken from the first paragraph of the first story.)
- Appositions: Zo leefden zij, de prinses met haar koning Lijsterbaard, nog lang en gelukkig. (Taken from the final paragraph of the first story.)
- Cause relation using the cue word 'zo ..., dat': Hij rukte zo krachtig aan het touw, dat het brak. (Taken from the fifth paragraph of the fourth story.)
- Temporal relation using the cue word 'of': Hij had het dier nog niet lang op stal, of hij merkte wel... (Taken from the sixth paragraph of the fourth story.)

Furthermore the stories contain many verbs that require an additional clause (using a conjunction such as 'dat' and 'of'), including:

- vragen of: De koning vroeg of de markies er niet wat voor voelde met de prinses te trouwen. (Taken from the final paragraph of the second story.)
- doen alsof: De kat deed alsof zij erg onder de indruk was. (Taken from the fifth paragraph of the second story.)
- vergeten dat: De jongemen vergat nooit dat hij zijn rijkdom aan de kat te danken had. (Taken from the final paragraph of the second story.)

Finally some single sentences are described in a way which is not supported by the system yet. However, in order to select one of these constructions we need to know the intensity of the internal state or the intensity of the adverb in case of an action. The constructions include the following:

- Wat was hij blij! (Taken from the final paragraph of the first story.)
- De prinses had zich nog nooit zo ellendig gevoeld. (Taken from the fourth paragraph of the first story.)
- Hij rende weg zo hard als hij kon. (Taken from the eighth paragraph of the fourth story.)

Obviously not all of these constructions will be added to the Narrator agent, but some of them may be useful extensions because they add more variety to the generated stories.

4.3.6 Conclusions sentence and paragraph structures

First of all analysis of sentence structures confirms that combining at most three clauses into a sentence is reasonable. Furthermore it turned out that most of the rhetorical relations used *within* sentences are already supported by the Surface Realizer. The example stories showed that each clause is related to another clause by one of the following relations: additive, cause, contrast, purpose, temporal, condition, elaboration and elaboration2, which means that using this set of relations is satisfactory. Furthermore I will ignore the condition relation for the moment, because this type of information will not be available in the input, and I will express the elaboration relation implicitly (by simply using two separate sentences). This means that only the elaboration2 relation should be implemented in the Surface Realizer, and this type of relation can best be expressed by a relative clause.

The stories also showed that most rhetorical relations *between* sentences are expressed implicitly, but expressing some of them explicitly can make the stories more coherent. Furthermore paragraph breaks can occur when there is a change of time, location or character.

Finally there are some syntactic constructions which cannot be created by the Surface Realizer yet. First of all, relative clauses appear very often as a special kind of elaboration. Furthermore constructions using 'zo ..., dat' can be used as a special case of the cause relation and may lead to a greater variety of sentences. The description of two actions or events which occur almost simultaneously using 'of' (such as "Hij had het paard nog niet lang op stal, of hij merkte wel dat het zijn vroegere leerling was") may also lead to more variety, although it is not entirely sure if this can be realized based on the input fabula structure. The decision can be based on the time arguments of the different plot elements, but this way the construction might be used in cases in which it is not appropriate. Therefore we will ignore this construction for the moment, but it may be added once the input contains sufficient information. Finally there are some additional ways to describe states, such as "Wat was ze bedroefd!" or "Ze had zich nog nooit zo ellendig gevoeld".

4.4 Referring Expressions

The third section of each appendix shows which referring expressions are used. For each story the referring expressions have been analyzed, but this is described in detail only for the first story; for the other stories I will only describe the remarkable situations.

4.4.1 'Koning Lijsterbaard'

In the first story there are three important characters; the princess, her father and the king Lijsterbaard. The appendix shows that the princess is referred to by a noun phrase 17 times and by a pronoun 41 times. This may sound rather pointless, so I will try to find out *when* a noun phrase is used and *when* a pronoun is used. I will first analyze the referring expressions for the princess. She is the only female character, so it would be possible to use pronouns each time the princess is being referred to (since there is no other possible referent). Therefore analyzing those referring expressions will show best when a noun phrase can be used. Furthermore there are two male characters (the kings), so when generating a referring expression for one of them, there are some additional requirements: the referring expressions have to include enough information to make clear which king is being referred to.

Referring expressions for the princess

The first paragraph introduces the princess with the indefinite noun phrase 'a princess'. All other references in the paragraph are pronouns.

The second paragraph contains mainly noun phrases. First of all this may have been done, because the noun phrase is used to introduce her father and to specify whose behavior and whose window is described. Furthermore the use of noun phrases may indicate that the paragraph is told from a different point of view; the first paragraph is described from the point of view of the princess, and the second paragraph may be told from the father's point of view. On the other hand, this is not entirely true, since the father could not know that Lijsterbaard had heard him, so not the whole paragraph is told from his point of view. Therefore a final

possibility is that a different character is central in both paragraphs; in the first paragraph this is the princess and in the second paragraph it may be her father. The final reference to the princess in this paragraph is also a noun phrase ('zijn dochter'); this reference could have been a pronoun, but using this noun phrase the relationship between the two characters is expressed explicitly (the father/daughter relationship).

The third paragraph uses a pronoun for the first time, even though this is against our intuition. A possible reason for this is that the paragraph break shows a change of location instead of a change of time. This means that the princess is still fresh in the reader's mind and therefore a pronoun may be used. The paragraph contains nine pronouns before the first noun phrase is used. Furthermore the next referring expression is also a noun phrase, because the phrase contains the adjective 'ongelukkige'. This noun phrase describes the internal state of the princess and therefore a noun phrase is necessary. Finally the other references in the paragraph are all pronouns.

In the fourth paragraph the first reference is a noun phrase and then six pronouns are used. The next reference is a noun phrase, because many pronouns have been used in between. Then five pronouns are used again and the final reference is a noun phrase again. Note that both times a noun phrase is used, the previous noun phrase is three sentences before, so this may be a proper distance.

The final paragraph uses a noun phrase for the first reference. Next three pronouns are used and then a noun phrase is used again. At the end some more noun phrases are used; first in the noun phrase to refer to her father, then in the phrase 'zijn dochter' to show their relationship, one in subject position, and finally in the apposition in the final sentence.

Referring expressions for the kings

The first paragraph only contains a reference to Lijsterbaard, so he is introduced using this name. The second paragraph introduces the other king by saying that he is the father of the princess. Then this king is referred to by a pronoun twice, since he is the only male referent mentioned in the paragraph sofar. Then Lijsterbaard is referred to using his name to differentiate him from the other king. The final sentence contains references to both kings, but it also contains pronouns: "Toen de koning dat zag, haalde hij de muzikant in het paleis en liet zijn dochter met hem trouwen." This sentence therefore shows that it is sometimes allowed to use pronouns when the sentence contains references to several entities of the same gender. The first referring expression is the expression 'the king' to make clear the princess's father is meant. The second referring expression is 'hij'; since the father is used most recently it is obvious that the father is meant. Then 'de muzikant' is used to refer to Lijsterbaard. Next the subject is removed by performing ellipsis, but it makes sure that 'zijn' refers to the father again. The final referring expression is 'hem' which appears as object in the sentence. The referring expression 'de muzikant' also appeared as object, so the referent Lijsterbaard is most likely the correct one (since there is a preference for parallelism [CS98]).

The next two paragraphs only contain references to Lijsterbaard, so the analysis is similar to the referring expressions for the princess, since there is only one recent male referent. The paragraphs show that many different noun phrases are used; 'haar nieuwe echtgenoot', 'de muzikantkoning', 'haar man', 'de koning' etc.

The final paragraph contains references to both kings. The text uses a pronoun if the previous reference was to the same king and a noun phrase if the previous reference was to the other king.

4.4.2 'De Gelaarsde Kat'

In the second story there are five characters; the boy, the cat, the king, the princess and the giant. The referring expressions consisting of a noun phrase used in this story all use the same noun, except for the princess which is described by both 'prinses' and 'dochter'.

The story contains two important male characters (the boy and the king), so especially at the end (when the characters have met each other) many noun phrases are used to make sure the reader knows which entity is talked about.

4.4.3 'De ring van de koningsdochter'

In this story there are three main characters; the princess, the king and the farmer. Furthermore the two courtiers are male characters, so rather often two male characters appear in the same paragraph. This leads

to some interesting constructions, including the following sentence (taken from the fifth paragraph): "Toen hij hoorde dat het boertje de ring had, wilde hij hem niet doorlaten, of hij moest hem het derde deel beloven van de beloning, die hij bij de koning zou bedingen."

This sentence contains references to two male characters, and still six pronouns are used. The first 'hij' refers to the courtier, which can be derived from the context. The second clause contains the pronouns 'hij' and 'hem'. Since it is most likely that the referents appear in the same order as they do in the previous clause (see for example Grosz's Centering theory [MT87], but also the algorithm of Lappin and Leass [LL94]), these pronouns can easily be resolved. The third clause, however, also contains the referring expressions 'hij' and 'hem', but in this clause they refer to other referents. The only way to resolve these pronouns is by looking at the meaning of the sentence. When this is done automatically, the Referring Expression Generator would probably generate some noun phrases instead of pronouns in order to avoid confusion.

4.4.4 'De jongen die lezen en schrijven leerde'

This story uses many different referring expressions for the two main characters, because they can change themselves into anything they like. Their visual appearance changes the whole time, so many different expressions can be used throughout the story. These expressions are listed in the table in the appendix, but it is not useful to compare the results with the results of the other three stories.

4.4.5 Conclusions referring expressions

The stories show that there are many differences between the referring expressions. For example, the first story uses more pronouns than noun phrases, while the second story uses more noun phrase than pronouns.

Furthermore it seems that noun phrases are used in the following cases:

- At the beginning of a paragraph.
- If a pronoun has been used a number of times (about five or six times).
- If the antecedent has been referred to by a pronoun for the last two sentences.
- If the antecedent has not been mentioned for two sentences.
- If the use of a pronoun would lead to ambiguity which referent is being referred to, since there are two salient characters of the same gender.
- If the referring expression includes additional information, such as adjectives.
- If the referring expression shows a relation between two characters, such as the father/daughter relationship.

Finally, additional use of noun phrases may indicate a different point of view or a time gap between the two consecutive references.

4.5 Rhetorical Structure Theory

In this section two paragraphs are selected to check if the texts can be represented in the same way as done in RST [MT87] using the relations specified in section 4.3.1. This is necessary because our representations (called document plans, which are described in more detail in the next chapter) will look somewhat like RST structures. Section 4.5.1 analyzes the beginning of the first paragraph of the first story and section 4.5.2 analyzes the first paragraph of the third story.

4.5.1 Example 1: Taken from 'Koning Lijsterbaard'

In the tree in figure 4.1 the numbers refer to the following underlying propositions:

1. Er was eens een prinses.
2. Zij was heel verwaand.
3. Ze vond zichzelf heel mooi.
4. Ze lachte iedereen uit die er wat minder mooi uitzag.
5. Elke dag verschenen er prinsen, edelmannen en koningen aan het hof.
6. Zij wilden haar hand vragen.
7. Op allen had ze wel iets aan te merken.

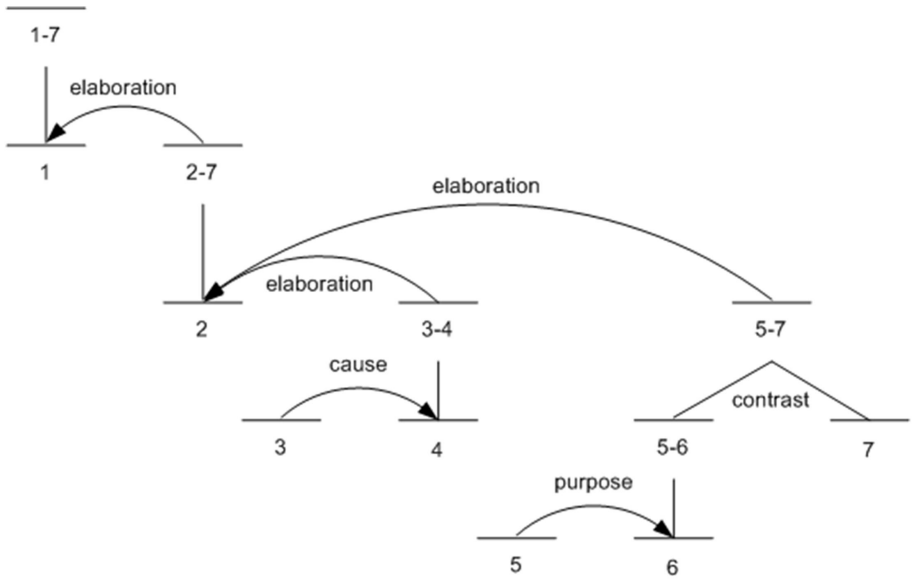

Figure 4.1: Example tree for the beginning of the first story

The first two clauses are combined by an elaboration relation. The second sentence contains the 'zo ..., dat' construction (which represents a causal relation) and the whole sentence is related to the second clause of the first sentence by means of an elaboration relation; the sentence tells more about the fact that the princess is conceited. The third sentence, containing a purpose relation and a contrast relation, further elaborates on being conceited, and is therefore also connected to the second clause of the first sentence by an elaboration relation.

4.5.2 Example 2: Taken from 'De jongen die lezen en schrijven leerde'

In the tree in figure 4.2 the numbers refer to the following underlying propositions:

1. Er was eens een jongen.
2. Hij kon niet lezen en schrijven.
3. Hij wilde het graag leren.

4. Hij vond niemand die hem onderwijs wilde geven.

5. Eindelijk ontmoette hij een heerschap

6. Het heerschap wilde hem in dienst nemen.

7. Het heerschap wilde hem lezen en schrijven leren.

8. De jongen nam dit graag aan.

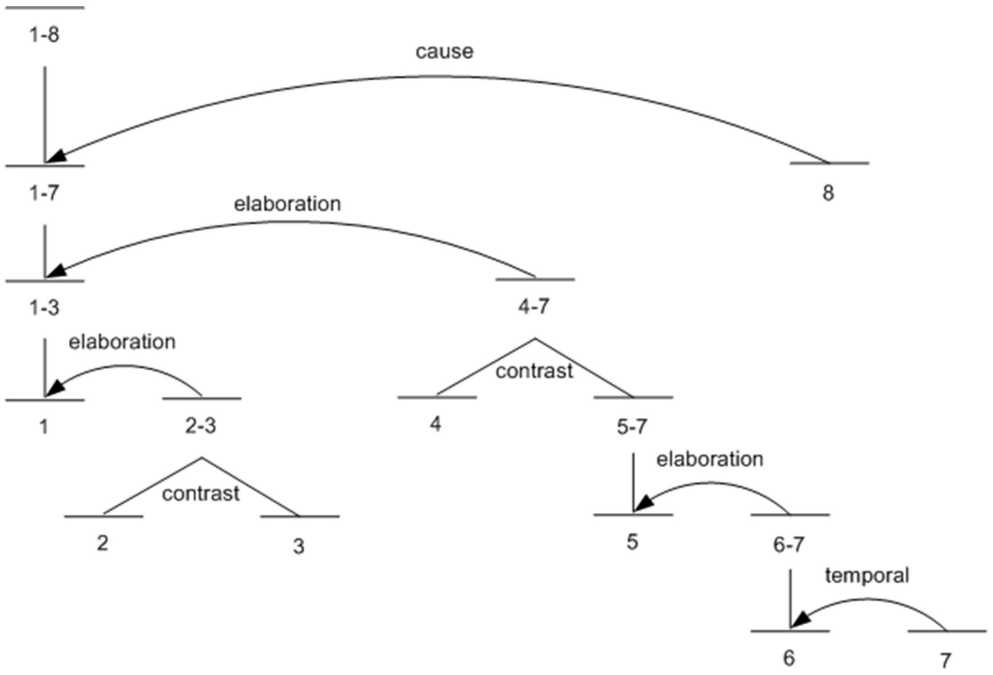

Figure 4.2: Example tree for the beginning of the third story

The beginning of the paragraph is rather straightforward. The second and the third clause can be connected by a contrast relation, and the result can then be related to the first clause using an elaboration relation. Furthermore the sixth and seventh clause can be connected by a temporal relation (assuming that the master will first hire the boy and *then* teach him how to read and write). This result can be connected to the fifth clause by an elaboration relation and this result can then be related to the fourth clause by a contrast relation.

Finally it is not entirely clear how the final sentence in this example should be related to the other sentences. The pronoun 'dit' ('this') refers to the relative clause of the preceding sentence, but it seems impossible to connect the sentence only to this relative clause. Therefore I chose to relate the sentence to the entire tree using a cause relation; the boy accepted this *because* he didn't know how to read and write, but he did want to learn it.

4.5.3 Conclusions Rhetorical Structure Theory

The two examples in this section show that it is possible to describe fairy-tales using RST. Note that there may be different correct possibilities, but the point is that similar structures can be used in the Virtual Storyteller.

4.6 Conclusions

The analysis described in this chapter looked at some existing stories from three different points of view. First of all, the stories were looked at by examining the whole story and determining which information will be available and where. This analysis provides a better understanding of which information will be available from the start and which information has to be included by the Narrator agent.

The second point of view is at the level of paragraphs and sentences. Analysis of sentence structures affirms that combining at most three sentences into one is reasonable. Furthermore it turned out that relative clauses, cause relations expressed using the 'zo ..., dat' construction and temporal relations expressed using the 'of' construction may be useful additions. Finally there are some more ways to describe states, such as "Wat was ze bedroefd!" or "Ze had zich nog nooit zo ellendig gevoeld".

The final point of view is at the level of words and phrases. This analysis only focussed on referring expressions, but this may be extended to include other phrases as well. Analysis of referring expressions led to some conclusions about when to use a noun phrase and when to use a pronoun. Noun phrases are generally used at the beginning of a paragraph, if a pronoun has been used a number of times, if the antecedent has not been mentioned in the preceding two sentences and if it is unclear which referent is being referred to. Furthermore they may be used if additional information is available (such as adjectives describing the internal state of the referent) or if a referring expression can express a relation between two characters explicitly (such as father/daughter). Finally it appears that (if a sentence contains two references to the same referent) the *first* referring expression in the sentence should be the noun phrase.

Chapter 5

Project Goals and Architecture

Using the results of the literature study and the analysis of human-written fairy-tales, I will give a more detailed description of the project goals in section 5.1. Section 5.2 then describes the global architecture of the Narrator agent, which is represented as a pipeline in such a way that the different components of the pipeline can accomplish the goals given in the first section. Finally section 5.3 specifies the inputs and outputs of the components of the pipeline, and the following three chapters describe the design and implementation of the components in more detail.

5.1 Detailed Project Goals

As mentioned before our ultimate goal is to produce texts which are comparable in quality to human-written fairy-tales. Since this is not realizable in a project like this one we will redefine this goal to the goal of generating narratives which are attractive for the reader, and I believe the most important aspects of this term are variety and coherence. In order to achieve this we first have to combine the two already existing parts: the part that generates the fabula structure and the Surface Realizer. The Surface Realizer already takes care of syntactic aggregation, referring expression generation and surface form generation, so the only tasks left are the tasks of content determination, assigning the correct rhetorical relations and lexicalization. According to Reiter and Dale [RD00] I will implement the natural language generation process as a pipeline consisting of a Document Planner, a Microplanner and a Surface Realizer, as will be described in more detail in the following section. Therefore the main goal is to design and implement an initial Document Planner and an initial Microplanner.

Furthermore the original Surface Realizer was meant to combine only a few simple sentences into more complex ones. This means that it expected at most three dependency trees as input and tried to combine them into one dependency tree. If we want to generate longer texts the original algorithm fails to produce correct output, so the Surface Realizer has to be extended to support more complex input as well.

Once first versions of the Document Planner and Microplanner have been built and the Surface Realizer can deal with larger input, the system will be able to generate simple fairy-tales. However, these stories will still be rather monotonous because the plot elements are translated into text in the exact same way throughout the story. This will result in stories such as "Plop had honger. Plop zag een appel, dus hij at de appel." ("Plop was hungry. Plop saw an apple, so he ate the apple.") If we compare such results with the stories analyzed in the previous chapter, there is still an obvious difference. When we combine the results from the analysis with the results from the literature we can construct a large number of additional goals, which all take place either in the Document Planner, the Microplanner or in the Surface Realizer.

Additional goals of the Document Planner are adding some more detail to the generated texts, such as adjectives and adverbs. Furthermore it should add background information when necessary and it should decide which rhetorical relations hold between different parts of the text. Finally the Document Planner should decide when to add paragraph breaks if the generated texts get longer.

Furthermore we have to make sure that the Microplanner will have access to a rather large lexicon in order to choose different synonyms once in a while. If possible it should also have different ways of expressing

the same content, for example by supporting active voice as well as passive voice or by supporting different ways of describing an internal state.

Finally the Surface Realizer will also be improved, especially by extending the Referring Expression Generator. The original algorithm failed to produce a correct referring expression in some cases (such as the situation mentioned in section 3.3.2 in which a noun phrase was used instead of a pronoun), so this has to be corrected. Furthermore the grammar can be improved by making sure that less sentences start with the subject in order to avoid monotony (see section 2.1.3). Additionally the grammar has to be extended in order to inflect adjectives as well. Finally, analysis of human-written fairy-tales shows that relative clauses are used frequently, so supporting this construction is also one of our project goals.

Obviously there are far more possible project goals but due to the limited amount of time I will only focus on the issues mentioned sofar. Some other options are discussed in the future work section in chapter 9.

5.2 Global Architecture of the Narrator Agent

As described in the previous section the architecture of the Narrator agent is represented as a pipeline. The pipeline is based on the pipeline defined by Reiter and Dale [RD00] and is shown in figure 5.1. This pipeline consists of the same components as the pipeline described by Reiter and Dale, but the tasks performed by each component differ somewhat. Furthermore the tasks marked with an asterisk have already been implemented.

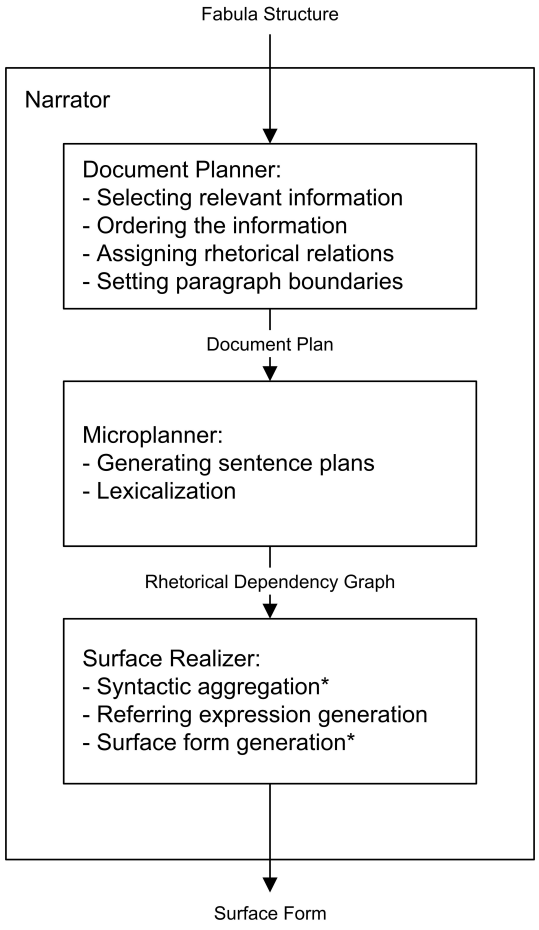

Figure 5.1: Global architecture of the Narrator agent

As described in chapter 3 Hielkema [Hie05] has already implemented a Surface Realizer that can be used in the Narrator agent. The component's main task is performing syntactic aggregation. Reiter and Dale place aggregation in the Microplanner, but since the Virtual Storyteller only performs *syntactic* aggregation, Hielkema believes the best place to perform this kind of aggregation is in the Surface Realizer. The generation of referring expressions can best be done *after* syntactic aggregation for the following two reasons. First of all, generating referring expressions after performing syntactic aggregation is necessary because otherwise ungrammatical sentences could be generated. Take for example the propositions "De prinses was bang." ("The princess was scared.") and "De prinses schreeuwde." ("The princess screamed.") connected by a cause relation. If these propositions would be converted into two separate sentences noun phrases can be used in both sentences. However, if the propositions would be combined into one sentence, only one noun phrase could be used: the sentence "De prinses was zo bang, dat de prinses schreeuwde." ("De princess was so scared that the princess screamed.") implies that the noun phrases refer to different characters (following Chomsky's Government and Binding Theory [Cho82]) and since in this example the noun phrases refer to the *same* princess the sentence is ungrammatical. This example shows that the choice of referring expression partly depends on the fact whether syntactic aggregation can be performed or not. Secondly, generating referring expressions after performing syntactic aggregation is simply more efficient, because otherwise the Referring Expression Generator could generate complicated referring expressions which are then removed by the component performing syntactic aggregation. Therefore the task of generating referring expressions is taken care of in the Surface Realizer.

The final difference is the fact that the lexicalization process has been split into two separate parts. The first part is carried out in the Microplanner, in the same way as done in the pipeline defined by Reiter and Dale, and the second part is carried out in the Surface Realizer when generating referring expressions.

5.3 Inputs and Outputs

The information passed between the different components of the pipeline is represented in different formats. Since the language generation process is represented as a pipeline, the output of one component is the input to the next component. In this section I will first describe the inputs and outputs to the different components as specified by Reiter and Dale [RD00] and then the inputs and outputs used in the Narrator agent.

5.3.1 Inputs and outputs in a standard NLG system

In this section I will describe the inputs and outputs to the different components of a natural language generation system as specified by Reiter and Dale.

Input NLG System - Input Document Planner

The input to a natural language generation system consists of a number of databases and knowledge bases containing information available to the system, and the goal of the text to be generated.

Output Document Planner - Input Microplanner (Document Plans)

Next it is the task of the Document Planner to convert this input into a document plan by performing content determination and document structuring. A document plan specifies how messages are to be grouped together and is typically represented as a tree structure. The leaves of the tree represent the information (such as sentences or predicates) and the internal nodes represent groups of related messages and may correspond to sections and paragraphs. An internal node may also specify rhetorical relations between the node's children (such as cause or sequence).

Output Microplanner - Input Surface Realizer (Text Specifications)

Then the Microplanner converts the document plan into a text specification which contains all information needed by the Surface Realizer to transform this specification into text in natural language. Like document

plans text specifications are usually represented as tree structures, but in such trees the leaf nodes are phrase specifications which represent sentences. A phrase structure can be specified at different levels of abstraction:

1. An *orthographic string* is simply a sentence in natural language, including punctuation and capitalization.

2. *Canned text* is also a sentence in natural language, but punctuation and capitalization may still be needed.

3. At the next level there are *abstract syntactic structures* in which the words to be used and the grammatical roles are present, but the actual sentence still has to be constructed by the Surface Realizer. An example is shown in figure 5.2 for the sentence "The courier delivered the green bicycle to Mary". You can see that the words are known, even though the words are still uninflected. Furthermore the grammatical roles, such as subject and object, are known and the remaining necessary information can be represented using other features.

4. The final level of abstraction is represented by *lexicalized case frames* (see figure 5.3). These look somewhat similar to the abstract syntactic structures, but instead of syntactic roles (such as subject and object) lexicalized case frames specify semantic roles (such as possessor). An advantage of using semantic roles rather than syntactic roles is that the system can generate a greater variety of sentences (e.g. the system can still decide to present the sentence in active or passive form).

Finally the aforementioned representations can be combined, for example when it is useful to represent some parts as canned text while other parts can be represented by other structures.

$$\begin{bmatrix} head: & |deliver| \\ subject: & [head: |courier|] \\ object: & \begin{bmatrix} head: |bicycle| \\ mod: [head: |green|] \end{bmatrix} \\ indirectobject: & [head: |Mary|] \end{bmatrix}$$

Figure 5.2: Example abstract syntactic structure, taken from [RD00]

$$\begin{bmatrix} head: & |deliver| \\ participants: & \begin{bmatrix} arg1: & [head: |courier|] \\ arg2: & \begin{bmatrix} head: |bicycle| \\ mod: |green| \end{bmatrix} \\ arg3: & [head: |Mary|] \end{bmatrix} \end{bmatrix}$$

Figure 5.3: Example lexicalized case frames, taken from [RD00]

Output Surface Realizer - Output NLG System

The output of the natural language generation system is a text in natural language. Most systems pass this resulting text to a document presentation system which makes sure the text is presented on the screen properly. If this is the case the Surface Realizer not only has to generate the text, but should also include mark-up symbols which can be understood by the document presentation system. For example, if the document presentation system expects input in HTML-format (which is for example done in StoryBook, see section 2.3.3), the Surface Realizer has to add <p> for each paragraph break.

5.3.2 Inputs to the Narrator

As described in section 3.1 the Plot agent generates the fabula structure together with the Character agents and the World agent. This structure is then passed to the Narrator agent and represents the main part of the input to the Narrator. Apart from the fabula structure the Narrator can include information from the Story World and from the Character Information modules, so these modules are also described in this section.

Fabula Structure

The fabula structure is a causal network based on the General Transition Network Model of Trabasso et al. [TVdBS89]. Figure 5.4 shows which elements may appear in a fabula structure and the possible relations which can hold between the elements. Figure 5.5 shows an example of a fabula structure based on a real story (see [Swa06] for the complete story and explanation).

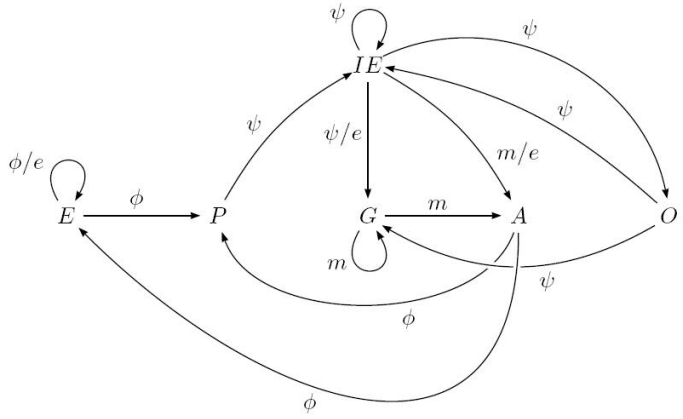

Figure 5.4: Fabula structure, taken from [Swa06]

The nodes in the fabula structure are *plot elements* and the edges are the causal relations that hold between the plot elements. The fabula structure may contain the following plot elements:

- A *goal* (G) is a desired or undesired state or activity for a character. Characters can have goals at the initiation of the story, but characters can also adopt new goals based on their internal states, beliefs and emotions.
- An *outcome* (O) is the result of a goal. This outcome does not have any contents like the other plot elements do; it is just a positive, a negative or a neutral outcome, depending on whether the goal has been fulfilled successfully.
- An *internal element* (IE) is an emotion, a cognition or a belief. Examples are being hungry or believing that somebody tries to kill you.
- An *action* (A) changes the world state and is initiated by a character. An action can either be explained by an attempt to reach a certain goal or by an internal state.
- An *event* (E) is similar to an action. However, an event is not initiated by a character, but simply happens coincidentally.
- A *perception* (P) is noticing that something in the world has changed. Examples of perceptions are seeing and hearing.

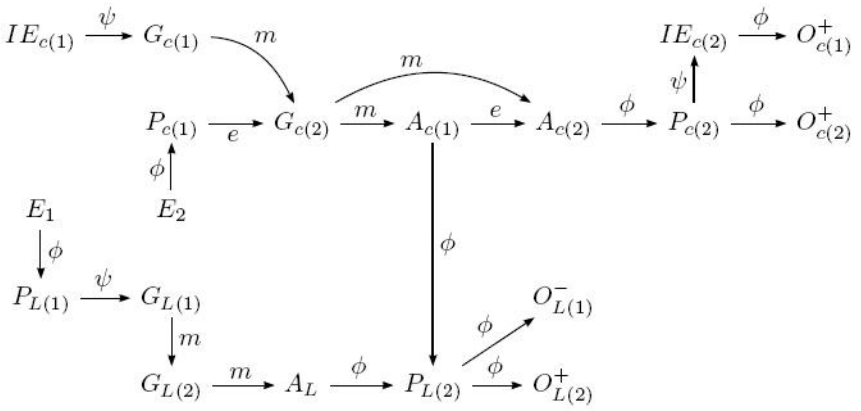

Figure 5.5: Example of an actual fabula structure, taken from [Swa06]

Furthermore the fabula structure contains the *causal relations* that hold between the plot elements. The following relations are used:

- A plot element *enables* (e) another plot element if the latter plot element has become possible because of the first plot element. An example is an event enabling another event; in this case the first event changes the world, and because of these changes the preconditions of the second event may be satisfied.
- A plot element may also *psychologically cause* (ψ) another plot element, for example a perception leading to an internal element.
- Furthermore a plot element may *physically cause* (ϕ) another plot element. This is the case when a plot element automatically causes another plot element which has nothing to do with beliefs or internal states. An example is an event physically causing a perception.
- An example of a plot element *motivating* (m) another plot element is a goal motivating an action. In this example the action is carried out in order to reach the goal, so the action is motivated by the goal.

Note that the motivate and enable relations are no official cause relations; RST [MT87] supports separate relations for cause, enable and motivate. On the other hand the enable and motivate relations can be expressed using the same cue words as the causal relations (for example the cue word 'omdat' ('because') for a motivate relation), and therefore these relations can be seen as special cases of the cause relation.

The plot elements in the fabula structure are represented in OWL. OWL (The Web Ontology Language) [W3C04] is a markup language for publishing and sharing data on the Internet using ontologies. It can be used by applications that need to process the content of the information instead of just presenting the information to the user and therefore it is useful for the representation of plot elements. Examples of some plot elements represented in OWL are given in figure 5.6.

In the figure you can see that a plot element has a number of properties. Each type of plot element has a different set of arguments:

- An event has a *time* argument specifying the moment at which the event occurred. Furthermore it has a *starttime* and an *endtime* in case the event takes some time (such as an earthquake). Finally an event has an *agens* argument specifying the object causing or doing the event.
- A perception also has a *time* argument. Furthermore it has a *character* as argument specifying the character that has the perception, and the *contents* argument specifies what the perception entails.

```
(rdf:type goal52 AttainGoal)
(fabula:time 4)
(fabula:character goal52 knight1)

(rdf:type action126 Rescue)
(fabula:hasContext action126 goal52)
(fabula:character action126 knight1)
(fabula:patiens action126 princess2)

(rdf:type action135 GoTo)
(fabula:starttime 5)
(fabula:endtime 7)
(fabula:motivates goal52 action135)
(fabula:agens action135 knight1)
(fabula:target action135 castle3)

(rdf:type action136 Open)
(fabula:starttime 8)
(fabula:endtime 8)
(fabula:motivates goal52 action136)
(fabula:agens action136 knight1)
(fabula:patiens action136 gate4)
```

Figure 5.6: Examples of plot elements represented in OWL

- An internal element has the same arguments as a perception: it has a *time*, a *character* and a *contents* argument.
- A goal also has a *time*, a *character* and *contents* argument.
- An action also has a *time* argument, but like events an action has a *starttime* and an *endtime* argument because actions may also take some time. The *agens* specifies the character performing the action, the *patiens* is the character undergoing the action, the *target* is for example a destination for a 'goto' action and the *instrument* is the object with which the action is performed (note that this can also be a set of instruments).
- An outcome only has a *time* argument which specifies when the outcome occurred.

Note that the *contents* argument is represented by the inverse relation $fabula : hasContext$. Furthermore each specification of a plot element begins with a line specifying the type of the element using the $rdf : type$ relation. An example of such a type specification is the first line in the second paragraph which tells that $action126$ is an action of type *Rescue*. Finally the figure shows that the causal relations between the different plot elements are represented in the exact same way as the other properties; as a relation which holds between two arguments (such as the *motivates* relation between $goal52$ and $action135$).

Character Information

The Narrator agent can use the Character Information module to retrieve more information about the characters, such as the names and properties of the story's characters. This information can then be included in the story in order to generate better referring expressions or to add entire sentences and relative clauses. The Character Information module also contains the relations that hold between different characters. Examples are the father-of relation (such as the fact that $king01$ is the father of $princess01$), but also the part-of relation (such as the fact that $gate01$ is part of $castle01$). All of this information can be combined with the information available in the fabula structure and included in the document plan.

Story World

The Story World module contains a description of the world, a model of what the world looks like at every moment in time. Examples of facts stored in the Story World are locations of objects and facts where people

live. All of these facts can be combined with the facts from the fabula structure and added to the document plan in order to generate more coherent texts.

Emotion Model

In future versions the input will be annotated with the emotions of all characters in the story. The system differentiates between eight different types of emotions and the intensity of an emotion is modelled by a number between -100 and +100. Once the input has been annotated with these numbers this information can be used for a number of different things. At some places in the document I will refer to these numbers and explain how they can be used in the Narrator agent.

5.3.3 Document plans (output of the Document Planner)

The Document Planner has to select the relevant information from the fabula structure and the other input modules, and has to build a document plan containing all this information. Like the document plans of Reiter and Dale they are represented as tree structures in which the leaves are the plot elements (such as actions, goals and perceptions) and the other nodes represent the rhetorical relations which hold between their child nodes. An example of a document plan is given in figure 5.7.

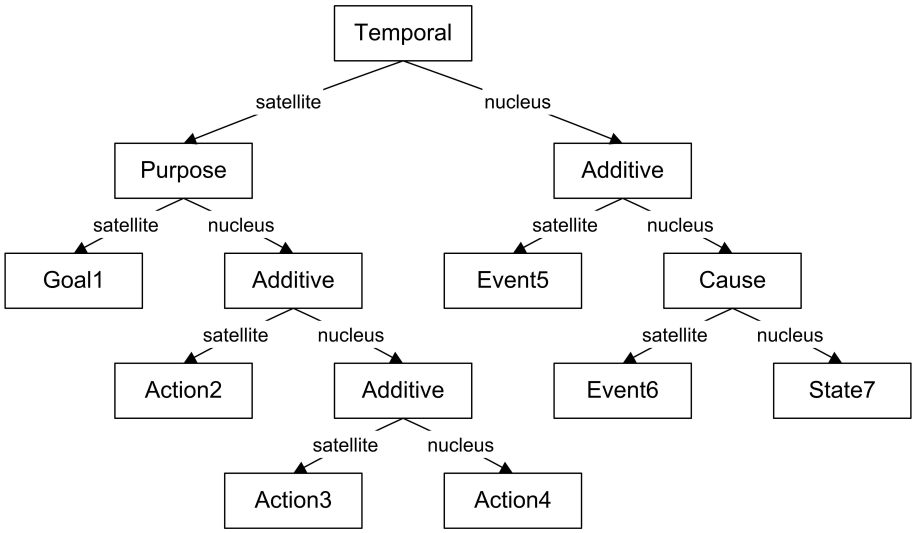

Figure 5.7: Example of a document plan

5.3.4 Rhetorical dependency graphs (output of the Microplanner)

The Microplanner has to convert a Document Plan into a Rhetorical Dependency Graph, which is a coherent tree structure in which dependency trees are connected by rhetorical relations. Reiter and Dale use the term text specification, but I will use the term Rhetorical Dependency Graph according to Hielkema [Hie05], see also section 3.2. The Rhetorical Dependency Graph looks very much like a document plan, but the leaves are replaced by dependency trees.

5.3.5 Output of the Narrator

The final output of the Narrator agent is the text in natural language which can be converted into speech by the Speech agent. In the future this text should include annotation, such as the emotions of the characters. Since the input is not annotated with the emotions yet (see the end of section 5.3.2), the output will not be annotated either, but once the input has been annotated, this can easily be extended.

Chapter 6

Document Planner

This chapter describes the design and implementation of the first module of the Narrator pipeline, the Document Planner. This includes an explanation of the architecture and all of its components. The chapter ends with an example in order to illustrate the functioning of the different components.

6.1 Architecture

As explained in the previous chapter the Document Planner is responsible for selecting the relevant information, ordering the information, adding rhetorical relations and setting paragraph boundaries. All of these tasks are performed by the Initial Document Plan Builder which creates an initial document plan. Once this document plan has been created the following three kinds of transformations can be performed on the document plan: adding background information, changing the way in which an internal state is expressed and adding sentences to create a mood. After performing these transformations the resulting document plan may still contain long branches, so the final component of the Document Planner is the Branch Remover which makes sure that long branches are split in shorter branches. The architecture of the Document Planner is shown in figure 6.1 and all of its components are described in more detail in this chapter.

6.2 Initial Document Plan Builder

The Initial Document Plan Builder takes as input a fabula structure as described in more detail in section 5.3.2. Its main task is to convert this structure into an initial document plan (see section 5.3.3), specifying what information should be present in the final text and specifying how the information should be related. In this section I will first explain how all information about the plot elements can be retrieved from the fabula structure and how all this information can be stored in one Java Plot Element structure. Then I will explain how the entire fabula structure can be converted into a document plan and finally I will explain how the different tasks of the natural language generation process are performed when creating the document plan.

6.2.1 Retrieving all information about the plot elements

As explained in the previous chapter the fabula structure is represented as one large OWL file containing tuples such as $(rdf : type\ Goal52\ AttainGoal)$ and $(fabula : hasContext\ Action126\ Goal52)$. This OWL file can be loaded using the Java Theorem Prover (see [FJF04] and [FJF03]), an object-oriented reasoning system implemented in Java. Using JTP the data stored in an OWL file can be loaded into a reasoning system and this system can then be queried.

Knowledge in JTP is represented in conjunctive normal form and an example of a literal used in JTP is $(rdf : type\ Goal52\ AttainGoal)$. In this example $rdf : type$ is the relation which holds between the arguments $Goal52$ and $AttainGoal$. Once the OWL file has been loaded an example of a query that can be

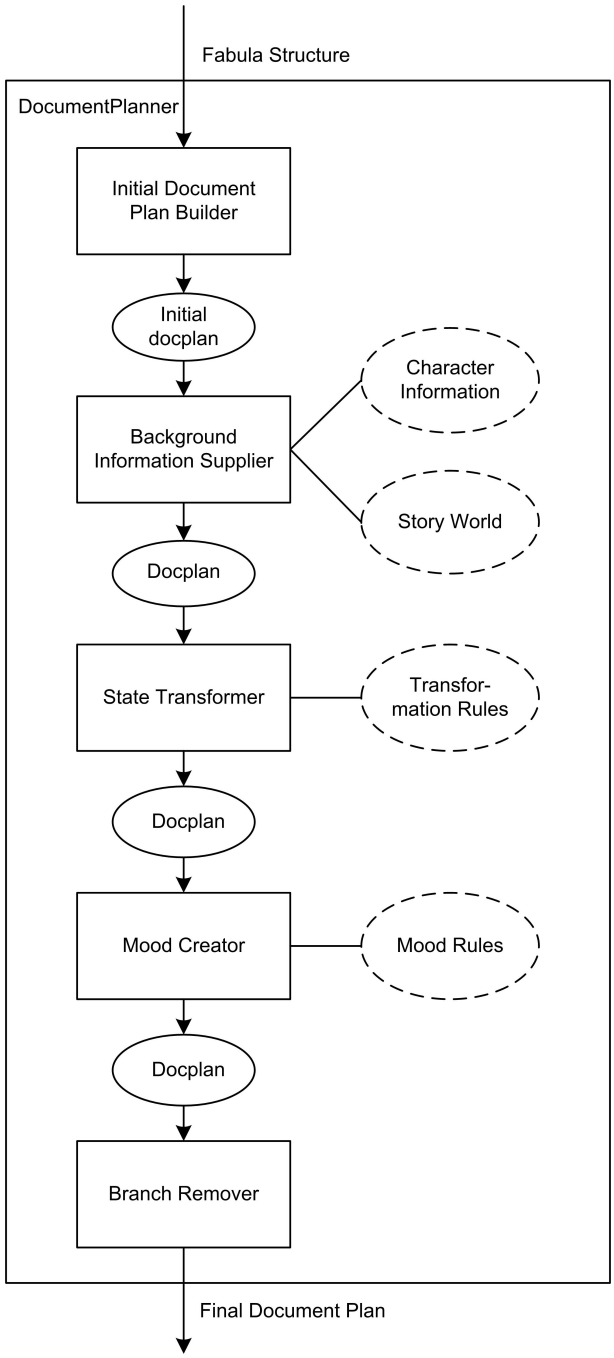

Figure 6.1: Architecture of the Document Planner

asked is "What type has Goal52?", represented as $(rdf : type\ Goal52\ ?x)$, and the answer will be stored in the variable $?x$.

In section 5.3.2 has been explained that each plot element has a different set of arguments. Examples of each of the plot elements are shown in figure 6.2. Using these representations each plot element can be modelled by a Plot Element structure which consists of five Strings and a Vector: the type of element, the name, the agens, the patiens, the target and a set of instruments, some of which may be null. Perceptions,

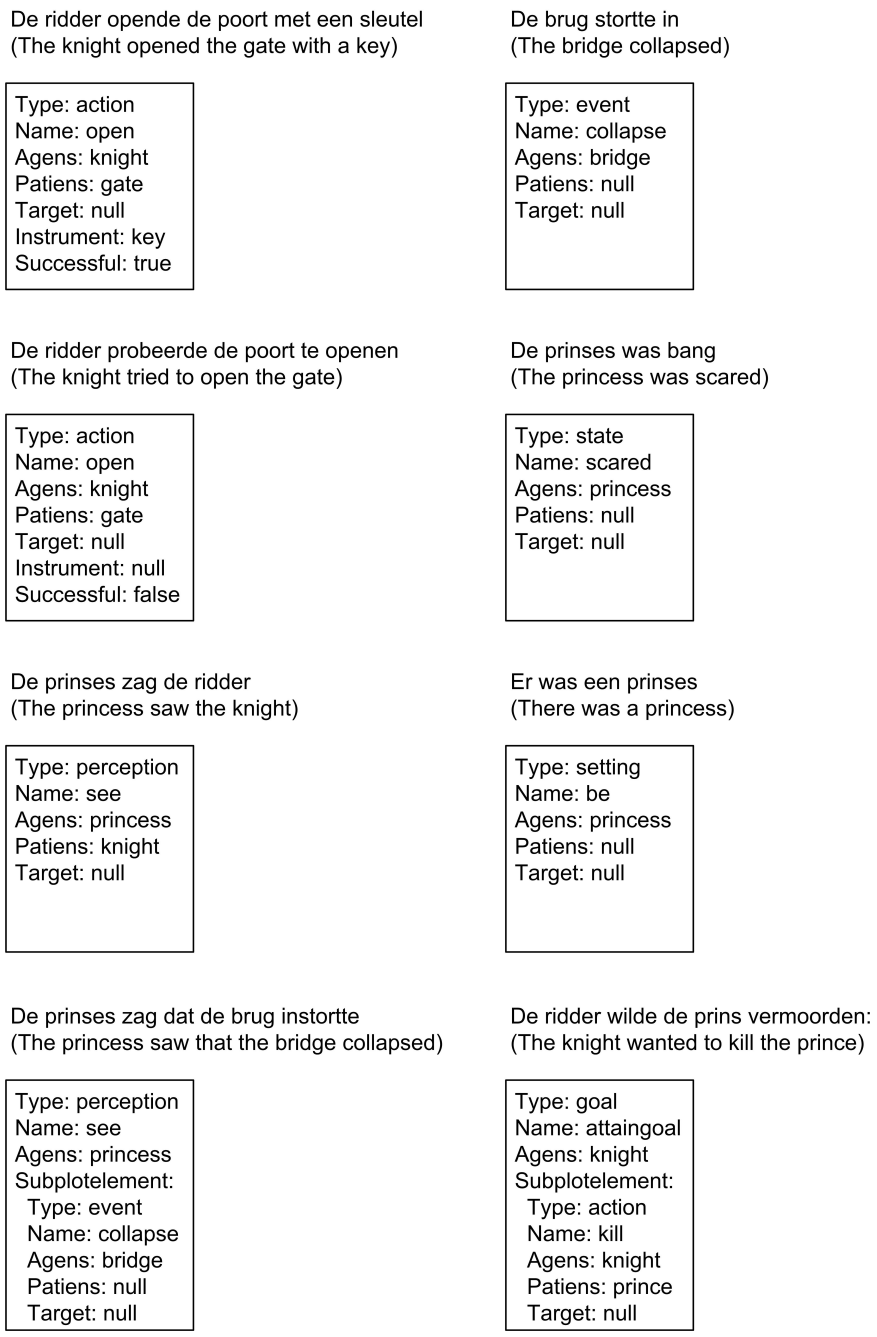

Figure 6.2: Examples of plot elements

beliefs and goals differ from the other types of elements since they can have another plot element as one of their arguments. Such a sub plot element is related to the original plot element by the $fabula : hasContext$ relation, and can be an entire plot element or a relation that holds between two elements, such as $isLocated$. Examples of both types of sub plot elements are given in figure 6.3. The example on the left side of the figure represents the goal of eating an apple, and the example on the right side of the figure represents the belief that there is an apple in the house. If the sub plot element is an entire plot element (as the example left in the figure) the algorithm for retrieving a plot element from the fabula structure is simply called recursively.

If the sub plot element is represented as the example on the right side of the figure, all arguments (in this case the apple and the house) are retrieved and the fabula is queried to get the relation that holds between the arguments (in this case the *isLocated* relation). Then a new plot element is created with the name set to this relation and the agens and target set to the arguments. In the figure you can see that entities can also have a $fabula : hasInterpretation$ relation with another entity, so if such a relation exists we have to make sure the correct entities are used (the ones at the bottom of the figure). Finally note that sub plot elements can have sub plot elements as well, in order to generate complex sentences such as "De ridder wilde dat de prinses zou denken dat hij haar wilde vermoorden." ("The knight wanted the princess to think that he wanted to kill her.").

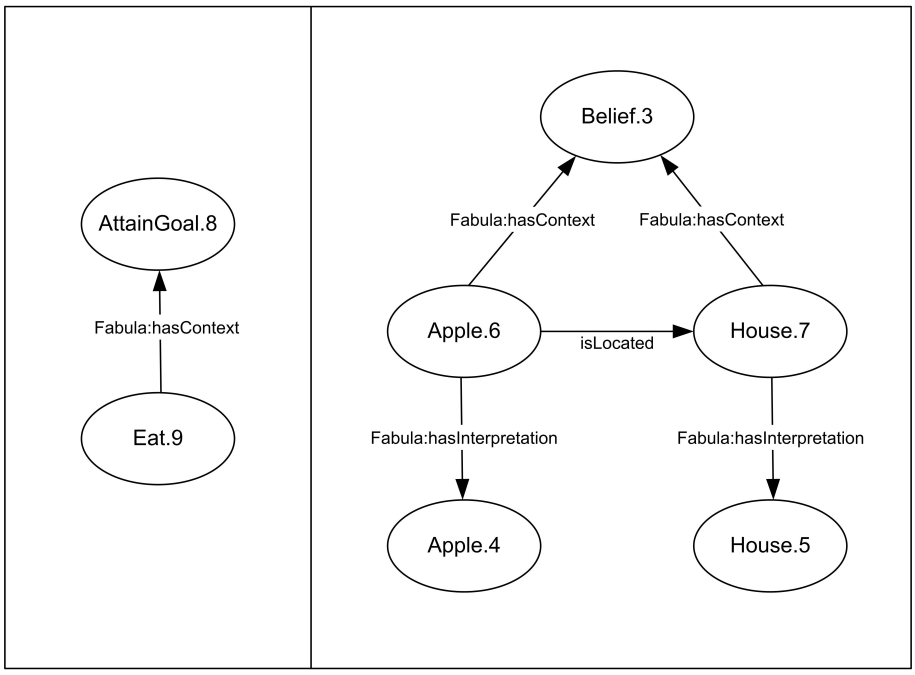

Figure 6.3: Examples of sub plot elements

This way we can query the fabula structure in order to get all arguments of a particular plot element and all of these arguments can be stored in a single Java Plot Element structure. However, if we translate such a Plot Element structure directly into a dependency tree (and therefore into a sentence in natural language), we will still get rather monotonous and stilted texts. All sentences will look somewhat like this: "De ridder wilde de prinses ontvoeren. Hij ging naar het kasteel. Hij bracht de prinses naar een brug. De brug stortte in." ("The knight wanted to kill the princess. He went to the castle. He brought the princess to a bridge. The bridge collapsed.") In this example all sentences start with the subject and only main clauses are used. If we focussed on generating instructional texts this might be satisfactory, but since our goal is to generate attractive fairy-tales we need to include more details. Therefore the Document Planner extends the Plot Elements with a Details Vector which contains (concept, detail)-pairs that specify certain details about a concept. Some of these details can be taken from the Character Information module, but currently most of the details are added manually.

The concepts in the (concept, detail)-pairs can be entities in which case the detail element is either an adjective (which tells more about the entity) or a noun in order to create appositions (such as "De prinses, een lief meisje, ging naar het bos." ("The princess, a sweet girl, went to the forest.")). If the detail element is an adjective, this adjective can also appear in the list together with an adverb which tells something about the adjective. An example of this is the word 'erg' ('very') in the noun phrase 'de erg oude vrouw' ('the very old lady'). Finally the concepts in the list can be verbs in which case the detail element is an adverb ("De prinses rende *snel*." ("The prince ran *quickly*.")), a time expression ("*Op een dag* zal de prins wraak

nemen." ("*One day* the prince will take revenge.")), or a place expression ("*Aan de overkant* zag de prinses haar vader staan." ("*At the other side* the princess saw her father.")).

Furthermore some of the plot elements in the fabula structure may have an additional property. Perceptions have the boolean property $isNonperception$ and beliefs have the boolean property $isNonbelief$. If the value of one of these properties is *true* the modifier 'niet' ('not') is added to the details Vector in order to make sure that the Microplanner will add the modifier to the dependency tree. Finally actions have the property $isSuccessful$ which indicates whether the action has taken place successfully. If the property is set to $false$ a mark is added to the Plot Element that the action failed, so the Microplanner will be able to select the correct template (see secation 7.2.1).

6.2.2 Converting the fabula into an initial document plan

In the previous subsection I explained how the arguments of a plot element can be retrieved from the fabula structure. Furthermore the OWL file contains the relations which hold between the different plot elements, so the complete structure can also be retrieved from the fabula. The fabula structure is represented as one large causal network, so all relations that appear in the fabula are causal relations. As explained in the previous chapter the fabula structure may contain four different types of causal relations; enable, motivate, physically cause and psychologically cause. The Surface Realizer, however, can only distinguish between a volitional and a non-volitional cause. A volitional cause relation is a cause relation in which one of the plot elements presents a purposeful or intended action caused by the other plot element, and in a non-volitional cause relation one plot element simply causes the other plot element without any purpose or intention. The causal relations used in the fabula structure can be mapped onto the causal relations supported by the Surface Realizer in the following way. The enable and physically cause relations are both mapped onto a non-volitional cause relation (since there is no purpose or intention present), and the motivate and psychologically cause relations are both mapped onto a volitional cause relation (since they all deal with goals or internal states).

Idea behind the algorithm

Converting a fabula structure into a document plan means converting a network containing nodes with possibly several incoming and outgoing edges, into a document plan in which all internal nodes have exactly one or two children. Moreover in a fabula structure all nodes represent plot elements and the edges represent the relations that hold between the plot elements. In a document plan, however, the relations are also represented by nodes (the internal nodes to be more precisely) and the plot elements only appear at the leaf nodes.

Before describing the entire algorithm for converting a fabula structure into a document plan, let's first look at a small example to illustrate the idea behind the algorithm. Suppose the fabula structure only consists of one plot element causing two other plot elements (see figure 6.4). In this example the first plot element may be the goal of eating an apple, and the other two plot elements may then be the actions of taking the apple and eating the apple. In this example the first plot element causes the other two plot elements both, so these latter two elements should be connected by an additive relation (or a temporal relation if the plot elements do not take place at the same time). The result of this should then be connected to the first plot element by means of a cause relation, as shown at the right side of figure 6.4.

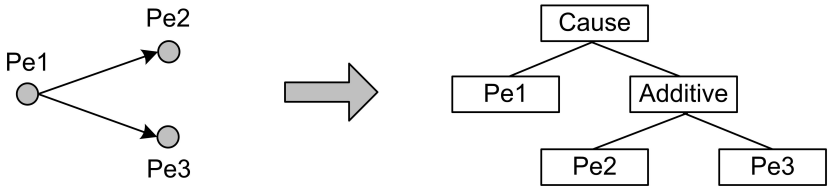

Figure 6.4: Example of a fabula containing a plot element that causes two other plot elements

In the exact same way a node with several incoming edges represents a plot element that is caused by several other plot elements *together*. All of these elements should be connected by additive relations, and the result of this should be connected to the final plot element by means of a cause relation. An example of a situation in which two plot elements cause another plot element together is given in figure 6.5. In this example the first two plot elements may be the internal state of being hungry and the perception of seeing an apple, and the final plot element may then be the goal of eating the apple.

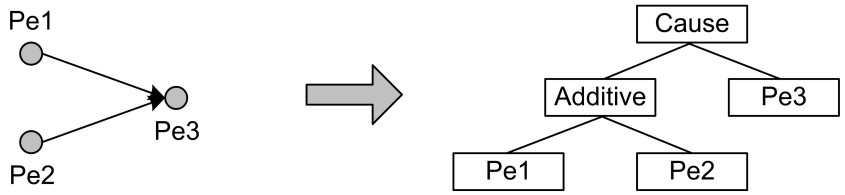

Figure 6.5: Example of a fabula containing two plot elements that together cause another plot element

Algorithm

The examples given in the previous subsection look rather simple, but since a certain node can have any number of incoming and outgoing edges, there are many different possibilities which also depend on the tree generated sofar. Furthermore when a node has several incoming edges one of these can also point to a *branch* of nodes instead of pointing to a single node, in which case the entire branch should be added to the document plan. This final problem requires the additional possibility of building the document plan backwards.

In order to make sure that the story begins with an introduction of the characters or settings the document plan is initialized with a settings node containing the main character. This node is then used to create the first sentence of a fairy-tale "Er was eens een ..." ("Once upon a time there was a ..."). Furthermore one node in the fabula structure will be marked as the starting node of the story. This can be achieved by retrieving all plot elements from the fabula structure and selecting the one which happened first, based on the time arguments of the plot elements. For this starting node the algorithm determines the number of incoming and outgoing edges and depending on these numbers it decides how to add the node to the tree generated sofar. In the following explanation of the algorithm we assume that *all* plot elements are to be included in the document plan.

The algorithm for converting a fabula structure into a document plan is shown in algorithm 1 and it works as follows. The algorithm is called recursively for each plot element that is to be added to the document plan. The function has therefore three arguments: *curr* (the id of the plot element currently to be added to the document plan), *tree* (the document plan generated sofar) and $forward$ (a boolean indicating whether the tree is created forwards or backwards). Furthermore the variable *result* represents the document plan after inserting the current plot element; its value is initialized to *tree*, then it is updated according to the new incoming and outgoing edges and at the end of the algorithm its value is returned. Finally the algorithm uses the function CreateRhetRel(*rel*, *tree*1, *tree*2) which creates a rhetorical relation node (representing the relation *rel*) with the children *tree*1 and *tree*2. The arguments *tree*1 and *tree*2 can thus be either another rhetorical relation node or a plot element node.

The first step of the algorithm is deciding the number of incoming edges and outgoing edges of the current node. A Vector is kept with the ids of all plot elements that have already been added to the document plan, so using this Vector and the incoming and outgoing edges the Vectors *prev* and *next* can be determined, representing the *new* previous nodes (the nodes connected to the current node by an incoming edge) and the *new* next nodes (the nodes connected to the current node by an outgoing edge). The functions DetermineNewPrevNodes() and DetermineNewNextNodes() are responsible for this, but are not shown in detail in the outline of the algorithm.

After initialization of *result*, *prev* and *next* the actual Document Plan will be updated. This process is subdivided into two cases depending on the boolean $forward$. Figure 6.6 shows all possible situations if the

boolean is set to *true*. Note that we assume that the plot elements that appear in *tree* all occur before the other previous nodes of *curr*, which means that *tree* always appears left from the other previous nodes of *curr*. This is not necessary, so the actual algorithm will order the nodes based on the time arguments. For clarity I have not included the situations in which *tree* should be added after a *prev* node, because these cases are exactly the same, except that two nodes are interchanged.

Figure 6.6 shows the following situations:

1. The first situation is the one in which the current node does not have any new previous or next nodes, which is the easiest situation possible. In this case *curr* is simply added to the document plan generated sofar by means of a cause relation, and the algorithm finishes without calling the same function recursively.

2. If the current node does have a new next node, *curr* is added to *tree* in the exact same way, but the function $CreateDocumentPlan()$ is called recursively for the new next node. The argument *tree* for the recursive function call is set to the combination of *tree* and *curr*, and the argument $forward$ is simply set to *true*. The result of this recursive function call is then the document plan for the new next node *including* the tree generated sofar.

3. If the current node has more than one new next node, the function $CreateDocumentPlan()$ has to be called for each next node. In this case the argument *tree* is set to *null*, so the result of such a function call is a tree representing one of the new next nodes. The resulting trees are then connected by additive relations (or temporal relations if the next nodes do not happen at the same time), and finally added to the result.

4. If the current node has a new previous node, this node is first turned into a tree by calling the function $CreateDocumentPlan()$ with the argument *tree* set to *null* and the boolean $forward$ set to $false$ (see the explanation below). The resulting tree is then added to *tree* by an additive relation in order to show that the trees cause the current node together. Finally, *curr* is added to the result by a cause relation.

5. Having looked at a number of situations, the remaining situations are rather straightforward, so these are described in less detail. The situation in which *curr* has one new previous and one new next node connects *tree* to the tree representing the new previous node (by calling the function recursively) by an additive relation, then adds *curr* by means of a cause relation and finally calls $CreateDocumentPlan()$ recursively for the new next node with the argument set to the tree generated sofar.

6. If the current node has one new previous and more than one new next node, the algorithm is the same, except that the function is called for both new next nodes (both times with the argument *tree* set to *null*.

7. If *curr* has more than one new previous node, the trees representing these nodes are all connected using additive relations and finally added to *tree*, in order to show that all of these nodes together cause the current node. Finally *curr* is added to the result with a cause relation.

8. This situation is the same as the previous one, except that the function is also called for the new next node with the argument *tree* set to the combination of *tree* and all new previous nodes.

9. This situation is also the same, apart from the fact that the function is called once more for *all* new next nodes, with the argument *tree* set to *null*.

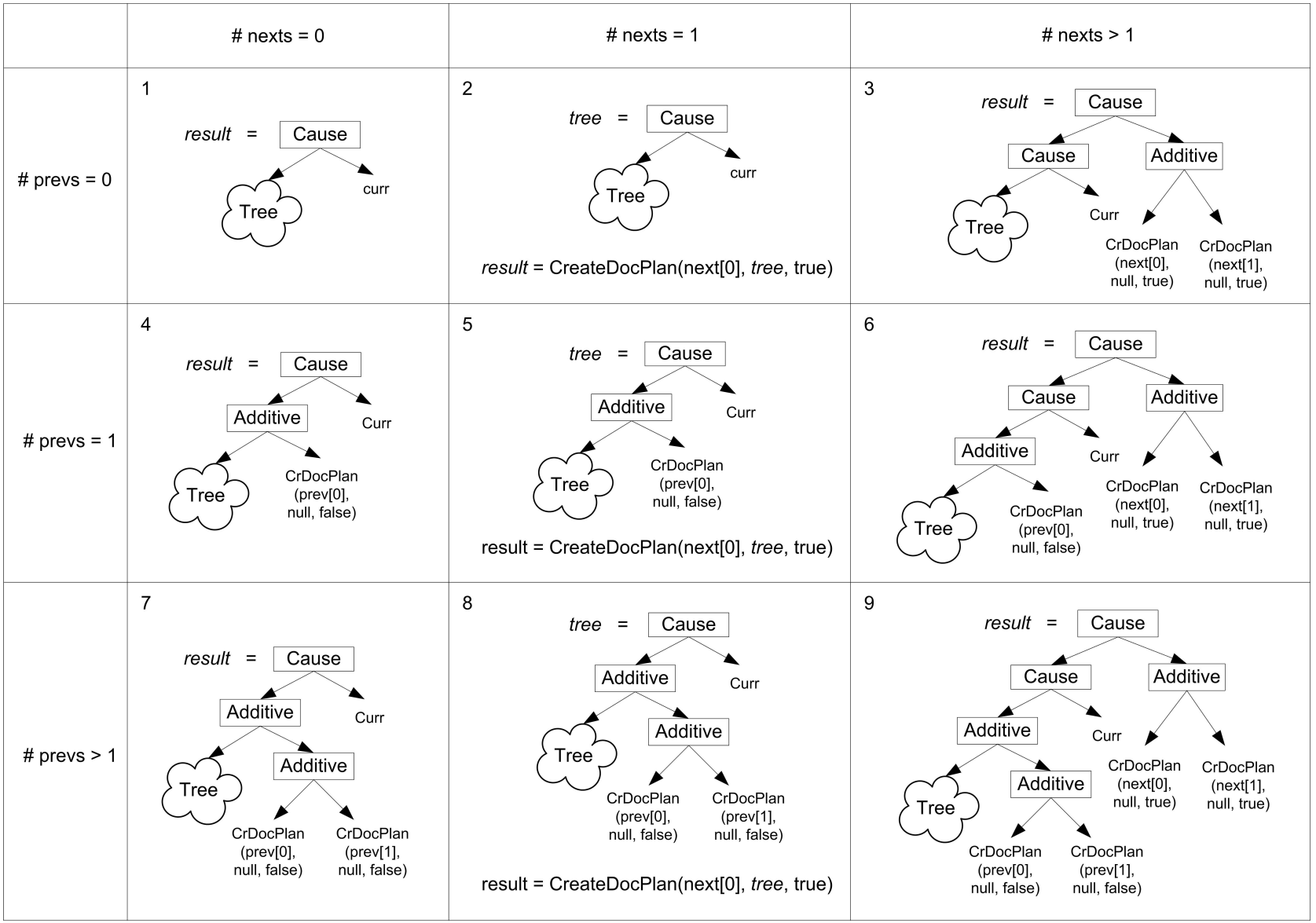

Figure 6.6: Situations in CreateDocumentPlan(curr, tree, forward)

All of these cases show that first the new previous nodes are turned into trees by calling the function $CreateDocumentPlan()$ for each new previous node with the boolean $forward$ set to $false$. Then the resulting trees are connected by means of additive relations to show that they cause the current node together. The resulting tree is then added to *tree* by an additive relation as well. Next the current node is added to the result by a cause relation since the result (representing all incoming edges) *causes* the current node. The remainder of the algorithm depends on the number of new next nodes. If there are no new next nodes the algorithm simply finishes without calling the function recursively. If there is exactly one new next node the function is called recursively with the argument *tree* set to the document plan generated sofar (the combination of *tree* and the new previous nodes); the result of this function call is the entire document plan and the algorithm finishes. Finally, if there are several new next nodes each of these nodes is turned into a tree by calling the function recursively with the argument *tree* set to *null*. The resulting trees are then connected by additive or temporal relations and finally these are added to the result by a cause relation.

Note that in all cases the branches are added to the document plan chronologically which can be achieved using the time arguments of the different plot elements. Furthermore the outline of the algorithm and the situations in figure 6.6 only show the case in which there are exactly two new previous or next nodes, but the actual algorithm also supports more nodes.

As mentioned before the function $CreateDocumentPlan()$ can also be called with the boolean $forward$ set to $false$, in which case the tree has to be created backwards. Currently, this algorithm is somewhat simplified, but it does always generate a correct document plan. The algorithm simply builds two separate trees; one for the (new) previous nodes and one for the (new) next nodes. In the same way as done when creating the tree forwards, the trees are simply created by calling the function $CreateDocumentPlan()$ recursively with the boolean $forward$ set to *true* for new next nodes and set to $false$ for new previous nodes. The resulting trees are then connected by additive or possibly temporal relations. At the end of the algorithm the tree representing all incoming edges (*prevstree*) is connected to the current node and then this result is connected with the tree representing all outgoing edges (*nextstree*) and the final result is returned.

Example of execution of the algorithm

In the description of the algorithm sofar we only looked at independent steps of the algorithm, so in this subsection I will give a short example of the entire conversion of a fabula structure into a document plan. For this example I will use the same examples as the ones used in the description of the idea behind the algorithm.

The first example is the easiest one and is shown in figure 6.7 (the same as the example in figure 6.4). In this example node $pe1$ is the starting node of the fabula structure, so the algorithm begins with calling $CreateDocumentPlan(pe1,\ null,\ true)$. In this function call *curr* is set to $pe1$ and this node has no new previous nodes but it does have two new next nodes, such as situation 3 from figure 6.6. Therefore a tree is built with a cause node with $pe1$ as first node and an additive relation as second node. The children of this additive node are then created by calling the function $CreateDocumentPlan()$ recursively for $pe2$ and $pe3$ with *tree* set to *null* and $forward$ set to *true*. The results of these function calls are single nodes representing the plot elements $pe2$ and $pe3$ (because both nodes do not have any new previous or next nodes; situation 1 in figure 6.6). These nodes are then added to the tree generated sofar and the result looks like the document plan at the bottom of figure 6.7 which is exactly the same as the one in figure 6.4.

The second example uses the same fabula structure as the one in figure 6.5 and is shown in figure 6.8. Node $pe1$ is the starting node and this node only has one new next node (situation 2 from figure 6.6). Therefore the function is called recursively for $pe3$ with the argument *tree* set to a node representing the plot element $pe1$ and the argument $forward$ set to *true*. The node $pe3$ has one new previous node (situation 4 from figure 6.6), so first of all the function is called recursively for $pe2$ with *tree* set to *null* and $forward$ set to $false$. This node does not have any new previous or next nodes, so the result of this function call is simply a node representing $pe2$. This node is then connected to $pe1$ by an additive relation and finally $pe3$ can be added to the result by means of a cause relation. The result looks like the final document plan in figure 6.8, which is again exactly the same as the document plan in figure 6.5.

Algorithm 1 CreateDocumentPlan(*curr*, *tree*, *forward*)

```
result ← tree

prev ← DetermineNewPrevNodes()
next ← DetermineNewNextNodes()

if forward then
  if #prev = 1 then
    result ← CreateRhetRel("additive", result, CreateDocumentPlan(prev[0], null, false)
  else if #prev > 1 then
    result ← CreateRhetRel("additive", result, CreateRhetRel("additive", CreateDocumentPlan
             (prev[0], null, false), CreateDocumentPlan(prev[1], null, false)))
  end if
  if #next = 0 then
    result ← CreateRhetRel("cause", result, curr)
  else if #next = 1 then
    result ← CreateRhetRel("cause", result, curr)
    result ← CreateDocumentPlan(next[0], result, true)
  else if #next > 1 then
    tmpresult ← CreateRhetRel("temporal", CreateDocumentPlan(next[0], null, true), Create-
                DocumentPlan(next[1], null, true))
    result ← CreateRhetRel("cause", result, curr)
    result ← CreateRhetRel("cause", result, tmpresult)
  end if
else
  prevstree ← null
  nextstree ← null
  if #prev = 1 then
    prevstree ← CreateDocumentPlan(prev[0], null, false)
  else if #prev > 1 then
    prevstree ← CreateRhetRel("additive", CreateDocumentPlan(prev[0], null, false), Create-
                DocumentPlan(prev[1], null, false))
  end if
  if #next = 1 then
    nextstree ← CreateDocumentPlan(next[0], null, true)
  else if #next > 1 then
    nextstree ← CreateRhetRel("temporal", CreateDocumentPlan(next[0], null, true), Create-
                DocumentPlan(next[1], null, true))
  end if
  if prevstree ≠ null then
    result ← CreateRhetRel("cause", prevstree, curr)
  end if
  if nextstree ≠ null then
    if result ≠ null then
      result ← CreateRhetRel("cause", result, nexts)
    else
      result ← CreateRhetRel("cause", curr, nexts)
    end if
  end if
  if result = null then
    result ← curr
  end if
end if
return result
```

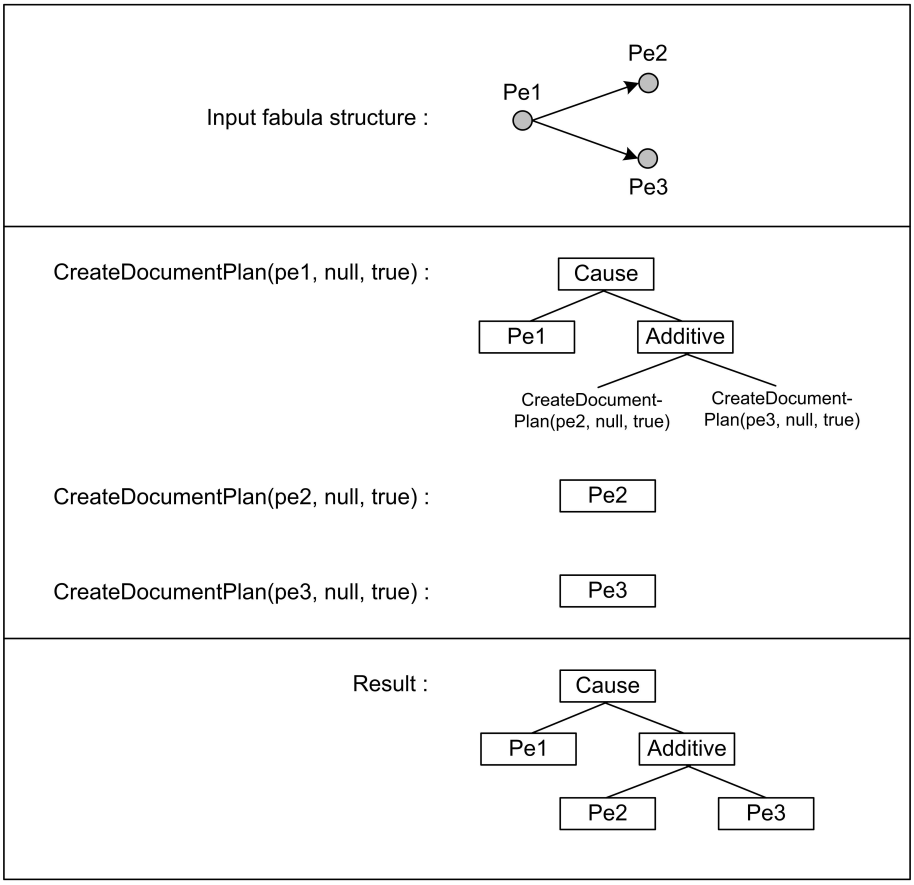

Figure 6.7: Example of converting the fabula from figure 6.4

6.2.3 Performing the NLG tasks

As mentioned before the Initial Document Plan Builder has to carry out four different tasks which are all performed simultaneously when building the initial document plan. The fabula structure is one coherent graph which means that all plot elements in the fabula are connected and are all relevant for the story. This also means that all plot elements should be included in the final story, so the task of *selecting the relevant information* is performed by simply adding *all* nodes, with a few exceptions though. First of all successful actions always lead to perceptions that the action succeeded which then lead to beliefs that the action succeeded. Obviously this should not be narrated each time an action is performed, so if the fabula contains such a structure only the action is included in the document plan. Furthermore it may be redundant to include the supergoal as well as the subgoal if the subgoal is a generalization of the supergoal. An example is the supergoal of 'eating something' and the subgoal of 'eating an apple'. In this case it is better to narrate only the subgoal in order to avoid texts such as "Plop wilde iets eten, dus hij wilde een appel eten." ("Plop wanted to eat something, so he wanted to eat an apple.").

The module's second task is *ordering the information*, which has already been explained in the previous section with the explanation of the algorithm for converting the fabula into a document plan. Branches of nodes without any subbranches follow each other directly. Furthermore if a node has more than one incoming edge the incoming branches are told chronologically, and if the node has more than one outgoing edge the outgoing branches are also told chronologically. This means that the story is mainly told chronologically, but if there are two characters which perform some different actions approximately at the same time, the actions which are carried out by the same character are told together and the actions carried out by the other

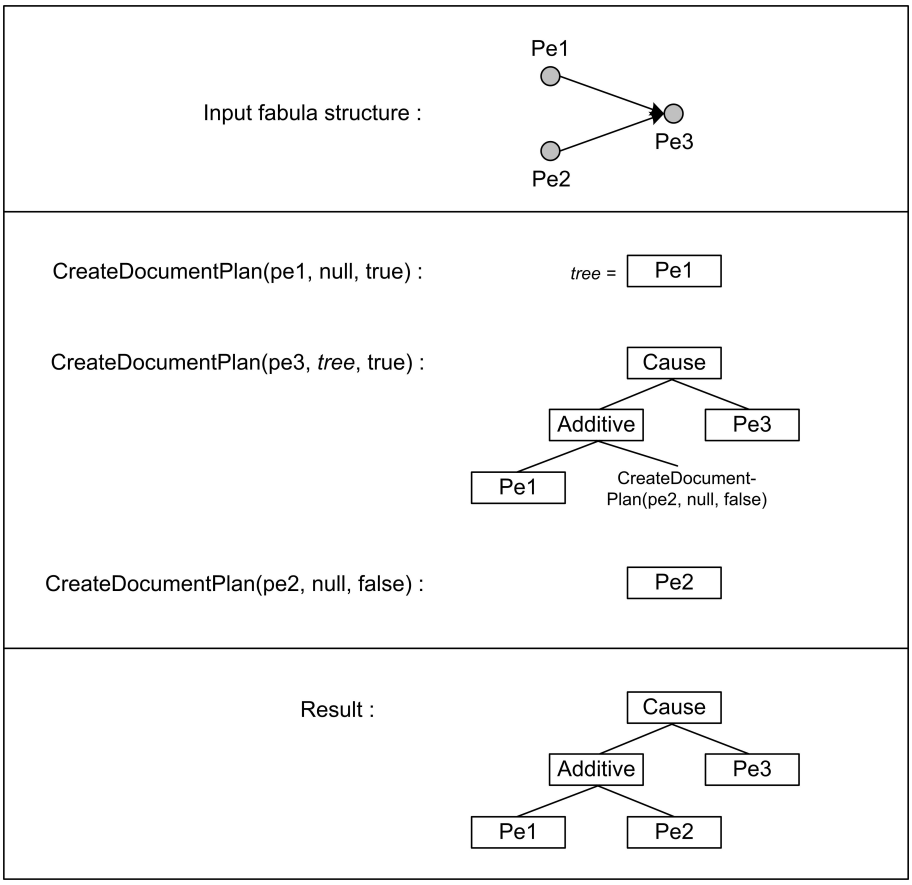

Figure 6.8: Example of converting the fabula from figure 6.5

character are also told together. This is done in order to avoid switching back and forth between different characters (or even different locations) over and over again.

The third task to be carried out by the Initial Document Plan Builder is *assigning the correct rhetorical relations*. The causal relations are specified in the fabula structure, but if all of these relations are expressed explicitly this leads to rather boring texts containing the same cue words in every single sentence. Furthermore the branches of two incoming edges are always connected by an 'additive' relation in order to make clear that they cause the next plot element *together*. However, in some cases different relations should be used. Take for example the plot elements "De ridder was verliefd op de prinses." ("The knight was in love with the princess.") and "De prinses was verliefd op een prins." ("The princess was in love with a prince."), which together lead to the knight's goal of killing the princess. In this case it is better to connect the plot elements with a contrast relation than an additive relation. In order to achieve this the plot elements should be compared, and based on the differences between the plot elements a contrast relation should be selected. Currently this has not been included yet, but it may be a useful extension. Furthermore the branches of two outgoing edges are normally combined with an 'additive' relation, but if the plot elements do not happen at the same time a 'temporal' relation is chosen.

Finally the Initial Document Plan Builder has to *add paragraph boundaries*. Currently this has not been added yet, because the fabula structures are still rather small. In section 4.3.4 we saw that paragraph breaks usually appear at switches of character, time or location, so once the fabula structures are more complicated paragraph breaks may be added based on these properties. Furthermore paragraph breaks may be added based on the number of incoming and outgoing edges, or in cases in which a node has two outgoing edges which both point to rather large branches. In such a case one of the branches will be told first and then the

algorithm has to go back which may lead to some large time differences. In chapter 9 some larger examples will be given in which the input is partly modelled, so in these examples the paragraph breaks have been added manually. In the remainder of this document we can therefore assume that the fabula has already been subdivided into paragraphs.

6.3 Background Information Supplier

After an initial document plan has been generated, three different kinds of transformations can be performed on this document plan. The first transformation is carried out by the Background Information Supplier, whose main task is to add background information. Furthermore the module keeps a discourse history, a list containing all plot elements mentioned sofar, which is used for the following three reasons: to prevent the system from telling facts more than once, to add some more discourse markers and to include information that has already been told to refresh the reader's memory.

6.3.1 Adding background information

The module takes as input an initial document plan and then traverses the tree from left to right. For each plot element the module checks if it contains a new entity. If this is the case the Character Information and the Story World modules are queried to find out if some additional information is available.

Examples of information taken from the Character Information are the names and properties of characters. For each piece of background information a new node is added to the document plan, connected to the original plot element node by an elaboration relation. This way the plot element introduces the new entity and then the additional information is narrated. An example is a plot element describing the fact that a knight hits a princess and the only background information available about the princess is that she is beautiful. In this case the fact that the princess is beautiful is told *after* the original plot element, such as "De ridder sloeg de prinses, die mooi was." ("The knight hit the princess, who was beautiful."). Note that this component adds a complete plot element to the document plan, usually representing only a property of a certain entity. The State Transformer (described in the next section) will then decide if the new plot element can be combined the other plot element, resulting in the sentence "De ridder sloeg de mooie prinses." ("The knight hit the beautiful princess.").

Background information can also be taken from the Story World module. Examples of this kind of information are the locations of objects. If such information is available also a new node is added to the document plan, but in this case the node is added right *before* the current plot element node. An example in which this is the correct order is the scenario in which a knight goes to the forest motivated by the goal of having a sword. At first sight this doesn't make any sense, but if the information that there is a sword in the forest is added to the text it becomes clear immediately.

6.3.2 Discourse history

Furthermore the Background Information Supplier keeps a discourse history (a list of all previous plot elements) which can be used to detect if facts have already been told.

Adding discourse markers based on repetition

In some cases it might be useful to tell facts several times, while in other cases it may lead to monotony. Therefore the module first checks if the current plot element has been told recently; it checks if the plot element is the same as one of the three last plot elements in the discourse history. Two plot elements are the same if the type of plot element is the same (such as an action or internal state), and if the agens, patiens, target and instruments are the same. This means that two actions with the same arguments, but which do not happen at the same time, are considered the same, since they may be lexicalized in exactly the same way.

If the current plot element has been recently told, the module updates the document plan as follows, depending on the type of plot element:

- The plot element is a goal or setting.

  In this case the module will delete the plot element from the document plan. An example is a series of actions all motivated by the same goal. It would be useless to mention the goal each time an action is described, so we will simply tell the goal only once.

- The plot element is an action or an event.

  If the plot element is an action or an event, it may be important that the action is carried out several times. Therefore the module will include the plot element, but will also add the cue word 'weer' or 'nog een keer' ('again') to the plot element. An example is a knight hitting a princess several times: "De ridder sloeg de prinses, waardoor zij bang was. Daarna sloeg hij haar *nog een keer*." ("The knight hit the princess, so she was scared. Then he hit her *again*.").

- The plot element is an internal state.

  In this case it may also be important to mention the internal state again, for example to show that the character experiences the state for some period of time. Therefore the plot element is included in the document plan and the cue word 'nog' or 'nog steeds' ('still') is added to the plot element.

  A useful extension would be to describe the state somewhat differently, based on the intensity the emotion is experienced and on the fact whether the intensity has just increased or decreased. An example is someone being angry, followed by some event that leads to becoming even angrier. This could be lexicalized as "Hij was nog steeds boos." ("He was still angry."), but it would be better to use "Hij werd nog bozer." ("He became even angrier."). As explained in section 5.3.2 the input will be marked with the emotions and their intensities in the future, so once these intensities have been added this new way of describing internal states can be included.

Adding the discourse marker 'also'

The discourse history can also be used to decide whether the discourse marker 'ook' ('also') has to be included. If two consecutive sentences are very similar, the use of the word 'also' is obligatory. Therefore the discourse marker will be added when the plot elements differ in only one argument. This may be any of the arguments, under the condition that the plot elements have the same type:

- Verb: The knight hit the princess. → The knight *also* kicked the princess.
- Noun phrase: The knight hit the princess. → The king *also* hit the princess.
- Adjective: The princess was scared. → The princess was *also* sad.

The condition that the plot elements are of the same type is necessary, because otherwise the discourse marker would be added to the following text: "De prinses was bang. Ze schreewde ook." ("The princess was scared. She also screamed."), which is obviously not desirable.

Note that there is an additional requirement for adding the cue word 'also' as the following example shows: "Plop pakte een appel op. Hij at de appel." ("Plop took the apple. He ate the apple."). In this example the first action *enables* the second action and the two sentences are therefore connected by a causal as well as a temporal relation. In this case it is inappropriate to add the cue word 'also'. On the other hand, the two sentences "De prinses ging naar het bos. De ridder ging *ook* naar het bos." ("The princess went to the forest. The knight *also* went to the forest.") are connected by a motivate relation, so they are also connected by a causal relation as well as a temporal relation. However, in this case the cue word 'also' does lead to a more coherent text and should therefore be included. This may be due to the fact that the first two actions were related by an enable (causal) relation and the other two actions by a motivate (causal) relation. The two examples may also be different because in the first example the type of *actions* are different and in the other example the *subjects* are different. For the moment I will solve this problem by always adding the

cue word 'also' if the subjects differ, and only adding the cue word if the time arguments are the same in case the only difference is the verb (describing two different actions carried out simultaneously).

The second situation in which the word 'also' should be added applies when the arguments are exactly the same, but when two of them are swapped. These arguments can be any two of the arguments as the following examples show:

- John introduced Pete to Bill → John *also* introduced Bill to Pete
- John introduced Pete to Bill → Pete *also* introduced John to Bill
- John introduced Pete to Bill → Bill *also* introduced Pete to John

These three examples all require the word 'also' while the text "John introduced Pete to Bill. Pete introduced Bill to John." should not include the word 'also'.

In order to decide whether one of these cases applies, the current plot element is compared to the last plot element in the discourse history. The algorithm then checks if the two plot elements are of the same type. If this is the case, it counts the number of differences. If this difference is exactly one, the plot elements are almost the same and the word 'also' has to be included (corresponding to the first set of examples). If the number of differences is two *and* the difference is a swap of arguments the word 'also' is needed too (corresponding to the second set of examples). Note that the algorithm also checks if the plot elements contain the modifier 'not', because the cue word 'also' should not be added if one of the plot elements contains the modifier and the other one does not.

In the exact same way the plot element is compared to the one-but-last plot element in the discourse history. This is necessary as the following example shows: "De prinses ging naar het bos. Omdat de ridder de prinses gevangen wilde nemen, ging hij *ook* naar het bos." ("The princess went to the forest. Because the knight wanted to capture the princess, he went to the forest *too*"). In this example there is an intervening clause, so the two consecutive plot elements do not satisfy the requirements for adding 'also'. However, if the word 'also' is added to the final clause, the text becomes far more coherent, so the plot element is compared to the last *two* plot elements in the discourse history.

Repeating background information

Finally the discourse history can be used to repeat facts in order to refresh the reader's memory. Obviously this is only useful when an entity has not been mentioned for a long time. Therefore the module checks if the entity has not been mentioned for at least ten clauses and if this is the case, it retrieves the first fact from the discourse history containing that entity. An additional requirement is that this fact is a state, because otherwise actions (or other types of plot elements) may be repeated when this is inappropriate. One final requirement of repeating the background information is that the state still applies.

An example of a situation in which repeating background information is useful, is given in chapter 9. The example shows a rather large story about a princess and in the first paragraph is mentioned that she is in love with a prince. Then for two paragraphs the prince is not mentioned at all, but in the final paragraph he turns out to be the hero. In that case the fact that she was in love with him is repeated which results in a somewhat more coherent and understandable story.

6.4 State Transformer

The second type of transformation that can be performed on an initial document plan is carried out by the State Transformer, which decides how an internal state can be described best. Usually a state such as 'scared(princess)' is described as "De prinses was bang." ("The princess was scared."), but in some cases the state can be combined with another plot element. For example, if the input consists of the state "De prinses was bang." ("The princess was scared.") and the action "De prinses ging naar het bos." ("The princess went to the forest."), this input can be described in the following ways:

1. Normally, using two separate sentences:
   "De prinses was bang. Zij ging naar het bos."
   ("The princess was scared. She went to the forest.")

2. As an adjective, combined with the other sentence:
   "De bange prinses ging naar het bos."
   ("The scared princess went to the forest.")

3. As a relative clause, connected to the other sentence:
   "De prinses, die bang was, ging naar het bos."
   ("The princess, who was scared, went to the forest.")

4. As an adverbial adjunct within the other sentence:
   "Met een bonzend hart ging de prinses naar het bos."
   ("The princess went to the forest with a pounding heart.")

5. As an action, using two separate sentences:
   "De prinses ging naar het bos. Haar hart bonkte in haar keel."
   ("The princess went to the forest. Her heart was pounding in her throat.")

Like the Background Information Supplier the State Transformer takes as input a document plan and it traverses the tree from left to right. Doing this it checks if the document plan contains a state and an action which are connected. If this is the case, it has to decide which of these ways is best in the current context. How this decision will be done, is not known yet, so for the moment the State Transformer chooses one of the possibilities at random.

If the State Transformer decides to tell the information following the first way, it leaves the document plan unchanged. If the second way is chosen, the state is transformed into a detail element and this element is added to the details Vector of the action node. The original rhetorical relation node is then replaced by the new action node. If the state will be described using the third way, the plot elements simply have to be related by the elaboration relation, so only the label in the relation node has to be updated.

In order to generate the other two ways an additional database is required, which maps states to adjuncts or to complete sentences. If an adjunct is selected, this is added to the details Vector of the action plot element (as canned text), and the original relation node is then replaced by this new action node. If the state is transformed into a new sentence, such as the fifth example, the action node is left unchanged, but the state node is replaced by a canned text node representing the new sentence. Note that such a sentence may contain a pronoun such as 'haar' ('her') in the example. Obviously it would be best if the referring expression generator would generate this pronoun, but since the rest of the sentence is stored as canned text this would make things more complicated than necessary. Therefore I solved this by simply storing a male and a female version of the sentence; one sentence containing the pronoun 'zijn' ('his') and one containing the pronoun 'haar' ('her').

6.5 Mood Creator

Fairy-tales and stories differ from other types of texts in the sense that they try to create a mood. This means that the text is not simply an enumeration of actions and events, but some information is only added in order to make the stories somewhat more attractive for the reader. Examples of such information are single words which make the sentences sound more fluently, but also complete sentences can be added. Therefore the final type of transformation that can be performed on an initial document plan is the addition of mood sentences.

In order to decide which sentences can be added, the Mood Creator keeps track of a mood model, modelling the current mood of the story. In this model the current location and the internal state of the agens of the current action are stored. In future versions this mood model may be extended, but for the moment only those two variables are used. Based on the combination of the values of these variables mood sentences can be added, such as the following examples:

- Location is forest and internal state is happy:
  De vogeltjes floten vrolijk op de takken van de bomen.
  (The birds whistled happily on the branches of the trees.)
- Location is forest and internal state is scared:
  De gure wind waaide door de takken van de bomen.
  (The rough wind was blowing through the branches of the trees.)
- Location is a path and the internal state is happy:
  Er groeiden talloze kleurrijke bloemen langs de weg.
  (Countless colorful flowers were growing along the path.)
- Location is a path and the internal state is scared:
  Op het angstaanjagend donkere pad drong de maan nauwelijks door de dichte bomen door.
  (On the frightening dark path the moonlight could barely penetrate the dense trees.)

Note that some of these example sentences require additional information. Take for example the sentence about the moon which cannot appear in a story that happens during the day. Currently the system does not support day or night, but once this has been added to the Plot agent, the Mood Creator has to make sure that the generated sentences are not in conflict with the story settings.

These new sentences are stored as canned text nodes in the document plan. This is possible because the sentences are only meant to create a mood and do not have to be combined with other sentences. Since the entire input is not used yet, it is almost impossible to know the location and the states of each character at each moment in the story. Therefore the Mood Creator has been implemented, but it has not been integrated into the final system yet. However, chapter 9 gives an example of a story in which the Mood Creator is switched on.

6.6 Branch Remover

The algorithm for converting a fabula structure into a document plan always results in a correct document plan in the sense that the Microplanner and Surface Realizer can translate the document plan into text in natural language. However, the resulting document plan may contain long branches in which only the nodes at the bottom of the tree can be combined in order to get more complex nodes. Since the Background Information Supplier can add entire plot elements to the document plan some nodes may be replaced by two nodes connected by an elaboration relation. In such a case these can be combined, but if the Background Information Supplier cannot add any nodes, the document plan may still contain long branches which will result in many simple sentences. However, it would be much better if the Surface Realizer would still be able to combine some other nodes as well. Take for example the fabula structure from figure 6.9 in which the entire fabula structure is represented as one large branch. The nodes in the fabula structure represent the following sentences:

- $IE(Hungry)$ - Plop was hungry.
- $G(Eat)$ - Plop wanted to eat something.
- $A(Goto)$ - Plop went to the forest.
- $P(See)$ - Plop saw an apple.
- $A(Eat)$ - Plop ate the apple.

Note that this example is created manually; it is only meant for illustrative purposes. The Initial Document Plan Builder will convert this fabula structure into the initial document plan shown in the middle of the figure which is also one large branch. If this document plan would be the final output of the Document Planner, only the nodes $IE(Hungry)$, $G(Eat)$ and $A(Goto)$ could be combined. A possible result is then the following text: “Plop was hungry, so he wanted to eat something and therefore he went to the forest.

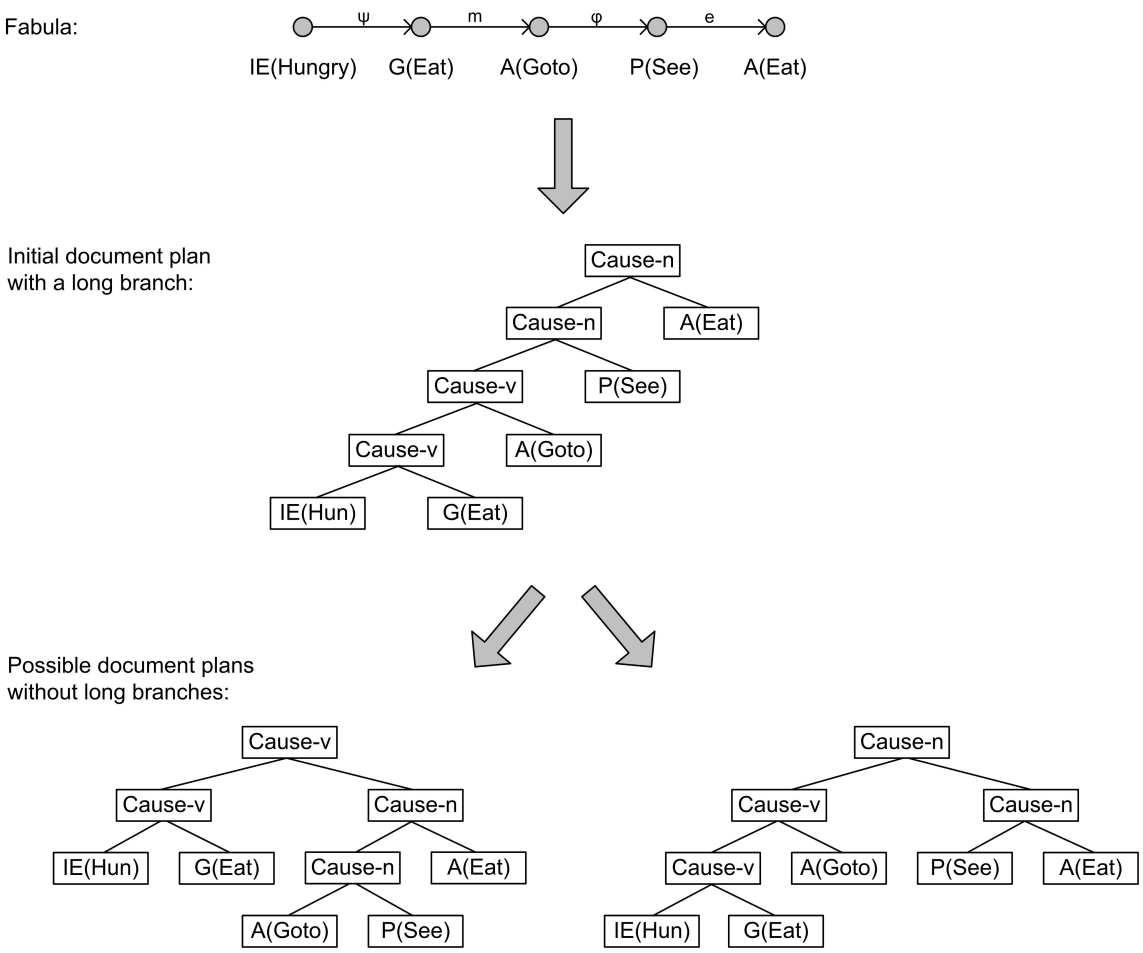

Figure 6.9: Example of removing a long branch from a document plan

He saw an apple. He ate the apple". If the branch is even longer, for example seven or eight nodes long, also three nodes could only be combined, so the result would be even worse. Furthermore *all* relations used in this example are causal relations, since they are taken from the fabula structure. This means that there is no reason to believe that a certain relation is stronger than another relation (i.e. that it would be better to combine nodes $IE(Hungry)$ and $G(Eat)$ than the nodes $P(See)$ and $A(Eat)$). Therefore I added an additional module to the Document Planner that recognizes long branches and splits them into smaller branches.

As described in section 2.1.3 it is best to generate sentences with varying length. Furthermore the Surface Realizer will combine *at most* three dependency trees into one. Therefore the Branch Remover splits the branches into branches of two or three nodes. In order to achieve this, the Branch Remover traverses the document plan depth first and keeps the length of the current branch. If it recognizes a branch with a length of four or higher it decides whether to split it into branches of two nodes or branches of three nodes. Take for example the branch of five nodes shown in figure 6.9. The algorithm decides to split this into a branch of two nodes and a branch of three nodes, both results shown at the bottom of the figure.

The algorithm for removing long branches by splitting them into separate shorter branches works as follows (see figure 6.10). If the algorithm recognizes a branch and decides to split it into a branch of three nodes and a branch of the remaining nodes, it takes the first three nodes of the branch and stores this in the variable $tree1$. Then the remaining part of the branch is stored in variable $tree2$, by moving the first element to its parent node. In the example this means moving the node with the edge one level up. Next

these trees are both added to a new rhetorical relation node *result*. Then the original nodes from the branch are removed from the tree while the parent node is stored in the variable *store*. Finally the rhetorical relation *result* is added to *store*. The result will then look like the document plan at the bottom of figure 6.10.

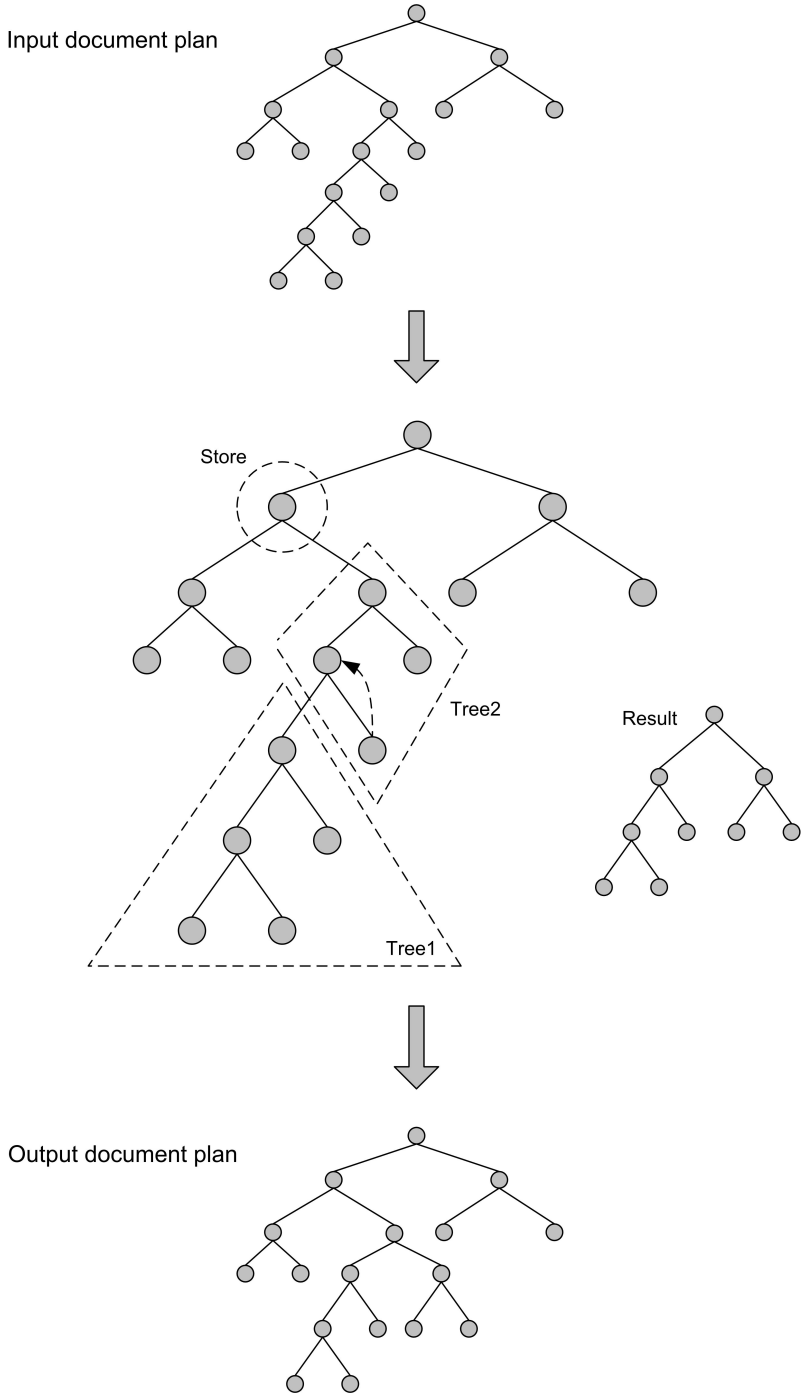

Figure 6.10: Algorithm for removing a long branch from a document plan

6.7 Example

In order to show how the different modules of the Document Planner work I will give a short example. Figure 6.11 shows an example of a fabula structure for a story about a dwarf who is hungry and wants to eat an apple. The nodes in the fabula structure represent the following sentences:

Hungry	The dwarf was hungry.
Belief	The dwarf believed there was an apple in the house.
Goal	The dwarf wanted to eat an apple.
Take apple	The dwarf took the apple.
Eat apple	The dwarf ate the apple.

Appendix B.1 shows the OWL representation of this fabula structure. The story is extremely simple, but it contains enough information to illustrate the different components of the Narrator agent. The example is also used throughout the following chapters to illustrate those components as well, and the final output of the example is given in figure 8.7.

Since this example is very simple, chapter 9 gives a number of somewhat more complicated examples, which show the functioning of the other components as well.

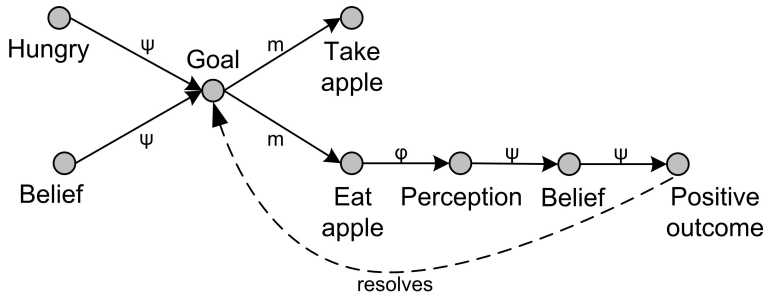

Figure 6.11: Example of a fabula structure for the Plop story

6.7.1 Initial Document Plan Builder

The Initial Document Plan Builder has to convert the fabula structure into an initial document plan. As explained in section 6.2 the module first creates a settings node in order to introduce the dwarf and then builds the initial document plan using the fabula structure.

In this process the nodes that represent the fact that the dwarf is hungry and the fact that he believes there is an apple in the house together cause the goal of eating the apple. Therefore these nodes are first connected by an 'additive' relation and then the goal node is connected to this rhetorical relation node by means of a 'cause' relation. This goal node has two new outgoing edges (one to the action of taking the apple and one to the action of eating the apple), so new trees are created for both branches separately. The first branch only consists of the node describing the action of taking the apple. The second branch consists of several nodes, but all of these nodes represent the sequence of an action which leads to the perception that the action has taken place which then leads to the belief that the action has taken place successfully. Therefore the entire branch is also represented as a single action node. These two action nodes are then combined using a 'temp-after-sequence' relation, because the action of taking the apple has to take place before the action of eating the apple. Finally the resulting node is added to the document plan generated sofar by a 'cause' relation. At this moment the entire fabula structure has been processed and the final document plan is shown in figure 6.12.

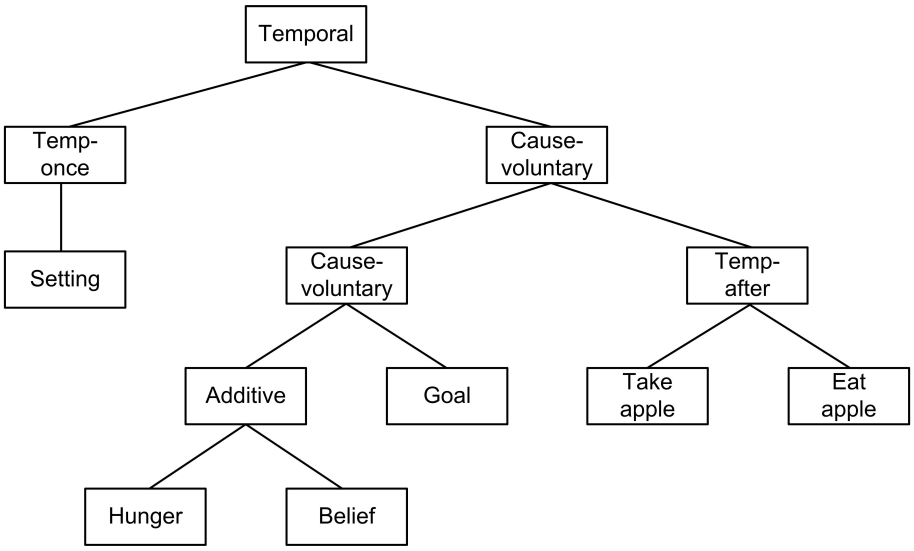

Figure 6.12: Output of the Initial Document Plan Builder for the Plop story

6.7.2 Background Information Supplier

The second module of the Document Planner is the Background Information Supplier which has access to the Story World and Character Information modules. If the Character Information module contains the fact that the dwarf is called Plop the setting node in the document plan is replaced by a rhetorical relation node with the setting node and a new node for the name as its children. Furthermore no background information is available and when comparing consecutive nodes to each other it appears that no discourse markers have to be added, so the result of the Background Information Supplier is the document plan shown in figure 6.13.

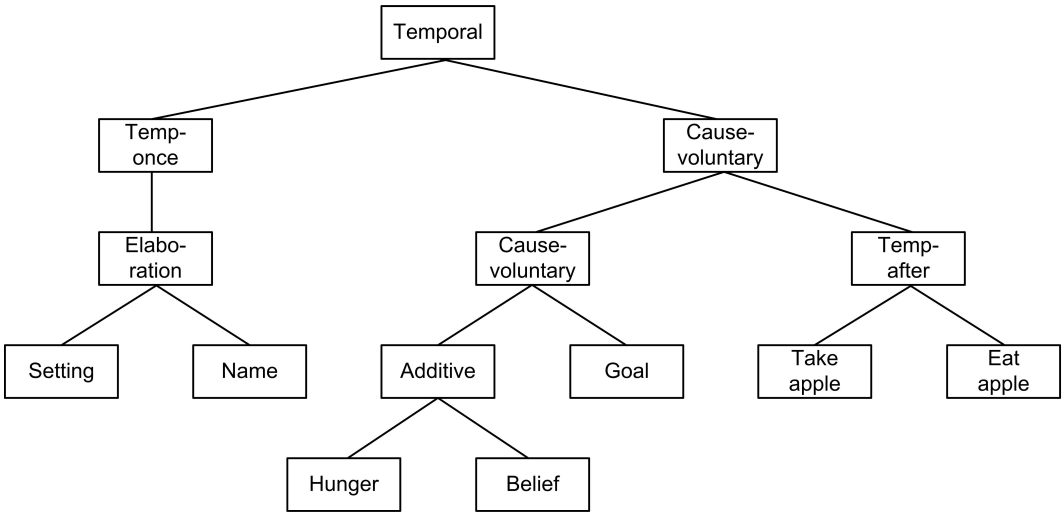

Figure 6.13: Output of the Background Information Supplier for the Plop story

6.7.3 State Transformer

The State Transformer can combine actions and internal states, but in this example there is no such combination. Therefore the State Transformer will leave the document plan unchanged and passes it to the Mood Creator. Chapter 9 will give an example in which actions and states are combined.

6.7.4 Mood Creator

The use of the Mood Creator is only useful in longer stories with several characters and different locations. Therefore I have turned off this component in the example, but chapter 9 gives an example in which the Mood Creator is used.

6.7.5 Branch Remover

The final component of the Document Planner is the Branch Remover. The current example does not include any long branches, so the Branch Remover cannot remove any branches. The result of this component is therefore exactly the same as the document plan shown in figure 6.13.

Chapter 7

Microplanner

This chapter describes the design and implementation of the Microplanner. The first section shows the architecture of the module and in the following sections the components are described in more detail. Finally the functioning of the module is explained using an example.

7.1 Architecture

The Microplanner converts the document plan generated by the Document Planner into a rhetorical dependency graph containing all information needed by the Surface Realizer. A rhetorical dependency graph is represented in the same way as a document plan, except that the plot elements are replaced by dependency trees. Therefore the algorithm traverses the document plan depth-first and replaces the leaves by dependency trees. In order to achieve this the algorithm first applies templates to generate sentence plans and then these sentence plans are transformed into complete dependency trees by performing the first part of the lexicalization. Figure 7.1 shows the architecture of the Microplanner.

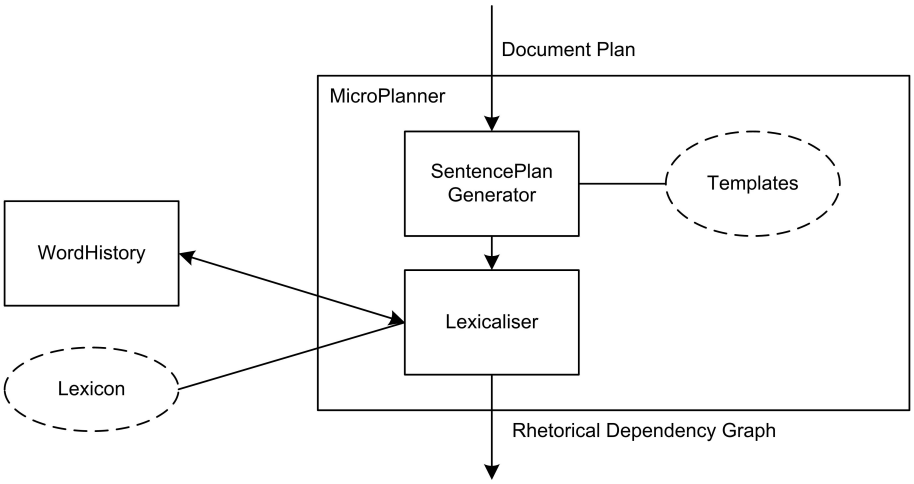

Figure 7.1: Architecture of the Microplanner

7.2 Sentence Plan Generator

In order to convert a plot element into a dependency tree different templates are used. For each plot element a template has been specified which exactly tells how the arguments of the plot element should appear in the dependency tree. The templates for an action, an event, a failed action, a setting, a state and a perception are shown in figure 7.2.

7.2.1 Actions and events

Successful actions and events use the same template, since they are practically the same (except that an action is initiated by a character and an event simply occurs). They place the name of the action (e.g. Goto) in a hd-node, the agens in a su-node, the patiens in an obj1-node and the target in a target node. This target node may contain a preposition, but this depends on the action name. The label 'label' is used to show that the name of the label depends on the verb of the sentence, so the actual label is retrieved from the lexicon when the verb is retrieved (see the following section in which the Lexicalizer is described). Finally, if the instruments Vector of the action is not empty, a pp-node is created containing a preposition ('met' ('with') for a make action and 'via' for a move action). This new pp-node contains all elements of the instruments Vector, possibly combined with comma's or the word 'en' ('and'). An example is the action "De ridder opende de poort *met een sleutel en een code*" ("The knight opened the gate *with a key and a code*"). The template in the figure shows the case in which the instruments Vector contains two elements.

A failed action is converted into a sentence containing the verb 'proberen' ('to try'), for example "De ridder probeerde de poort te openen." ("The knight tried to open the gate."). The template for such a sentence shows that a hd-node is added to the smain-node for the verb 'to try'. Furthermore an inf-node is added with the root-tag set to the name of the action. Note that the morph-tag of this node is set to 'teinf' instead of 'infinitive' in order to create the phrase '*te* openen' ('*to* open').

Furthermore there is a template for an action in passive voice, but this template is not included in the figure. It looks exactly like the template for an action in active voice, except that the agens and patiens are interchanged. In section 2.1.3 we saw that it is usually better to use active voice than passive voice, so we have to make sure that passive voice will not be used too often. However, an example in which it can be used is the situation in which a certain character is central and that character is the patiens of the current action. In order to decide whether it is appropriate to use passive voice the Document Planner should include a model of character centrality. Since this has not been added to the system yet, this is done manually.

7.2.2 Internal states

Figure 7.2 also shows a template for an internal state. First of all, the template adds a node for the name of the state. Then a hd-node is added based on the part of speech of the internal state. If the name of the state represents an adjective, the verb 'to be' is added, and if the name of the state represents a noun, the verb 'to have' is added. An example of the first type is the state *scared*(*princess*) which should be translated into "De prinses *is* bang" ("The princess *is* scared"). An example of the second type is the state of being hungry, which should be lexicalized as "De prinses *heeft* honger." ("The princess is hungry.", note that this internal state simply uses the verb 'to be' in English).

Apart from this standard way to describe states, I included the following two ways which can be used if the character experiences the state intensely: "Wat was ze blij!" and "Ze was nog nooit zo gelukkig geweest!" ("She had never been so happy before!"). The templates for these states are not shown separately. The template for the first one simply adds the modifier 'wat' to the standard tree for states and marks the top node as an exclamation. Using this mark it can be made sure that the sentence will not be combined with another sentence in the Surface Realizer, and secondly, the component which takes care of orthography can add an exclamation mark at the end of the sentence. The template for the second example adds some more nodes; one for the verb 'geweest', an ap-node with the children 'nog' and 'nooit' and an ap-node with the children 'zo' and one for the name of the state (in this case 'gelukkig' ('happy')). The Microplanner supports these two ways to describe states, but in order to decide when one of them can be used, we need the intensity numbers of the emotions as explained in section 5.3.2. Therefore this is done manually for the moment.

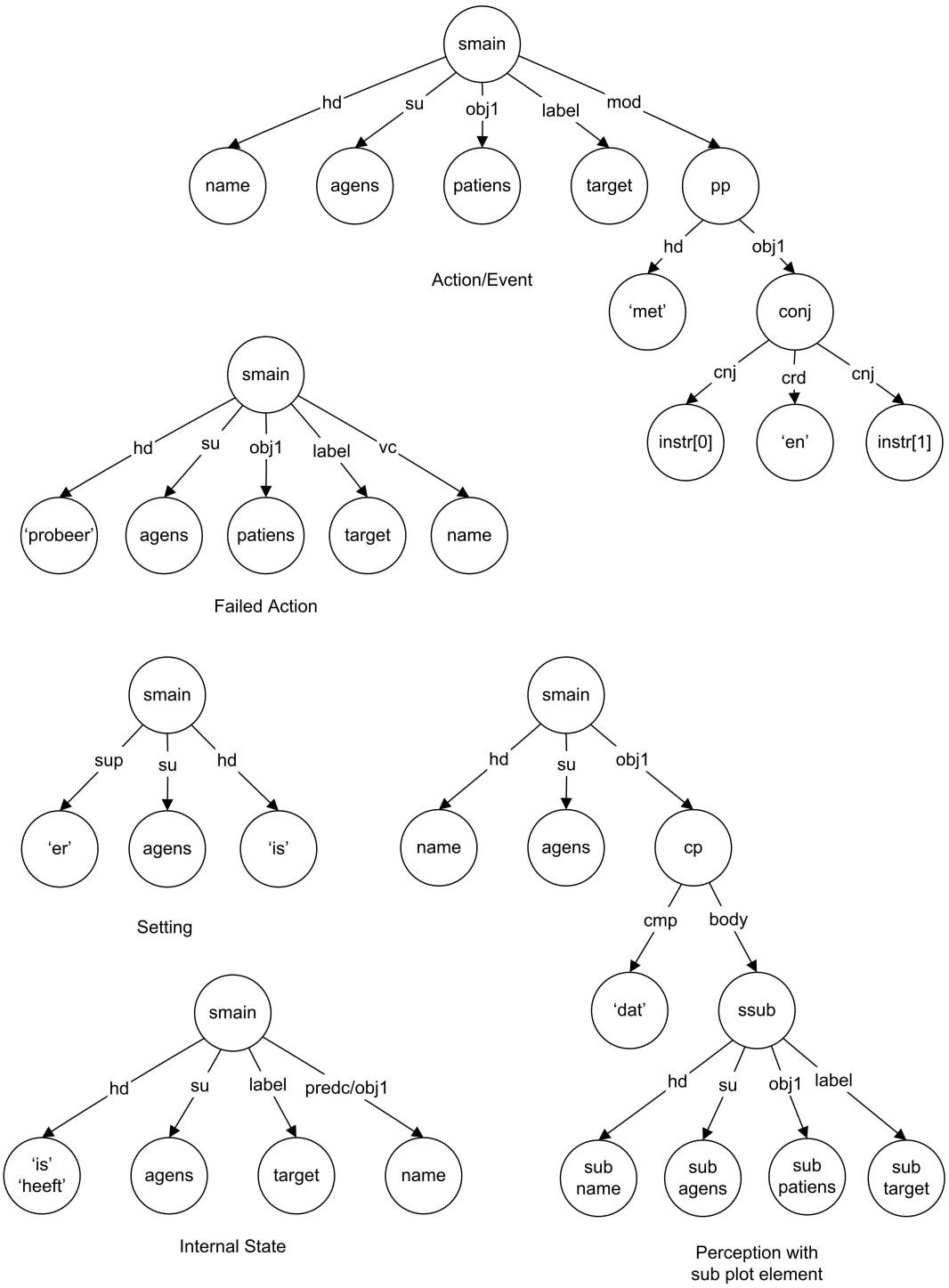

Figure 7.2: Templates for each type of plot element

7.2.3 Perceptions and beliefs

There are two different types of perceptions, one in which the perception has the usual arguments and one in which the perception has a plot element as one of its arguments. An example of the first one is "De prinses zag de ridder." ("The princess saw the knight."). The template for this type of perception is not shown in the figure, because it looks exactly like the template for a successful action in active form; the kind of perception (such as seeing or hearing) is the hd-node of the action.

An example of the other type of perception is "De prinses zag *dat* de brug ingestort was." ("The princess saw that the bridge had collapsed."), containing the word 'dat'. The first part of the sentence is the main clause and is created by generating a su-node for the agens and a hd-node for the name of the perception. Then the sub plot element is transformed into a dependency tree as well and copied to a ssub-node, representing the subordinate clause. Finally the dependency tree for the subordinate clause is connected to the root of the main clause by an edge labelled 'obj1'.

Beliefs are created in the exact same way as perceptions containing the word 'dat', but the main verb is set to 'denken' ('to think').

7.2.4 Goals

The fabula structure distinguishes four different types of goals; an attain goal, a sustain goal, a leave goal and an avoid goal. Additionally, all of these goals can have either an action, an internal state or an object as argument, which results in twelve different types of goals. Furthermore the character of the goal can be the *same* character as the action's agens or state's character, or a *different* character. If the characters are the same, the goal can be lexicalized using a simple sentence, but if the characters are different the sentence should include a subordinate clause as well. Table 7.1 gives an overview of possible lexicalizations of all twenty types of goals.

	Action	**State**	**Object**
Attain Goal	Hij wilde lachen. Hij wilde dat ze lachte (zou lachen).	Hij wilde gelukkig zijn. Hij wilde dat ze gelukkig was (zou zijn).	Hij wilde de appel hebben.
Sustain Goal	Hij wilde (blijven) lachen. Hij wilde dat ze lachte (zou blijven lachen).	Hij wilde gelukkig blijven. Hij wilde dat ze gelukkig was (zou blijven).	Hij wilde de appel houden.
Leave Goal	Hij wilde niet meer huilen. Hij wilde dat ze niet meer huilde (zou huilen).	Hij wilde niet meer bang zijn. Hij wilde dat ze niet meer bang was (zou zijn).	Hij wilde de appel niet meer hebben.
Avoid Goal	Hij wilde niet (gaan) huilen. Hij wilde dat ze niet huilde (zou huilen).	Hij wilde niet bang zijn (worden). Hij wilde dat ze niet bang was (zou worden).	Hij wilde de appel niet hebben (krijgen).

Table 7.1: Examples of lexicalizations of all types of goals

Figure 7.2 does not include a template for a goal, because it can use the other templates. If the characters of the goal and the sub plot element are different, the same template is called used for perceptions, with the name set to 'willen' ('to want'). Furthermore a modifier may be added to the subordinate clause, such as the modifier 'niet meer' ('no more') for a leave goal and the modifier 'not' ('not') for an avoid goal.

If the character of the goal is the same as the character of the sub plot element, or if the argument is an object, a simple sentence will be created. For the generation of such a sentence the same template is called as the template for successful actions. The action's name is set to 'willen' ('to want') and another verb node is added for the main verb with the morphology tag set to 'infinitive'. The root of this verb depends on the type of goal: an attain goal with an action as sub plot element simply takes the name of the action ('wilde

lachen' ('wanted to laugh')), and a sustain goal with an internal state as sub plot element uses the verb 'blijven' ('to stay'). The other roots can simply be taken from the table.

7.2.5 Settings

Figure 7.2 also shows a template for creating settings, such as the sentence "Er is een prinses." ("There is a princess."). Such a sentence is needed to create the first sentence of a fairy-tale "Er was eens een prinses." ("Once upon a time there was a princess."). Furthermore the template can be used for the generation of general descriptions, such as the locations of objects: "Er ligt een zwaard in het bos." ("There is a sword in the forest."). The template is similar to the template of an action in active form, except that a sup-node is added to the top-node, representing the anticipatory subject 'er' ('there').

7.2.6 Adding modifiers

Finally the Sentence Plan Generator checks if the details Vector contains any details about the name of the current plot element. If this is the case the modifier is added to the root of the dependency tree with the label 'mod'. One exception is a detail element that tells something about the name of an internal state, such as the adverb 'erg' ('very') for the internal state "De prinses was bang." ("The princess was scared."). In this case the modifier's scope is not the entire sentence, but the constituent 'bang' ('scared'). Therefore the sentence should be translated into "De prinses was erg bang." ("The princess was very scared."), which can be achieved by creating a new ap-node with the modifier and the original state node as its children.

7.3 Lexicalizer

Once the templates have been filled in, the first step of the lexicalization has to be performed. In this step the concepts are mapped onto Dutch words using a lexicon. Furthermore the chosen words are stored in a word history in order to detect word repetition and possibly prevent the repetition. The nodes representing entities are not converted yet, because this will be done in the Referring Expression Generator (see section 8.3).

7.3.1 Lexicon

In order to translate plot elements into text we need to map concepts to actual words in natural language using a lexicon. Sowa [Sow00] defines the lexicon as the bridge between a language and the knowledge expressed in the language. Each language has a different vocabulary, but every language also provides a grammar in order to combine the words from the vocabulary correctly. Grammars and words are thus language dependent and belong to the area of linguistics, and the concepts they express belong to the extra-linguistic knowledge about the world. Thus, for each language the lexicon must map these language independent concepts to language dependent grammars and words.

A lexicon is usually represented as a list of entries which map concepts to words. The level of detail of the information needed in the lexicon depends on the type of application. For a simple syntactic parser it is sufficient to store the part of speech and word features, but a semantic interpreter requires a more detailed representation including the semantics. The entries in our lexicon consist of the following properties, some of which may be null:

- the concept
- the root of the lexical item onto which the concept can be mapped
- the part of speech tag, such as 'verb', 'noun' or 'adjective'
- the determiner (only for nouns)
- the gender (only for nouns), which also includes the value 'place' in order to create correct referring expressions

- the preposition (only for verbs), such as the preposition 'naar' ('to') for the concept Goto
- the verb particle (only for verbs), such as '*op*pakken' ('to pick *up*')
- the dependency label used in the target node (only for verbs with a preposition), such as the label 'ld' for the concept Goto

The lexicon may contain several entries for a single concept in which case the words stored in the entries represent synonyms for the concept. With the selection of the words stored in the lexicon we have made sure that the words are easy to understand. In section 2.1.3 we saw that it is better to be clear and to use simple words, and this is especially the case in fairy-tales because these are often written for children.

7.3.2 Word History

In the word history all used words are stored. It keeps a list containing word count elements consisting of the following two properties:

- word: the Dutch word used to lexicalize a concept
- count: the number of times the word has been used sofar

Using this history the Microplanner can detect word repetition and if possible it should select different words once in a while.

7.3.3 Lexical Chooser

The Lexical Chooser manages these two knowledge bases and can be queried to get an entry from the lexicon by passing it a concept. The module will then try to find an entry in the lexicon which has not been used recently. If it finds an entry in the lexicon, it checks if this entry has been used before by checking if the word stored in the entry (as root) appears in the word history. If this is the case it stores the entry in a temporary variable and it tries to find another entry. If an entry has been found which has not been used before, this entry is returned. Once it reaches the end of the lexicon and all entries have already been used, it returns the first entry with a chance of $(100 - \#entries \times 20)\%$ and another entry with a chance of 20%. An example is the concept happy for which three entries are stored in the lexicon; the entries representing the Dutch words 'blij', 'vrolijk' and 'gelukkig'. The first one is chosen with a chance of 60% and the other entries with a chance of 20%.

After the Lexical Chooser has selected an appropriate entry for the entity or concept, this entry has to be converted into a node in the dependency tree. A concept can simply be converted into a single node with the root set to the word stored in the lexicon and the other tags set correctly.

7.4 Example

In this example we use the same input as in the example of the Document Planner, see section 6.7. The input to the Microplanner is then exactly the same as the output of the Document Planner and is shown in figure 6.13.

7.4.1 Sentence Plan Generator

The Sentence Plan Generator traverses the document plan depth-first and selects a template for each plot element specifying how the arguments should appear in the final dependency tree. Since the rhetorical dependency graph becomes rather large, I only included the sentence plan for the first plot element in figure 7.3. The first plot element is the settings node for the introduction of the dwarf. The template adds a sup-node for the anticipatory subject, a hd-node for the verb 'to be' and an entity node for the dwarf.

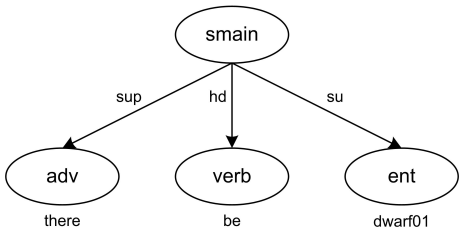

Figure 7.3: Output of the Sentence Plan Generator for the Plop story

7.4.2 Lexicalizer

After the sentence plans have been generated the Lexicalizer selects the appropriate words from the lexicon. In most cases the English word is simply mapped onto a Dutch word and nodes representing entities are left unchanged because they are lexicalized in the Referring Expression Generator. If the lexicon contains more than one entry the module also has to decide which entry is best. For the same reason as with the Sentence Plan Generator I will only give the dependency tree for the first plot element, see figure 7.4. In this example the concept 'there' is simply mapped onto the Dutch word 'er', the verb 'to be' is mapped onto the verb 'zijn' with the root 'ben', and the entity node is left unchanged. Appendix B.2 shows the dependency trees as they appear in the rhetorical dependency graph which is passed to the Surface Realizer.

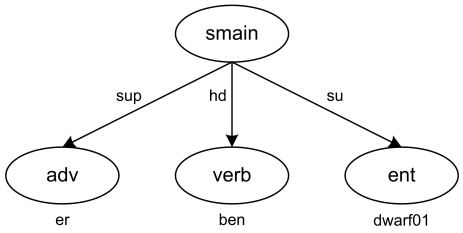

Figure 7.4: Output of the Lexicalizer for the Plop story

Chapter 8

Surface Realizer

The Surface Realizer is the final component in the Narrator architecture. Like the previous two chapters this chapter begins with an explanation of the architecture of the module. Then the components of the module are explained in more detail and the chapter ends with an example illustrating the functioning of the different components of the module.

8.1 Architecture

The Surface Realizer [Hie05] has to convert the rhetorical dependency graph created by the Microplanner into the final text. Doing this the module performs syntactic aggregation, referring expression generation and surface form generation. The architecture of the Surface Realizer is shown in figure 8.1 and all of its components are described in this chapter (recall that the components marked with an asterisk had already been implemented). Furthermore the module has been extended to generate relative clauses which affected several components, so this will be described in section 8.5.

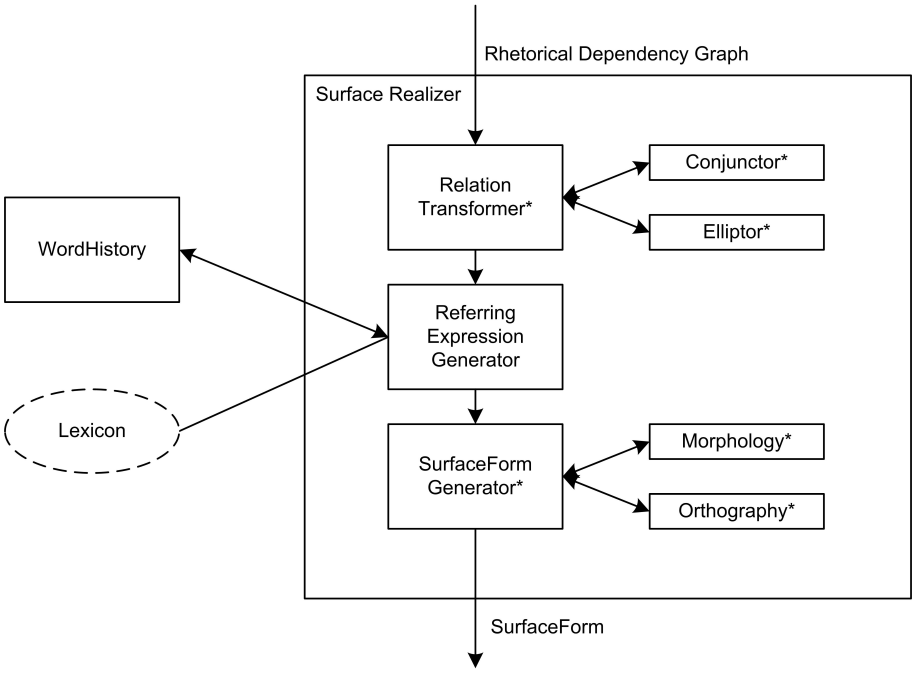

Figure 8.1: Architecture of the Surface Realizer

8.2 Syntactic Aggregator

The syntactic aggregation is performed by the first three components together (the Relation Transformer, Conjunctor and Elliptor), so these components are all described in this section.

8.2.1 Relation Transformer

The rhetorical dependency graph exactly specifies how two dependency trees are related, so by means of which rhetorical relation. The Relation Transformer decides how these rhetorical relations will be expressed in the final text.

Possibilities for expressing rhetorical relations

The module can choose between the following possibilities to express a rhetorical relation between two dependency trees. First of all the two dependency trees can be combined into one tree by adding a conjunction, such as 'omdat' ('because') for a cause relation, or 'maar' ('but') for a contrast relation. Secondly, the module can generate two separate trees and add a cue word as an adjunct to one of the trees, such as 'daarom' ('therefore') for a cause relation, or 'echter' ('however') for a contrast relation. Finally the module can choose not to express the relation explicitly, in which case two separate sentences are generated.

To this list of possibilities I added a combination of the first two ways: using the conjunction 'en' ('and') in combination with an adjunct, such as 'daarom' ('therefore'). This new possibility is especially useful when no conjunction can be used. An example is the following text: "De ridder tilde de prinses op. Hij bracht haar vervolgens naar de brug." ("The knight lifted up the princess. He then brought her to the bridge."), which can then be told like "De ridder tilde de prinses op en vervolgens bracht hij haar naar de brug." ("The knight lifted up the princess and then he brought her to the bridge."). Since the cue word 'en' ('and') is used ellipsis can be applied, so even more variety can be added to the generated texts using this new construction. Note that this construction is only used when the relation is a temporal or a causal relation, in order to avoid sentences such as "De ridder probeerde de poort te openen en echter was die op slot." ("The knight tried to open the gate and however the gate was locked.").

Finally I added cue words to the first sentence of a paragraph in order to show the relation between the two consecutive paragraphs.

Process the tree depth first

The original algorithm [Hie05] looked for rhetorical relations which had two dependency trees as their children (since a relation cannot be processed otherwise). Once it found such a relation the algorithm transformed the relation with its two children into a single dependency tree and placed the resulting tree in the dependency graph. If the two trees could not be combined into one, one of the trees was written to the output and the other tree was placed in the dependency graph. This way the algorithm started at the bottom of the tree at different places and worked its way up to the root by translating those relations which had two dependency trees as their children. For small trees this is satisfactory, but for larger trees this results in texts which are told in a rather random way.

In order to create the correct result, I changed the existing algorithm. The new algorithm also looks for relations which have two dependency trees as their children, but it will not write anything to the output; it will only update the dependency graph. The result of this algorithm is therefore not an ordered list of dependency trees, but a rhetorical dependency graph in which some relations have been transformed. After updating the graph, the nodes will be converted into text from left to right, which can be achieved by processing the tree depth first.

This way only the trees at the bottom of the graph will be connected, but this is satisfactory as the following example shows. Figure 8.2 shows an example dependency graph based on the first paragraph of the final story in appendix A. In this graph the nodes are actually dependency trees, but for clarity the complete sentences in natural language are shown. The Surface Realizer will first combine nodes 1, 2 and 3, then nodes 4 and 5 and finally nodes 6, 7, and 8. This results in the graph shown in figure 8.3. The other trees cannot be combined, since the generated trees would be too complex, but some relations can still be

expressed explicitly by adding an adjunct (such as 'daarna') instead of a conjunction (such as 'nadat'). In order to achieve this the algorithm traverses the trees in the dependency graph once more and transforms some final relations. Furthermore the Branch Remover in the Document Planner makes sure that the final document plans do not contain any long branches, so simply combining the trees at the bottom of the graph is satisfactory.

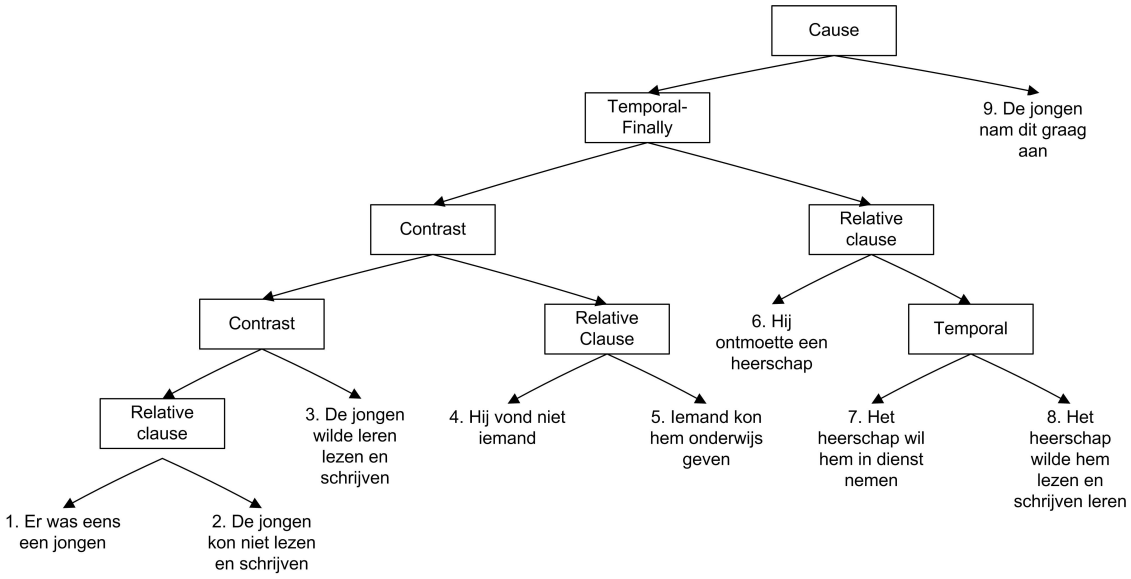

Figure 8.2: Example of a rhetorical dependency graph

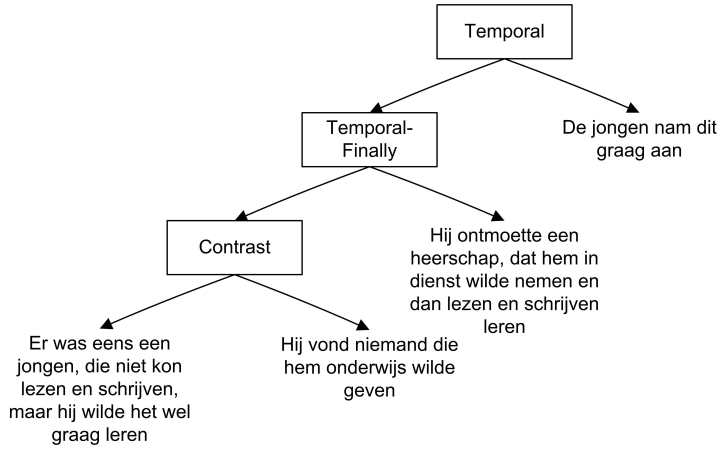

Figure 8.3: Example of a rhetorical dependency graph after combining some trees

The original libraries contained all cue words in one list, so this list included conjunctions as well as adjuncts. I changed this to keep two separate lists: one with the conjunctions and one with the adjuncts. Using these lists the Relation Transformer first tries to combine dependency trees using cue words from the list of conjunctions. Therefore the module traverses the tree twice and combines as many trees as possible.

If no conjunction can be used the algorithm may choose to use the conjunction 'en' ('and') in combination with an adjunct. Then the module traverses the tree once more and adds some final cue words from the list of adjuncts. Finally a cue word may be added to the first sentence of the paragraph which shows the relation to the previous paragraph. In order to achieve this the Relation Transformer searches for the leftmost dependency tree and adds the adjunct to this tree.

Additional requirement for combining two dependency trees into one tree

Hielkema used as requirements for combining two dependency trees that the resulting tree should not contain more than three clauses. This can be achieved by checking that only one of the trees is complex and consists of exactly two clauses. Furthermore the selection of the cue word was based on the cue words that had already been used. In order to achieve this a list is kept with the cue words that have been chosen recently, and if the cue word appears in the list a different cue word is selected.

I added the requirement that a *subordinating* conjunction may only be added to a simple sentence (a sentence consisting of only one clause). An example is a tree containing the following three actions all combined by temp-after-sequence relations: "De ridder pakte de prinses op. Hij zette haar op zijn paard. Hij bracht haar naar een brug." ("The knight lifted up the princess. He placed her on his horse. He took her to a bridge."). If the first two clauses have already been combined into "De ridder pakte de prinses op en vervolgens zette hij haar op zijn paard." ("The knight lifted up the princess and then he placed her on his horse."), the original Relation Transformer would try to combine this sentence with the third clause by adding the cue word 'nadat' ('after') to the first sentence. In this case the algorithm should transform the complex sentence (consisting of two main clauses) into a subordinate clause, but since the Relation Transformer fails to do this correctly, the following sentence is generated: "Nadat de ridder de prinses op had gepakt en vervolgens zette hij haar op zijn paard, bracht hij haar naar een brug." ("After the knight had lifted up the princess and then he placed her on his horse, he took her to a bridge."). The correct sentence would be "Nadat de ridder de prinses op had gepakt en vervolgens op zijn paard had gezet, bracht hij haar naar een brug." ("After the knight had lifted up the princess and placed her on his horse, he took her to a bridge."), but this sentence is still rather complex. In order to avoid this problem we will only add subordinating conjunctions to simple sentences, which results in this example in the following text: "De ridder pakte de prinses op en vervolgens zette hij haar op zijn paard. Daarna bracht hij haar naar een brug." ("The knight lifted up the princess and then he placed her on his horse. Afterwards he took her to a bridge.").

8.2.2 Conjunctor

The Conjunctor is responsible for the actual transformation of rhetorical relations and the combination of two dependency trees into one. This module has remained mainly the same, but I included the cue word 'zo ..., dat' to the cause relation and I changed the order in which the clauses are told.

Adding the cue word 'zo ..., dat' to the cause relation

Many existing stories use the construction 'zo ..., dat' in order to lead to a heightened suspense. An example of such a construction is the sentence "De prinses was zo bang, dat ze schreeuwde." ("The princess was so scared, that she screamed."). The construction is implemented as an additional cue word in the cause relation, since it is a special case of this relation. The algorithm is very similar to the cause relation which uses a different cue word such as 'omdat' ('because'); the only difference is that the word 'zo' ('so') has to be added to the first clause. This can be done by taking the node with the part of speech 'adv' and replacing this node by an ap-node containing the original adv-node and a node representing the word 'zo'. Furthermore it is made sure that the order of the generated sentences is correct, which can be done using the second modifier label described in section 8.4.1. If the original 'mod' label would have been used, the resulting sentence would be "Dat de prinses schreeuwde, was ze zo bang." ("That the princess screamed, was she so scared."), but using the 'modb' label, the phrases are generated in the correct order.

Leaving the order of the nodes unchanged

The original Surface Realizer selected a cue word from the list of cue words and placed the subordinate clause after the main clause, or embedded the subordinate clause in the main clause. The advantage of doing this is that the generated texts become more varied than always placing the subordinate clause and the main clause in a fixed order, but in some cases it may also lead to somewhat more complicated texts. Take for example the sentence from the example story of figure 3.4: "Hij sloeg Amalia, want, doordat Amalia het zwaard op pakte, werd hij bang." ("He hit Amalia, because, because/when Amalia picked up the sword, he got scared."). In this example the subordinate clause is embedded in a main clause and furthermore the clauses are not told in chronological order, so the result is a more complicated text than necessary. If the order of the nodes would have been left unchanged the resulting text would be as follows: "Omdat Amalia het zwaard op pakte, werd Brutus bang en daarom sloeg hij haar." ("Because Amalia picked up the sword, Brutus got scared and therefore he hit her."). Obviously this text is easier to understand and therefore I changed the algorithm in such a way that it leaves the order of the nodes unchanged, which automatically means that the nodes are always told in chronological order.

The child nodes of a rhetorical relation in the rhetorical dependency graph are called the satellite and the nucleus. Furthermore for each cue word has been stored whether it should be added to the satellite or nucleus, so this information can be used to make sure that the order of the nodes will not be changed (i.e. that the satellite is always told before the nucleus). The clause containing the cue word is the subordinate clause and the other one is the main clause. This means that if the cue word is added to the satellite, the first clause becomes the subordinate clause and should therefore be placed *before* the main clause. This can be achieved by using the 'moda' label instead of the standard 'mod' label as described in section 8.4.1. On the other hand, if the cue word is added to the nucleus, the subordinate clause should be placed *after* the main clause, which can be achieved by the 'modb' label, also described in section 8.4.1.

A disadvantage of keeping the order of the nodes unchanged is caused by the fact that the Surface Realizer only supports a Cause relation rather than a Cause relation as well as a Consequence relation. For that reason each cause relation is expressed by first telling the cause and then telling the consequence. Since many cue words have been stored there are still many different possibilities to express such a relation, but the cue word 'want' will never be used anymore.

8.2.3 Elliptor

The Elliptor is responsible for the syntactic aggregation. If the Conjunctor combines two dependency trees into one, the Elliptor checks if syntactic aggregation can be applied. This module has been mainly left unchanged, so for the implementation details see [Hie05]. The only difference is that the original Elliptor compared different nodes by checking if the part-of-speech tag and the root were the same, and the new Elliptor will check if the entities are the same. Checking whether the root was the same was satisfactory because the sentences always contained only one entity for each concept and because the same word was used to refer to an entity each time.

In the current version however comparing the roots results in two problems. First of all, two different entities which are an instance of the same concept can be described by the same noun. If this is the case the original algorithm would treat the entities as if they were the same entity and performs ellipsis in cases in which this is not appropriate. An example of this is the sentence "De mooie prinses was blij en de lelijke prinses was verdrietig" ("The beautiful princess was happy and the ugly princess was sad") which would be turned into "De mooie prinses was blij en verdrietig." ("The beautiful princess was happy and sad."). Secondly, if one entity is described using different referring expressions (for example by retrieving different words from the lexicon), no ellipsis can be performed since the roots are different. Therefore I included the entity (a unique identifier) to the nodes in the tree and changed the Elliptor in order to check for similarity of this entity instead of the root.

8.3 Referring Expression Generator

One of the main tasks of a natural language generation system is the generation of referring expressions, so the system has to decide how to refer to an entity. This can be done using very simple algorithms, but there are also more complicated algorithms. Since the generation of referring expressions is a considerably large part of the project, I will discuss it in more detail than the other tasks.

This section first gives an overview of some literature about the generation of referring expressions, including the kinds of referring expressions and some existing algorithms. Then the algorithm designed for the Virtual Storyteller is described in more detail in section 8.3.2. Section 8.3.3 then describes how the algorithm can be extended in order to generate some other types of referring expressions as well. Finally section 8.3.4 gives some examples of referring expressions generated by the Referring Expression Generator.

8.3.1 Existing algorithms for referring expression generation

The first task of the Referring Expression Generator is deciding what type of referring expression will be generated. First of all, the referring expression can be a description, such as the noun phrase 'de mooie prinses' ('the beautiful princess'). Furthermore the character's name can be used (assuming that the character has a name) such as in the expression 'prinses Amalia' ('princess Amalia'). Finally a personal pronoun can be used, such as 'zij' ('she') or 'haar' ('her').

Obviously the Referring Expression Generator should only generate a pronoun if no confusion can arise about which entity is being referred to. Furthermore the module has to make sure that the referring expressions contain enough information to find out which entity is being referred to. This means that nouns should be selected properly and that correct adjectives should be added if the story contains different entities which are an instance of the same concept. Therefore Dale [Dal92] came up with three principles the generated referring expressions should satisfy. The first one is the principle of sensitivity, which means that the expression should be understandable to the hearer or reader. This also includes selecting a correct determiner, since using an incorrect determiner may also lead to confusion. The second principle is the principle of adequacy which means that the referring expression should include enough information to find out which referent is being referred to. The final principle is the principle of efficiency which means that the referring expression should not contain more information than necessary. Later algorithms have dropped this final principle, because experiments showed that people often use referring expressions containing redundant information; hearers are more concerned with the principle of sensitivity than with the principle of efficiency [Rei95].

Dale's Incremental algorithm

Reiter and Dale [RD92] designed an algorithm that can generate referring expressions which satisfy the principle of sensitivity and the principle of adequacy. The algorithm takes an entity r, a distracter set C (the set of entities from which r has to be distinguished) and an ordered list of preferred attributes P. The algorithm first chooses the best noun to refer to the entity; this is the noun which rules out most elements of the distracter set. If the distracter set is not empty (so if there are different entities which belong to the same concept), the algorithm goes through the list of properties and checks for each property whether its value rules out at least one other element of the distracter set. If the property does rule out another element, the value of the property is added to the generated referring expression and the next property is examined. This continues until either the distracter set is empty (in which case an adequate referring expression has been generated) or the complete set of properties has been examined (in which case there are still different possible referents).

In order to illustrate their algorithm Reiter and Dale use the following example. Assume the context contains the following three entities and a referring expression has to be generated for $Object1$:

- $Object1: \ < type\ chihuahua >, \ < size\ small >, \ < color\ black >$
- $Object2: \ < type\ chihuahua >, \ < size\ large >, \ < color\ white >$
- $Object3: \ < type\ siamesecat >, \ < size\ small >, \ < color\ black >$

In this example the variables are set as follows: $r = Object1$ and $C = \{Object2,\ Object3\}$. Furthermore assume that $P = \{type,\ color,\ size\}$. When the algorithm for generating a referring expression is called, the algorithm first looks at the first element of the list of properties P and chooses the best value for the property *type*; it chooses the best noun to refer to r which is *dog*. This noun rules out $Object3$, so the distracter set is set to $C = \{Object2\}$. Then the next property in P is examined which is *color*. The best value for this property is *black* and this rules out $Object2$. At this point the distracter set is empty and the algorithm returns a specification for 'the black dog'.

The algorithm described in this section assumes that each entity can be represented as a list of properties. Furthermore there may be subsumption hierarchies for the values of some of the attributes, and one of these values is called the *basic level value*. Take for example the subsumption hierarchy of the attribute *type*; this may specify that a chihuahua is a kind of dog, and that a dog is a kind of animal. In this example the value 'dog' is the basic level value of the attribute 'type'.

Modification of the Incremental algorithm

The Incremental algorithm has some limitations and therefore Krahmer and Theune [KT00] modified the algorithm. They focus on contexts in which there are many different entities which belong to the same concept, such as a dog show with hundreds of different dogs. In such contexts an unambiguous referring expression may become as complex as 'the large black long-haired sausage dog'. The original Incremental algorithm would generate this referring expression each time an expression were to be generated for the dog, because the distracter set was equal to *all* entities in the context. However, it would be much better if the algorithm generated simpler expressions if a particular dog is more salient than the other dogs. Therefore Krahmer and Theune extended the original algorithm with salience values. This means that only those entities are added to the distracter set which have a salience value higher than or equal to the salience value of the entity for which a referring expression is to be generated. This also means that in the initial situation the distracter set contains all entities, since the entity has to be distinguished from *all* possible entities. For subsequent references a simpler referring expression can be used since the distracter set only contains a small number of entities. This results in shorter referring expressions which contributes to the principle of efficiency, although the generated referring expressions are still not necessarily optimally efficient.

Salience-based algorithm for pronoun resolution

As already said the modified incremental algorithm makes use of salience values. There are many different algorithms that use salience values in order to determine which referents are prominent. One of these algorithms is the algorithm of Lappin and Leass [LL94] which is actually meant to *resolve* pronouns in a text. The algorithm processes a text and each time a pronoun is used, it generates a set of possible referents and calculates which referent is most likely the correct one. This process is actually the opposite of generating a referring expression, but the salience values used in the algorithm may be useful for our Referring Expression Generator.

Jurafsky and Martin [JM00] describe a number of constraints and preferences for referring expressions and their referents when resolving pronouns. The *constraints* include first of all agreement of number, person, case and gender. Secondly there are syntactic constraints when the antecedent appears in the same clause as a referring expression. An example is the sentence "John bought him a new car" in which 'him' cannot refer to 'John'. Finally there are selectional restrictions, for example when a verb requires an animate object as one of its arguments.

The *preferences* defined by Jurafsky and Martin include first of all recency, which means that more recent referents are more salient than other entities. The preference for grammatical role means that the phrases in a sentence are ordered by their grammatical roles; e.g. entities which appear in subject position are preferred to entities which appear in object position. Thirdly, entities will become more salient when they are being referred to repeatedly. Next there is a preference for parallelism, which means that a referent is more likely to refer to an entity that appears in the same grammatical role than another entity. Finally some verbs emphasize certain arguments, such as 'telephoning' which emphasizes the subject or 'criticizing' which emphasizes the object. An example of this final preference is as follows: the sentence "John telephoned Bill" emphasizes the subject, so if a sentence containing the referring expression 'he' follows, the pronoun most

likely refers to John. On the other hand, a pronoun following the sentence "John criticized Bill" most likely refers to the object Bill.

The algorithm of Lappin and Leass takes a number of these preferences into account. The algorithm keeps a discourse model with all entities mentioned sofar and for each entity a salience value that models the level of saliency. The salience value is calculated as the sum of the weights assigned by a number of salience factors. The salience factors with their corresponding salience values are as follows: sentence recency (100), subject emphasis (80), existential emphasis (70), accusative emphasis (50), indirect object and oblique complement emphasis (40), head noun emphasis (50) and non-adverbial emphasis (80). Each time a reference to a referent is encountered (either a pronoun or a noun phrase) the associated value is updated and the algorithm moves on to the next reference. When the end of a clause is reached all salience values are cut in half which models the preference for more recent referents. Furthermore the algorithm implements the parallelism preference by adding *temporarily* 35 points to a possible referent if this referent fulfills the same grammatical role as the referring expression. Finally the algorithm adds a negative score to cataphora (references in which the antecedent appears after the pronoun).

8.3.2 Algorithm for referring expression generation in the Virtual Storyteller

Using the algorithms described sofar I designed an algorithm which is based on the modified incremental algorithm. In the outline of the algorithm (see algorithm 2) a number of functions are called, which are described in the following subsections. You can see that the first step of the algorithm is choosing the kind of referring expression. If a pronoun can be used the correct pronoun is returned, otherwise the algorithm decides whether to generate a noun phrase containing the entity's name or to generate a noun phrase without the entity's name. Depending on the type of referring expression the correct function is called and the correct referring expression is returned.

Algorithm 2 GenerateReferringExpression(r)

```
if Pronominalize(r) then
  return CreatePronoun(r)              // e.g. 'zij'
else
  if UseName(r) then
    if OnlyName(r) then
      return CreateName(r)             // e.g. 'Amalia'
    else
      return CreateNameNoun(r)         // e.g. 'prinses Amalia'
    end if
  else
    return CreateNounPhrase(r)         // e.g. 'de mooie prinses'
  end if
end if
```

Choose the kind of referring expression

The first step of the algorithm is choosing what kind of referring expression will be generated. Analysis of human-written fairy-tales (see section 4.4) has led to a number of conclusions about when to use a pronoun and when to use a noun phrase. First of all pronouns can be used when no confusion will arise about which referent is being referred to, but using noun phrases can make the stories more varied. This results in the following cases in which a noun phrase can be used:

- At the beginning of a paragraph.
- If a pronoun has been used a number of times (about four times) *and* the referring expression is the first one in the sentence.

- If the antecedent has not been mentioned for two sentences.
- If it is unclear which referent is being referred to when using a pronoun, since there are two salient characters of the same gender.
- If the referring expression should include additional information such as adjectives. These can be added by the Document Planner, for example when describing a state.
- If the referring expression shows a relation between two characters, such as father/daughter.

Furthermore many existing algorithms for pronominalization can be found in the literature. Appendix C includes a chapter based on the results of some research I did for another course. The chapter gives an overview of a number of existing theories and algorithms, and combines the advantages of some of them into a new algorithm. The algorithm described in the appendix can be combined with the results of the analysis, and the resulting algorithm is shown in algorithm 3. The algorithm is a simple if-then-else algorithm and returns 'true' if a pronoun can be used and returns 'false' when it is better to use a noun phrase. The algorithm contains one new condition; one that states that a noun phrase should be used when a relative clause is defined for the entity. This is necessary to avoid sentences such as "Zij, die lachte, ging naar het bos." ("She, who laughed, went to the forest.").

The final if-statement in algorithm 3 checks if the referent has the highest salience value. This is done using the salience factors and values used in the algorithm of Lappin and Leass (described in section 8.3.1). Furthermore the salience value of r is not compared to the salience values of the entities that have already been mentioned in the same clause, because they cannot corefer with r. An example is the sentence "Nadat de ridder naar de prins was gegaan, sloeg hij hem." ("After the knight had gone to the prince, he hit him."). In this example the entities do not appear in strong parallelism, because the prince is an indirect object in the first clause and a direct object in the second clause. However, the subject is the same in both clauses and can be pronominalized in the second clause (since the knight is the entity with the highest salience value). Once a referring expression is to be generated for the prince, the knight is still the entity with the highest salience value, so the prince could not be pronominalized. However, since the subject 'he' cannot corefer with the object 'him' it is allowed to use a pronoun for the prince as well, and this can be achieved by ignoring the entities that have already been mentioned in the same clause.

Algorithm 3 Pronominalize(r)

```
if first reference to r in current paragraph
   or antecedent has not been mentioned for two sentences
   or first reference in sentence and a pronoun has been used 4 times
   or referent contains a relative clause
   or adjective should be added (determined by the Document Planner) then
   return false
end if
if r has not been mentioned in current sentence then
   if strong parallelism with previous sentence then
      return true
   end if
else
   if strong parallelism with first clause
      or r appears in causal relation then
      return true
   end if
end if
if r has highest salience value then
   return true
end if
return false
```

Generate pronouns

The algorithm to generate a pronoun is really simple; it simply uses the grammatical role in the sentence and the gender of the referent. If the word is an object it also checks the determiner to select the correct pronoun. The algorithm is shown in algorithm 4.

The algorithm only generates a pronoun for an object if it appears in subject or direct object position, such as "De ridder probeerde de poort te openen, maar *die* was op slot." ("The knight tried to open the gate, but *it* was locked.") or "Plop pakte de appel op en at *hem*." ("Plop picked up the apple and ate *it*"). This means that it does not generate a pronoun when the object appears in a prepositional phrase. I added this requirement in order to avoid sentences such as "De prinses ging naar het bos en de ridder ging ook naar *hem*." ("The princess went to the forest and the knight went to *him* too."). In the future this may be solved by generating expressions such as 'daarnaartoe' ('there').

Algorithm 4 CreatePronoun(r)

```
gender ← GetGender(r)
rel ← GetRel(r)
det ← GetDeterminer(r)
if gender = female then
  if rel = su then
    return 'zij'
  else if rel = obj1 then
    return 'haar'
  else if rel = poss then
    return 'haar'
  end if
else if gender = male then
  if rel = su then
    return 'hij'
  else if rel = obj1 then
    return 'hem'
  else if rel = poss then
    return 'zijn'
  end if
else if gender = neutral then
  if rel = su then
    if det = 'het' then
      return 'dat'
    else
      return 'die'
    end if
  end if
end if
```

Generate noun phrases

The first step of the algorithm for generating a noun phrase is to decide what *kind* of noun phrase is to be generated, so to decide whether the name of the entity should be included or not (assuming the entity has a name). This decision is made randomly; 25% of the references uses the name and the other 75% uses a description. If the algorithm decides to generate a referring expression containing the entity's name, there are still two possibilities: simply the name (such as 'Amalia'), or a noun phrase containing the name (such as 'prinses Amalia'). This final construction can only be used when the noun describes a function, such as princess, king or knight. If this is the case the algorithm includes the noun, otherwise it will only generate the name. This means that the function UseName() in algorithm 2 returns *true* with a chance of 25% and $false$

with a chance of 75%. Furthermore the function OnlyName() returns *true* if the referent is not a 'title' noun such as princess or king, and $false$ otherwise. Note that a referring expression containing the entity's name does not include any adjectives. This means that the algorithm will not add any *distinguishing* adjectives either (these are adjectives selected by the Incremental algorithm in order to distinguish the entity from the other entities in the context), but this is satisfactory since we assume that all entities have unique names. In the real world this is obviously not the case, but since we are focussing on fairy-tales this is reasonable.

If the algorithm decides to generate a noun phrase without the name the resulting algorithm consists of the following three steps: first a noun will be selected, then a number of adjectives (possibly zero) will be added to the noun phrase and finally a determiner will be added. The entire algorithm for generating a noun phrase is shown in algorithm 5.

Algorithm 5 CreateNounPhrase(r)

```
result ← null

// Step 1: select a noun
noun ← GetEntry(r)
result ← CreateNpNode(noun)

// Step 2a: add distinguishing adjectives
C ← all entities with the same concept as r and a higher salience than r
for each property P of r do
   outruled ← RulesOut(C, P, V)          // V is value of property P
   if outruled not empty then
      C ← C − outruled
      result ← AddAdjective(result, V)
   end if
   if C empty then
      return true
   end if
end for

// Step 2b: add internal state adjectives
for each adjective A specified in Document Planner do
   result ← AddAdjective(result, A)
end for

// Step 2c: add remaining adjectives
if result contains no adjectives then
   adj ← GetPossibleAdjective(r)
   if adj not null then
      result ← AddAdjective(result, adj)
   end if
end if

// Step 3: add the correct determiner
if AlreadyMentioned(r) then
   result ← AddDefiniteDeterminer(result)
else
   result ← AddIndefiniteDeterminer(result)
end if

return result
```

The first step of the algorithm for generating a noun phrase without the entity's name consists of selecting an appropriate *noun*. If the lexicon contains more than one entry, the same rule is used to decide which

entry is best as was done with the selection of entries for adjectives and verbs; the first entry is chosen with a chance of $(100 - \#entries \times 20)\%$ and the other entries are chosen with a chance of 20%.

The second step of the algorithm is adding *adjectives* to the noun phrase. This step can also be subdivided into three different steps: adding distinguishing adjectives (which are necessary in order to create an unambiguous referring expression), adding internal state adjectives (which are added by the Document Planner when describing a state) and adding other adjectives (which are added by the Surface Realizer based on a list of nouns and for each noun a list of possible properties).

The modified incremental algorithm (described in section 8.3.1) is used to generate the distinguishing adjectives. The algorithm takes as only argument the referent for which a referring expression has to be generated (referent r). The set of distracters is then calculated by taking all entities with a salience value higher than the salience value of referent r. Finally the list of preferred attributes is ignored; the properties which are defined for r are simply examined in the same order as they have been stored in the Character Information. This means that we assume that the properties are ordered with respect to their importance. This way the properties are examined until an adequate referring expression has been generated or all properties have been examined. Furthermore when introducing a *new* character all properties available in the Character Information are also added to the referring expression, because they can be used as distinguishing adjectives later in the story. These adjectives are therefore no distinguishing adjectives like the ones meant by Reiter and Dale, but we will call these adjectives distinguishing adjectives as well, since they have the same function.

As described in section 6.4 the Document Planner supports different ways to describe a state. This can be done using a complete sentence, but in some cases the state is converted into an adjective which should be added to the referring expression. If this is the case the entity node includes a mark that an adjective should be added and the Referring Expression Generator then adds this adjective to the generated noun phrase.

The algorithm just described generates adequate referring expressions, which means that the referring expression is usually unambiguous and includes additional information such as internal states. Since the stories generated by the Virtual Storyteller will not often contain different characters which are an instance of the same concept, the first part of the algorithm will often return a noun phrase which only contains a noun (without any adjectives). Therefore the Narrator agent will include a list of nouns and for each noun a number of possible properties. This list will only include neutral entities (such as gates and bridges) rather than characters as well. The main reason for this is that characters already have a number of properties so simply adding new properties may lead to confusion. Furthermore adjectives are only added to nouns for which no properties have been specified, otherwise these properties can be used. Using the list of possible properties an adjective is chosen when an entity is used for the first time, and this adjective is then stored in a separate list. Each subsequent reference to the entity may include this adjective or use no adjective at all (this is also done randomly). This way only one adjective for each entity can be used throughout the story and therefore no confusion can arise. Take for example the sentence "De ridder probeerde de zware poort te openen, maar de groene poort was op slot" ("The knight tried to open the heavy gate, but the green gate was locked"). In this sentence there are two references to the same gate, but since different adjectives are used the reader might think there are two different gates. By always taking the same adjective (or using no adjective at all) no confusion can arise. On the other hand, the gate can be heavy and green at the same time, but if this information is relevant the adjectives have already been added in other steps of the algorithm.

The final step of the algorithm for generating noun phrases is choosing a correct *determiner* and adding this to the noun phrase generated sofar. This is done by keeping an entity history and using an indefinite article when the entity has not been mentioned before, and using a definite article when the entity has already been mentioned. This means that different nouns can be used to refer to an entity and still the correct determiner is chosen, which is shown in the following example: "Er was eens een koning. *De* vorst woonde in een ver land." ("Once upon a time there was a king. The ruler lived in a faraway country."). In the first sentence the entity (e.g. king01) is introduced using the noun 'koning' ('king') and the indefinite article 'een' ('a') is used. Then the second sentence tells something about the same entity, but uses the noun 'vorst' ('ruler'). Since the choice of the determiner depends on the *entity* a definite article is chosen correctly. Note that this is similar to the generation of determiners for aliases in StoryBook (see section 2.3.3).

In the future the Story World can also be used to find out if a definite article can be used. If an entity has not been mentioned before the Story World can be queried to find out if that entity is the only entity of that type, and if that is the case a definite article can be used. An example is a referring expression for a king in a story in which there is only one king; in this case the first referring expression may also use a definite article even though this is not necessary.

Include relation nouns

The just described algorithm generates correct referring expressions, but for some entities has been specified that they are related to other entities. Examples are the part-of relation, such 'de poort van het kasteel' ('the gate of the castle'), the belong-to relation 'de slaapkamer van de prinses' ('the bedroom of the princess') and the related-to relation 'de vader van de prinses' ('the father of the princess'). The first two of these examples simply specify *which* gate and *which* bedroom is meant, but the third example represents a different way for describing a king.

After the noun phrase has been generated the algorithm checks if such information is available in the Character Information module. If this is the case the algorithm for generating a referring expression is also executed for the new entity and both referring expressions are combined with the preposition 'van' ('of'). This is done recursively in order to generate more complicated referring expressions as well, such as 'het slot van de poort van het kasteel' ('the lock of the gate of the castle').

8.3.3 Inference-based referring expression generation

The algorithm described sofar generates adequate referring expressions in the sense that the generated expressions usually are unambiguous and they are (at least to some extent) varied. When we look at the referring expressions used in human-written texts however, there are far more different kinds of expressions than the ones generated in the Virtual Storyteller sofar. In this section we will look at the different types of definite descriptions and the generation of some of these. The descriptions can be subdivided into four familiarity classes using the definitions of hearer status and discourse status, so let's first define these concepts.

When telling a story the speaker has to make sure the hearer understands everything that is being told. *Information status* has to deal with the shared knowledge of discourse participants; it is defined as the speaker's assumptions about the hearer's knowledge. When focusing on natural language generation information status is thus defined as the system's assumptions about the reader's knowledge. Prince [Pri81] believes that information status can be considered old or new with respect to the *hearer* or with respect to the *discourse*. Hearer-old entities are then entities which are already known to the hearer and hearer-new entities are new to the hearer. Furthermore discourse-old entities have already been mentioned in the discourse and discourse-new entities have not.

Note that the concepts hearer status and discourse status are partially independent of each other. The concept discourse-new does not say anything about hearer status and the concept hearer-old does not say anything about discourse status. On the other hand, the concept discourse-old does imply hearer-old and the concept hearer-new implies discourse-new. Table 8.1 shows examples of all possible combinations of discourse status and hearer status.

	discourse-new	discourse-old
hearer-new	I bought *a new car* yesterday.	-
hearer-old	*The sky* is blue.	I bought a new car yesterday. *The car* is red.

Table 8.1: Examples of discourse status and hearer status

Types of definite descriptions

Gardent and Striegnitz [GS03] focussed on the generation of different types of definite descriptions. Firstly, they state that a definite description should satisfy the criteria of uniqueness and familiarity. *Uniqueness* means that the referring expression should be adequate in order to decide which entity is being referred to, so the intended referent should be the only salient referent satisfying the definite description. *Familiarity* means that the intended referent should be known to the reader. Dale's Incremental algorithm [Dal92] (described in section 8.3.1) generates referring expressions which satisfy these two criteria, but it generates only a small subset of the possible types of referring expressions. Using the definitions of hearer status and discourse status defined earlier in this section, Gardent and Striegnitz subdivide all of the possible expressions into the following four familiarity classes: coreferential (direct or indirect), bridging, larger situation and unfamiliar uses.

Coreferential uses refer to an entity that has been mentioned before in the discourse (a discourse-old entity). This can be done using a hearer-old description (direct coreferential use), or using a different noun in a hearer-new description (indirect coreferential use). Gardent and Striegnitz also distinguish between two different kinds of indirect coreferential use: one in which the second reference is a hypernym of the first reference (a generalization such as 'actress' → 'woman') and one in which the second reference is a hyponym of the first reference (including more information such as 'car' → 'Volvo'). Poesio and Vieira [PV98] add two more types to this set of indirect coreferential uses: synonyms (such as 'trousers' → 'pants') and reformulations (for example by turning a verb into a noun such as 'travelled' → 'the journey'). This final type can also be seen as some sort of bridging, so we will ignore this kind of coreferential use. This results in the following types of coreferential uses:

- Direct coreferential uses: A woman came in. *The woman* was wearing a hat.
- Indirect coreferential uses:
  - Hypernymy: An actress came in. *The woman* was wearing a hat.
  - Hyponymy: I bought a new car. *The Volvo* is blue.
  - Synonymy: Fred was wearing trousers. *The pants* were green.

In a *bridging use* the referent is discourse-new, but somehow related to a discourse-old entity via some inferrable relation, the *bridging relation*. In order to recognize these relations the reader needs to reason based on world knowledge and the discourse context. Clark [Cla75] identified the following types of bridging relations:

- Set-membership bridging: I met two people. *The tall one* told me a story.
- Part-of bridging:
  - Necessary part: John entered the room. *The ceiling* was high.
  - Probable part: John entered the room. *The windows* were large.
  - Inducible part: John entered the room. *The chandelier* was sparkling brightly.
- Roles bridging:
  - Necessary role: John was murdered. *The murderer* got away.
  - Optional role: John died. *The murderer* got away.

All of the aforementioned bridging relations are relations between two entities, such as a room and its ceiling, but Asher and Lascarides [AL99] and Clark [Cla75] also focus on bridging between complete sentences based on the rhetorical relation between those sentences. This kind of bridging is much more complicated than the other kinds and since it is probably not necessary to generate those bridging descriptions I will ignore them.

In *situation uses* the entity being referred to has not been mentioned before, but is assumed to be part of the hearer's world knowledge; this means that the used definite description refers to a discourse-new but hearer-old object. Poesio and Vieira [PV98] distinguish between immediate situation uses and larger situation uses. In immediate situation uses the referent may be visible or not:

- Visible immediate situation: Pass *the salt* please.
- Non-visible immediate situation: Beware of *the dog*.

Furthermore in larger situation uses the speaker refers to some entity which is not apparent in the immediate situation, but is part of shared knowledge. Finally Gardent and Striegnitz [GS03] add references to unique entities to this subclass. This results in the following examples of larger situation uses:

- General world knowledge (larger situation): Have you seen *the bridesmaids*? (at a wedding)
- Unique entities (larger situation): *The sun* is rising.

Finally the *unfamiliar uses* contain all definite descriptions that cannot be categorized as belonging to one of the aforementioned familiarity classes. This class includes all uses where the referent of the definite description is neither discourse-old or hearer-old nor related to some discourse-old entity. Poesio and Vieira subdivide this class into the following categories:

- Np-complements (in which the head noun is complemented): John is amazed by *the fact that his father is black*.
- Nominal modifiers (in which the nominal modifier refers to the class to which the head noun belongs): I like *the color red*.
- Referent-establishing relative clauses (in which the new entity is uniquely identified by the limiting relative clause): *The man John met yesterday* is interesting.
- Associative clauses (which can be seen as cases of bridging in which the related entity is mentioned as well): John swam to *the bottom of the sea*.
- Unexplanatory modifiers use (some other modifiers which require the use of 'the', such as 'same'): My wife and I share *the same interests*.

Applying world knowledge for inference

In the previous subsection we saw that all definite descriptions can be subdivided into four familiarity classes using the concepts hearer status and discourse status. We saw that a definite description can be familiar either because it refers to some known entity, or because it refers to an entity which is related to a known entity. With coreferential use the description directly refers to a discourse-old entity and with situational use the description directly refers to a hearer-old, but discourse-new entity. Furthermore with bridging the description refers to an entity that is implicitly related to a known entity, such as 'the ceiling' when a room has just been mentioned. Finally with unfamiliar use the description also refers to an entity that is related to a known entity, but in this case the relation is expressed explicitly, such as 'the man John met yesterday'.

The indirect coreferential uses and bridging uses differ from the other types of referring expressions, because processing these requires reasoning based on world knowledge and the discourse context. With indirect coreferential uses the reader must be able to infer which hearer-old entity is being referred to using a new description. With bridging uses the reader should infer the implicit relation between the referent of the definite description and some other discourse-old or hearer-old entity.

In order to *generate* these kinds of referring expressions we also need to apply world knowledge. Poesio and Vieira [PV98] set up an experiment to assess the number of occurrences of all types of referring expressions. They found that almost 24% of the definite descriptions require inference: 8.5% are bridging descriptions and about 15% are indirect coreferences. Therefore adding such expressions to the Referring Expression Generator used in the Virtual Storyteller is definitely a useful extension.

The algorithm for generating referring expressions described in the previous subsection makes use of the following modules to generate referring expressions: the discourse history, the fabula structure, the Character Information and the Story World. In order to generate indirect coreferential descriptions and bridging descriptions we need an additional knowledge base containing rules the system can reason with. Some of these rules are already present in the Story World module, but for clarity I will call this module the Inference Rules module. The module will contain rules such as:

- A princess is a girl:
  $\forall\ x\ .\ Princess(x)\ \rightarrow\ Girl(x)$
- Castles have gates:
  $\forall\ x\ .\ Castle(x)\ \rightarrow\ \exists\ y\ .\ Gate(y)\ \wedge\ Has(x,y)$
- A book has more than one page:
  $\forall\ x\ .\ Book(x)\ \rightarrow\ \exists\ y\ z\ .\ Page(y)\ \wedge\ Page(z)\ \wedge\ Has(x,y)\ \wedge\ \ Has(x,z)\ \wedge\ y \neq z$
- The sun is a unique entity:
  $\forall\ x\ .\ Sun(x)\ \rightarrow\ \neg\ \exists\ y\ .\ Sun(y)\ \wedge\ x \neq y$

The first one of these rules is needed for the generation of indirect coreferential descriptions (for the generation of hypernyms), the second and third rules are needed for the generation of bridging descriptions and the final rule can be used to determine whether an entity is unique. The rules have been stored in a separate file in KIF-format [GF92] and the first two of these rules are represented in KIF-format as follows:

- A princess is a girl:

```
(forall (?x) (=> (type princess ?x) (type girl ?x)))
```

- Castles have gates:

```
(forall (?x) (=> (type castle ?x) (exists ?y (and (type gate ?y) (has ?x ?y)))))
```

For the application of these rules we use the Java Theorem Prover (see [FJF04] and [FJF03]). Using JTP a reasoning system is created which can then be queried to find out if any rules have been stored, in the exact same way as the fabula can be queried (see section 6.2.1).

JTP is especially useful for this type of reasoning because it automatically applies all available rules. Take for example a reasoning system containing a rule that a princess is a girl and a rule that a girl is a person. If a particular princess is then added to the reasoning system (for example by telling the fact (*princess* *princess*01)), the system automatically also stores the facts (*girl* *princess*01) and (*person* *princess*01). If the reasoning system is finally queried to get all entities of type *person* it will also return the entity *princess*01.

Generating coreferential uses

The algorithm for generating referring expressions described in the previous subsection is only able to generate expressions of the direct coreferential type, but when applying world knowledge the algorithm can be extended to generate some new kinds of referring expressions as well.

Hypernymy relations can be generated using a subsumption hierarchy. This hierarchy contains rules such as 'a princess is also a girl' (stored as $\forall\ x\ .\ Princess(x)\ \rightarrow\ Girl(x)$) and 'a girl is also a person' ($\forall\ x\ .\ Girl(x)\ \rightarrow\ Person(x)$). As explained before the Inference Rules module contains such rules and these can be used to choose a more general concept for the entity. Doing this we assume that each entity belongs to a particular concept and that this concept is the most specific concept and simultaneously the most basic level concept. This seems reasonable because fairy-tales usually contain characters such as princesses and knights. We will thus normally use this most specific concept, but in order to add some variety to the

chosen nouns we will use a more general concept once in a while (obviously only when no confusion can arise). In order to determine whether a more general concept can be used the algorithm checks if there is another entity in the distracter set which is also an instance of the more general concept. If there is no such entity, the algorithm can choose between the most specific concept and the more general one. Consider the situation in which there are a knight and a princess in the distracter set and a noun is to be selected for the princess. In this case the algorithm can either use the concept 'princess' or 'girl', but not 'person' because a knight is also a kind of person.

In order to generate indirect coreferences in which the nouns are related by *hyponymy*, we actually need additional information. We could introduce an entity using a somewhat more general concept than available and then use the most specific concept later. This can for example result in a sentence such as "The knight hit a girl. The princess cried." Using such constructions will indeed lead to more variety, but it may also lead to confusion. Therefore we will not focus on this type of referring expressions.

Synonymy can easily be generated by storing several entries for the same concept in the lexicon, as described earlier in this section. It is best to store a preferred entry and possibly another entry which will only be used occasionally. An example is the concept 'king' with the preferred entry for the Dutch word 'koning' and an entry for the Dutch word 'vorst' which will only be used when the word 'koning' has been used for a number of times in a row.

Generating bridging uses

Entities described in bridging descriptions are related to an already mentioned entity by a bridging relation. Earlier in this section we distinguished three kinds of bridging relations: set-membership, part-of and roles relations. Clark [Cla75] states that the part-of relation is the most common kind of bridging. Currently, the Narrator does not support sets yet and we would need additional information in order to generate roles bridging relations, so we will only focus on the part-of relation. These part-of relations are stored in the Inference Rules module and include rules such as 'Castles have gates' (stored as $\forall\, x\, .\, Castle(x) \;\rightarrow\; \exists\, y\, .\, Gate(y) \;\wedge\; Has(x, y)$).

In order to generate bridging descriptions the standard algorithm for generating referring expressions is executed. If a referring expression is to be generated for an entity it is first checked if that entity is related to another entity. If this is the case *and* the other entity has already been mentioned, the function ChooseKindOfExpression is called (see algorithm 6). This function checks if a bridging description can be used or that the expression should include the related entity as well. The algorithm returns the *kind* of expression that should be generated: DEF/INDEF tells whether to use a definite or indefinite determiner and VAN/NOVAN tells whether to include the related entity as well. The examples in the outline of the algorithm are explained in more detail in the following section. Note that only the expressions of type DEF-NOVAN and INDEF-NOVAN are bridging descriptions, so this algorithm checks *whether* bridging can be applied.

The arguments of the function are the entity for which a description is to be generated ($e1$) and the entity which it is related to ($e2$). Take for example as $e1$ a gate and as $e2$ a castle that has been mentioned before. The algorithm then checks if the Inference Rules contains a rule that specifies that an entity of the type of $e2$ usually has an entity of the type of $e1$, so in our example it checks if there is a rule that specifies that castles have gates. Then it checks if there is another salient entity that can have an $e1$, so it checks if there is another entity that can have a gate (note that this can be another castle, but also an entity of a completely different concept). Finally it queries the Inference Rules module to find out if the entity $e2$ has exactly one $e1$ in which case a definite article can be used; if this is not the case an indefinite article will be used. Depending on the rules stored in the Inference Rules module the algorithm returns one of the values INDEF-VAN, INDEF-NOVAN, DEF-VAN or DEF-NOVAN as can be seen in the outline of the algorithm.

Generating situational uses

The immediate situational uses usually occur in utterances, for example in conversations. Since the characters in the Virtual Storyteller are not able to talk yet, it is not necessary to implement this kind of definite descriptions (at least not for the moment). The larger situational uses were subdivided into two subclasses:

Algorithm 6 ChooseKindOfExpression($e1$, $e2$)

```
if a rule has been specified that an e2 has an e1 then
  if e2 is the only salient entity that can have an e1 then
    if e2 has exactly one e1 then
      return DEF-NOVAN          // e.g. de poort
    else
      return INDEF-NOVAN        // e.g. een pagina
    end if
  else
    if e2 has exactly one e1 then
      return DEF-VAN            // e.g. de slaapkamer van de prinses
    else
      return INDEF-VAN          // e.g. een been van de prinses
    end if
  end if
else
  return INDEF-VAN              // e.g. een poort van de dierentuin
end if
```

the descriptions based on shared knowledge and unique entities. The first of these classes contains references in which an entity has not been mentioned before, but in which the entity is part of shared knowledge. Shared knowledge in the fairy-tale domain may contain characters which appear very often in fairy-tales, such as princesses, kings and villains. In stories it is therefore common to refer to a king as 'the king' if there is only one king in the story. Such definite descriptions can be generated by simply querying the Story World to find out if there is only one such entity.

The generation of references to unique entities is very similar to the generation of situational uses based on shared knowledge; we only have to use a definite article instead of an indefinite article, even if the entity has not been used before. To achieve this the algorithm queries the Inference Rules module to find out if there is only one entity of a particular type.

Unfamiliar uses

The final category contained five subclasses. It seems that none of them is useful for the stories generated by the Virtual Storyteller, so for the moment we will ignore all of them.

8.3.4 Results of the referring expression generation

The Referring Expression Generator can generate direct coreferential uses, indirect coreferential uses, bridging descriptions and larger situational uses. This subsection gives some examples of the referring expressions that can be generated.

1. Indirect coreferential use

   - Hypernymy:
     "Er was eens een prinses, die buitengewoon mooi was. *Het meisje* woonde in een groot kasteel."
     ("Once upon a time there was a princess, who was extremely beautiful. *The girl* lived in a large castle.")
     In this example the princess has just been mentioned. Furthermore there are no other salient entities which are an instance of the concept 'girl', so this more general concept can be used.
   - Synonymy:
     "Er was eens een koning, die heel rijk was. *De vorst* woonde in een groot kasteel."
     ("Once upon a time there was a king, who was very rich. *The ruler* lived in a large castle.")

In this example the word 'koning' has been used to lexicalize the concept 'king' in the first sentence. In the second sentence a pronoun could be used, but the referring expression generator may also choose to use a synonym, such as 'vorst'.

2. Bridging descriptions

   - DEF-NOVAN:
     "Op een nacht ging de ridder naar het kasteel. Hij probeerde *de poort* te openen."
     ("One night the knight went to the castle. He tried to open *the gate*.")
     In this example the castle has just been introduced and is therefore very salient. The Inference Rules module contains a rule that specifies that castles have gates and a rule that specifies that castles usually have only one gate. Therefore the 'van'-part can be dropped and the expression uses a definite article.
   - INDEF-NOVAN:
     "John had een boek gekocht. *Een pagina* ontbrak."
     ("John had bought a book. *A page* was missing.")
     In this example there is a rule that specifies that books have pages and there is only one salient entity that can have pages, so the 'van'-part can be dropped again. However, books usually have more than one page, so this time an indefinite article is used.
   - DEF-VAN:
     "De ridder klom in een boom en sprong vervolgens *de slaapkamer van de prinses* binnen."
     ("The knight climbed in a tree and jumped then into *the bedroom of the princess*.")
     In the Inference Rules has been stored that people have bedrooms. Since the knight (who is also a person) is salient, the 'van'-part has to be included and no bridging description can be used. Furthermore a definite article is used, because the princess has only one bedroom.
   - INDEF-VAN:
     "De ridder hakte *een been van de prinses* af."
     ("The knight chopped off *a leg of the princess*.")
     In this example no bridging description can be used either, because the knight is another salient entity that can have legs, so the 'van'-part has to be included again. Furthermore the princess has two legs, so an indefinite article is used.
   - INDEF-VAN
     "Buiten de stad ligt een dierentuin. *Een poort van de dierentuin* was vernield."
     ("Outside of the city is a zoo. *A gate of the zoo* had been destroyed.")
     In this example the entity 'poort' ('gate') is related to the entity 'dierentuin' ('zoo'), but no rule has been specified that this is a common relation, so the 'van'-part has to be included and an indefinite article is used.

3. Larger situational uses:

   - Shared knowledge:
     - "Er was eens een prinses, die heel verwaand was. *De koning* vond dit vervelend."
       ("Once upon a time there was a princess, who was very conceited. *The king* thought this was annoying.")
       A king is not a unique entity, but in the Story World has been stored that there is only one king *in this particular story*. Therefore a definite article is chosen, even though the king has not been mentioned before.
   - Unique entities:
     - "*De zon* schijnt fel."
       ("*The sun* is shining brightly.")
       In the Inference Rules module has been stored that the sun is a unique entity, so a definite article is chosen, even though the sun has not been mentioned before.

8.4 Surface Form Generator

The final component in the Surface Realizer is the Surface Form Generator and this component has to transform the final rhetorical dependency graph into text in natural language. At this point as many dependency trees as possible have already been combined and the referring expressions have already been generated. In order to transform the graph into text, the Surface Form Generator traverses the dependency trees once more and decides for each tree the order in which the nodes should appear in the sentence. Doing this the module also applies morphology (word inflection) and orthography (adding punctuation and capitals).

8.4.1 Ordering the nodes in a dependency tree

The nodes in the dependency trees are ordered using a grammar which specifies the word order for each phrase and the order of the phrases in a sentence. The grammar had already been implemented, but as will be described in section 8.5 I added a class for the relative clauses category. The Smain library class is the class that specifies in which order the constituents should appear in a main clause. In order to create more varied sentences I changed this class by making sure that (one of) the modifier(s) is placed at the beginning of the sentence, rather than starting with the subject in every single sentence.

The dependency trees used by Alpino [vdBBD+02] and CGN [HMR+03] use the label 'mod' for everything that modifies something else. This can be an adjective, an adverb, a time expression and even entire subordinate clauses. All of these phrases get the same label 'mod', so the grammar cannot distinguish between these different kinds of modifiers. This means that a noun phrase which consists of a determiner, a noun and a modifier always places the words in the following order: determiner - modifier - noun. If the modifier is an adjective this is the correct order (such as 'de mooie prinses', 'the beautiful princess'). On the other hand, if the modifier is a complete prepositional phrase (such as 'van een ver land', 'of a faraway country') this results in the phrase 'de van een ver land prinses' ('the of a faraway country princess') which is obviously incorrect. In order to solve this problem I added a second modifier label which makes sure the modifier is placed at the end of the phrase.

The 'modb' label can also be used to make sure the subordinate clause is placed at the end of the sentence. Furthermore a 'moda' label has been added to make sure that the corresponding constituent is placed at the beginning of a sentence. This label is used when the sentence should start with the subordinate clause, for example when generating a sentence containing the cue word 'nadat' ('after'). In the original Surface Realizer several sentences could be embedded such as in the following example, taken from the example story in figure 3.4: "Brutus sloeg Amalia, want, doordat Amalia het zwaard op pakte, werd hij bang." ("Brutus hit Amalia, because, when Amalia picked up the sword, he got scared."). Using the new modifier labels this sentence is changed into the less complex text: "Brutus sloeg Amalia, want hij werd bang, doordat Amalia het zwaard op pakte." ("Brutus hit Amalia, because he got scared, when Amalia picked up the sword.").

8.4.2 Morphology

Apart from the Library classes that can be used to order the nodes in a dependency tree, the Library contains classes to inflect nouns, verbs and determiners. However, the inflection of adjectives had been ignored, so I added a class in order to inflect adjectives. The algorithm makes sure that the following kinds of adjectives are inflected correctly:

- If the last-but-one and last-but-two characters of the root are the same vowel and the last character is not a vowel, then remove one occurrence of the vowel: groot - grote, mooi - mooie
- If the last character is not a vowel and the last-but-one character is a vowel (but not the combination 'ig'), then add another occurrence of the last character: snel - snelle, vlug - vlugge, aardig - aardige
- If the last character is an 's' and both of the characters before are vowels (or the combination 'i' and 'j'), then replace the 's' with a 'z': boos - boze, grijs - grijze

- If the last character is an ‘f’ and both of the characters before are vowels (or the combination ‘i’ and ‘j’), then replace the ‘f’ with a ‘v’: lief - lieve, stijf - stijve
- If the root ends with ‘en’ and the character before is not a vowel, then leave the root unchanged: verlegen - verlegen, gemeen - gemene

The algorithm to inflect an adjective is called in all cases, except the case in which the determiner ‘een’ is used while the original determiner is ‘het’, as illustrated in table 8.2.

	originally ‘de’	originally ‘het’
definite	de grote jongen	het grote meisje
indefinite	een grote jongen	een groot meisje

Table 8.2: Examples of inflection of adjectives

8.4.3 Orthography

While converting the nodes into sentences orthography is applied; punctuation is added and it is made sure that each sentence starts with a capitalized letter. As will be described in section 8.5 I extended the module to include commas with relative clauses. Furthermore the module has been extended to include commas with appositions as well and to include an exclamation mark with the two special ways to describe internal states, such as the internal state “Wat was ze blij!”.

8.5 Relative Clauses

Analysis of human written fairy-tales (see chapter 4) shows that relative clauses are used extensively. The original Surface Realizer was not able to generate relative clauses, so I extended the module to support relative clauses as well. This affected several components in the Surface Realizer which is described in the following sections, but let’s first take a closer look at the different types of relative clauses.

8.5.1 Types of relative clauses

There are many different types of relative clauses. First of all one can distinguish between non-limiting relative clauses and limiting clauses. A limiting relative clause has the function of identifying the object being referred to; a non-limiting relative clause does not have such a function, but is merely used to provide additional information. Furthermore a non-limiting relative clause is placed between commas and a limiting relative clause is not placed between commas. An example of a limiting relative clause is the sentence “De vrouw die bij het raam staat, is mijn moeder.” (“The woman who is standing near the window, is my mother.”) in which the relative clause “die bij het raam staat” (“who is standing near the window”) identifies which woman the speaker is talking about. An example of a non-limiting relative clause is “Mijn vrouw, die 24 jaar is, is dokter.” (“My wife, who is 24 years old, is a doctor.”). In this example the speaker has only one wife, so the relative clause just provides some additional information about the woman.

The difference between these two kinds of relative clauses is very important for the implementation of relative clauses. Non-limiting relative clauses simply combine two sentences which can also be told separately. An example is the sentence “De ridder ging naar de prinses, die in een kasteel woonde.” (“The knight went to the princess, who lived in a castle.”) which could also be told using two separate sentences as in “De ridder ging naar de prinses. Zij woonde in een kasteel.” (“The knight went to the princess. She lived in a castle.”). Adding limiting relative clauses on the other hand changes the meaning of the sentence and therefore the sentence cannot be told separately. An example is “De prinses lachte iedereen uit die lelijk was.” (“The princess laughed at everyone who was ugly.”). This sentence is obviously correct, but it cannot be derived from the two separate sentences “De prinses lachte iedereen uit.” and “Iedereen was lelijk.” (“The princess laughed at everyone.” and “Everyone was ugly.”). Since my goal is to combine simple sentences such as the

ones above, I will only focus on generating non-limiting relative clauses which provide additional information without having a second function.

When only focussing on non-limiting relative clauses, there are still many different kinds of relative clauses as illustrated by the following examples:

- Clauses in which the relative pronoun is the subject:
  - A clause related to the subject of the main clause:
    De prins, die mooi was, sloeg de prinses.
    (The prince, who was beautiful, hit the princess.)
  - A clause related to the object of the main clause:
    De prins sloeg de prinses, die mooi was.
    (The prince hit the princess, who was beautiful.)
- Clauses in which the relative pronoun is the object of the relative clause:
  - A clause related to the subject of the main clause:
    De foto, die de prinses ontving, viel op de grond.
    (The picture, that the princess received, fell on the floor.)
  - A clause related to the object of the main clause:
    De prins sloeg de prinses, die hij niet leuk vond.
    (The prince hit the princess, who he disliked.)
- A clause in which the relative pronoun is accompanied by a preposition:
  De prins, op wie de prinses verliefd was, was mooi.
  (The princess, who the princess was in love with, was beautiful.)
- A clause using a place expression as relative pronoun:
  Het land, waar de prinses geboren was, is onbekend.
  (The country, where the princess was born, is unknown.)

Additionally, the first three of these examples can also use the relative pronoun 'dat' if the noun is a neutral word (e.g. "De ridder sloeg het meisje, *dat* mooi was.").

8.5.2 Algorithm for generating relative clauses

Relative clauses differ from the already implemented rhetorical relations in the sense that the other relations specify how two *sentences* are related. Relative clauses on the other hand specify something about an *entity* used in the main clause, and therefore the second sentence should be added to a node representing a noun phrase instead of to the top node of the sentence. Though this is quite a difference, I chose to implement the relative clauses in the same way as the other rhetorical relations, because they do combine two separate sentences which could also be told separately (on the condition that I will only focus on non-limiting relative clauses).

The algorithm for combining two dependency trees into one using a relative clause is shown in algorithm 7. The algorithm consists of three steps: deciding which nodes are the same, deciding which relative pronoun is to be used (possibly combined with a preposition) and finally copying the relative clause to the correct node of the main clause. Note that at this point the referring expressions have not been generated yet, so the algorithm looks for *entity* nodes which are the same.

The first step of the algorithm traverses both dependency trees and tries to find an entity node which appears in both trees. Once it finds such a node the variables *mainid* and *subid* are set to the ids of the corresponding nodes.

In the second step the correct relative pronoun is selected. First the algorithm checks if the entity appeared in a pp-node in the clause that is being turned into a relative clause. If this is the case the relative pronoun should contain the preposition. If the noun is a person, a pp-node is created with the preposition set

Algorithm 7 GenerateRelativeClause(*mainclause*, *subclause*)

```
relpron ← null

//step 1: Determine which nodes are the same
for each ent1 in mainclause do
  for each ent2 insubclause do
    if ent1 = ent2 then
      mainid ← GetId(ent1)
      subid ← GetId(ent2)
    end if
  end for
end for

//step 2: Determine which relative pronoun should be used
if subid appears in pp-node then
  prep ← GetPrep(pp-node)
  if ent1 is male or female then
    relpron ← CreatePpNode(prep, 'wie')
  else if ent1 is neutral then
    relpron ← CreateNode('waar+prep')
  else if ent1 is place then
    relpron ← CreateNode('waar')
  end if
else
  if ent1 is 'het'-word then
    relpron ← CreateNode('dat')
  else
    relpron ← CreateNode('die')
  end if
end if

//step 3: Copy the relative clause to the main clause
entnode ← CreateNode()
relnode ← CreateNode()
insert entnode between topnode and mainnode
add relpron to relnode
for each node in subclause do
  if node is not related to subid-node then
    copy node to relnode
  end if
end for
```

to the preposition used in the original clause and a node for the relative pronoun 'wie' (such as "De prins, *op wie* de prinses verliefd was, was mooi." ("The prince, who the princess was in love with, was beautiful.")). If the noun is not a person, the preposition is combined with the word 'waar' into one word which is the relative pronoun (for example "Het paard, *waarop* de ridder zat, ging naar het bos" ("The horse, which the knight was sitting on, went to the forest.")). Finally if the noun represents a place, the relative pronoun is just 'waar' and the preposition is left out (such as "Het land, *waar* de prinses geboren is, is klein" ("The country, where the princess was born, is small.")). If the entity does not appear in a pp-node, the relative pronoun is either 'die' or 'dat' and this decision is based on the article of the noun.

The third step of the algorithm copies the clause that is turned into a relative clause into the main clause by first inserting a new entity node between the top node and the original entity node. Then a rel-node for the relative clause is created and added to the new entity node. Next the relative pronoun node (created in the second step of the algorithm) is copied into the rel-node as well. Finally the algorithm traverses the nodes in the original clause and copies those nodes which are not related to the entity, since the other nodes have already been turned into a relative pronoun.

8.5.3 Example execution of the algorithm

An example of execution of this algorithm is shown in figure 8.4. The two input trees represent the sentences "De prins is mooi." ("The prince is beautiful.") and "De prinses is verliefd op de prins." ("The princess is in love with the prince."). The first clause is the main clause and the second one is the relative clause, so the clauses should be combined into "De prins, op wie de prinses verliefd is, is mooi." ("The prince, who the princess is in love with, is beautiful."). The first step of the algorithm traverses the entity nodes of the main clause. The first entity it encounters is the node 3 with entity 'prince'. For this node it traverses the entity nodes of the relative clause and checks if there is a node with the same entity. First it encounters the princess node (node 7) which represents a different entity, but then the algorithm moves on to the prince node (node 10) and finds out that these nodes refer to the same entity. *Mainid* is therefore set to 3 and *subid* to 10. The subid node appears in a pp-node and the entity represents a male character, so the relative pronoun is 'wie' combined with the preposition 'op'. The final step first inserts a new ent-node between the topnode (node 1) and the mainid node (node 3) and adds a new rel-node to this node. Then it creates a new pp-node with a prep-node 'op' and a node 'wie' as its children and adds this node to the rel-node. Finally the algorithm copies the nodes 6 and 7 to the rel-node as well. This means that nodes 8, 9 and 10 are ignored, because these have already been used to create the correct relative pronoun.

8.5.4 Adding a library class

In order to make sure the words in the relative clauses appear in the correct order, a new library class has been added. For other subordinate clauses the class SsubLib can be used, but that library makes sure that the subordinate clause always begins with the subject and then the other phrases. A relative clause, on the other hand, should always start with the relative pronoun (possibly preceded by a preposition), no matter if this is the subject, the object or even something else. In order to achieve this a new grammatical category 'Rel' is added to the grammar and this category contains rules which make sure the relative pronoun is the first constituent of the sentence.

8.5.5 Adding punctuation

Finally the relative clause should be placed between commas. This is done in the Surface Realizer when generating the surface form. Doing this the dependency tree is processed from left to right, so a comma can easily be added right before the relative pronoun node is processed. The comma at the end of the relative clause can be added by checking if the node being processed is the final node of the relative clause.

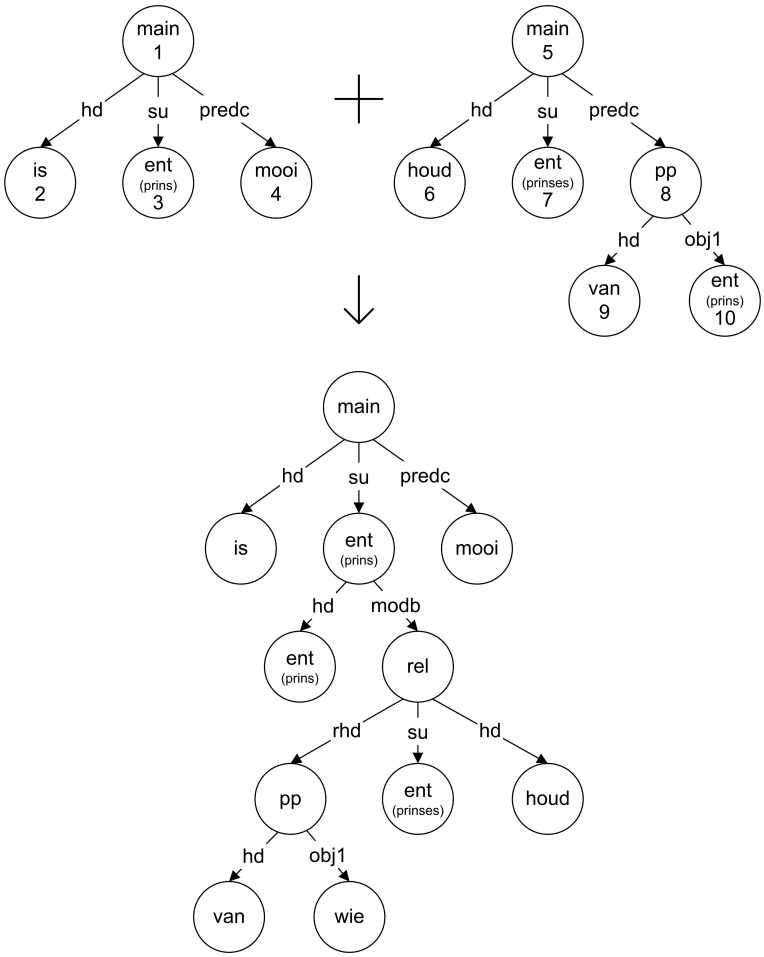

Figure 8.4: Example of generating a relative clause

8.6 Example

In order to illustrate the components of the Surface Realizer we use the same example used throughout the previous chapters. The input to the Surface Realizer is the same as the output of the Microplanner, as shown in figure 7.4.

8.6.1 Syntactic Aggregator

The syntactic aggregator is responsible for performing syntactic aggregation. The component first tries to combine as many dependency trees into one as possible, with a maximum of three, and if possible it performs ellipsis as well. In our example the first two dependency trees can be combined; they are related by an elaboration relation, so the second clause is turned into a relative clause with the relative pronoun 'die'. Furthermore the resulting dependency tree has a 'temp-once' rhetorical relation node as its parent, so the cue word 'eens' ('once') is also added to the tree. Next, the third and fourth clause are combined; they are related by an additive relation, so the resulting dependency tree simply consists of two main clauses connected by the cue word 'en' ('and'). Since this dependency tree consists of two main clauses, ellipsis can be applied in which the subject in the second clause is removed. Finally the final two dependency trees are related by a temporal relation, so these trees are turned into a single tree consisting of a subordinate clause containing the cue word 'nadat' ('after') and a main clause.

Next, the Relation Transformer checks if any rhetorical relations between consecutive sentences can be expressed explicitly. In this case the only adjunct that can be added is the cue word 'daarom' ('therefore') to the dependency tree describing the goal of eating the apple.

Finally, the module can add a cue word to the very first dependency tree of the rhetorical dependency graph in order to show the relation to the previous paragraph (if the story consists of several paragraphs). In this example the entire story is represented as a single paragraph, so no cue word will be added to the first dependency tree in the graph.

Figure 8.5 shows the dependency tree for the first sentence after combining the first two dependency trees and adding the cue word 'eens' ('once') to the tree.

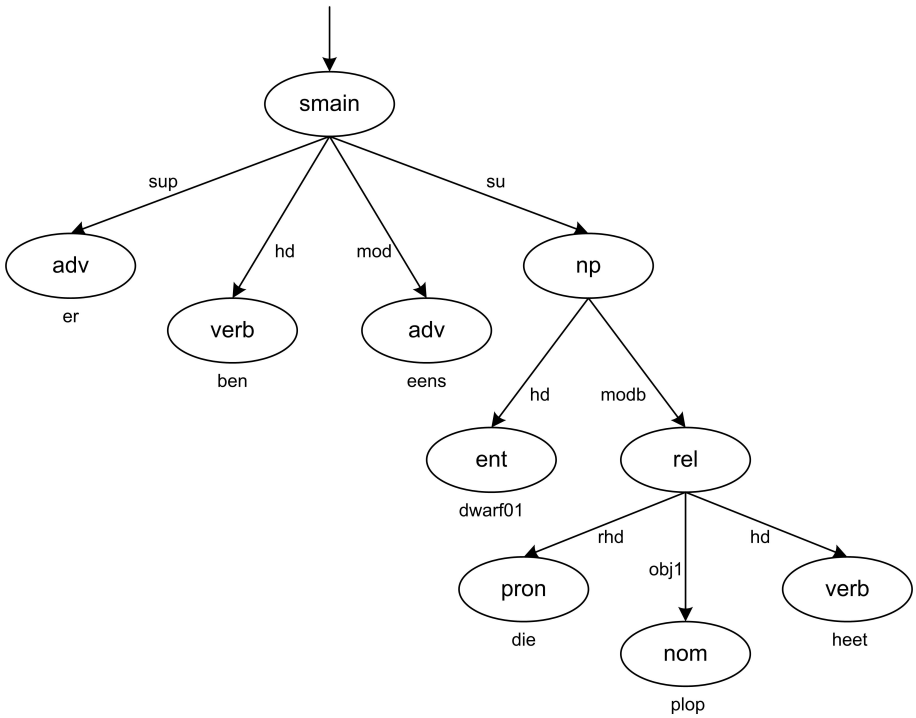

Figure 8.5: Output of the Syntactic Aggregator for the Plop story

8.6.2 Referring Expression Generator

The Referring Expression Generator lexicalizes the entity nodes, so it replaces the entity node by either a pronoun node or a more complicated noun phrase node. If a noun phrase is generated it can also add adjectives, but in this example no adjectives have been added. However, chapter 9 gives an example in which adjectives are added.

In the first sentence of the example the only entity node is the node labelled 'dwarf01'. This is the first reference to the dwarf, so an indefinite article is selected, and the entity node is replaced by a noun phrase node for 'een kabouter' ('a dwarf'). The result is shown in figure 8.6.

Appendix B.3 shows the dependency trees after generating referring expressions for all sentences in the story. The second dependency tree contains another reference to the dwarf and since he is the only entity mentioned sofar, he can be referred to by a pronoun. Furthermore the tree contains a reference to an apple, which has not been mentioned before, so the entity node is replaced by a node representing 'een appel' ('an apple'). In the third sentence the dwarf can be referred to by a pronoun too and the referring expression for the apple selects a definite article. The referring expression for the dwarf in the final sentence uses a noun phrase, because the entity has been referred to by pronouns for the last two sentences. The algorithm

chooses to generate an expression containing the name and adds the noun as well. Finally, the second clause in the final sentence appears in strong parallelism with the first clause (in both sentences the subject is the dwarf and the object is the apple), so both entities can be referred to by a pronoun.

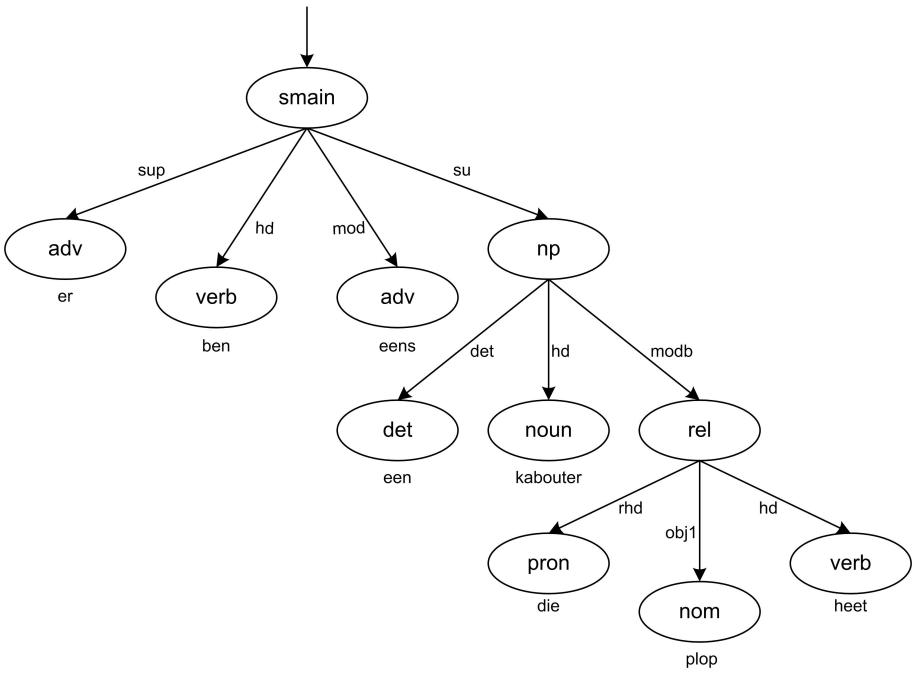

Figure 8.6: Output of the Referring Expression Generator for the Plop story

8.6.3 Surface Form Generator

The final component of the Surface Realizer has to convert the dependency trees in the rhetorical dependency graph into sentences in natural language. The result for the entire rhetorical dependency graph is shown in figure 8.7.

Plop example
Er was eens een kabouter, die Plop heette.
Hij had honger en dacht dat er een appel in een huis lag.
Daarom wilde hij de appel eten.
Nadat kabouter Plop de appel op had gepakt, at hij hem.

Figure 8.7: Output of the Surface Form Generator for the Plop story

Chapter 9

Results and Evaluation

The example used throughout the previous chapters was extremely simple, so in this chapter I will give a number of somewhat more detailed examples. In section 9.1 I will give an example of a story that can be generated based on a modelled fabula structure and I will show how the results can be improved with only a few changes. In section 9.2 I will give the results for the example story given in chapter 3. Finally I will give a small example in order to illustrate the functioning of the Mood Creator and the State Transformer in section 9.3.

9.1 Example 1: Knight Story

In this section I will give an example of a story about a knight who wants to capture a princess. First I will give the output when a modelled fabula structure is used. This result shows that a somewhat larger fabula can also be translated into text correctly, but it also shows that a number of things should be improved in future versions, such as the task of assigning more varied rhetorical relations. Then I will give the output of the same story, but using modelled document plans instead of a modelled fabula structure. This example shows that with a few changes the generated stories become much better.

9.1.1 Using a modelled fabula structure

For the first example I will use a modelled fabula structure. This means that the structure is represented in the exact same format as the actual output of the Plot agent, but it has been created manually. Furthermore some plot elements have been removed, such as the perception and the belief that an action has succeeded after carrying out the action. The modelled fabula structure used for this example is given in figure 9.1.

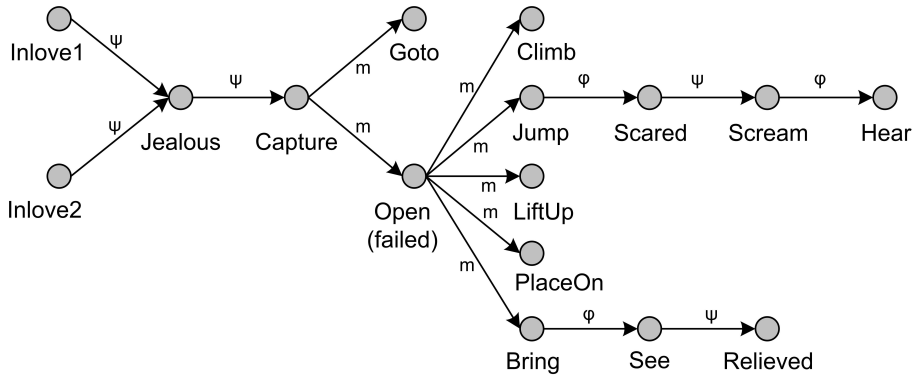

Figure 9.1: Modelled fabula structure for the Knight story

The plot elements in the figure can be mapped onto the following sentences:

*Inlove*1	A knight was in love with a princess.
*Inlove*2	The princess was in love with a prince.
Jealous	The knight was jealous.
Capture	The knight wanted to capture the princess.
Goto	The knight went to a castle.
Open	The knight tried to open a gate.
Climb	The knight climbed in a tree.
Jump	The knight jumped into the princess's bedroom.
Scared	The princess was scared.
Scream	The princess screamed.
Hear	Nobody heard the princess.
LiftUp	The knight lifted up the princess.
PlaceOn	The knight placed the princess on his horse.
Bring	The knight brought her to a bridge.
See	The princess saw the prince.
Relieved	The princess was relieved.

First of all, I will give the results without performing syntactic aggregation, generating proper referring expressions and adding any details. The results are then comparable to the result of the first version of the Virtual Storyteller that used templates for the natural language generation. The result is given in figure 9.2, and like the example story in section 3.3.1 the resulting text only contains sentences that start with the subject and include no details.

> Er was een prinses. Een ridder was verliefd op de prinses. De prinses was verliefd op een prins. De ridder was jaloers. De ridder wilde de prinses ontvoeren. De ridder ging naar een kasteel. De ridder probeerde een poort van het kasteel te openen. De ridder klom in een boom. De ridder sprong een slaapkamer van de prinses binnen. De prinses was geschrokken. De prinses schreeuwde. Niemand hoorde de prinses. De ridder pakte de prinses op. De ridder zette de prinses op een paard van de ridder. De ridder bracht de prinses naar een brug. De prinses zag de prins. De prinses was opgelucht.

Figure 9.2: Example output without performing aggregation and referring expression generation

If we do apply syntactic aggregation, generate better referring expressions and add some more detail (such as adjectives and background information) in the same way as described in the previous chapters, the Narrator generates the output shown in figure 9.3.

> Er was eens een buitengewoon mooie prinses, die Amalia heette. Een ridder van een ver land was verliefd op haar en zij was verliefd op een jonge prins. De ridder was jaloers, dus hij wilde haar ontvoeren. Omdat de prinses in een groot kasteel woonde, ging hij naar het kasteel. Hij probeerde de zware poort te openen. De ridder klom in hoge een boom. Hij sprong haar slaapkamer binnen, zodat de prinses geschrokken was. Doordat zij schreeuwde, hoorde niemand haar. De ridder pakte haar op. Hij zette het meisje op zijn paard. Hij bracht haar naar een oude en smalle brug, dus zij zag de prins, op wie zij verliefd was. Daarom was de prinses opgelucht.

Figure 9.3: Example output after performing aggregation and referring expression generation

You can see that the resulting output text is much better. First of all, background information has been added such as the name of the princess and the fact that she lives in a castle. Furthermore different referring expressions are used and a number of rhetorical relations are expressed explicitly.

There are however a number of things to note. First of all, it would be better if some different rhetorical relations were chosen. An example is the relation between the facts that the knight loves the princess and the fact that she loves someone else; in this example they have simply been connected by an additive relation because the facts together cause the internal state of being jealous, but it would be better if the facts were related by a contrast relation.

Additionally, the fact that the princess screamed does *cause* the event that nobody heard her (i.e. the screaming enables the event), but it seems rather strange to lexicalize this by a causal relation. It would be much better if a contrast relation would have been used, such as in "De prinses schreeuwde, *maar* niemand hoorde haar." ("The princess screamed, *but* nobody heard her."). The reason why this sentence sounds strange is caused by the fact that the actual fabula structure should also include the reason that no one heard her (for example because the walls were too thick). In that case the plot element describing the action of screaming and the plot element describing the reason for the non-perception would then together cause the event that nobody heard her. In this example, however, I have left out the reason, so in the fabula the event is directly caused by the action.

Furthermore it would be better if the reason would have been given why the action of opening the gate failed. Since the Plot agent knows why a certain action fails, the agent should include the reason in the fabula structure. A possible solution is always adding the sentence "maar het lukte niet" ("but (s)he failed") for a failed action, but it is better if the reason is available in the fabula structure.

The results would also be better if the location of the prince was available. The reader will assume that the prince is near the bridge (since the action of bringing the princess to the bridge leads to the perception of seeing the prince), but it would be better if this was told explicitly.

Finally it seems that simply selecting a cue word for a particular cause relation does not always result in the selection of the *best* cue word. Take for example the action of bringing the princess to the bridge that physically causes the perception of seeing the prince. In this example the cue word 'dus' has been chosen, but it would be better to use the cue word 'waardoor'. Since the Surface Realizer only distinguishes between a non-volitional and a volitional cause relation, both cue words can be used, and one of them is selected randomly. A possible solution to this problem is supporting four different types of causal relations (the same ones as used in the fabula structure). *If* the relations can always be mapped onto the same cue words (for example that physically cause relations always use the cue word 'waardoor' and that enable relations always use the cue word 'dus'), this is a rather easy extension; the two lists of possible cue words simply have to be split into four lists of possible cue words. However, the fact which cue word is best in which cases may also be more complicated, for example because it may depend on the context. Therefore this needs to be analyzed in more detail.

9.1.2 Using modelled document plans

In order to show that small changes such as the ones mentioned above result in better texts, I give the results of the same story once more, but in this case not the fabula structure has been modelled, but the initial document plan. One of the tasks of the Document Planner is to add paragraph boundaries, but since this has not been implemented yet (see section 6.2.3), this is done manually. Furthermore the reason for failure of the action of opening the gate has been added, and some details for the plot elements are added to the details Vectors of the elements. The initial document plans for the paragraphs are shown in figure 9.4 and the output text that has been generated using this input is given in figure 9.5. Note that cause-v means a volitional cause relation, cause-n a non-volitional cause relation and cause-st a cause relation that is expressed using the 'zo ..., dat' construction.

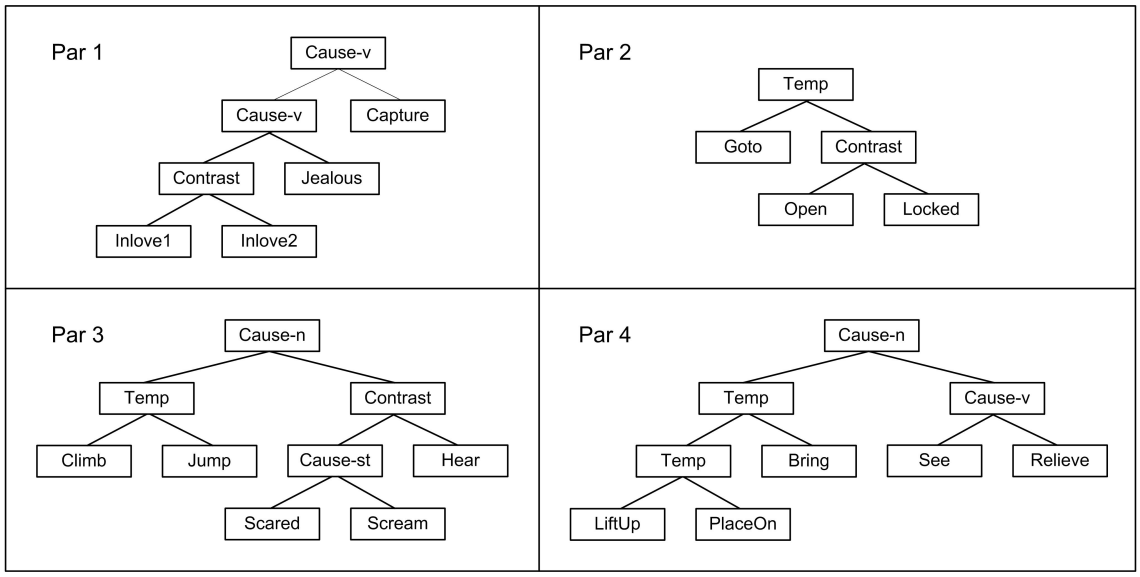

Figure 9.4: Modelled initial document plans for the Knight story

(1) Er was eens een buitengewoon mooie prinses, die Amalia heette. Een ridder van een ver land was verliefd op haar, maar zij was verliefd op een jonge prins. De ridder was jaloers, dus hij wilde haar ontvoeren.

(2) De prinses, een lief meisje, woonde in een groot kasteel. Op een nacht ging de ridder naar het kasteel. Voorzichtig probeerde hij de zware poort te openen, maar die was op slot.

(3) Nadat de ridder in een hoge boom was geklommen, sprong hij de slaapkamer van de prinses binnen. Zij was zo geschrokken, dat zij hard schreeuwde, maar niemand hoorde haar.

(4) Hardhandig pakte de ridder de tegenstribbelende prinses op en vervolgens zette hij haar op zijn paard. Daarna bracht hij haar naar een oude en smalle brug. Aan de overkant zag zij de prins, op wie zij verliefd was. Wat was prinses Amalia opgelucht!

Figure 9.5: Example output with modelled document plans

9.1.3 Evaluation of the Knight story

In the remainder of this section I will evaluate the generated story on a number of different criteria. For this evaluation I will mainly use the final version of the story, so the one in which some additional information has been added manually.

Adding background information

The examples given in this chapter show the advantages of the Background Information Supplier. First of all, the module is responsible for adding the name of the princess which can be used later in the story to refer to the princess. Secondly the module adds the fact that she lives in a castle, which makes the story far more coherent. In the first example this sentence was not present in the final text so the action of going to

the castle was rather unexpected, especially because an indefinite article was used in the referring expression for the castle. In the final story this has been solved by first telling that the princess lives in a castle which is then the reason for going to the castle.

The Background Information Supplier is also responsible for repeating the background information that the princess is in love with the prince. This fact has been told in the very first paragraph and then the prince is not mentioned until the final paragraph. Since the reader might have forgotten about the prince, the background information that she is in love with him is added to the document plan.

Types of referring expressions

The final version of the example story shows that the use of varying referring expressions leads to better results. First of all, different types of referring expressions are used: noun phrases, pronouns and names. The first reference in a paragraph is always a noun phrase, but after this reference pronouns can be used as well. Furthermore the algorithm makes sure that a pronoun is only used when no confusion can arise. Take for example the first reference to the knight in the final sentence of the first paragraph; if a pronoun would have been used it could be unclear whether the pronoun referred to the knight or the prince, so a noun phrase is used.

Additionally, if an entity has been referred to by a pronoun for the last two sentences, a noun phrase is generated, such as the noun phrase 'prinses Amalia' in the final sentence. This final example also shows that referring expressions including the name can be generated if the name for the entity is known.

Finally, the story contains an apposition in the second paragraph: "de prinses, een lief meisje," ("the princess, a sweet girl,"). Currently, the apposition has been added to the details Vector manually by adding the details ($princess01$, $girl01$) and ($girl01$, $sweet$) and the actual generation has been done automatically. Note that it would be better if the apposition would have been generated in the Referring Expression Generator. In this module the fact that the princess is a girl is available (using the subsumption hierarchy), and the fact that the princess is sweet is also available (using the Character Information), so this should not be too complicated.

Adjectives used in referring expressions

The generated referring expressions are not only varied in their types, but also in the number and kinds of adjectives they use. The referring expressions used in the story contain adjectives of all three types:

- Distinguishing adjectives

  The story does not include two characters which are an instance of the same type (such as two knights), so no actual distinguishing adjectives can be added. Recall from section 8.3.2 that adjectives from the Character Information are always added when introducing a character, because these adjectives could be used as distinguishing adjectives later in the story. Examples are the adjective 'mooi' ('beautiful') when introducing the princess, and the combination 'oude en smalle' ('old and narrow') when introducing the bridge.

- Internal state adjectives

  An example of an internal state adjective is 'tegenstribbelende' ('struggling') in the final paragraph, and this adjective has been added by the Document Planner. Currently, adjectives of this type are simply added manually, but in future versions this should also be done automatically.

- Adjectives used for creating a mood

  The adjectives 'groot' ('large') for castle and 'zwaar' ('heavy') for the gate are examples of adjectives added by the Surface Realizer to create a mood, and are taken from the list of possible properties. You can see that the nouns are introduced using such an adjective and then the second reference may include the adjective or may drop it.

Inference-based referring expressions

The example also shows four examples of referring expressions for entities which are related to other entities:

- The first example is 'een ridder van een ver land' ('a knight of a faraway country') in which both parts have not been mentioned before, and therefore both entities use the indefinite determiner 'een' ('a').
- The second referring expression in which an entity is related to another entity is an example of a bridging description ('de poort'), see section 8.3.3. The first version of the story (figure 9.2) generated the expression 'een poort van het kasteel' ('a gate of the castle'), but the final version can generate the expression 'de poort' ('the gate'). This can be done because the castle is very salient when a description for the gate is to be generated (i.e. it has just been mentioned twice), so the part including the castle can be left out. Furthermore a castle usually has only one gate, so a definite article can be used.
- The third example is the reference to the princess's bedroom. In this case the entities are related by a belong-to relation instead of a part-of relation, so the related entity has to be included (i.e. no bridging can be applied). A definite article can be used, because a person usually has only one bedroom. Finally the princess has not been mentioned in the current paragraph, so she should be referred to by a noun phrase ('de slaapkamer van de prinses' ('the princess's bedroom')).
- The final example is the reference to the knight's horse. In this case no bridging can be applied either, but the knight is salient, so a pronoun can be used to refer to the knight ('zijn paard' ('his horse')).

Types of plot elements

The story contains all different types of plot elements: a setting, an internal state, a goal, an action, a failed action and a perception. The story does not include a belief, but since a belief is generated in the exact same way as a perception (and the example used throughout the previous chapters contained a belief), beliefs are also generated correctly.

The final paragraph of the story also contains a description of a state using one of the additional ways ("Wat was prinses Amalia opgelucht!"). This can be achieved using the intensity numbers for internal states, but since these have not been added to the input yet (see section 5.3.2), I have marked the internal state manually.

Used rhetorical relations

Finally the story contains all different types of rhetorical relations; additive, cause, contrast, temporal and elaboration relations. The elaboration relation is expressed using a relative clause and the story contains two of these constructions. The first one appears in the very first sentence and simply uses the relative pronoun 'die'. The second relative clause appears in the final paragraph when the background information is repeated. This relative clause uses a relative pronoun combined with a preposition ('op wie').

In the third paragraph the story contains a cause relation that is lexicalized using the 'zo ..., dat' construction. In this example the 'zo ..., dat' construction can be used, because the princess is extremely scared. Once intensity numbers for the internal states have been added to the input, this type of rhetorical relation could be assigned based on these numbers. Currently, the relation's label has been added to the document plan manually.

Furthermore you can see that the same rhetorical relation is usually expressed differently each time the relation is used. Take for example the rhetorical relation 'temp-after-sequence' ('temp' in the figure), which appears four times in the document plans and is expressed differently each time:

- The first relation is the one between the action of going to the castle and the failed action of opening the gate. In this case the failed action has already been combined with the reason for failure, so the temporal relation between the actions is not expressed explicitly.
- The second time a temporal relation has been specified, is the relation between the action of climbing in a tree and the action of jumping into the princess's bedroom. This time the cue word 'nadat' ('after') is

chosen, so the two clauses are combined into a complex sentence with a main clause and a subordinate clause (ordered chronologically).

- The third temporal relation is the one between the action of lifting up the princess and placing her on the horse. This time the cue word 'nadat' ('after') has just been used, so the cue word 'en' ('and') is chosen combined with the adjunct 'vervolgens' ('then'), which results in a complex sentence consisting of two main clauses.

- The final temporal relation is the relation between the action of placing the princess on the horse and bringing her to the bridge. This final relation is expressed by the adjunct 'daarna' ('then') and results in a simple sentence.

Language use throughout the story

The evaluation criteria focussed on in this section all deal with different aspects of the Narrator agent. However, in section 2.1.3 we saw a number of directives for language use in narratives, so in this section I will evaluate the generated stories on these criteria.

- Sentences should not begin with the words 'There' or 'It' (apart from the very first sentence in a fairy-tale).

  The very first sentence begins with 'er' ('there'), but none of the other sentences start with one of these words. If locations of objects are added by the Background Information Supplier, these sentences would also start with 'er' (such as "Er ligt een appel in het huis.", "There is an apple in the house."), but these sentences will occur only occasionally.

- Many action verbs should be used and it is better to use active voice than passive voice.

  Currently, the system generates all sentences in active voice, but it can generate a sentence in passive voice as well. This should be decided by the Document Planner, so this module should not decide to generate a sentence in passive voice too often.

- It is better to be clear and to use simple words, because the use of many difficult words may lead to confusion and redundancy.

  With the creation of the lexicon we have made sure that only simple words have been added, so this problem will not arise.

- The prose should be fluent, varied in rhythm, and suitable in tone to the type of story.

  I have not taken into account this directive, because different components should take care of the criterion. Take for example the directive for variety in rhythm. This criterion depends partly on the number of constituents in the sentence, but also on the size of each of the constituents. It is possible to add more or less modifiers in order to generate sentences with a more varied number of constituents. On the other hand, at the time of adding modifiers (which takes place in the Document Planner), the system does not know the size of each of the constituents. Furthermore the rhythm may depend on the prosody which will not be taken care of in the Narrator agent, but in the Speech agent. Therefore it will be rather hard to model this criterion correctly, so it has simply been ignored.

- Descriptive language is often used to enhance the story and create images in the reader's mind.

  This should include descriptions of characters and objects. Currently the Character Information and Story World modules are not very detailed, but once detailed sources of information are available this could be extended.

- The text should include many linking words, especially linking words that have to do with time.

  The fabula structure only contains causal relations. If two plot elements cause another plot element, these elements are related by an additive relation, and if a plot element causes two other plot elements, these elements are related by a temporal relation. This results in many causal relations and only a few

temporal relations. However, the results of the final version of the story do include many temporal relations, because they are assigned manually. These results are much better, so future versions of the Narrator agent should assign more temporal relations, which can be achieved using the time arguments of the plot elements.

- Stories are normally written in past tense, but dialogues between characters may change the tense to present or future.

  The entire story is written in past tense. Since the stories do not include dialogues, the tense does not change to a different tense.

- The narrative can be written in first person (I, we) or third person (he, she, they), but in short stories it is best to keep the same narrative style throughout the story.

  The entire story is written in third person, so the story does not switch to a different narrative style.

- Showing is better than simply telling.

  This directive is also rather hard to take into account. The example given in section 2.1.3 showed that for the description of internal states it is usually better to *show* the behavior rather than simply telling it. For example, if a character is awkward, it is best to give examples of his awkward behavior. Since the fabula contains causal relations, the plot element describing the fact that a character is awkward would be caused by a number of other plot elements which may be seen as the reason for being awkward. This means that in such a case the reason is narrated automatically.

  Another possibility is to include a large model with sentences that can be added based on the internal state being described. For example, if a character is awkward the Narrator could add a sentence telling how the character just tripped over something. This would then be comparable to the sentences stored in the Mood Creator, but since it is rather hard to decide which sentences can be added, this directive has been ignored.

- Paragraphs should be about the same topic and should not be too long.

  Since we have assigned the paragraph boundaries manually, the final version of the story satisfies this criterion. Once the paragraph boundaries are assigned automatically, this criterion should be taken into account.

- Sentence length should vary as much as possible.

  First of all, the Surface Realizer combines at most three dependency trees into one. Furthermore it will not add any subordinating cue words to complex sentences, so many sentences consisting of two clauses are also generated. Finally, a number of clauses cannot be combined with another sentence at all, for example because the clause has been marked as an exclamation, or because all possible cue words have been recently used. This means that sentences consisting of one, two and three clauses will all appear in the final texts. The final example shows that most sentences consist of one or two clauses and that one sentence consists of three clauses. This means that sentences of all possible lengths are generated indeed, so it seems that the sentence lengths vary properly.

- There should be much variety in the sentences beginnings.

  Looking at the final version of the story we have succeeded in generating texts with varying sentence beginnings. The grammar has been changed in order to place a modifier at the beginning of a sentence, if the sentence contains a modifier. This results in sentences starting with time expressions (‘op een nacht’ (‘one night’)), adverbs (‘voorzichtig’ (‘carefully’) and ‘hardhandig’ (‘roughly’)), subordinate clauses (‘Nadat hij in een boom was geklommen’ (‘After he had climbed a tree’)), place expressions (‘aan de overkant’ (‘at the other side’)), adjuncts (‘daarna’ (‘then’)) and simply with the subject (‘de prinses’ (‘the princess’) and ‘zij’ (‘she’)).

9.2 Example 2: Story from Chapter 3

In this section I will use the same example as given in chapter 3, see figure 3.3 and 3.4. For the story it is not relevant that there is a sword in the forest and one in the mountains, so I have removed these facts from the story. Furthermore I have split the story into four paragraphs and the initial document plans for each paragraph used to model the input are shown in figure 9.6. The resulting output text is then the text given in figure 9.7.

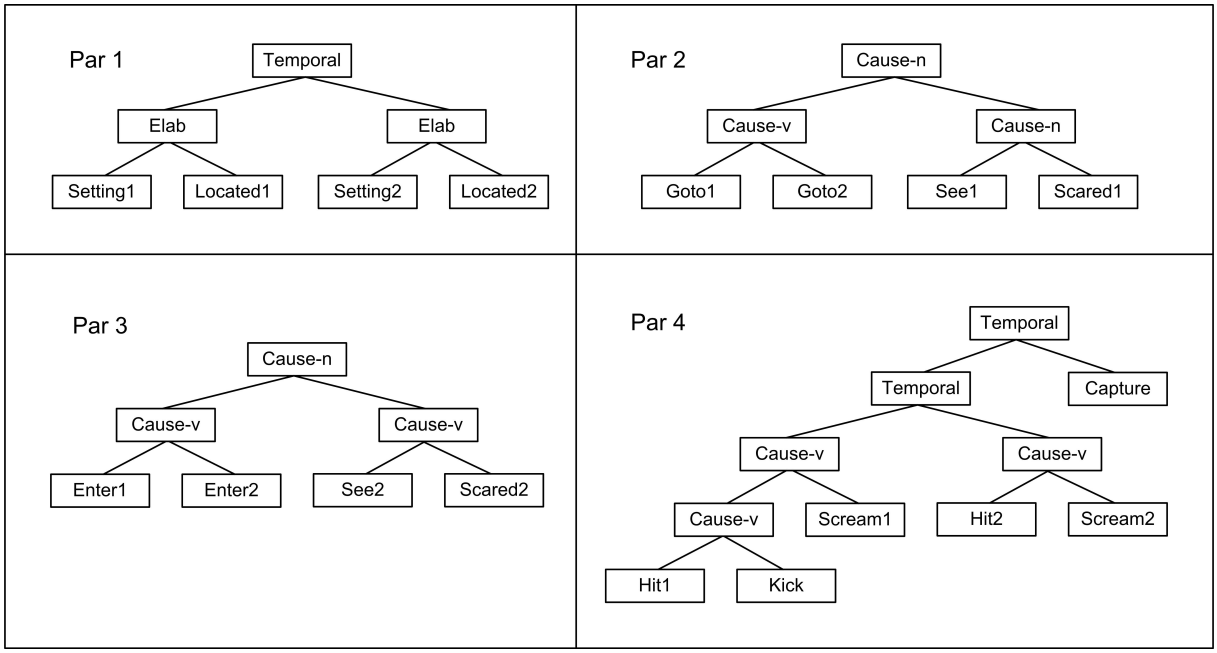

Figure 9.6: Modelled initial document plans for the story from Chapter 3

The plot elements used in the figure can be mapped onto the following sentences in natural language:

*Setting*1	There was a princess.
*Located*1	The princess was in a forest.
*Setting*2	There was a villain.
*Located*2	The villain was in a swamp.
*Goto*1	The princess went to a desert.
*Goto*2	The villain went to the desert.
*See*1	The princess saw the villain.
*Scared*1	The princess was scared.
*Enter*1	The princess entered a castle.
*Enter*2	The villain entered the castle.
*See*2	The princess saw the villain.
*Scared*2	The princess was scared.
*Hit*1	The princess hit the villain.
Kick	The villain kicked the princess.
*Scream*1	The princess screamed.
*Hit*2	The villain hit the princess.
*Scream*2	The princess screamed.
Capture	The villain captured the princess in the castle.

(1) Er was eens een buitengewoon mooie prinses, die Amalia heette. Zij was in een klein bos. Er was ook een schurk, die Brutus heette. Hij was in een moeras.

(2) Terwijl de prinses naar een woestijn ging, wandelde Brutus ook naar de woestijn. Zij zag hem, dus zij was bang.

(3) Omdat de prinses een groot kasteel binnen ging, ging de schurk het ook binnen. Weer zag zij hem en nog steeds was zij daarom bang.

(4) Prinses Amalia sloeg Brutus en dus schopte hij haar. Zij schreeuwde. Ook sloeg Brutus haar, dus weer schreeuwde zij. Uiteindelijk sloot hij het meisje op in het grote kasteel.

Figure 9.7: Example output of the story from Chapter 3

The resulting text shows a number of functionalities not present in the example of the previous section, but it also shows some shortcomings of the current system. First of all, the results show that a referring expression for the princess containing the name does include the noun 'prinses' ('princess'), but that such a referring expression for the villain does not include the noun. This is done because 'princess' is a title noun as explained in section 8.3.2 and 'villain' is not.

Additionally, the final sentence in the example shows that hypernymy can be applied in cases in which there are no other salient entities that also belong to the more general type. In this example the word 'meisje' ('girl') can be used to refer to the princess, because no other female entities have been mentioned in the story recently.

Furthermore it appears that the addition of cue words based on the discourse history (the cue words 'also', 'again' and 'still') make the story far more coherent. However, the example also shows that simply adding these cue words based on repetition is not correct when two cue words are added to the same sentence. Take for example the final sentence in the third paragraph: by adding the cue word 'again' to the first clause and adding the cue word 'still' to the second clause, the causal relation is not correct anymore. The best way to solve this problem is using the intensity numbers of the emotions (see section 5.3.2), and recognizing that the princess became even more scared because she saw the villain again. In that case the result would be: "Doordat zij hem weer zag, werd zij nog banger." ("Because she saw him again, she got even more scared.").

The same sentence also shows that the modifiers are still placed in a rather random order. The sentence would be much better if the modifiers in the second clause were swapped, such as "en daarom was zij nog steeds bang." ("and therefore she was still scared."). Since the grammar cannot distinguish between these modifiers they are placed in a random order, so the best solution to this problem is supporting more modifier labels (see section 10.2.2).

One final advantage of supporting more modifier labels is caused by the fact that currently one of the modifiers is always placed at the beginning of the sentence (if the sentence contains a modifier). This is done in order to avoid generating sentences that all start with the subject. In some sentences this leads to better results (such as the final sentence with the modifier 'uiteindelijk' ('finally')), but in other sentences it would be better if the modifier was placed at the end (such as the one-but-last sentence containing the modifiers 'ook' ('also') and 'weer' ('again')).

9.3 Example 3: Using the Mood Creator and State Transformer

In this final section I will give an example in order to illustrate the functioning of the Mood Creator and the State Transformer. These modules have been implemented, but the input fabula structures do not contain enough information to show the functioning of the modules properly.

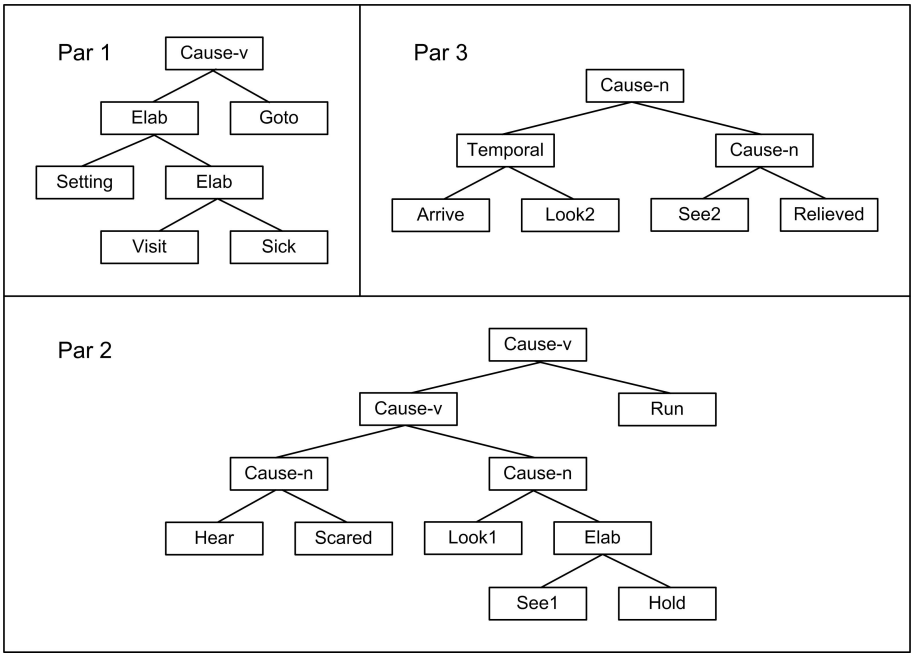

Figure 9.8: Modelled document plans for the example using the Mood Creator and State Transformer

Figure 9.8 shows the initial document plans used for this example, and the results without using the Mood Creator and State Transformer are shown in figure 9.9. The plot elements in the figure can be mapped onto the following sentences:

Setting	There was a princess.
Visit	The princess wanted to visit her grandmother.
Sick	The grandmother was sick.
Goto	The princess went to a forest.
Hear	The princess heard a strange sound.
Scared	The princess was scared.
*Look*1	The princess looked around.
*See*1	The princess saw a knight.
Hold	The knight held a sword.
Run	The princess ran away.
Arrive	The princess arrived at the border of the forest.
*Look*2	The princess looked around.
*See*2	The princess did not see the knight.
Relieved	The princess was relieved.

Currently, the input has not been annotated with the intensity numbers of the characters' emotions (see section 5.3.2), so we will assume that this has been done. In this example the first paragraph would then be annotated with the fact that the princess is extremely happy, the second paragraph with the fact that she is scared, and in the final paragraph the annotation would switch from being scared to being relieved.

(1) Er was eens een prinses, die Amalia heette. Zij wilde haar oma, die ziek was, bezoeken. Op een zonnige ochtend ging zij naar een bos.

(2) Plotseling hoorde de prinses een vreemd geluid, zodat zij bang was. Doordat zij rond keek, zag zij een gemene ridder, die een groot zwaard vast hield. Snel rende prinses Amalia daarom weg.

(3) Nadat de prinses bij een rand van het bos aan was gekomen, keek zij weer rond. Doordat zij de ridder niet zag, was zij opgelucht.

Figure 9.9: Example output of the example without using the Mood Creator and State Transformer

Furthermore the location throughout the entire story is the forest, which is also needed by the Mood Creator. Figure 9.10 then shows a possible result with the Mood Creator and the State Transformer switched on. In this figure the sentences and adjuncts added by one of these components are italicized.

(1) Er was eens een prinses, die Amalia heette. Zij wilde haar oma, die ziek was, bezoeken. Op een zonnige ochtend ging zij naar een bos. *De vogeltjes floten vrolijk op de takken van de bomen.*

(2) Plotseling hoorde de prinses een vreemd geluid, zodat zij bang was. Doordat zij *op trillende benen* rond keek, zag zij een gemene ridder, die een groot zwaard vast hield. Snel rende prinses Amalia daarom weg.

(3) Nadat de prinses *met een bonzend hart* bij een rand van het bos aan was gekomen, keek zij weer rond. Doordat zij de ridder niet zag, was zij opgelucht.

Figure 9.10: Example output of the example using the Mood Creator and State Transformer

This example shows that adding mood sentences and adjuncts describing a character's internal state do result in more varied texts and therefore a system including the Mood Creator and State Transformer modules may indeed lead to better results. On the other hand, the modules should be extended considerably, because they should decide when to add which modifiers automatically. Furthermore not too many of these constructions should be added to a single text, because the resulting text may then become somewhat artificial. Therefore a more detailed analysis of human-written stories and the frequency of such sentences and adjuncts is required.

Chapter 10

Conclusions and Future Work

As the examples in the previous chapter show we have succeeded in creating a module that can convert a fabula structure into natural language. The example used throughout chapters 6 to 8 uses a fabula structure that has also been generated automatically. This example is extremely simple, but it shows that a short fairy-tale can be generated automatically from scratch. The examples in the previous chapter are somewhat more detailed and complicated, but they require additional information. There are a number of things that can make the Virtual Storyteller perform better which will be described in this chapter, but first I will give some conclusions.

10.1 Conclusions

One of the project's main goals was to connect the two already existing parts; to convert a fabula structure generated by the other agents in the Virtual Storyteller into a rhetorical dependency graph. The examples given in the previous chapter show that the Narrator agent can indeed convert a fabula structure represented in OWL into text in natural language correctly. First of all, this shows that the pipelined architecture used in the Narrator agent is satisfactory as an architecture for a natural language generation system. This pipeline is based on the pipeline defined by Reiter and Dale, although it differs in the following aspects. The main difference is the fact that syntactic aggregation and referring expression generation have been moved to the Surface Realizer. Hielkema [Hie05] has shown that the Surface Realizer is the best place to perform syntactic aggregation, so I have adopted this view. Furthermore I believe referring expression generation should be performed after syntactic aggregation in order to avoid generating ungrammatical sentences or generating complex referring expressions which may be removed when performing ellipsis. The final difference is that I have split the lexicalization process in two separate parts; the first one is carried out in the Microplanner (following Reiter and Dale) and the final part is carried out in the Surface Realizer (when generating referring expressions). Using this modified pipeline I have succeeded in generating acceptably coherent texts, so I believe this is a correct architecture for the natural language generation process.

Apart from the goal of simply converting the fabula structure into a rhetorical dependency graph, one of the main goals was to generate narratives that are attractive for the reader. The term 'attractive for the reader' may be rather vague, but I believe the most important aspects are variety and coherence. Even though I have not carried out a detailed evaluation comparing the new results to human-written stories, possibly with test subjects, we can definitely say that the results given in the previous chapter are more varied and coherent than the results of the original template-based design.

The results of the new Narrator agent are more attractive for the reader for the following reasons. First of all, Hielkema has already shown that syntactic aggregation improves the generated texts considerably. By generating sentences of varying length and expressing the relations that hold between different plot elements explicitly, the texts become more coherent and more varied. Furthermore the new Referring Expression Generator makes sure that many different referring expressions can be used. First of all, the module distinguishes between noun phrases and pronouns. Additionally the Referring Expression Generator can generate a large number of different noun phrases; it can add three different types of adjectives, it can include the charac-

ter's name, and it can apply inference in order to generate bridging descriptions or indirect coreferential descriptions. Supporting all of these types of referring expressions obviously contributes to the variety of the generated stories. Furthermore the addition of background information (such as locations of objects), and the use of additional cue words based on the discourse history (such as 'also' and 'again') make the stories more coherent. Since the fabula structure is a causal network containing different types of plot elements, the characters' actions can be explained by their goals and their emotions, which also leads to more coherent texts. Finally adding details (such as adjectives, adverbs and different kinds of expressions) and supporting different ways of expressing the same content (such as different words to lexicalize the same concept, but also different templates for the same type of plot element) make the stories more varied as well. All of these functionalities thus make the stories more varied and more coherent, and therefore more attractive for the reader. However, there are many things left to be done which are described in the remainder of this chapter.

10.2 Suggestions for Future Work

In this section some suggestions for future work are given which can be subdivided into four different kinds of extensions. First of all, section 10.2.1 gives some suggestions for extending the input with additional information. Then section 10.2.2 gives some suggestions for extensions to already existing components of the Narrator agent. Section 10.2.3 gives suggestions for new functionalities that can be added to the Narrator agent. Finally some suggestions for completely new projects are given in section 10.2.4.

10.2.1 Extending the input with more information

In this section some useful extensions to the input are described. These do not require any changes in the Narrator agent, but will make the module perform better.

Using more detailed Character Information and Story World modules

Most of the adjectives added by the Narrator agent are derived from the Character Information module. If this module is limited only a few adjectives can be added, but once the module contains detailed information about the characters' properties more varied referring expressions may be generated. For the same reason a more detailed Story World module will result in more varied and detailed referring expressions. Furthermore using a detailed Story World module will result in adding more background information which will lead to more coherent stories.

In order to generate some additional adjectives the Referring Expression Generator has been extended with a list of possible properties for a number of nouns. First of all, this list only includes the nouns that appear in the story, so once a new story is generated by the system, this list of adjectives should be extended. Furthermore simply choosing an adjective randomly from a list of possible adjectives may lead to inconsistency. Since the Referring Expression Generator will only include an adjective when no properties have been specified, this chance is rather small, but the problem may still arise. Therefore providing the system with a more detailed Character Information as input (generated by the Plot agent), will also make sure that the generated stories are consistent throughout the whole story.

Using a more detailed emotional model

The Character Information module only contains *static* information about the characters, such as their names, looks and ages, all properties that can be added to the story at any time. Apart from these static properties the characters also have *dynamic* properties which may change over time, such as being scared or being hungry.

The Virtual Storyteller uses a rather complicated emotional model, including eight different types of emotions. The intensity of the emotions are then modelled by assigning a number between -100 and +100 and these numbers are updated every single time step. Currently, this information has not been included in the input though, so the Narrator agent cannot access these exact numbers. Once this is added to the input, the Narrator can use more varied ways of expressing the characters' internal states, for example by saying

that a certain character becomes angrier and angrier. Finally the numbers can be used to decide whether one of the additional ways of describing a state can be used. If the value of the property 'happy' is (almost) 100 points for a certain character, this can for example be lexicalized as "Wat was ze blij!", or "Ze was nog nooit zo gelukkig geweest!".

10.2.2 Extending existing elements of the Narrator

In this section some additions to the current system will be described which will make the Narrator perform better. These additions are all extensions to already existing parts of the Narrator.

Extending the State Transformer and the Mood Creator

Currently the State Transformer is rather limited; it only supports three different states (being happy, angry or scared) and for each state only two adjuncts and two sentences have been stored. Once the more detailed emotion model has been included in the input, more types of states can be added and more adjuncts or complete sentences can be added. Furthermore the module could choose the way the state will be expressed based on the emotions' intensity numbers rather than doing this randomly.

For the same reasons the Mood Creator is also very limited. It also supports three different internal states (happy, angry and scared) and only two locations (the forest and on a path). First of all the module could be extended by supporting more states and more locations, and by storing more sentences for each combination. More importantly however, the module should be extended by keeping track of more different types of variables, such as the time of day or the season. Once all of these variables are known, there may be fewer possibilities for adding a mood sentence, but the generated texts will obviously be more coherent.

Using more types of rhetorical relations

Currently the Surface Realizer supports the following rhetorical relations: additive, cause, contrast, elaboration, purpose and temporal. First of all the set of rhetorical relations could be extended, for example by distinguishing between a cause and a consequence relation. An advantage of supporting both relations is that the most important node can be told first. Recall that by leaving the order of the nodes unchanged each cause relation is expressed by first telling the cause and then telling the consequence (see section 8.2.2). If the Surface Realizer would support both relations, simply the most important node may be told first (which is either the cause or the consequence).

Furthermore the rhetorical relations currently supported by the Surface Realizer all support sub relations as well, for example the temp-after-sequence or temp-suddenly relations for the temporal relation. Currently the Surface Realizer uses some of these sub relations, but this can also be extended. Using the time arguments of the plot elements it is possible to select better temporal relations, such as choosing temp-suddenly when an event suddenly occurs, or choosing temp-during when two actions happen (partly) simultaneously. Selecting other types of relations, on the other hand, requires a more advanced algorithm. Take for example the first story from chapter 9 in which two internal elements cause another plot element; the knight being in love with the princess and the princess being in love with someone else together lead to the knight's internal state of being jealous. Since they *together* cause the internal state, the plot elements are combined with an additive relation, but it would be much better if the plot elements would have been combined with a contrast relation. This can be achieved by comparing the plot elements and recognizing that they describe two contradicting facts (or actually recognizing that the fact that the princess loves a prince implies the fact that she does *not* love the knight, and finally that this implied fact contradicts with the fact that the knight loves the princess).

Extending the grammar with more modifier labels

As explained in section 8.4.1 the original Surface Realizer supported only one modifier label and used this label for adjectives, adverbs, all kinds of expressions and entire subordinate clauses. I extended this with two more modifier labels in order to make sure that a certain modifier appeared at the beginning or at the end of the sentence, but this can be extended even more. It would be best if the modifier label depended

on the type of modifier instead of on the fact whether the modifier should appear at the beginning or end of the sentence. Extending the grammar with all of these types of modifiers requires adding a rule for each modifier. Take for example the following rule:

```
SMAIN -> { SU, HD, MOD }
```

This rule says that a dependency tree consisting of a subject, a head and a modifier should be translated into a sentence in which the constituents are placed in the order SU, HD, MOD. Suppose that the grammar will be extended in order to support five different kinds of modifiers. This would then result in adding five more rules (one for each modifier label). Rules containing two modifiers will result in adding even more rules, since the modifiers may usually appear at different places in the sentence.

Note that supporting different modifier labels would be necessary if the system generated the stories in English (see section 10.2.3). In English there is a requirement for placing the time expression after the place expression (at least at the end of the sentence). If the same label 'mod' would be used for both expressions, the grammar would not be able to distinguish between these two kinds of expressions and would place them in a random order, which could result in ungrammatical sentences.

Using text files as input

Currently the background information is added to the system in Java code, but it would be better if this was done using simple text files. Examples of such input are the grammar, the lexicon, the templates used for the generation of sentence plans, mood sentences used by the Mood Creator and the sentences and adjuncts used by the State Transformer if an internal state can be told differently.

First of all text files are easier to extend than Java code, especially by people who are not familiar with the system. Furthermore if text files are supported the entire system does not have to be recompiled when an entry has been added. Finally the lexicon could even be stored as an OWL file in the exact same way as the fabula structure. An advantage of this is that all information needed by the different components can then be stored in one single reasoning system. Furthermore there are some useful programs such as Protégé in which OWL files can be presented graphically to the user which makes them even easier to extend.

10.2.3 Adding new functionalities to the Narrator

In this section some other additions to the current system will be described. These additions however are all meant to add new functionalities to the Narrator.

Adding different perspectives

There is a large amount of literature about different types of narrators and perspectives used in human-written stories. StoryBook (see section 2.3.3) also supports different narrators, such as narrators using first person and ones using third person, embodied and disembodied narrators, omniscient and limited narrators, and reliable and unreliable narrators. Another possibility is telling the story in a different order than the chronological order.

A possible follow-up project is to find out how different narrators affect the story to be generated. For example it may be interesting to model the factors as a set of parameters that can be initialized before the story is generated (one parameter for person, one for embodiment etcetera). This way one fabula structure may result in different stories, which also makes the system more attractive for the user.

Generating the stories in English

Earlier versions of the system were able to generate English texts, but these systems used fixed sentences for the language generation process, so the entire sentences could be translated into English. Since the current system uses more sophisticated natural language generation techniques, it will be more complicated to extend the system to generate English texts as well. On the other hand, it may be a very useful extension, especially because the system could gain more international interest.

The most obvious changes are the lexicon and the grammar, since these components are completely language dependent. The lexicon can easily be turned into an English lexicon by changing the roots to English words and by changing some morphological information. The grammar, however, may be more complicated to change, because the grammar has to support different modifier labels, as described in the previous section.

The Document Planner is mainly language independent, so this component could be left unchanged. The templates used by the Microplanner may change somewhat, but these changes are also rather small. The most important changes are therefore in the Surface Realizer, for example in the Relation Transformer. In this component two dependency trees are combined into one with the addition of a proper cue word. Especially the generation of relative clauses will be rather different in English. In both languages the relative clause always begins with the relative pronoun, but in Dutch the pronoun cannot be separated from the preposition (if the pronoun is accompanied by a preposition). Take for example the Dutch sentence "De prins, *op wie* de prinses verliefd was, ging naar het bos.". This sentence should be translated into the English sentence "The prince, *who* the princess was in love *with*, went to the forest.". This example shows that the Dutch language combines the relative pronoun with a preposition, while the English language can also move the preposition to the end of the sentence. This is quite a difference, but it is rather easy to extend the Relation Transformer and grammar to generate relative clauses correctly.

Finally, the Referring Expression Generator may change as well. The decision of when to pronominalize is language independent, so the pronominalization algorithm does not have to be changed. The referring expressions used however are language dependent, so the actual generation of the expressions should be changed. On the other hand, this will not be very complicated either, because (parts of) many algorithms can still be used, such as the Incremental algorithm for the generation of distinguishing adjectives which was originally designed for the English language.

10.2.4 Suggestions for completely new projects

During the design phase of a project ideas for all different kinds of new projects arise. In this last section some suggestions for related projects are given.

Supporting dialogues

Analysis of human-written fairy-tales shows that dialogues appear very often in fairy-tales. Callaway [Cal00] has done some research in generating dialogues automatically and supporting different ways of describing speech. Examples are direct speech (such as "John said: 'I love you, Mary'."), indirect speech (such as "John told Mary that he loved her."), or a mixture of both. Currently the characters in the Virtual Storyteller are not able to talk yet, because the Action ontology does not include a talk action. Once this has been added to the system, the Narrator agent can be extended by turning the talk actions into natural language. First of all this requires a more advanced module for adding punctuation, such as quotation marks. Furthermore the grammar has to be extended, because exclamations and incomplete sentences may appear in speech. An example of an incomplete sentence is an answer to a question; if someone asks the question "Where are you going?", a correct answer is "To my grandmother." instead of repeating part of the question as "I am going to my grandmother.". This example shows that sentences in speech may consist of only one constituent, so the grammar should be able to recognize such a sentence as a correct one.

Generating speech and visualization

Currently there is some ongoing work into converting the final text generated by the Narrator agent into speech using speech generation. This is a useful extension since it makes the system more attractive for the user, especially for small children. If a Speech agent will be added to the system, the Narrator could also annotate the stories with information that can be used by the Speech agent. Examples of such additional information are prosodic information, such as which words to emphasize, and the emotions of the characters which can be used to choose the correct tone of the sentence. Finally it may be useful to support visualization, exactly for the same reasons as supporting speech generation; it makes the system more attractive for the user.

Bibliography

[AL99] N. Asher and A. Lascarides. Bridging. *Journal of Semantics*, 15:83–113, 1999.

[Cal00] C.B. Callaway. *Narrative Prose Generation*. PhD thesis, North Carolina State University, Raleigh, NC, 2000.

[Cho82] N. Chomsky. *Some concepts and consequences of the Theory of Government and Binding*. MIT Press, 1982.

[CL01] C.B. Callaway and J.C. Lester. Narrative prose generation. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, pages 1241–1248, Seattle, WA, 2001.

[Cla75] H. Clark. Bridging. *Theoretical issues in natural language processing*, 1975.

[CS98] G. C. Chambers and R. Smyth. Structural parallelism and discourse coherence: A test of centering theory. *Journal of Memory and Language*, 39:593–608, 1998.

[Dal92] R. Dale. *Generating Referring Expressions - Constructing descriptions in a domain of objects and processes*. MIT Press, 1992.

[dJS74] E. de Jong and H. Sleutelaar. *Nieuwe Sprookjes van de Lage Landen*. De Bezige Bij, Amsterdam, 1974.

[Ehr80] K. Ehrlich. Comprehension of pronouns. *Quarterly Journal of Experimental Psychology*, 32:247–255, 1980.

[Faa02] S. Faas. Virtual storyteller: an approach to computational storytelling. Master's thesis, University of Twente, 2002.

[FJF03] R. Fikes, J. Jenkins, and G. Frank. Jtp: A system architecture and component library for hybrid reasoning. 2003.

[FJF04] G. Frank, J. Jenkins, and R. Fikes. Jtp: An object-oriented modular reasoning system, http://www.ksl.stanford.edu/software/jtp/, July 2004.

[GDAPH05] P. Gervas, B. Diaz-Agudo, F. Peinado, and R. Hervas. Story plot generation based on case based reasoning. *Knowledge-Based Systems*, 18:235–242, 2005.

[GF92] M.R. Genesereth and R. Fikes. *Knowledge Interchange Format, Version 3.0 Reference Manual*. Computer Science Department, Stanford University, Stanford, CA, 1992.

[GJW95] B. J. Grosz, A. K. Joshi, and S. Weinstein. Centering: a framework for modelling the local coherence of discourse. *Computational Linguistics*, 21(2):203–226, 1995.

[GS03] Claire Gardent and Kristina Striegnitz. Generating bridging definite descriptions. In H. Bunt and R. Muskens, editors, *Computing Meaning*, volume 3. Kluwer, Dordrecht, The Netherlands, 2003.

[Haj93] E. Hajičová. Issues of sentence structure and discourse patterns. *Theoretical and Computational Linguistics*, 2, 1993.

[HCP00] R. Henschel, H. Cheng, and M. Poesio. Pronominalization revisited. In *COLING*, pages 306–312, 2000.

[HF86] J.R. Hayes and L.S. Flower. Writing research and the writer. *American Psychologist*, 41(10):1106–1113, October 1986.

[Hie05] F. Hielkema. Performing syntactic aggregation using discourse structures. Master's thesis, University of Groningen, 2005.

[HMR+03] H. Hoekstra, M. Moortgat, B. Renmans, M. Schouppe, I. Schuurman, and T. van der Wouden. *Syntactische Annotatie voor het Corpus Gesproken Nederlands*, 2003.

[JM00] D. Jurafsky and J.H. Martin. *Speech and language processing - an introduction to natural language processing, computational linguistics and speech recognition.* Prentice Hall, 2000.

[Keh02] A. Kehler. *Coherence, Reference, and the Theory of Grammar.* CSLI Publications, 2002.

[Kil92] Crawford Kilian. Advice on novel writing: http://www.steampunk.com/sfch/writing/ckilian/, November 1992.

[KKH97] I. Kruijff-Korbayova and E. Hajicova. Topics and centers: A comparison of the salience-based approach and the centering theory. *Prague Bulletin of Mathematical Linguistics*, 67:25–50, 1997.

[KP00] R. Kibble and R. Power. An integrated framework for text planning and pronominalization. In *Proceedings of the International Conference on Natural Language Generation (INLG)*, pages 77–84, 2000.

[KT00] E. Krahmer and M. Theune. Efficient context-sensitive generation of referring expressions. *Information Sharing: Reference and Presupposition in Language Generation and Interpretation*, pages 223–264, 2000.

[LL94] S. Lappin and H. Leass. An algorithm for pronominal anaphora resolution. *Computational Linguistics*, 20(4):535–561, 1994.

[Moh00] C.J. Mohan. The writing process, 2000.

[MS99] K. F. McCoy and M. Strube. Generating anaphoric expressions: Pronoun or definite description? In Dan Cristea, Nancy Ide, and Daniel Marcu, editors, *The Relation of Discourse/Dialogue Structure and Reference*, pages 63–71. Association for Computational Linguistics, New Brunswick, New Jersey, 1999.

[MT87] W. C. Mann and S. A. Thompson. Rhetorical structure theory: A theory of text organization. *The Structure of Discourse*, 1987.

[Onl98] English Online. Fairy tales: http://english.unitecnology.ac.nz/resources/units/fairytales/home.html, 1998.

[Pem89] L. Pemberton. A modular approach to story generation. In *4th European Conference of the Association for Computational Linguistics*, pages 217–224, Manchester, 1989.

[Pri81] E. F. Prince. Toward a taxonomy of given-new information. *Syntax and semantics: Radical Pragmatics*, pages 223–255, 1981.

[PV98] M. Poesio and R. Vieira. A corpus-based investigation of definite description use. *Computational Linguistics*, 24(2):183–216, 1998.

[RD92] E. Reiter and R. Dale. A fast algorithm for the generation of referring expressions. In *Proceedings of the fifteenth International Conference on Computational Linguistics (COLING-192)*, volume 1, pages 232–238, Nantes, France, 1992. International Committe on Computational Linguistics.

[RD00] E. Reiter and R. Dale. *Building Natural Language Generation Systems*. Cambridge University Press, 2000.

[Rei95] E. Reiter. Nlg vs. templates. In *Proceedings of the 5th European Workshop on Natural Language Generation (EWNLG'95)*, pages 95–106, Leiden, The Netherlands, 1995.

[Ren04] S. Rensen. De virtuele verhalenverteller: agent-gebaseerde generatie van interessante plots. Master's thesis, University of Twente, 2004.

[Sow00] J.F. Sowa. *Knowledge Representation: Logical, Philosophical, and Computational Foundations*. Brooks/Cole Publishing Co., Pacific Grove, CA, 2000.

[Str18] W. Strunk. *The elements of style*. Ithaca, N.Y.: Priv. print, 1918.

[Str98] M. Strube. Never look back: An alternative to centering. In *COLING-ACL*, pages 1251–1257, 1998.

[SW00] M. Strube and M. Wolters. A probabilistic genre-independent model of pronominalization. In *ANLP*, pages 18–25, 2000.

[Swa06] I. M. T. Swartjes. The plot thickens - bringing structure and meaning into automated story generation. Master's thesis, University of Twente, the Netherlands, March 2006.

[TVdBS89] T. Trabasso, P. Van den Broek, and S. Y. Suh. Logical necessity and transitivity of causal relations in stories. *Discourse Processes*, 12:1–25, 1989.

[Uij06] J. R. R. Uijlings. Designing a virtual environment for story generation. Master's thesis, University of Amsterdam, the Netherlands, March 2006.

[vdBBD+02] L. van der Beek, G. Bouma, J. Daciuk, T. Gaustad, R. Malouf, G. Van Noord, R. Prins, and B. Villada. Algorithms for linguistic processing. Technical report, NWO PIONIER Progress Report, Groningen, 2002.

[VH] E. Van Hengel. Sprookjes pagina: http://members.chello.nl/e.vanhengel/index.htm.

[W3C04] W3C. Owl web ontology language, http://www.w3.org/tr/owl-features/, February 2004.

Appendix A

Human-Written Stories for Analysis

A.1 Koning Lijsterbaard

A.1.1 Story

(1) Er was eens een prinses, die heel verwaand was. Ze vond zichzelf zó mooi dat ze iedereen die er wat minder mooi uitzag uitlachte. Elke dag verschenen er prinsen, edelmannen en koningen aan het hof om haar hand te vragen, maar op allen had ze wel iets aan te merken: de een was te lang, de ander te kort, te mager, te dik, de neus was te lang, de benen krom... Vooral om één koning moest zij hard lachen omdat hij zo'n raar puntig baardje had; spottend noemde zij hem koning 'Lijsterbaard'.

(2) De vader van de prinses vond het gedrag van zijn dochter erg akelig en tenslotte werd hij zó boos op haar dat hij zei: "Mijn geduld is op! Nu trouw je met de eerste de beste man die aan het hof komt!" Koning Lijsterbaard had dit gehoord; hij vermomde zich als muzikant en ging onder het raam van de prinses zijn liedjes spelen. De prinses ging kijken wie die mooie muziek maakte en toen de koning dat zag haalde hij de muzikant in het paleis en liet zijn dochter met hem trouwen.

(3) Huilend en jammerend volgde zij haar nieuwe echtgenoot naar een armoedig hutje in het bos. Daar moest zij het huishouden doen, koken, spinnen en weven, maar omdat zij dat nooit had geleerd bracht zij er niet veel van terecht. Toen liet de muzikantkoning haar potten bakken om te verkopen, maar toen zij met haar waren op de markt stond, kwam er een ruiter die dwars over haar potten heen reed en alles brak (die ruiter was natuurlijk koning Lijsterbaard, die zich had vermomd). Huilend ging de prinses naar huis. "Kan je dan ook niets goed doen?" vroeg haar man, "Dan moet je maar naar het paleis gaan en vragen of je in de keuken mag werken". Het ongelukkige prinsesje werd aangenomen als keukenhulp en nu moest zij afwassen, dweilen en met zware ketels sjouwen. Behalve haar loon mocht zij ook restjes eten van de tafel van de koning, die zij in haar schort verzamelde, mee naar huis nemen.

(4) Op een dag gaf de koning een groot diner met bal na. Voor al het personeel in de keuken was het een drukke dag. Ook de prinses moest de hele dag en avond hard werken, maar zij kon wel allerlei lekkere hapjes onder haar schort verzamelen om thuis met haar man een heerlijk maaltje te kunnen eten. Tijdens het dansen moest zij in de balzaal hapjes rondbrengen voor de gasten. De koning, en natuurlijk was dit niemand anders dan onze koning Lijsterbaard, kwam naar haar toe en vroeg haar ten dans. De prinses wilde weigeren, maar de koning had haar hand al gegrepen en leidde haar naar het midden van de dansvloer. Daar gebeurde iets vreselijks: haar schort scheurde en alle etensrestjes die zij had verzameld, vielen om haar heen op de grond. Alle gasten begonnen hard te lachen; het was ook zo'n raar gezicht. De prinses had zich nog nooit zo ellendig gevoeld.

(5) Koning Lijsterbaard vond dat de verwaande prinses nu haar lesje wel had geleerd en hij vertelde haar de waarheid; hoe hij zich vermomd had en haar een heel moeilijke tijd had laten doormaken. De prinses had wel even tijd nodig om dit alles te verwerken, maar toen was zij toch ook wel heel blij: de laatste weken had zij dikwijls met spijt terug gedacht aan alle mensen die zij had uitgelachen en aan koning Lijsterbaard nog wel het meest! Ze waren al getrouwd, maar de bruiloft werd opnieuw gevierd; dit keer met een groot feest. De vader van de prinses behoorde ook tot de gasten. Wat was hij blij dat zijn dochter zo veranderd was en

dat zij ondanks alles zo'n goede echtgenoot had gekregen. Nooit was de prinses meer verwaand of lachte zij om iemands uiterlijk. En zo leefden zij, de prinses met haar koning Lijsterbaard, nog lang en gelukkig.

A.1.2 Rhetorical relations

paragraph	# clauses	sentences
1	2	elaboration2
	2	cause
	4	purpose, contrast, elaboration
	3	cause, cause
2	3	temporal, cause
	" ... "	
	3	cause, temporal
	4	temporal, temporal, temporal
3	1	
	3	contrast, cause
	8	purpose, contrast, temporal, elaboration2, additive, elaboration, elaboration2
	1	
	" ... "	
	2	temporal
	2	elaboration2
4	1	
	1	
	3	contrast, purpose
	1	(or two temporal)
	3	elaboration, additive
	3	contrast, additive
	3	elaboration, cause
	2	cause
	1	
5	4	cause, elaboration, additive
	4	contrast, cause, additive
	3	contrast, elaboration
	1	
	3	cause, additive
	2	additive
	2	apposition

Table A.1: Rhetorical relations used in 'Koning Lijsterbaard'

A.1.3 Referring expressions

referent	paragraph	referring expressions
prinses	1	een prinses, ze, ze, haar, ze, zij, zij
	2	de prinses, zijn dochter, haar, de prinses, de prinses, zijn dochter
	3	zij, haar, zij, zij, zij, haar, zij, haar, haar, de prinses, haar, het ongelukkige prinsesje, zij, haar, zij, zij, haar
	4	de prinses, zij, haar, haar, zij, haar, haar, de prinses, haar, haar, haar, zij, haar, de prinses
	5	de verwaande prinses, haar, haar, haar, de prinses, zij, zij, zij, de prinses, zijn dochter, zij, de prinses, zij, de prinses
koning 1	2	de vader van de prinses, zijn, hij, hij, de koning, hij, zijn
	5	de vader van de prinses, hij, zijn
koning 2	1	één koning, hij, koning Lijsterbaard
	2	koning Lijsterbaard, hij, zijn, de muzikant, hem
	3	haar nieuwe echtgenoot, muzikantkoning, koning Lijsterbaard, haar man
	4	de koning, haar man, de koning, onze koning Lijsterbaard, de koning
	5	koning Lijsterbaard, hij, hij, koning Lijsterbaard, zo'n goede echtgenoot, haar koning Lijsterbaard

Table A.2: Referring expressions used in 'Koning Lijsterbaard'

A.2 De gelaarsde kat

A.2.1 Story

(1) Er was eens een arme molenaar, die twee zoons had. Toen de oude man stierf, was er dan ook niet veel om te verdelen: de oudste zoon kreeg de molen en voor de jongste bleef niets anders over dan de kat. Met de kat trok de jongeman de wijde wereld in.

(2) Deze kat was een bijzonder dier; zodra zij de molen hadden verlaten, begon zij te spreken: “Meester, als je mijn raad steeds opvolgt zal ik je rijk maken en misschien trouw je dan nog wel met een prinses!” De verbaasde jongen besloot om te doen wat de kat zei. Van zijn laatste geld kocht hij een rode cape, een hoed en een paar laarzen voor haar. Intussen had de kat een konijn gevangen en, uitgedost in haar nieuwe kleren, bracht zij dat naar het kasteel van de koning: “zeg maar tegen de koning dat het een geschenk is van de markies van Karrabas,” zei ze. Een paar dagen later bracht ze een mooie fazant naar het kasteel en weer zei ze: “Een geschenk van de markies van Karrabas”.

(3) De koning was nu wel nieuwsgierig geworden naar deze onbekende markies en besloot hem een bezoek te gaan brengen. Samen met zijn dochter reed hij uit en al spoedig kwamen zij langs graanvelden waar boeren aan het werk waren. “Aan wie behoren deze graanvelden?”, vroeg de koning. “Aan de markies van Karrabas”, antwoordden de boeren. Even later kwamen zij langs weilanden waar koeien en schapen stonden te grazen. “Aan wie behoren deze weilanden en het vee?”, vroeg de koning aan de herders. “Aan de markies van Karrabas”, luidde het antwoord van de herders. De slimme kat was de koning voor geweest en had de boeren en herders precies verteld wat zij tegen de koning moesten zeggen als hij vragen zou stellen.

(4) In werkelijkheid was een oude, lelijke reus de baas over de landerijen. Ondertussen was de kat teruggegaan naar de jongeman. “Als er een rijtuig aankomt, moet je je uitkleden en in het meer springen”, zei ze tegen hem. De jongen vond dit wel vreemd, maar toen het rijtuig van de koning aan kwam rijden, deed hij toch precies wat de kat hem had opgedragen. “Help, help!”, riep de kat nu, “Mijn meester, de markies van Karrabas, verdrinkt!” De dienaren van de koning haalden de jongeman uit het water en gaven hem droge kleren. “Majesteit, wilt u mijn meester naar zijn kasteel brengen?”, vroeg de kat nu aan de koning en wees daarbij naar het kasteel van de reus. Natuurlijk wilde de koning dat wel doen.

(5) De kat rende nu langs een andere weg heel snel naar het kasteel van de reus. “Ik heb gehoord dat je een beetje kan toveren, maar dat wil ik wel ’s zien voor ik het geloof!”, zei ze tegen de reus. De reus veranderde zichzelf in een leeuw. De kat deed alsof zij diep onder de indruk was: “Dat was heel knap van je, maar kan je je ook in iets heel kleins veranderen, een muis bijvoorbeeld?” Dat deed de domme reus. De kat ving de muis/reus en at hem op met huid en haar.

(6) Net op tijd trouwens, want daar kwam het rijtuig van de koning al door de poort van het kasteel. De kat ontving de koning, de prinses en de jongeman en liet door de dienaren van de reus (die maar wat blij waren met een nieuwe meester) een feestmaal bereiden. Toen de koning alle rijkdommen in het kasteel zag, vroeg hij of de markies er niet wat voor voelde om met zijn dochter te trouwen. Dat wilde de jongeman maar al te graag want hij vond haar heel mooi en lief. De prinses was al verliefd geworden op de jongeman zodra zij hem zag en dus kon de bruiloft al gauw worden gevierd. De jongeman vergat nooit dat hij zijn rijkdom en geluk aan de kat te danken had en zorgde altijd heel goed voor haar. En zo leefden zij nog lang en gelukkig.

A.2.2 Rhetorical relations

paragraph	# clauses	sentences
1	2	elaboration2
	4	temporal, elaboration, additive
	1	
2	3	elaboration, temporal
	“ ... ”	
	1	
	1	
	2	temporal
	“ ... ”	
	2	temporal
3	2	cause
	3	temporal, elaboration2
	“ ... ”	
	2	elaboration2
	“ ... ”	
	3	temporal, condition
4	1	
	1	
	“ ... ”	
	3	contrast, temporal
	“ ... ”	
	2	temporal
	“ ... ”	
	2	temporal
	1	
5	1	
	“ ... ”	
	1	
	1	
	1	
	2	temporal
6	1	
	3	temporal, elaboration2
	2	temporal
	2	cause
	3	temporal, cause
	2	cause
	1	

Table A.3: Rhetorical relations used in ‘De gelaarsde kat’

A.2.3 Referring expressions

referent	paragraph	referring expressions
jongen	1	de jongste, de jongeman
	2	de verbaasde jongen, zijn, hij
	4	de jongeman, hem, de jongen, hij, hem, de jongeman, hem
	6	de jongeman, de markies, de jongeman, hij, de jongeman, hem, de jongeman, hij, zijn
kat	1	de kat, de kat
	2	deze kat, zij, de kat, haar, de kat, haar, zij, ze, ze, ze
	3	de slimme kat
	4	de kat, ze, de kat, de kat, de kat
	5	de kat, ze, de kat, zij, de kat
	6	de kat, de kat, haar
koning	2	de koning
	3	de koning, zijn, hij, de koning, de koning, de koning, hij
	4	de koning, de koning, de koning, de koning
	6	de koning, de koning, de koning, hij, zijn
prinses	3	zijn dochter
	6	de prinses, zijn dochter, haar, de prinses, zij
reus	4	een oude lelijke reus
	5	de reus, de reus, de reus, de domme reus, de muis/reus, hem
	6	de reus

Table A.4: Referring expressions used in 'De gelaarsde kat'

A.3 De ring van de koningsdochter

A.3.1 Story

(1) Dicht bij zee woonde vroeger een koning, die over heel Friesland regeerde. Hij had een dochter, een heel mooi meisje, dat zou trouwen met een prins uit een ander land. Hij had haar een prachtige gouden ring gegeven, en daar was ze erg blij mee.

(2) Als het mooi weer was, ging die koningsdochter graag spelevaren op zee, en op een keer verloor ze daar haar ring. Wat was ze bedroefd! De koning liet overal bekendmaken, dat zijn dochter haar ring in de zee had verloren en dat degene, die hem vond, een flinke som geld zou krijgen.

(3) Iedereen ging zoeken, maar niemand kon de ring terugvinden. Toen liet de koning bekendmaken, dat degene, die hem vond, met de prinses mocht trouwen, want hij dacht, dat ze de ring anders nooit terug zouden krijgen; 't was maar een lokmiddel. Daarop gingen er nog veel meer op zoek, maar zij vonden hem niet.

(4) Een visser bracht een zalm mee naar huis. Zijn vrouw ging ermee naar de markt. Een boertje, van wie men dacht, dat hij niet goed bij zijn hoofd was, kocht die vis, sneed hem open en vond de ring.

(5) Hij ging met de ring naar het paleis en meldde zich aan. Bij de eerste deur stond een hoveling. Toen hij hoorde, dat het boertje de ring had, wilde hij hem niet doorlaten, of hij moest hem het derde deel beloven van de beloning, die hij bij de koning zou bedingen. De hoveling wist wel, dat er een vette prijs op stond. Het boertje zag geen kans om er anders door te komen, en beloofde het.

(6) Bij de tweede deur stond weer een hoveling, en die wou hem ook niet doorlaten, of hij moest de helft krijgen van wat het boertje bedong. De boer zag geen kans er door te komen, en beloofde ook dat.

(7) Toen kwam hij bij de koning en gaf de ring aan de koningsdochter. Zij was er erg blij mee, maar de koning zei dat hij kon haar niet tot vrouw krijgen, want ze was al verloofd.

(8) "Zeg maar wat je hebben wilt, dan zal ik het je geven." "Ach man," zei het boertje, "Ik wil haar ook helemaal niet hebben, want wat moet ik met twee vrouwen; aan een heb ik genoeg. Zeg maar hoeveel geld je me wilt geven."

Toen vroeg de koning, wat hij vond van vijfhonderd daalders.

"Daar ben ik tevreden mee," zei het boertje, "maar ik beding er vierhonderdvijftig stokslagen bij."

"Wat? Vierhonderdvijftig stokslagen?"

"Ja. De hoveling bij de eerste deur moet er honderdvijftig hebben, want die heb ik een derde portie beloofd van wat ik zou bedingen, en de hoveling bij de tweede deur moet er tweehonderdvijfentwintig hebben, want die eiste de helft. Geld krijgen ze niet, want dat heb ik niet bedongen."

De koning sprong op, toen hij dat hoorde. "Wat een rakkers," zei hij. "We zullen ze geven wat ze toekomt, en wel meteen."

Toen werden die kerels geroepen. Ze konden er niet onderuit, en kregen zoveel stokslagen toegediend, dat ze op de grond bleven liggen. Het boertje kreeg zijn daalders en ging ermee op stap.

"Die heb ik te pakken," zei hij bij zichzelf. "Vijfhonderd daalders voor een gouden ring, dat was een goede handel."

A.3.2 Rhetorical relations

paragraph	# clauses	sentences
1	2	elaboration2
	3	apposition, elaboration2
	2	cause
2	3	condition, temporal
	1	
	2	additive
3	2	contrast
	3	cause, elaboration
	2	contrast
4	1	
	1	
	4	elaboration2, temporal, temporal
5	2	temporal
	1	
	4	temporal, condition, elaboration2
	1	
	2	cause
6	3	additive, condition
	2	cause
7	2	temporal
	3	contrast, cause

Table A.5: Rhetorical relations used in 'De ring van de koningsdochter'

A.3.3 Referring expressions

referent	paragraph	referring expressions
koning	1	een koning, hij, hij?
	2	de koning, zijn
	3	de koning, hij
	5	de koning
	7	de koning, de koning
prinses	1	een dochter, een heel mooi meisje, haar, ze
	2	die koningsdochter, ze, ze, zijn dochter, haar
	3	de prinses
	7	de koningsdochter, zij, haar, ze
boertje	4	een boertje, hij
	5	hij, het boertje, hem, hij, hij, het boertje
	6	hem, het boertje, de boer
	7	hij, hij

Table A.6: Referring expressions used in 'De ring van de koningsdochter'

A.4 De jongen die lezen en schrijven leerde

A.4.1 Story

(1) Er was eens een jongen, die niet kon lezen en schrijven, maar hij wilde het graag leren. Hij vond echter niemand die hem onderwijs wilde geven. Eindelijk ontmoette hij een heerschap dat hem in dienst wilde nemen en dan lezen en schrijven leren. De jongen nam dit graag aan.

(2) Toen hij een half jaar bij de heer had gediend, was hij al tamelijk gevorderd in de kunst. Nu beval de heer hem, dat hij eens moest opschrijven hoe men zich in allerlei dingen kan veranderen. Dit stond te lezen in een groot boek, dat de heer hem gaf. De jongen keek daar vreemd van op.

(3) Maar toen hij alles had opgeschreven, leerde hij het van buiten en toen hij alles in zijn hoofd had, kreeg hij zin om zich ook eens te veranderen. Hij veranderde zich in een koe, en ging in het weiland van een van de boeren liggen.

(4) Toen de boer de volgende morgen in het veld kwam, zag hij daar een vreemde koe. Er kwam niemand opdagen die zich als eigenaar van het dier liet gelden, en daarom zag de boer er geen been in met de koe naar de markt te gaan om haar te verkopen. Hij deed dat en verkocht haar voor een goede prijs aan een veehandelaar. Die ging met het beest aan een touw op weg naar zijn woonplaats.

(5) Maar toen ze een poos hadden gelopen, werd de koe halsstarrig en rukte zo krachtig aan het touw, dat het brak. Ze nam de vlucht naar een bosje, waar ze zich voor de koopman kon verschuilen, en veranderde daar weer in een jongen. De jongen liep de man tegemoet en toen deze hem vroeg of hij ook een koe had gezien, antwoordde hij: "Nee", en ging verder.

(6) Maar hij vond deze kunst zo mooi, dat hij zich kort daarna in een paard veranderde. Dit werd gekocht door de heer die de jongen het lezen en schrijven had geleerd. Hij had het dier nog niet lang op stal, of hij merkte wel dat het niemand anders was dan zijn vroegere leerling. Daarover was hij zeer boos, want hij wilde niet dat iemand buiten hem de kunst zou kennen.

(7) Hij nam het paard van stal en ging ermee naar de smid om het te laten beslaan. Toen dit karwei klaar was, zei de heer: "Baas, maak nu eens een groot stuk ijzer gloeiend en geef mijn paard daarmee een paar flinke brandmerken." Op die manier wilde hij de jongen straffen.

(8) De smid stak een groot ijzer in het vuur, maar voor hij daarmee klaar was, veranderde de jongen zich in een haas en rende weg zo hard hij kon. Op slag veranderde de heer zich nu in een hond en liep de haas achterna om die te vangen. Maar voor hij dit kon doen, veranderde de jongen zich in een vlieg.

(9) Hierop maakte de heer zich tot een zwaan en wilde zo de vlieg vangen, maar het lukte niet; de vlieg was te vlug. Boven een tuin gekomen veranderde de jongen zich in een gouden ring, die op de grond viel. Net op dat ogenblik wandelde daar een meisje, en zij vond de ring.

(10) De heer, die in een zwanenpak stak, veranderde zich nu in een koopman en vroeg het meisje of zij de ring wilde verkopen. "Nee," zei ze, en liet toen de ring per ongeluk vallen. De ring veranderde eensklaps in een gortkorrel, en de koopman werd een haan, die de gortkorrel wilde oppikken. Maar eer hij dit kon doen, veranderde de gortkorrel in een vos, die de haan de kop afbeet.

(11) Dit was de beloning, omdat de heer de jongen lezen en schrijven had geleerd.

A.4.2 Rhetorical relations

paragraph	# clauses	sentences
1	3	elaboration2, contrast
	1	
	3	elaboration2, temporal
	1	
2	2	temporal
	3	'dat', 'hoe'
	2	elaboration2
	1	
3	4	temporal, temporal, temporal
	2	temporal
4	2	temporal
	3	cause, purpose
	2	temporal
	1	
5	4	temporal, cause, cause
	3	elaboration2, temporal
	4	temporal, temporal, additive
6	2	cause
	1	
	2	temporal
	2	cause
7	3	temporal, purpose
	2	temporal
	" ... "	
	1	
8	4	contrast, temporal, additive
	3	temporal, purpose
	2	temporal
9	4	purpose, contrast, cause
	2	elaboration2
	2	temporal
10	3	elaboration2, temporal
	2	temporal
	3	temporal, elaboration2
	3	temporal, elaboration2
11	2	cause

Table A.7: Rhetorical relations used in 'De jongen die lezen en schrijven leerde'

A.4.3 Referring expressions

referent	paragraph	referring expressions
jongen	1	een jongen, hij, hij, hem, hij, hem, de jongen
	2	hij, hij, hem, hij, hem, de jongen
	3	hij, hij, hij, hij, hij
	4	een vreemde koe, het dier, de koe, haar, haar, het beest
	5	de koe, ze, ze, de jongen, hem, hij, hij
	6	hij, hij, de jongen, het dier, zijn vroegere leerling
	7	het paard, er(mee), het, de jongen
	8	de jongen, hij, de haas, die, de jongen
	9	de vlieg, de vlieg, de jongen, de ring
	10	de ring, de ring, de ring, de gortkorrel, de gortkorrel
	11	de jongen
heer	1	een heerschap
	2	de heer, de heer, de heer
	6	de heer, hij, hij, zijn, hij, hij, hem
	7	hij, de heer, hij
	8	de heer, hij
	9	de heer
	10	de heer, de koopman, hij, de haan
	14	de heer
veehandelaar	4	een veehandelaar, die, zijn
	5	de koopman, de man, deze
smid	7	de smid
	8	de smid, hij
meisje	9	een meisje, zij
	10	het meisje, zij, ze

Table A.8: Referring expressions used in 'De jongen die lezen en schrijven leerde'

Appendix B

The Plop Example

First of all, this appendix shows the input fabula structure for the Plop story. Furthermore it shows all dependency trees after generating the sentence plans and performing the first step of the lexicalization. Finally it gives all dependency trees after performing syntactic aggregation and referring expression generation.

B.1 Fabula Structure

Figure B.1 shows the relevant part of the fabula structure, the input to the Narrator agent. The fabula is represented as one large OWL file, and this file contains all plot elements (the normal plot elements as well as the sub plot elements), all of their arguments and finally the relations that hold between the different plot elements.

Take for example the final plot element in the fabula structure which represents the action of eating the apple, which is the sub plot element of the belief that the action of eating the apple has taken place successfully:

```
<fabula:Eat rdf:about="http://www.owl-ontologies.com/FabulaKnowledge.owl#ind_Eat.plop.15">
   <fabula:hasContext rdf:resource="http://www.owl-ontologies.com/FabulaKnowledge.owl#ind_BeliefElement.plop.14"/>
   <fabula:agens rdf:resource="http://www.owl-ontologies.com/StoryWorldCore.owl#ind_Humanoid.plop.1"/>
   <fabula:patiens rdf:resource="http://www.owl-ontologies.com/StoryWorldCore.owl#ind_FruitOrVegetable.plop.6"/>
   <fabula:sameConcept rdf:resource="http://www.owl-ontologies.com/FabulaKnowledge.owl#ind_Eat.plop.13"/>
</fabula:Eat>
```

In this example you can see that the second line tells that this plot element is the sub plot element of the Belief plot element. Furthermore the third and fourth line specify the agens and the patiens of the action (in this example is the dwarf represented as 'humanoid' object and the apple as a 'fruit or vegetable' object).

B.2 Dependency Trees after Lexicalization

The figures in this appendix show the dependency trees for each of the plot elements after generating a sentence plan and performing the first part of the lexicalization. The rhetorical dependency graph that is passed to the Surface Realizer has the same structure as the document plan shown in figure 6.13, but the plot elements have been replaced by the dependency trees given in this appendix.

B.3 Dependency Trees after Referring Expression Generation

The figures in this appendix show the dependency trees after performing syntactic aggregation and referring expression generation. These dependency trees are the final dependency trees that are transformed into text by the Surface Form Generator.

```
<rdf:RDF
  xmlns:swc="http://www.owl-ontologies.com/StoryWorldCore.owl#"
  xmlns:fabula="http://www.owl-ontologies.com/FabulaKnowledge.owl#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <rdf:Description rdf:about="http://www.owl-ontologies.com/FabulaKnowledge.owl#ind_BeliefElement.plop.3">
    <fabula:psi_causes>
      <swc:AttainGoal rdf:about="http://www.owl-ontologies.com/StoryWorldCore.owl#ind_AttainGoal.plop.8">
        <fabula:motivates>
          <fabula:Eat rdf:about="http://www.owl-ontologies.com/FabulaKnowledge.owl#ind_Eat.plop.11">
            <fabula:phi_causes>
              <fabula:See rdf:about="http://www.owl-ontologies.com/FabulaKnowledge.owl#ind_See.plop.12">
                <fabula:psi_causes>
                  <fabula:BeliefElement rdf:about="http://www.owl-ontologies.com/FabulaKnowledge.owl#ind_BeliefElement.
                    plop.14">
                    <fabula:psi_causes>
                      <fabula:Outcome rdf:about="http://www.owl-ontologies.com/FabulaKnowledge.owl#ind_Outcome.plop.16">
                        <fabula:character rdf:resource="http://www.owl-ontologies.com/StoryWorldCore.owl#ind_Humanoid.
                          plop.1" rdf:type="http://www.owl-ontologies.com/StoryWorldCore.owl#Humanoid"/>
                        <fabula:resolves rdf:resource="http://www.owl-ontologies.com/StoryWorldCore.owl#ind_AttainGoal.
                          plop.8"/>
                      </fabula:Outcome>
                    </fabula:psi_causes>
                    <fabula:character rdf:resource="http://www.owl-ontologies.com/StoryWorldCore.owl#ind_Humanoid.plop.1"/>
                  </fabula:BeliefElement>
                </fabula:psi_causes>
                <fabula:character rdf:resource="http://www.owl-ontologies.com/StoryWorldCore.owl#ind_Humanoid.plop.1"/>
              </fabula:See>
            </fabula:phi_causes>
            <fabula:agens rdf:resource="http://www.owl-ontologies.com/StoryWorldCore.owl#ind_Humanoid.plop.1"/>
            <fabula:patiens>
              <swc:FruitOrVegetable rdf:about="http://www.owl-ontologies.com/StoryWorldCore.owl#ind_FruitOrVegetable.
                plop.6">
                <fabula:hasContext rdf:resource="http://www.owl-ontologies.com/FabulaKnowledge.owl#ind_BeliefElement.
                  plop.3"/>
                <swc:isLocated>
                  <swc:GeographicArea rdf:about="http://www.owl-ontologies.com/StoryWorldCore.owl#ind_GeographicArea.
                    plop.7">
                    <fabula:hasContext rdf:resource="http://www.owl-ontologies.com/FabulaKnowledge.owl#ind_BeliefElement.
                      plop.3"/>
                    <fabula:hasInterpretation rdf:resource="http://www.owl-ontologies.com/StoryWorldCore.owl#ind_
                      GeographicArea.plop.5" rdf:type="http://www.owl-ontologies.com/StoryWorldCore.owl#GeographicArea"/>
                  </swc:GeographicArea>
                </swc:isLocated>
                <fabula:hasInterpretation rdf:resource="http://www.owl-ontologies.com/StoryWorldCore.owl#ind_
                  FruitOrVegetable.plop.4" rdf:type="http://www.owl-ontologies.com/StoryWorldCore.owl#FruitOrVegetable"/>
              </swc:FruitOrVegetable>
            </fabula:patiens>
          </fabula:Eat>
        </fabula:motivates>
        <fabula:motivates>
          <fabula:TakeFrom rdf:about="http://www.owl-ontologies.com/FabulaKnowledge.owl#ind_TakeFrom.plop.10">
            <fabula:agens rdf:resource="http://www.owl-ontologies.com/StoryWorldCore.owl#ind_Humanoid.plop.1"/>
            <fabula:patiens rdf:resource="http://www.owl-ontologies.com/StoryWorldCore.owl#ind_FruitOrVegetable.plop.6"/>
          </fabula:TakeFrom>
        </fabula:motivates>
        <fabula:character rdf:resource="http://www.owl-ontologies.com/StoryWorldCore.owl#ind_Humanoid.plop.1"/>
      </swc:AttainGoal>
    </fabula:psi_causes>
    <fabula:character rdf:resource="http://www.owl-ontologies.com/StoryWorldCore.owl#ind_Humanoid.plop.1"/>
  </rdf:Description>
  <fabula:Eat rdf:about="http://www.owl-ontologies.com/FabulaKnowledge.owl#ind_Eat.plop.9">
    <fabula:agens rdf:resource="http://www.owl-ontologies.com/StoryWorldCore.owl#ind_Humanoid.plop.1"/>
    <fabula:patiens rdf:resource="http://www.owl-ontologies.com/StoryWorldCore.owl#ind_FruitOrVegetable.plop.6"/>
    <fabula:hasContext rdf:resource="http://www.owl-ontologies.com/StoryWorldCore.owl#ind_AttainGoal.plop.8"/>
  </fabula:Eat>
  <rdf:Description rdf:about="http://www.owl-ontologies.com/FabulaKnowledge.owl#ind_Hunger.plop.2">
    <fabula:psi_causes rdf:resource="http://www.owl-ontologies.com/StoryWorldCore.owl#ind_AttainGoal.plop.8"/>
    <fabula:character rdf:resource="http://www.owl-ontologies.com/StoryWorldCore.owl#ind_Humanoid.plop.1"/>
  </rdf:Description>
  <fabula:Eat rdf:about="http://www.owl-ontologies.com/FabulaKnowledge.owl#ind_Eat.plop.13" fabula:isSuccessful="true">
    <fabula:hasContext rdf:resource="http://www.owl-ontologies.com/FabulaKnowledge.owl#ind_See.plop.12"/>
    <fabula:agens rdf:resource="http://www.owl-ontologies.com/StoryWorldCore.owl#ind_Humanoid.plop.1"/>
    <fabula:patiens rdf:resource="http://www.owl-ontologies.com/StoryWorldCore.owl#ind_FruitOrVegetable.plop.4"/>
  </fabula:Eat>
  <fabula:Eat rdf:about="http://www.owl-ontologies.com/FabulaKnowledge.owl#ind_Eat.plop.15">
    <fabula:hasContext rdf:resource="http://www.owl-ontologies.com/FabulaKnowledge.owl#ind_BeliefElement.plop.14"/>
    <fabula:agens rdf:resource="http://www.owl-ontologies.com/StoryWorldCore.owl#ind_Humanoid.plop.1"/>
    <fabula:patiens rdf:resource="http://www.owl-ontologies.com/StoryWorldCore.owl#ind_FruitOrVegetable.plop.6"/>
    <fabula:sameConcept rdf:resource="http://www.owl-ontologies.com/FabulaKnowledge.owl#ind_Eat.plop.13"/>
  </fabula:Eat>
</rdf:RDF>
```

Figure B.1: Example of the fabula structure for the Plop story in OWL representation

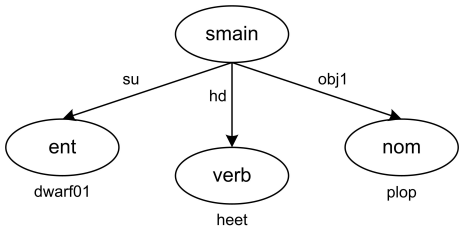

Figure B.2: Generated dependency tree for the plot element describing the dwarf's name

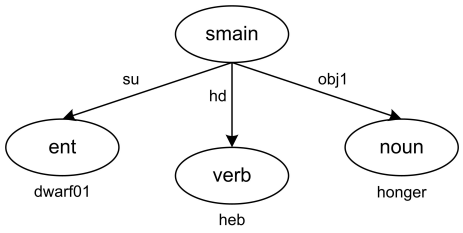

Figure B.3: Generated dependency tree for the plot element Hungry

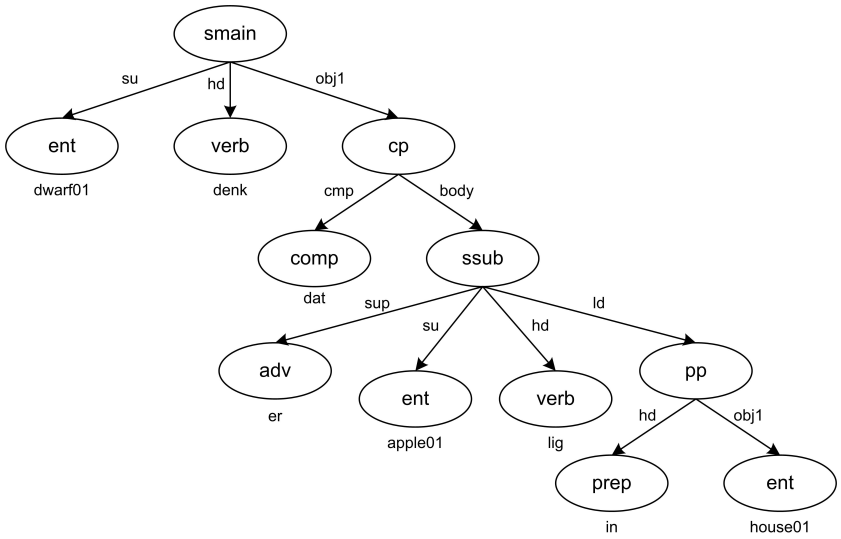

Figure B.4: Generated dependency tree for the plot element Belief

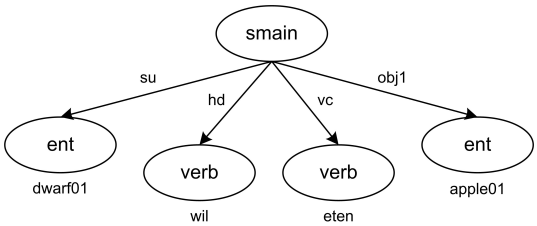

Figure B.5: Generated dependency tree for the plot element Goal

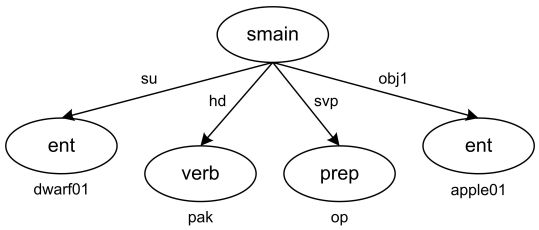

Figure B.6: Generated dependency tree for the plot element Take apple

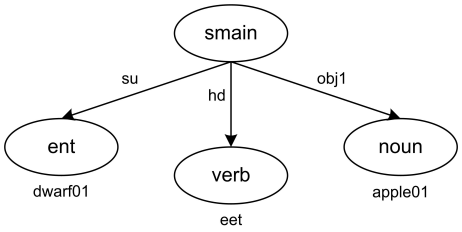

Figure B.7: Generated dependency tree for the plot element Eat apple

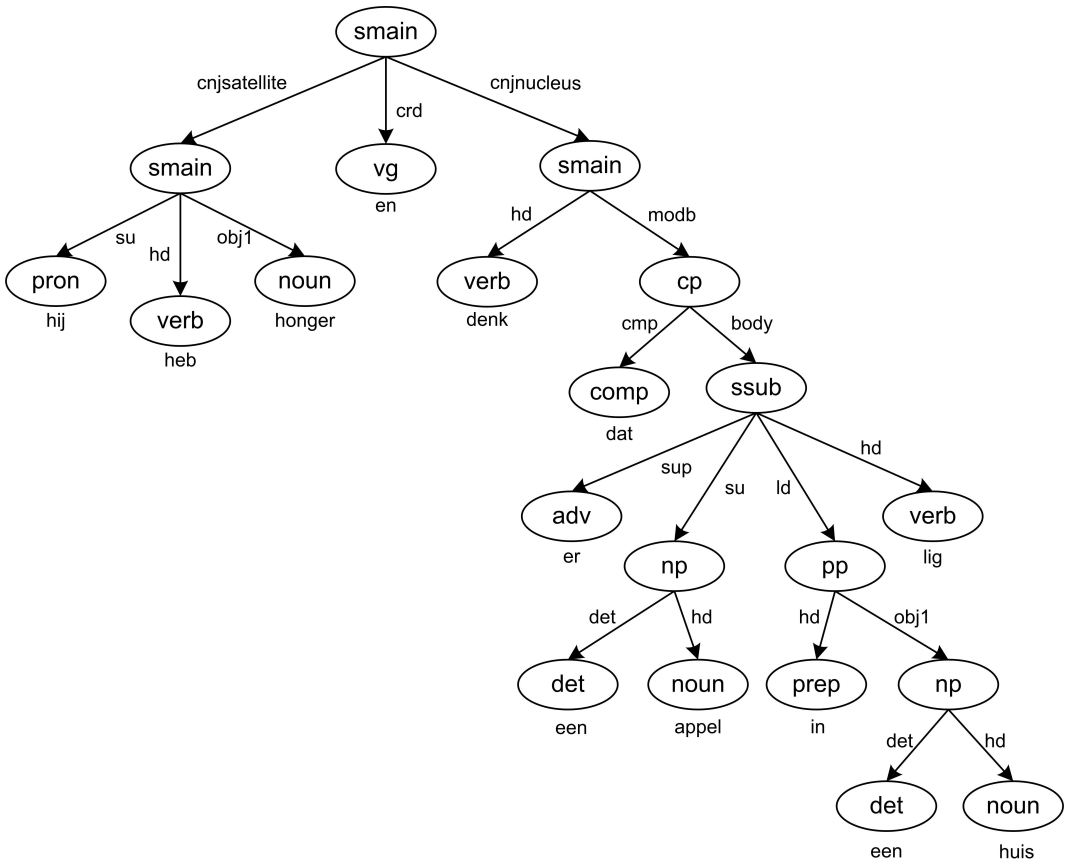

Figure B.8: Generated dependency tree for “Hij had honger en dacht dat er een appel in een huis lag.”

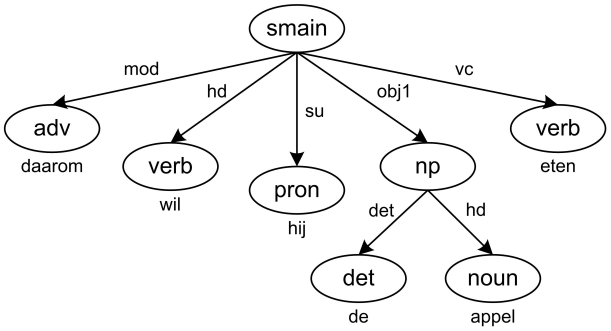

Figure B.9: Generated dependency tree for "Daarom wilde hij de appel eten."

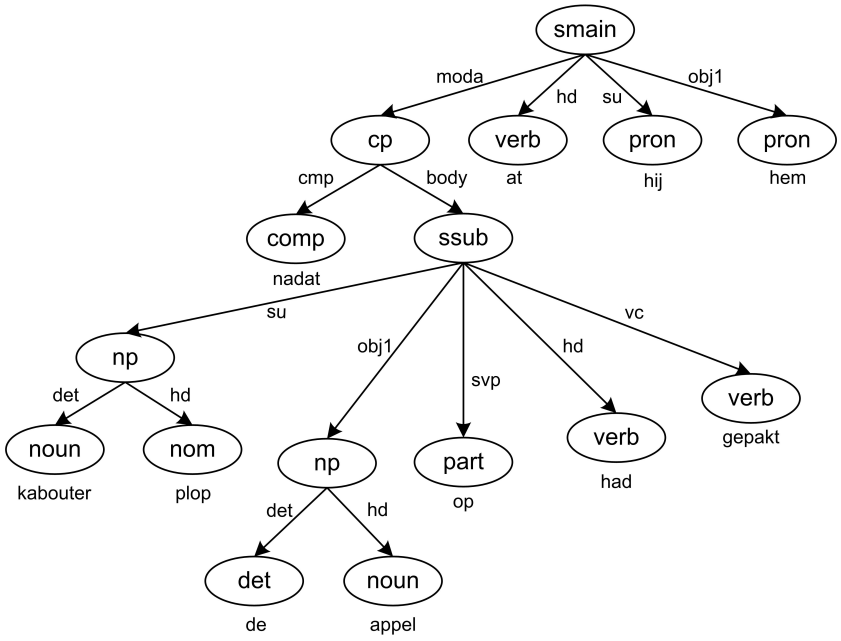

Figure B.10: Generated dependency tree for "Nadat kabouter Plop de appel op had gepakt, at hij hem."

Appendix C

Pronominalization

The generation of referring expressions is an important task of a natural language generation system. There are different ways to refer to an entity; first of all, it can be done using a pronoun, but it can also be done using a noun phrase (possibly containing the entity's name). When generating referring expressions it is important that the system does not generate too many pronouns, but not too many noun phrases either. The use of too many pronouns may lead to confusion which entity is being referred to, while using too many noun phrases may lead to a less coherent text or even to ungrammatical sentences. An important decision is therefore when to use a pronoun and when to use a noun phrase.

Reiter and Dale [RD00] claim that all pronominalization algorithms make mistakes which can be characterized as either *missed* pronouns or as *inappropriate* pronouns. In many algorithms there is a trade-off between these types of mistakes. An algorithm which generates only few inappropriate pronouns is called a conservative algorithm. An example of such an algorithm is only using a pronoun when the antecedent has been mentioned in the previous utterance and there is no other entity with the same person, gender and number.

Apart from this very simple algorithm numerous more complicated algorithms have been designed. The main focus of this paper is therefore to discuss a number of these algorithms (section C.1). Note that some of the algorithms are originally designed to *resolve* pronouns, but these can also be used for *generating* pronouns (possibly after some modification). Section C.2 describes the advantages of the algorithms and then tries to combine them into one algorithm.

C.1 Existing Theories and Algorithms

Many algorithms have been designed which decide when to use a pronoun and when to use a noun phrase. A number of these algorithms are described in this section.

C.1.1 Centering Theory

Grosz et al. [GJW95] claim that certain entities mentioned in an utterance are more central than others. In Centering Theory each utterance has a single *backward-looking center* $C_b(U)$ that represents the entity currently being focussed on after the sentence is interpreted. Furthermore every utterance has a set of *forward-looking centers* $C_f(U)$ consisting of all entities mentioned in the utterance, ordered by their grammatical role. The set of forward looking centers is thus a partially ordered list containing the entities which can serve as the backward looking center of the following utterance. The $C_b(U_n)$ is then the highest-ranked member of $C_f(U_{n-1})$ that is also realized in U_n. Finally the first element of the set of forward looking centers is the *preferred center* $C_p(U)$.

The elements of the forward looking center of an utterance are ordered by their grammatical roles as follows: subject → direct object → indirect object → others. This implies that Centering Theory prefers antecedents appearing in subject position to other possible antecedents.

Centering Theory furthermore states that the $C_b(U_{n+1})$ must be realized as a pronoun, if any element of $C_f(U_n)$ is realized as a pronoun in U_{n+1}. This also implies that no element in an utterance can be realized as a pronoun *unless* the backward-looking center of the utterance is also realized as a pronoun. Centering Theory was originally meant for language processing, but Kibble [KP00] proved that it can also be applied to language generation. Part of his study was the application of Centering Theory to the generation of referring expressions which took form in the decision on pronominalization. The result of his study was that a $C_b(U_n)$ must be pronominalized if $C_b(U_n)$ equals $C_b(U_{n-1})$ or if $C_b(U_n)$ equals $C_p(U_{n-1})$. As an example Kibble presents the following text containing both cases in which the $C_b(U_n)$ must be pronominalized:

1. John has had trouble arranging his vacation.

2. He cannot find anyone to take over his responsibilities. (C_b is John)

3. He called up Mike yesterday to work out a plan. (C_b is John, so $C_b(U_n)$ equals $C_b(U_{n-1})$ and must be pronominalized)

4. Mike has annoyed him a lot recently. (C_b is John, so $C_b(U_n)$ equals $C_b(U_{n-1})$ and must be pronominalized)

5. He called John at 5 am on Friday last week. (C_b is Mike, so $C_b(U_n)$ equals $C_p(U_{n-1})$ and must be pronominalized)

Kibble's result of pronominalization implies that no entity will be pronominalized if $C_b(U_n)$ cannot be pronominalized, but it is unclear whether several entities can be pronominalized when the $C_b(U_n)$ can be pronominalized.

C.1.2 Thread-based

Centering Theory bases its decision whether to pronominalize on the fact if a referent is *accessible*. McCoy and Strube [MS99] on the other hand found that in real texts many definite descriptions are used when pronouns could have been used as well. Therefore they state that generating appropriate referring expressions requires looking at more factors than just accessibility. In their research they have mainly focussed on New York Times news articles, but they believe that their ideas can be generalized to other types of text genres as well.

Like many algorithms the algorithm designed by McCoy and Strube does take distance into account; in long distance situations (in which the last reference to an entity was more than two sentences ago) a definite description is usually used, and in short distance situations (in which the last reference to an entity was in the same sentence) a pronoun is usually used. In other situations than these they look at the threaded structure of their texts. A thread describes a particular part of the story and can be interrupted by other threads and continued later. Possible indications for new threads are a shift of topic, a shift of time scale, a shift in spatial scale or a shift in perspective. Since McCoy and Strube only focussed on news paper articles they concentrated on the time-threaded discourse structure. Using these threads they hypothesize that definite descriptions should appear when the current reference to a discourse entity is in a different thread from the last reference to that entity, and pronouns should occur when the previous mention is in the same thread. Evaluation of this hypothesis showed that all definite noun phrases used in the texts can be explained by these changes in time.

Finally McCoy and Strube agree that the choice of referring expressions is also influenced by possible ambiguities. Some existing algorithms state that a referring expression is ambiguous if there is a competing antecedent in the previous or in the current sentence. McCoy and Strube found however that half of the ambiguous referring expressions were still realized as a pronoun. In order to solve this problem they turned to pronoun *resolution* algorithms and Strube's algorithm [Str98] in particular. The algorithm then chooses to generate a pronoun to refer to a particular entity if Strube's algorithm would choose that entity as the referent for the pronoun.

C.1.3 Local focus

In Centering Theory the backward looking center is often considered as the local focus of attention. Henschel, Cheng and Poesio [HCP00] agree with Centering Theory that the local focus is the entity that is most likely to be pronominalized, but they also found that many C_b's in real texts are not pronominalized, and secondly, that some non-C_b referents are pronominalized. They refer to the algorithm of McCoy and Strube as an algorithm that partly solves these problems. Henschel et al. however, focussed on descriptive texts (such as museum catalogues) rather than news paper articles, so in their texts there are no time changes. Therefore they propose a new algorithm that is also based on Centering Theory, but using a different definition of local focus. Furthermore their algorithm makes use of parallelism and the discourse status of the antecedent.

The discourse status of an entity is either *discourse-old* (if it has been mentioned before in the discourse), or *discourse-new* (otherwise). Obviously, an entity can only be pronominalized if it is discourse-old, but analysis shows that there is also a preference for *antecedents* which are discourse-old: 66% of the pronouns refers to a discourse-old antecedent. Furthermore the results of the corpus analysis showed a strong correlation between pronoun use and the grammatical function of the antecedent; 63% of the pronouns had a subject as antecedent.

Based on these results Henschel et al. give a new definition of *local focus* (the set of referents which are available for pronominalization): the local focus is the set of referents of the previous utterance that are either discourse-old or realized as subject. Quite often this set contains only one referent, the C_b used in Centering Theory. This new definition of local focus implies that newly introduced referents are never referred to by a pronoun immediately, unless they have been introduced as subject.

Using this definition of local focus the algorithm generated 91% of all short-distance pronouns used in their corpus correctly. After examining the other pronouns it turned out that they appeared in *strong parallel* contexts. A referring expression is parallel if its antecedent has the same grammatical function as the expression itself. If there is subject parallelism and object parallelism at the same time, this is called strong parallelism and in these cases the object may also be pronominalized.

Finally they solve the problem of ambiguity by stating that referents of the previous utterance which are not in the local focus do not disturb pronominalization, even if they have the same gender and number as the referent for which a referring expression is to be generated.

C.1.4 Parallelism

In Centering Theory there is a preference for antecedents in subject position, but Chambers and Smyth [CS98] found that there can also be a preference for parallelism. This means that antecedents appearing in the same grammatical role as the pronoun are preferred. Additional requirements are that both sentences have the same global constituent structure and that the thematic roles of the verbs in both sentences are the same.

Take for example the following sentence: "Jim surprised Paul and then Julie shocked him." Centering theory would prefer 'Jim' to be the antecedent of 'him' because he appears in subject position. The parallelism preference on the other hand prefers 'Paul' to be the antecedent, because 'Jim' and 'Julie' are parallel, and the verbs 'surprised' and 'shocked' have the same syntactic structure, so 'Paul' and 'him' should co-refer. In this example the parallelism preference thus chooses the correct referent.

The sentence "Jim tried to catch Paul and Julie bored him" shows that the parallelism effect is reduced when the verbs do not have the same syntactic features; the objects in the sentence differ in thematic roles, so the preference for parallelism is less obvious.

C.1.5 Plausibility

The algorithms and theories described sofar all looked at syntactic features of the antecedents. There are however many examples in which the *semantic* context is relevant too. An example of such a situation is the following:

1. John blamed Bill because *he* spilt the coffee.

2. John confided in Bill because *he* stole the money.

In both sentences the syntactic features of the possible antecedents in the first clause and the syntactic features of the pronoun in the second clause are the same. However, in the first sentence the pronoun 'he' refers to 'Bill' and in the second sentence it refers to 'John'. Both Centering and Parallelism would prefer 'he' to refer to 'John' in both sentences, so the only way to resolve the pronoun in the first sentence correctly is by looking at the meaning of the second clause.

Ehrlich [Ehr80] set up an experiment in which she tested whether the pronouns used in complex sentences were resolved correctly. In her experiment she focussed on sentences consisting of two clauses connected by different rhetorical relations (using the conjunction 'because', 'but' or 'and'). First of all she distinguished between NP_1 verbs which impute cause to the subject (such as 'to confide'), and NP_2 verbs which impute cause to the object (such as 'to blame'). Using these definitions she found that a sentence in which the first clause contains an NP_1 verb and the second clause contains a pronoun, this pronoun most likely refers to the subject of the first clause; with an NP_2 verb the pronoun most likely refers to the object of the first clause. Both examples can be explained by this categorization of verbs, but she also found that this is not a necessary requirement. Her experiment showed that sentences in which the subordinate clause contains information that is inconsistent with this categorization of verbs, did take longer to process, but were also interpreted correctly. An example is the sentence "John confided in Bill because he was understanding" (in which 'he' refers to 'Bill'). Even though this example is inconsistent with the idea that the pronoun 'he' most likely refers to the subject because 'to confide' is an NP_1 verb, the participants did select the correct referent. Therefore Ehrlich states that the reader also uses some general knowledge in order to choose the correct referent. She mainly looked at causal relations and she believes that pronouns are less easily resolved when the sentences are connected by other rhetorical relations.

Ehrlich thus states that readers first examine antecedents for features such as gender. If there is more than one possible antecedent, the reader will not only look at syntactic features or features of the main verb, but will also use his general knowledge to select a referent that is *plausible*. In most cases this is satisfactory, but if there are still several plausible antecedents the reader will retrieve a referent by analyzing the sentence more structurally, for example by looking at parallelism or syntactic roles.

C.1.6 Rhetorical relations

Kehler [Keh02] believes that Centering Theory, Parallelism Preference and the Plausibility Theory are all useful theories, but that they only apply in particular cases. More specifically he believes that each of the algorithms applies in *different* cases and that the choice of algorithm solely depends on the rhetorical relation used in the sentence.

First of all Kehler thinks a text is coherent when a hearer or reader can establish a relation between two consecutive utterances. He mainly focusses on complex sentences and the relation between the two clauses *in* the sentence. Moreover he believes there are only three classes of coherence relations: resemblance, cause-effect and contiguity.

A *resemblance* relation can be further subdivided into the following relations: parallel, contrast, exemplification, generalization, exception and elaboration. All of these relations are self-explanatory, but I will give some examples of the other relations. The *cause-effect* class can be subdivided into the following relations:

- Result: George is a politician, and therefore he's dishonest.
- Explanation: George is dishonest because he's a politician.
- Violated expectation: George is a politician, but he's honest.
- Denial of preventer: George is honest, even though he's a politician.

Finally the *contiguity* class only consists of the occasion relation which is somewhat less obvious than the other relations. Kehler gives two versions of this relation: one in which the final state of the first sentence can be inferred from the second sentence, and one in which the initial state of the second sentence can be inferred from the first sentence. As an example he gives the following text (somewhat simplified): "A campaign bus arrived in Iowa. Soon afterward, Bush gave his first speech of the season". In this example

assumptions are required that allow the final state of the first sentence to be identified as the initial state of the second sentence; the reader should infer that Bush went to Iowa by bus to give a speech.

Using these three classes of rhetorical relations Kehler says that each rhetorical relation is associated with one of the three theories of pronoun interpretation. To support this idea he gives the following examples:

1. Margaret Thatcher admires Hillary Clinton, and George W. Bush absolutely worships *her*.

2. The city council denied the demonstrators a permit because *they* advocated violence.

3. John hit Bill. Mary told *him* to go home.

The clauses in the first sentence are combined by a resemblance relation and therefore Kehler believes the pronoun resolution in this sentence can be explained by the parallelism preference. The second sentence is connected by a cause-effect relation, so he states the resolution can be explained by the plausibility theory. Finally the third sentence consists of a contiguity relation and can be explained by the syntactic hierarchy preference as in centering theory. Kehler admits there are exceptions, but he believes that the kind of rhetorical relation determines how a pronoun can be resolved.

C.1.7 Salience-based

There are some algorithms that determine which entity is the most likely antecedent based on *saliency*. In such algorithms a so-called *stock of shared knowledge* is kept containing all entities mentioned sofar in the discourse [KKH97]. Each entity then gets a salience value which represents how likely a pronoun refers to the entity. A rather simple example of an algorithm which uses salience values is the algorithm of Hajičová [Haj93]. In this algorithm each entity gets a *degree of activation*, so in this algorithm the entities get *lower* scores as they become more salient. The degrees of activation are based on the distance and on the position of the entity in the sentence. Another algorithm is the algorithm of Lappin and Leass [LL94] in which more salient entities have *higher* scores than less salient entities. Furthermore the algorithm combines a number of different preferences in one algorithm, such as the preference for grammatical role (like Centering Theory) and the preference for parallelism.

These algorithms were originally meant to resolve pronouns, but they can also be used as an algorithm for deciding when to pronominalize. Each time a referring expression has to be generated the algorithm simply checks if the antecedent is the entity with the highest salience value of the same person and gender (or the entity with the lowest degree of activation). If this is the case the standard resolution algorithm would resolve the pronoun correctly, so a pronoun can be used.

C.1.8 Probabilistic model

The final work we will discuss is the probabilistic model of Strube and Wolters [SW00] which is especially meant to be genre-independent. In order to design this model they analyzed twelve texts from four different genres. In this analysis they investigated nine different factors which can be subdivided into np-level factors and co-specification-level factors.

The *np-level factors* depend on the noun phrase to be constructed. They include agreement in person, gender and number, syntactic function and sortal class. Sortal classes are used to provide information about the discourse entity for which a referring expression is to be generated. Examples are person, group, physical object, concept, location, time, event, action, state and property.

The *co-specification factors* depend on sequences of referring expressions which co-specify with each other. The following factors are used: syntactic function of the antecedent, form of the antecedent, distance to last mention, parallelism and ambiguity.

First of all Strube and Wolters found a number of rather obvious effects. They found that masculine and feminine noun phrases are pronominalized more frequently than neuter and plural noun phrases. Furthermore they found that there is a preference for short distance to the antecedent, antecedents in subject position and parallelism. Finally they found that referring expressions that co-specify with an antecedent possessive pronoun are more likely to be pronominalized.

Deadend entities are entities mentioned only once, so there are no subsequent references to such an entity. Using this definition Strube and Wolters found that the sortal classes can be subdivided into three groups: Person/Group with the lowest rate of deadend entities and the highest percentage of pronouns, Location/PhysicalObject with about 65% deadend entities and a much lower pronominalization rate, and Concept/Action/Event/Property/State/Concept, with over 80% deadend entities.

They conclude that distance to last mention and agreement are the most important factors and to a certain extent the form of the antecedent. Using these factors they can build a probabilistic model that predicts about 93% of all pronouns correctly.

C.2 Combination of the Algorithms

In the previous section a number of algorithms have been described which can be used to decide whether a referring expression can be pronominalized. In this section I will point to the advantages of these algorithms and I will try to combine them into one new algorithm which can then be used in the Virtual Storyteller.

C.2.1 Advantages of the algorithms

McCoy & Strube (described in section C.1.2) and Henschel et al. (described in section C.1.3) both use a simple if-then-else algorithm to decide whether to pronominalize. I will simply take these algorithms as starting point and will try to extend them to include some ideas from the other algorithms as well.

Centering Theory (see section C.1.1) determines the backward looking center and the set of forward looking centers of each sentence, and bases its decision for pronominalization on these two variables. This means that the algorithm never looks back more than one sentence. In real texts however, there are many references to entities which were used two sentences before. Therefore our algorithm will keep a history with all entities mentioned in the discourse sofar.

The Parallelism preference (see section C.1.4) can be included by pronominalizing entities which appear in strong parallelism either with the previous utterance or with the first clause in the same utterance in case of a complex sentence. The algorithm will therefore check if the subject and the object of the previous utterance are the same as the subject and the object of the current utterance. If this is the case the algorithm will pronominalize both references in the second clause. This means that if a complex sentence contains the same two male characters in both clauses, both references in the second clause can be pronominalized even if an algorithm which uses salience values would pronominalize only one reference. An example is the sentence "John punched Bill, because he didn't like him". The algorithm currently used by the Virtual Storyteller would only pronominalize the reference to John (based on saliency), but the new algorithm will pronominalize both references in the second clause.

The Plausibility theory described by Ehrlich (see section C.1.5) and Kehler (see section C.1.6) can be included in complex sentences in which the clauses are connected by a causal relationship. Kehler focussed on pronoun *resolution*, so he did not explicitly state how his ideas can be used in referring expression generation algorithms. I do believe however that all references in the second clause of a sentence that also appear in the first clause of the sentence can be pronominalized when the clauses are connected by a causal relation. Since it is obvious that there is a causal relation the reader automatically tries to make the sentence as plausible as possible. Of course there are exceptions to this rule (sentences in which the interpretation is not as plausible as possible), but since these situations are rare I will ignore them.

Our algorithm will still use saliency to determine whether to pronominalize in other cases than the ones described sofar (see section C.1.7) and it uses the algorithm of Lappin and Leass for this. Finally Strube and Wolters (see section C.1.8) showed that distance to last mention and agreement were the most important factors that affected pronominalization, but these are already present in our algorithm, so we will simply ignore their model.

C.2.2 The combined algorithm

The final algorithm is shown in algorithm 8. Its input is the referent r for which a referring expression is to be generated and the algorithm returns $true$ if a pronoun can be used and $false$ otherwise. The algorithm

starts by determining the distance to the last reference; if r is the first reference in the current paragraph or the referent has not been mentioned for two sentences the algorithm returns $false$.

Then the algorithm checks if r has already been mentioned in the current sentence. If this is not the case it returns $true$ if there is strong parallelism with the previous sentence and it returns $false$ in cases of a change of thread (if this can be detected by the referring expression generator). Otherwise the algorithm bases its decision on the referent's salience value.

If the referent has been mentioned in the current sentence, it returns $true$ if the clauses are connected by a causal relationship or if the referent appears in strong parallelism with the first clause of the sentence. Otherwise the algorithm bases its decision on the referent's salience value again.

Algorithm 8 Pronominalize(r)

```
if first reference to r in current paragraph
   or antecedent has not been mentioned for two sentences then
   return false
end if
if r has not been mentioned in current sentence then
   if strong parallelism with previous sentence then
      return true
   else if change of thread then
      return false
   end if
else
   if r appears in causal relation or strong parallelism with first clause then
      return true
   end if
end if
if r has highest salience value then
   return true
else
   return false
end if
```