Improving the Efficiency and Capabilities of Document Structuring

Robert Marshall

October 2007

Submitted in total fulfilment of the requirements of the degree of Master of Science

Department of Computer Science and Software Engineering
The University of Melbourne
Victoria, Australia

Produced on archival-quality paper
Abstract

Natural language generation (NLG), the problem of creating human-readable documents by computer, is one of the major fields of research in computational linguistics.

The task of creating a document is extremely common in many fields of activity. Accordingly, there are many potential applications for NLG - almost any document creation task could potentially be automated by an NLG system. Advanced forms of NLG could also be used to generate a document in multiple languages, or as an output interface for other programs, which might ordinarily produce a less-manageable collection of data. They may also be able to create documents tailored to the needs of individual users.

This thesis deals with document structure, a recent theory which describes those aspects of a document’s layout which affect its meaning. As well as its theoretical interest, it is a useful intermediate representation in the process of NLG.

There is a well-defined process for generating a document structure using constraint programming. We show how this process can be made considerably more efficient. This in turn allows us to extend
the document structuring task to allow for summarisation and finer control of the document layout.

This thesis is organised as follows. Firstly, we review the necessary background material in both natural language processing and constraint programming. We then describe the implementation of our system, including a collection of constraints which specify the layout of the document, and an extension of the basic document structurer to enable summarisation. We also describe a number of strategies from constraint logic programming which we use in order to improve the efficiency of our system. Finally, we report on the results of our experimental work, and close by discussing our findings and identifying issues for future investigation.
Declaration

This is to certify that:

(i) the thesis comprises only my original work towards the Masters except where indicated in the Preface,

(ii) due acknowledgement has been made in the text to all other material used,

(iii) the thesis is approximately 30,000 words in length, exclusive of tables, maps, bibliographies and appendices.

Signed,

______________________________
Robert Marshall
17th October 2007
Acknowledgements

I would like to thank my supervisors, Steven Bird and Peter Stuckey, for reading countless drafts and providing useful guidance over the course of my candidature. Donia Scott provided some helpful suggestions in the early stages of this work and the Language Technology group at the University of Melbourne gave excellent feedback on a draft. I was supported by an Australian Postgraduate Award during my course, so I would like to thank the University of Melbourne for awarding it to me and to the Australian Government and taxpayers for funding it. I would also like to acknowledge my anonymous reviewers for their helpful suggestions on various aspects of this thesis. Finally, I would like to acknowledge my family for their understanding and support towards a perpetually distracted and nocturnal graduate student.
Contents

1 Introduction
  1.1 Natural language generation ........................................... 1
  1.2 Document structure ..................................................... 3
  1.3 Contribution ............................................................. 5
  1.4 Overview ................................................................. 7

2 Language technology background ........................................ 9
  2.1 Natural language generation ............................................ 10
    2.1.1 The consensus architecture of Reiter and Dale ................. 11
    2.1.2 Reference architecture for generation systems ............... 16
  2.2 Rhetorical structure .................................................. 19
  2.3 Document structure .................................................... 22
    2.3.1 Formal theory of document structure .......................... 22
    2.3.2 Discourse connectives ............................................ 24
    2.3.3 Generating a document structure ............................... 26
    2.3.4 Scoring document structures ................................... 28
    2.3.5 Summary .............................................................. 33
  2.4 Document summarisation ................................................. 34
  2.5 Corpora ................................................................. 39
  2.6 Summary ................................................................. 40
# CONTENTS

3 Constraint programming 43

3.1 Introduction to constraint programming 44
3.2 Formal definition of constraint programming 45
3.3 Constraint logic programming 50
3.3.1 Reified constraints 50
3.3.2 Labelling 51
3.4 Improving search performance 54
3.4.1 Caching search states 58
3.5 Constraint programming in natural language processing 59
3.6 Summary 61

4 Architecture 63

4.1 Overview 63
4.2 Implementing a document structurer 65
4.2.1 Modifications to the document structuring process 66
4.2.2 Constraint model 71
4.3 Realisation 79
4.4 Layout constraints 86
4.4.1 Positioning information 87
4.4.2 Word positioning 89
4.4.3 Redundant constraints 90
4.5 Summarisation 93
4.5.1 Modifications to layout constraints 95
4.5.2 Modifications to structure realisation 95
4.6 Improving performance 97
4.6.1 Labelling order 98
4.6.2 Limited-discrepancy search 98
4.6.3 Caching search states 99
CONTENTS

4.7 Prolog implementation .................................................. 102
  4.7.1 Constraining ......................................................... 102
  4.7.2 Labelling and Optimisation ..................................... 103
4.8 Comparison to Reiter and Dale’s consensus architecture ......... 106
4.9 Comparison to RAGS ...................................................... 113
4.10 Extensions ................................................................. 115
4.11 Summary ................................................................. 116

5 Experiments ................................................................. 117
  5.1 Corpus ................................................................. 118
  5.2 Sample output .......................................................... 120
  5.3 Basic document structuring .......................................... 121
    5.3.1 Labelling order .................................................. 121
    5.3.2 Search strategies ................................................ 124
  5.4 Summarisation .......................................................... 127

6 Conclusion ................................................................. 133
  6.1 Contribution ............................................................ 133
  6.2 Further work ............................................................ 135
Chapter 1

Introduction

1.1 Natural language generation

Natural language generation (NLG), the problem of creating human-readable documents by computer, is one of the major fields of research in computational linguistics.

The task of creating a document is extremely common in many fields of activity. Accordingly, there are many potential applications for NLG - almost any document creation task could potentially be automated by an NLG system. Advanced forms of NLG could also be used to generate a document in multiple languages, or as an output interface for other programs, which might ordinarily produce a less-manageable collection of data. They may also be able to create documents tailored to the needs of individual users.

The problem of NLG is related to several other fields of natural language processing. It can be considered as the inverse problem to that of natural language understanding. Other problems in natural language processing, such as summarisation and machine translation, can be considered as having a subproblem of NLG.
Reiter and Dale (2000) divide the NLG process into several subtasks, which fall into two categories: those which relate to the content to be included in the document, and those dealing with the structure of the document. Most NLG systems focus primarily on content-related issues. These revolve around converting the abstract input given to the system into linguistic information, including such tasks as choosing appropriate words and grammatical structures to represent the desired content.

However, we are concerned with structural issues, such as grouping of the text into paragraphs, sentences and other linguistic units, and the ordering of those units. These are vital to ensure the coherence of a long document, which might otherwise be little more than a sequence of bullet points.

The NLG process involves several different data types. We adopt the terminology of the Reference Architecture for Generation Systems (RAGS) (Mellish et al., 2004) here.

**Rhetorical representations** describe the relationships between different parts of the text. They generally take the form of a tree, with leaf nodes representing units of text, while internal nodes describe the relationships between these text spans. The most common type of rhetorical representation is known as a *rhetorical structure* (Mann and Thompson, 1986; 1987). Rhetorical structures use this tree format, and additionally the nodes are labelled as either nuclei or satellites, depending on their relative importance.

**Document representations** are a step closer to the actual document than rhetorical representations. They include textual units, such as sentences and paragraphs, along with some formatting information, such as their relative positions and indentations. They are still abstracted somewhat from the actual text, however. We are particularly concerned with one type of document representation, known as a *document structure*, which we introduce in more detail in the next section.
1.2. DOCUMENT STRUCTURE

Semantic representations represent the meaning of basic propositions. They can be quite abstracted from the output text — potentially even to the point of being language-independent. Semantic representations are specified in terms of lower-level linguistic objects, but in a sufficiently flexible way as to permit different theories of semantic representation. Although they are intended to describe the content of basic propositions, other forms may also be used for this purpose, such as conceptual or quote representations.

Syntactic representations represent the text which is to be realised from a proposition, rather than its meaning, but still in an abstract form. For example, rather than simply storing a word as it will be realised, a syntactic representation could contain the word stem and the necessary lexical information to generate the correct inflection, such as number and case.

1.2 Document structure

Recent work (Power et al., 2003) has demonstrated that the graphical organisation of a text – such as its headings, fonts, and linebreaks – can convey meaning. A simple illustration of this point appears in Figure 1.2 (Scott, pers. comm.). Power et al. contend that all texts have layout, even if it is very basic. They consider layout in text to be analogous to prosody in speech. However, prosody has been much more thoroughly studied.

They propose to capture those aspects of graphical organisation which carry meaning using a new level of representation called document structure. Power et
Elixir is safe to use since

- the medicine has been thoroughly tested and,
- it has no significant side effects.

(b) Output in bulleted list format

The medicine has been thoroughly tested; it has no significant side-effects. Therefore, Elixir is safe to use.

(c) Output in inline list format

Figure 1.2: Two Formats for an Excerpt from a Patient Information Leaflet, and the Underlying Rhetorical Structure (Power et al., 2003)

al. define document structure as “the organization of a document into graphical constituents like sections, paragraphs, sentences, bulleted lists, and figures; it also covers some features within sentences, including quotation and emphasis.”

They argue that the same document structure can be rendered into formatted text in multiple ways, as illustrated in the patient information leaflet text in Figure 1.2(b) and 1.2(c).

This is related to markup languages such as HTML and \texttt{LaTeX}, which allow us to describe the structure of a document independently of its presentation. However, such markup languages are only suggestive of document structure, as they
1.3. CONTRIBUTION

blur the distinction between descriptive and presentational markup (cf (Coombs et al., 1987)).

In addition to providing a theoretical description of the interplay between the layout of a document and its meaning, document structures can be used in NLG. This process, known as document structuring, uses constraint programming to establish the relationships between a rhetorical structure and a set of corresponding document structures.

There are many such document structures corresponding to a single rhetorical structure. Each of these express the same meaning, but with different formatting. In order to choose between them, the document structures are scored by counting the number of undesirable features they contain. For a given rhetorical structure, all possible document structures are generated and then ranked based on their defect count. The structure with the lowest number of defects is then used.

1.3 Contribution

Even a basic notion of document structure, allowing different parts of a document to be represented at different levels of importance, such as sections, paragraphs or sentences, or permitting different orderings of those parts, is likely to result in a constraint satisfaction problem, with a exponential space of possible solutions.

Problems such as these require sophisticated methods to solve efficiently. One area which has had considerable success in solving constraint satisfaction problems is constraint programming (Van Hentenyck, 1989, Marriott and Stuckey, 1998).

Power et al. describe their document structures as a constraint model, and use basic constraint programming to solve it, but do not use any advanced techniques
to improve their system’s performance. Accordingly, their system is only capable of rendering documents of approximately a paragraph in size.

In this thesis we present what we believe to be the first attempt to apply techniques of this sort to the problem of structuring an automatically-generated document. We will show that they allow the generation of larger documents, as well as more complex document structuring tasks.

Using these techniques, we are able to render considerably larger structures than the system of Power et al. (2003), up to around a page in size. We have also added the capability to constrain the output into a predetermined amount of space. In order to do this, we have produced constraints which describe the layout of the resulting document down to the level of individual characters, as well as the capacity to summarise the input structure. However, this adds to the complexity of the problem, reducing our system’s performance.

Our system is divided into two parts: the document structurer, and the syntactic realiser, as shown in Figure 1.3. The document structurer is further split into constraining and labelling stages.

The document structurer works similarly to that of Power et al. (2003), with some enhancements. It allows the rhetorical structure to be summarised, and can control the size of its output down to the word and character level, allowing very precise formatting. Moreover, it uses a variety of constraint programming strategies in order to improve its efficiency.

The division between constraining and labelling is common in constraint logic programming. The first of these stages expresses the relationship between the input rhetorical structure and any valid document structure, while the second actually chooses one possible structure and fully instantiates it. The syntactic realiser converts a document structure into actual text; it is primarily concerned with generating correct capitalisation, punctuation and indentation.
1.4 Overview

The remainder of this thesis is organised as follows. Firstly we review the necessary background material; Chapter 2 covers natural language processing (NLP). We are primarily interested in natural language generation, and concentrate on it and related topics in NLP. In particular, we focus considerable attention to the theories of rhetorical structure and document structure, and also briefly review document summarisation.

Chapter 3 deals with constraint programming. We briefly survey the field as a whole and give a formal definition of a constraint program. We then focus on the subfield of constraint programming used in our work, constraint logic programming over finite domains, and closely examine several techniques for improving the performance of these programs. The chapter concludes with a survey of previous applications of constraint programming to natural language processing.

In Chapter 4 we describe the implementation of our system. Initially, we explain the operation of a basic document structurer, incorporating the same functionality as that of Power et al. (2003). We then examine in detail the process of generating an actual document from the document structure. Next, we describe several enhancements which can be made to this document structurer.

We give a collection of constraints which specify the layout of the document, and explain how the system can be modified to enable summarisation. These
two enhancements allow us to generate document structures which fit into a given amount of space. We also describe a number of strategies from constraint logic programming which we use in order to improve the efficiency of our system. We conclude the chapter by comparing our architecture to two reference architectures for NLG systems, and describing some possible extensions which could be made to our system.

Having described our system, we report on the results of our experimental work in Chapter 5. We describe the corpus which we use to test the system. The experiments are divided into two parts - those on the basic document structurer, and those dealing with the extended document structurer, including summarisation and layout constraints.

We close by discussing our findings and identifying issues for future investigation in Chapter 6.
Chapter 2

Language technology background

In this chapter we review various fields of language technology which are relevant to our work. We defer other background material in constraint programming until Chapter 3.

Our primary interest is in natural language generation (NLG), the process of generating human-readable documents by computer. This is the focus of Section 2.1. We define it in more detail and review two reference architectures which are based on a large number of different systems, and give an overview of the most common approaches to the problem.

Many NLG systems, including our own, make use of rhetorical structures as an input or intermediate form of data. A rhetorical structure represents the relationships between different spans of text in a document in a simple form, and are easy to represent and manipulate computationally. We describe these structures in detail in Section 2.2.

The method which we use to implement NLG is based around the theory of document structure, which we examine in Section 2.3. It attempts to bridge the gap between a rhetorical structure and a fully realised document by accounting for aspects of the document’s layout which affect its meaning. This intermediate
representation effectively splits the task of NLG into two steps. A rhetorical structure is converted to a document structure, which is subsequently realised as an actual document.

We would like to be able to control the size of our output documents. This could be useful when targeting different output devices or users, for example. This necessitates summarisation, where we choose the most relevant portions of a document and only display those. We explain the basic principles of summarisation, focussing particularly on one method which operates on rhetorical structures, in Section 2.4.

In order to make use of these processes, we require a corpus of rhetorical structures, from which we can generate document structures and actual documents. We review the available data in Section 2.5. We then conclude by summarising the most important information from the chapter in Section 2.6.

2.1 Natural language generation

Natural language generation is the process of generating text in a human language, usually starting from some non-linguistic input (Reiter and Dale, 2000).

There are many natural language generation systems in existence. They have varying purposes, many being domain-specific, and use different methods to accomplish their aims. Rather than choosing specific systems, we examine two reference architectures, which we believe will better describe the aims and features of typical natural language generation systems.

In Section 2.3, we describe one particular system, the document structurer of Power et al. (2003).
2.1. NATURAL LANGUAGE GENERATION

2.1.1 The consensus architecture of Reiter and Dale

Reiter and Dale (2000) describe the process of building a natural language generation (NLG) system (Reiter and Dale, 2000). Based on a review of several NLG systems, they define a consensus architecture. It is compatible with the structures of many of the surveyed systems, and is also considered to be a good framework in which to build future systems. This architecture is shown in Figure 2.1.1. They identify several sub-tasks of the NLG problem, which are each assigned to one of three modules in their proposed system, although sometimes the boundaries between subtasks and their positioning within modules is left unspecified. The modules are intended to be executed sequentially, as shown in Figure 2.1.1.

The tasks are also divided into those dealing with the content which should be displayed, and those involved in structuring the output. This categorisation has less influence on their proposed architecture, being intended more to clarify the tasks for human readers.
Reiter and Dale (2000) claim, as many others have done previously (see for example (Mann and Thompson, 1986)), that the underlying structure of a text can be represented as a tree. Accordingly, the output of the document structuring process is a tree of messages, known as a document plan. The first stage in the pipeline, document planning, involves the creation of such a tree.

The messages in the tree are intermediate representations of the data which is to be realised. The input to an NLG system is entirely abstract, while the output takes the form of a sentence or some similar unit of text. The exact format of the message varies considerably between systems and domains.

The document plan takes the form of a tree, with messages at the leaf nodes and groups of related messages as internal nodes. An example of this is shown in Figure 2.2. These internal nodes may contain additional information — for example, specifying whether the node should be realised as a document, paragraph or some other linguistic structure, or the order in which its children should be displayed.
As shown in Figure 2.1.1, document planning consists of two subtasks. The first of these is *content determination*, which is simply the process of determining what information needs to be included in the resulting document.

The other document planning task is that of *document structuring*. As the term is used by Reiter and Dale\(^1\), this means organising the information which is to be displayed into an appropriate form. This process involves expressing the relations between the messages, and may also include ordering them, and determining whether they should be represented as sentences, paragraphs or other linguistic structures.

The intermediate step in Reiter and Dale’s NLG pipeline is known as *microplanning*. It is further divided into the subtasks of lexicalisation, referring expression generation and aggregation, which we describe below. Lexicalisation and referring expression generation are considered to be content-related tasks, while aggregation is a structuring task. The output of this process is known as a text specification.

Text specifications describe precisely what should be in the resulting document, but they are still in an abstract form. Like document plans, they are generally trees. The leaf nodes of a text specification correspond to linguistic units in the output, usually individual sentences. They may contain the actual text which is to be realised in the output, or a syntactic representation. Syntactic representations generally include a list of the lexemes which are to be displayed, along with grammatical information like number or tense. Non-leaf nodes specify the higher-level structure, such as paragraphs, sections and chapters.

Microplanning is the process of converting a document plan into a text specification. It is perhaps the least clearly defined of the three pipelined stages. An NLG

\(^1\)Note that “document structuring” in this context is not specifically referring to the process described by Power et al. (Power et al., 2003), which we examine in detail in Section 2.3. Reiter and Dale’s use of the term is much more broad, although both processes have similar aims.
system may perform some or all of the three subtasks at the document planning stage, relegating microplanning to a relatively minor role, or even removing the need for it entirely.

The first subtask is *lexicalisation*. This refers to the process of generating the actual words which are to be used to represent the desired information. Because a given fact can generally be expressed in multiple ways, an NLG system will require some procedure for choosing the most appropriate one.

The other content-related subtask of microplanning is *referring expression generation*. This is the problem of deciding how to refer to a given entity. It can be divided into two sub-problems. The first is introducing an entity which may have many possible descriptions; Reiter and Dale give the example of a person who could be variously introduced as a computational linguist, a visitor from overseas, or someone who is interested in Tai Chi.

The second issue is that of referring to a previously identified entity in a non-ambiguous way. Usually, an entity is introduced in some detail the first time it is referred to, while subsequent references are much simpler, often containing only a pronoun.

The last task in microplanning is that of *aggregation*. This is the procedure by which the document plan produced by the document structuring process is grouped into linguistic units, such as sentences and paragraphs.

The boundary between microplanning and document structuring is not very clearly defined. In particular, the ordering of messages may be determined in either task, as may decisions about what level (paragraph, sentence, etc) the facts should be realised. Reiter and Dale suggest that sentence and lower-level structures should be determined during aggregation, while higher-level choices should be made earlier, by the document structurer, but they do not take a rigid position on the issue.
2.1. NATURAL LANGUAGE GENERATION

The last of the three modules in Reiter and Dale’s architecture is that of surface realisation. This process generates the output document from the text plan. The complexity of this step varies considerably, depending on how abstract the leaf nodes of the text specification are. If the nodes are already text strings, then surface realisation will, at most, require orthographic changes, like capitalising initial letters and terminating sentences with full stops. A more abstract representation will require grammatical knowledge, and will involve choosing the appropriate form of each word, depending on its syntactic properties. It consists of two sub-tasks: linguistic realisation and structure realisation, which are content-based and structure-based, respectively.

The first of these tasks, linguistic realisation, is the process of converting the internal representations of the facts which the NLG system intends to convey into sentences in a natural language. NLG systems vary widely in the complexity of these internal representations, from some which use a highly abstracted form, to those which are essentially sentences themselves. The difficulty of linguistic realisation varies accordingly. Reiter and Dale also include orthographic requirements in this process, such as capitalising the first letter of a sentence, and beginning a paragraph on a new line.

The other task is that of structure realisation. This describes the insertion of extra markup required to correctly express the structure in the output format. For example, if the output is in HTML format, paragraphs will need to be marked with \texttt{<P>} tags.

Overall, Reiter and Dale describe an architecture which is intended to be used for future NLG systems, rather than to describe all of the prior work in the field. Nonetheless, their architecture still summarises the features of many existing systems quite well.
2.1.2 Reference architecture for generation systems

Mellish et al. (2004) describe a Reference Architecture for Generation Systems (RAGS), which is intended both as a description of the features of an NLG system which are commonly agreed upon by researchers, and as a base on which to build future systems. It is intended to allow different systems to share modules and data, and make comparison of the systems easier.

Mellish et al. review Reiter and Dale’s consensus architecture, which we described in the previous subsection, but reject it for use as a reference architecture for three main reasons. They claim that there is no agreement amongst other NLG implementations on the specifics of what each module should do and which of the six tasks should occur in which module. Moreover, the data interfaces between the modules are insufficiently specified. It is worth noting, however, that Reiter and Dale did not intend their architecture to model that of all pre-existing NLG systems, but rather to serve as a base for constructing new systems.

Subsequently, Mellish et al. (2004) describe their own architecture. While Reiter and Dale’s model is primarily concerned with the architecture that an implemented system might use, RAGS mostly deals with the data structures that will be used by the system. They identify six different types of data which an NLG system will use: conceptual, rhetorical, document, semantic, syntactic and quote representations. They rigorously define each of these representations in terms of various basic variables using set theory, but we only give an overview here.

They are much less prescriptive about the ways in which the data should be processed than the definitions of the data itself. They do not describe a pipeline for the data; they believe that there is not enough agreement in the literature to be able to define a common architecture which all NLG systems can use. Nonetheless, they name several modules which might be used in an NLG system along with their input and output data types, which we show in Table 2.1.
### 2.1. NATURAL LANGUAGE GENERATION

<table>
<thead>
<tr>
<th>Module description</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Content selection</td>
<td>None</td>
<td>Knowledge base identifier</td>
</tr>
<tr>
<td>Rhetorical structuring</td>
<td>Knowledge base identifier</td>
<td>Rhetorical representation</td>
</tr>
<tr>
<td>Document structuring</td>
<td>Rhetorical representation</td>
<td>Document representation</td>
</tr>
<tr>
<td>Aggregation</td>
<td>Any</td>
<td>Any</td>
</tr>
<tr>
<td>First stage of lexicalisation</td>
<td>Knowledge base identifier</td>
<td>Semantic representation</td>
</tr>
<tr>
<td>True lexical choice</td>
<td>Semantic representation</td>
<td>Syntactic representation</td>
</tr>
<tr>
<td>Referring expression generation</td>
<td>Knowledge base identifier</td>
<td>Semantic or syntactic representation</td>
</tr>
<tr>
<td>Surface realisation</td>
<td>Syntactic representation</td>
<td>Quote representation</td>
</tr>
</tbody>
</table>

Table 2.1: Modules suggested for implementation using RAGS (Mellish et al., 2004)

We have already given Mellish et al.’s definitions of several data types in the introduction. We will describe all of them here.

*Conceptual representations* are the input to the NLG system. They are presumed to refer to a knowledge base in some way, but the details are left unspecified. RAGS does, however, specify an interface for converting knowledge base identifiers to semantic representations, which we describe later.

*Rhetorical representations* are used to identify relationships between different parts of the text. Like the consensus architecture of Reiter and Dale, RAGS uses a tree structure for its rhetorical representations. An internal node contains a rhetorical relation which holds between its children, while the leaf nodes contain basic propositions. The content of these propositions can vary, depending on the needs of the system, and may include conceptual or semantic representations.

*Document representations* are similar, but they also contain some presentation-related information, like sentence and paragraph boundaries. Document representations are similar to both the text specifications of Reiter and Dale, and particularly the document structure of Power et al. (2003), which we describe in Section 2.3.
Semantic representations represent the meaning of conceptual representations. They are generally converted into syntactic representations, which define the actual text which is to be realised, but in a slightly abstract form, generally by storing word stems and lexical categories rather than just strings.

Each of these data types are formally specified using set notation. For example, a semantic representation is made up of a discourse referent, a set of predicates (which are collectively denoted by $SemType$) and values for semantic roles. This is expressed as follows:

$$SynRep = DR \times SemType \times SemAttr$$

(2.1)

Each of these three types are themselves defined in terms of other types. Eventually all types consist of primitives. These primitives are left unspecified so that they may be implemented in individual systems as their authors see fit.

The output is given in terms of quote representations are blocks of text or graphics which are directly copied to the output, without any other processing by the NLG system.

Mellish et al. (2004) also define a method for describing lower level data types, which are specific to individual NLG systems, which they call the objects-and-arrows model. Objects in this model are unrooted typed directed graphs. The nodes, or objects, take one of the six types described above, while the edges can take two types. The first of these, $n-el$, means that the child node of the edge is the $n$th element of the parent node. The second type, $refers\_to$, is a pointer to another object, possibly of a different type. For example, the leaf node of a rhetorical representation may refer to a semantic or quote representation.

This model allows different implementations to define their own data types, and to describe intermediate representations between the six major types. Mellish
et al. also describe how to describe a RAGS implementation in XML, which corresponds directly to the objects-and-arrows model.

In summary, RAGS identifies the areas of consensus and disagreement amongst current NLG systems, and describes in an abstract form their common data and functionality. It differs from Reiter and Dale’s architecture in that it is not intended as an aid to implementation so much as to enable comparison and integration between systems. Because of this, it is deliberately non-specific in some details where there is little agreement between existing systems.

2.2 Rhetorical structure

Rhetorical Structure (RS) was developed in the 1980s (Mann and Thompson, 1986; 1987). Before then, there was no theory describing the discourse structure of a text which was sufficiently well-defined to be implemented on a computer. RST is now used in many areas of computational linguistics, as well as in linguistics proper (Mann, 1999). We are particularly interested in its application to natural language generation.

The fundamental idea behind RST is that well-written texts have a high degree of coherence. Spans of text are not written at random, but have clear relationships to each other. For example, one span of text may provide evidence for assertions in another, or a contrasting point of view. In Figure 2.3 we provide an example of a text, with the spans denoted by closed brackets, and the corresponding rhetorical structure tree.

Rhetorical structure theory provides two different types of relations which may hold between spans of text: nucleus-satellite and multinuclear. Nucleus-satellite relations are those in which one of the spans is subordinate to the other; the spans are called the nucleus and satellite, respectively. For example, when one span
[The medicine has been thoroughly tested;] [it has no significant side-effects.] [Therefore, Elixir is safe to use.]

(a) Text

```
<table>
<thead>
<tr>
<th>evidence</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Therefore, Elixir is safe to use”</td>
</tr>
<tr>
<td>list</td>
</tr>
<tr>
<td>“the medicine has been thoroughly tested”</td>
</tr>
<tr>
<td>“it has no significant side effects”</td>
</tr>
</tbody>
</table>
```

(b) Rhetorical Structure

Figure 2.3: Text with accompanying rhetorical structure, from Power et al.

presents an assertion and another contains evidence for that assertion, the first span is more fundamental to the meaning of the text. In this case, the first span would be the nucleus, and the second the satellite. In Figure 2.3, the nuclei are shown in bold. Observe that the evidence relation is nucleus-satellite, so only one of its children is a nucleus, while the list relation is multinuclear, so both of its children are nuclei. Nucleus-satellite relations always have two children — the nucleus and the satellite — while multinuclear relations may have multiple children.

The set of relations which can be used varies between different applications of rhetorical structure (Mann and Thompson, 1986, Carlson and Marcu, 2001). Many relations are found in most or all applications, however, including list and evidence as used above.

We give a formal definition of a rhetorical structure below. It is consistent with — but more rigorous than — the description given by Mann and Thompson (1986).
2.2. RHETORICAL STRUCTURE

\[
RST = \text{NucleusSatellite} \cup \text{Multinuclear} \cup \\
\text{Elementary}
\]  

(2.2)

\[
\text{NucleusSatellite} = \text{Relation} \times RST \times RST
\]  

(2.3)

\[
\text{Multinuclear} = \text{Relation} \times RST^+
\]  

(2.4)

\[
\text{Elementary} \in \text{String}
\]  

(2.5)

Note that \textit{Elementary} data are simply strings, and we do not define the set of values a \textit{Relation} may take. The precise set of relations varies between different uses of rhetorical structure, although they generally have many common elements, including those used as examples here. Two such sets of relations are given by Mann (1999) and Carlson and Marcu (2001).

These rhetorical relations may hold between large spans of text, which can themselves contain smaller spans of text with connecting relations, and so on. This gives rise to a tree structure, with rhetorical relations at internal nodes, and spans of text at the leaves. This can be seen in Figure 2.3, with one of the spans of the \textit{Evidence} relation containing two spans linked by a \textit{List} relation.

While the theory was originally formulated and is generally expressed in terms of spans of text, we can also use some form of abstract, semantic representation for the leaf nodes. This is the viewpoint taken by Power et al. (2003), and is very similar to the rhetorical representations used by Reiter and Dale (2000), which we discussed in 2.1.1. This distinction does not have much effect on most tasks which can be performed on rhetorical structures; generally it simply adds an extra step, that of converting from text spans to the semantic form, or vice versa.

Perhaps the most basic task involving rhetorical structures is that of \textit{analysis}. This is the process of determining the rhetorical structure of a given text. An attempt to automate this process is described by Marcu (2000). The inverse
operation, can be considered as a form of natural language generation, although a rhetorical structure is typically an intermediate form of data in an NLG system, rather than an input, as we discussed in 2.1.

2.3 Document structure

2.3.1 Formal theory of document structure

The formal theory of document structure is based on rhetorical structure. A document structure contains the rhetorical structure of the text which it represents, retaining the same tree structure. All of the other data from the rhetorical structure is also kept: the different types of nodes (nucleus-satellite, multinuclear and elementary), the rhetorical relations and the textual or semantic information at the leaf nodes.

Each node on the document structure represents a document unit, which is a text-category such as a phrase, sentence, paragraph or chapter. In addition to the information retained from the rhetorical structure, document units are represented by four variables: level, indentation, position, and connective. A document structure containing each of these variables is shown in Figure 2.4. It corresponds to the inline text in Figure 1.2. We can formally describe the document structure of Power et al. as follows:

\[
DST = RR \times Level \times Indentation \times Position \times Connective
\]  

(2.6)

---

\(^2\)Power et al. allow the document structure trees to differ slightly from the rhetorical structure trees in some case, but we believe that these are better represented with the same tree, and present this material accordingly. We discuss this matter in more detail in Subsection 4.2.1.
We now describe each of the four document variables in more detail.

**Level:** This represents the importance of the unit in the realised document. Documents have a hierarchical structure: a paragraph is composed of sentences, sentences of text-clauses, and so on.
The level is an integer which can take values from zero to five, corresponding to realisation as a phrase (comma-terminated, of level zero), clause (semicolon-terminated, of level one), sentence (level two), paragraph (three), section (four), or chapter (five). In Figure 1.2, the children of the list relation are level zero in the first realisation, and level one in the second realisation.

**Indentation:** A document unit may be indented from its parent, allowing such items as bulleted lists. Power et al represent indentation as an integer, indicating how many times a unit has been indented relative to the root node. In Figure 1.2, the first list is indented in the first realisation, but not the second.

**Position:** The position variable indicates the order in which the nucleus and satellite occur. Because all of the children of a multinuclear relation are of the same importance, reordering them should have no effect on the realised document, so they are always rendered in the same order that they occur in the rhetorical structure. The two realisations in Figure 1.2 show alternate positionings of the nucleus and satellite of the evidence relation.

**Connective:** A discourse connective is always used to denote the rhetorical relationship between different document units. The only exception is for the leaf nodes of the document structure, which represent basic propositions. Figure 1.2 shows two different connectives which realise the evidence relation: “since” and “therefore”.

### 2.3.2 Discourse connectives

Discourse connectives (which are also known as discourse markers or just connectives) can be used to indicate the rhetorical relationships between consecutive
spans of text. We are particularly interested in their use in natural language generation.

Knott (1996), following earlier work by Quirk et al. (1972), lists five categories of discourse markers:

*Coordinators* are always found between the clauses which they connect. These clauses do not have to be in the same sentence. *Subordinators* introduce subordinate clauses. The subordinate clause can be either before or after the main clause, but the subordinator must always occur before the subordinate clause. *Conjunct adverbs* modify whole clauses. They can appear at several different points within clauses. *Prepositional phrases* refer back to the previous clause. Often they can be hard to distinguish from conjunct adverbs. *Phrases which take sentential complements* describe an intentional stance towards the text which follows them.

Using Knott’s work as a basis, Power et al. (1999) describe a theory classifying discourse connectives, which they apply to natural language generation. Their classification is similar to Knott’s, but simplified in several respects. They claim that theirs is the only work describing how to generate multiple discourse markers for texts spanning multiple sentences.

They side-step the question of whether a discourse connective should be realised overtly, insisting that there be a one-to-one relationship between rhetorical relationships in the rhetorical structure and discourse connectives in the generated text.

Discourse connectives are modelled with four attributes: relation, locus, phrase and syntax. *Relation* is the rhetorical relation which the connective represents. *Locus* specifies whether the connective should be attached to the nucleus or satellite. *Phrase* is simply the text of the connective. *Syntax* is based on a similar categorisation to that of Knott, representing the syntactic restrictions on how the connective can be used. Their categorisation is somewhat simplified and...
more clearly defined, making it more suitable for computational implementation. They only recognise three types: parenthetical, coordinating and subordinating; it seems that conjunctive adverbs have been merged with subordinators, and phrases taking sentential complements have been removed entirely.

Coordinating connectives force the units which they connect to be of level zero, which is a comma-terminated clause. An example of a coordinator, “although”, is given in Figure 2.5(b).

Subordinating connectives, such as “but”, require that the unit to which they are attached occurs last, as can be seen in Figure 2.5(a). Observe that the text spans can be of any level (phrase, clause or sentence), but that the text is clearly ungrammatical if the spans are reversed.

Parenthetical connectives also require that the attached unit occurs last, and also force the two text spans to be of level greater than zero. We give an example of a parenthetical connective, “however”, in Figure 2.5(c). Note that either using text-phrases or reversing the spans makes the text ungrammatical.

2.3.3 Generating a document structure

In addition to being of theoretical interest, document structure is a useful concept for NLG. Beginning from a rhetorical structure, one can construct a document structure, which is then readily converted into the output text. This process is an example of the document structuring referred to by Reiter and Dale (2000), although their definition is much more general.

The document structure itself corresponds to a document plan and a document representation, using the terminology of the consensus architecture of Reiter and Dale (2000) and RAGS (Mellish et al., 2004), respectively.

Power et al. examine the task of determining a document structure from a given rhetorical structure, which they call “document structuring” (Power, 2000, Power
The FDA bans Elixir, but the FDA approves Elixir+.
The FDA bans Elixir; but the FDA approves Elixir+.
The FDA bans Elixir. But the FDA approves Elixir+.
#But the FDA approves Elixir+. The FDA bans Elixir.

(a) A subordinating connective, “but”

The FDA approves Elixir+, although the FDA bans Elixir.
Although the FDA bans Elixir, the FDA approves Elixir+.
#Although the FDA bans Elixir. The FDA approves Elixir+.
#The FDA approves Elixir+; although the FDA bans Elixir.

(b) A coordinating connective, “although”

The FDA bans Elixir; however the FDA approves Elixir+.
The FDA bans Elixir. However the FDA approves Elixir+.
#The FDA bans Elixir, however the FDA approves Elixir+.
#However the FDA approves Elixir+. The FDA bans Elixir.

(c) A parenthetical connective, “however”

Figure 2.5: Correct and incorrect uses of the three types of discourse connective, with the latter denoted by a hash

et al., 2003). They implement it in a system called ICONOCLAST (Integrating Constraints on Layout and Style). The input consists of a collection of simple propositions organised into a rhetorical structure tree. The document structurer arranges these into a coherent collection of paragraphs, text-sentences and the like. This is then converted into an actual document by a syntactic realiser.

They use logic programming techniques to generate the document structure. Each node on the rhetorical structure tree corresponds to a node in the document structure tree. The four variables associated with the node are constrained in various ways. First, the level of a child unit must be less than or equal to that of its parent, and equal to that of its siblings. The only exception is when the child unit is indented, in which case its level is independent of its parent, although it must still be equal to that of its siblings. Second, the indentation of a child unit must be
either equal to or one greater than that of its parent. The positions of siblings must obviously be distinct. Third, the connective must realise the rhetorical relation. Finally, the type of the connective places additional constraints on the level and position of the children of the current unit, which we described in Subsection 2.3.2.

This process generates a large number of candidate document structures for a given rhetorical structure, without determining which to use. For example, Power et al. give an example of a rhetorical structure with one relation between two facts, with seven renderings, and one with three relations and four facts, which has 58 renderings. In fact, the number of document structures corresponding to a given rhetorical structure is exponential in the number of nodes in the structures.

2.3.4 Scoring document structures

Given that there are so many document structures corresponding to a given rhetorical structure, there is a need for some method of choosing between them. They identify six undesirable features which may occur in a document structure. Using these defects, they rank all the document structures from best to worst. We give descriptions and examples of these defects below. From a single rhetorical structure in Figure 2.6, we provide three different document structures, each containing some of the defects, in Figures 2.7 to 2.9.

Nucleus before satellite: The nucleus appears before the satellite. This is undesirable, according to Power et al., because the most important information in a sentence should be placed at the end, as per Quirk’s notion of end focus. They claim that this is the common practise in English, and psycholinguistic evidence which suggests that it makes the sentence more readable. Figure 2.8 shows a nucleus before satellite defect.
2.3. DOCUMENT STRUCTURE

concession

“some people might suffer a mild allergic reaction”
  concession

“the medicine has no significant side-effects”
  “one of the main ingredients is penicillin”

Figure 2.6: Common rhetorical structure

Left-branching structure: The left side of the document structure tree branches, while the right side does not. Power et al. claim that this constitutes a defect, because of psycholinguistic evidence showing that these structures are difficult to process, as well as their own informal observations.

Left-branching structures are shown in Figures 2.7 and 2.9. (Note that we display the trees according to the convention for rhetorical structures, with the nucleus first, not in the order in which they will be rendered.)

Power et al. suggest that this defect is an example of Quirk’s principle of end weight, which is related to, but distinct from end focus. The principle of end weight states that the part of a sentence with a higher information content should be placed at the end. This may or may not be the most important part of the sentence, so there can be a conflict between these two principles.

This conflict is reflected in the defect model. For certain rhetorical structures, it can be impossible to avoid both a nucleus before satellite and left-branching defect. If, at some point in the structure, the satellite branches and the nucleus does not, then we will incur a left-branching defect if the satellite is displayed first, and a nucleus before satellite defect if it does. This is the case in Figure 2.6 above; note that all of the realisations contain one of these two defects.
Although one of the main ingredients is penicillin the medicine has no significant side-effects. However, some people might suffer a mild allergic reaction.

```
concession
  Level: 3
  Position: 0
  Indentation: 0
Connective: however
```

“some people might suffer a mild allergic reaction”

```
concession
  Level: 2
  Position: 0
  Indentation: 0
```

“the medicine has no significant side-effects”

```
Level: 0
Position: 1
Indentation: 0
```

“one of the main ingredients is penicillin”

```
Level: 0
Position: 0
Indentation: 0
```

Figure 2.7: Document structure and realisation with a left-branching structure

**Lost rhetorical grouping:** The document structure conflates distinct levels of the rhetorical structure. In terms of the document structure variables, a parent and its children have the same value for their level variables. If this occurs at any point in the document structure, Power et al. call the document structure and rhetorical structure *homomorphic*; when all parents have (strictly) higher level or lower indentation than their children, they call the structures *isomorphic*.

A lost rhetorical grouping leaves the relative levels of some units in the rhetorical structure implicit. While the relationships between the different document units can still be inferred from the text, it can make it more difficult to read. Figure 2.8 contains two lost rhetorical groupings.
Some people might suffer a mild allergic reaction, although although one of the main ingredients is penicillin, the medicine has no significant side-effects.

It is worth noting that a rhetorical structure may have more distinct levels than are allowed in the document structure. If this is the case, then some parent and child nodes will have to be realised at the same level, unless they are multinuclear relations which can be indented instead. Therefore, lost rhetorical grouping defects may sometimes be unavoidable.

**Single-sentence paragraph:** A paragraph contains only one sentence. (A second sentence may sometimes be added to repair this defect.) Figure 2.8 gives an example of this defect.
One of the main ingredients is penicillin; however, the medicine has no significant side-effects. However, some people might suffer a mild allergic reaction.

```
concession
  Level: 3
  Position: 0
  Indentation: 0
Connective: however
```

```
“some people might suffer
  a mild allergic reaction”
  Level: 2
  Position: 0
  Indentation: 0
```

```
“the medicine has no
  significant side-effects”
  Level: 1
  Position: 1
  Indentation: 0
```

```
“one of the main ingredients
  is penicillin”
  Level: 1
  Position: 0
  Indentation: 0
```

Figure 2.9: Document structure and realisation with a left-branching structure, repeated connective and oversimple text-clause

**Oversimple text-clauses:** A sentence is composed of two text-clauses (clauses separated by a semi-colon), each of which expresses a single elementary proposition. Power et al. claim that, because a semicolon is a relatively uncommon device compared to a comma or full stop in human-authored text, it should used more sparingly. Accordingly, they score a defect for it when it is used to represent structures which do not need the extra level between sentence and phrase. When realising a more complex rhetorical structure which can make use of the added level, then a semicolon is allowed, with no penalty.

A single relation composed of two elementary structures can generally be readily rendered as a sentence of two (comma-separated) text-phrases, rather than
two text-clauses, so this defect is relatively easy to avoid. The only exception
is where a parenthetical connective is used to represent the relation, which will
prevent the elementary structures from being realised at level zero. The first
sentence of the second realisation in Figure 2.9 contains an oversimple text-clause.

**Repeated discourse connective:** A single rhetorical structure is represented
twice in the document structure, by the same connective, and in such a way that
one of the occurrences is a descendent node of the other. An example of this defect
is given in Figure 2.8.

The presence of these defects does not make a document structure invalid,
only less desirable. As such, they can be considered soft constraints, while those
earlier this section which describe the document structure are hard constraints.

Power et al. acknowledge that these defects are somewhat arbitrary, and lack
a sound theoretical basis. Instead, they are intended to demonstrate that not
all document structures are equally good, and that there are relatively simple
methods of choosing between them using only the document structure variables.
We consider how document structure scoring might be improved in Section 6.2.
For instance, Figure 2.8 shows a new defect we would like to introduce: two
connectives which are adjacent in the output.

### 2.3.5 Summary

We have extensively reviewed the theory of document structure. It is both a theory
of the relationship between document layout and meaning, and a practical method
of describing document layout for the purposes of natural language generation. A
document structure is based on rhetorical structure, but with additional variables
at each node representing layout-related information.
It is implemented using constraint programming to generate a large number of potential document structures from a single rhetorical structure. Of these, one can be selected by determining which has the least defects.

### 2.4 Document summarisation

The area of intersection between this thesis and the whole area of summarisation is relatively small with respect to either of the two, so we will not attempt to survey the entire field in depth.

Moreover, our aims are somewhat different from the usual goals of this field. Generally, summarisation is intended to be used on complete, human-authored documents, with no annotation. Any semantic or other abstract representation of the document to be summarised must be generated specifically for that purpose. By contrast, we plan to implement it as part of the natural language generation process, on documents for which we already possess rhetorical structures. Therefore, we will give a quick overview of the field before concentrating on one particular algorithm which operates on rhetorical structures.

There are two main approaches to this problem, which may be called *abstractive* and *extractive summarisation*. The first of these is what one might expect summarisation to be: taking a document, and producing an entirely new, smaller document which includes the most important information in the original. However, this process is quite difficult in the general case. It requires us to parse a document, to represent it in a knowledge base, to determine the most important items of the knowledge base, and then to output them as a new document (Hovy, 2002; Mani, 2001).

This process thus includes both natural language understanding and generation as subtasks, making it very difficult. Therefore, the problem is often simplified to
that of extraction — choosing the most relevant portions of the original document, and using them as a summary, or extract.

Extractive summarisation considerably reduces the need for natural language understanding and generation, by using the text of the existing document as the output. The only remaining problem is deciding which of the components (usually sentences) of the original text are important enough to include in the summary. While this can still benefit from being informed by a fully-fledged natural language understanding system, it can also be performed by much simpler, heuristic-based methods.

An example of the latter is given by Kupiec et al. (1995), which makes no attempt at understanding the text to be summarised. Instead, they score each sentence in a document on five categories: sentence length, occurrence of a few fixed phrases, sentence position within a paragraph, occurrence of the document’s most common words, and all-uppercase words. Using these categories, they train a Bayesian classifier on a corpus of documents with corresponding extracts, and use the results to rank the sentences of the document to be summarised, selecting a predetermined portion of it.

Another approach to this problem partway between the two extremes of attempting natural language understanding and using a purely heuristic method is described by Marcu (2000). It is of particular interest to us, because it operates on rhetorical structures.

The summarisation algorithm is based on the hypothesis that the nucleus of a given relation is more important than the satellite. Similarly, in a two level tree, the nucleus of the nucleus of the root is the more important that the nucleus of the satellite, which is more important than the satellite of the satellite. This idea can be extended recursively to a rhetorical structure of arbitrary depth.
We begin with a single rhetorical relation. Its children may be a nucleus and satellite, or both nuclei, depending on the type of relation. Also, they may be rhetorical relations themselves or elementary units. Each leaf node in the structure is assigned a weight as follows:

\[
score(u, T, d) = \begin{cases} 
0, & T \text{ is empty} \\
 d, & u \in \text{promotion}(P) \\
 d - 1, & u \in \text{parentheticals}(P) \\
 \max \left( \frac{score(u, L, d - 1)}{score(u, R, d - 1)} \right), & \text{otherwise}
\end{cases}
\]

In the above equation, \( u \) is the unit whose importance we are evaluating and \( T \) is the rhetorical structure tree, of depth \( d \). The rhetorical structure has left and right subtrees, which are called \( L \) and \( R \), respectively. The promotion set of a node is the set of that node, all of its nuclei, all nuclei of those nodes, and so on. We do not use parenthetical representations in our rhetorical structures, so they can be ignored.

Applying this equation recursively, we see that the weight of a leaf node is determined by the highest level node of which it is in the promotion set; the weight is the depth of the tree which begins at that node. Equivalently, we can define the weight of a given node as the depth of its first ancestor which is a satellite (including the node itself, if it is a satellite). If a node has no such ancestor, its weight is the depth of the rhetorical structure.

We give an example rhetorical structure with corresponding weights on each node in Figure 2.4. Nuclei are displayed in bold, and satellites in non-bold.
Observe that the conjunction is a multinuclear relation, and thus both of its children are in bold. The weights themselves are readily calculated as the height of the first satellite reached by iterating through the ancestors of a node.

The highest weight, five, is given to the only leaf node in the promotion set of the root, which contains the text “Therefore, Elixir is safe to use”. This does indeed seem to be the most important piece of information in the text. Similarly, the least important node, with a weight of one, contains the text “However, although it was introduced only three years ago”, which is probably the least useful information. The other three nodes, which are less easily ranked, have similar weights of three and four.

The weighting function is only specified for relations with two children, as Marcu’s definition of rhetorical structure only allows binary trees for rhetorical structures. However, it is trivial to extend it to relations with an arbitrary number of children.

This equation is only used for leaf nodes. In Marcu’s formulation of rhetorical structure, only the leaf nodes contain text. This includes discourse connectives, unlike the document structure of Power et al. (2003). While discourse connectives are still used to determine the rhetorical relations, they are not specifically required, and are grouped in with one of the descendants of the internal node which they help specify.

Given weights on each node of the rhetorical structure, we can produce a summary of the original document by selecting nodes in order of their weights until the total length of the selected nodes reaches the desired length for the summary. Note that this algorithm only uses the weights to rank individual nodes; it does not use the actual values at the nodes, only their importance in relation to other nodes.
Elixir contains gestodene. However, although it was introduced only three years ago, the medicine has been thoroughly tested, and it has no significant side-effects. Therefore, Elixir is safe to use.

(a) Text

(b) Corresponding rhetorical structure with node weights

Figure 2.10: A text with corresponding rhetorical structure and weights on each leaf node.
2.5 Corpora

A large corpus of rhetorical structures has been constructed by Carlson et al. (2002). It consists of 385 documents from the Penn Treebank (Marcus et al., 1993), each annotated as rhetorical structures. It is intended to be useful in a wide variety of applications. Unfortunately, it does not provide all the information required for document structuring. In particular, it does not presume a direct correspondence between discourse connectives and rhetorical relations, as this is not the case in human-authored text.

We are not aware of any comprehensive corpus of rhetorical relations and corresponding discourse connectives. Knott (1996) provides a list of connectives and the corresponding types in his PhD thesis, but without corresponding rhetorical relations.

There is also some data provided with the document structurer described in Power et al. (2003), which are based on the Compendium of Patient Information Leaflets ABPI (1997). They are given in our desired format, and have corresponding discourse connectives. These rhetorical structures are too small to be used as our main corpus, however, and are used primarily as examples.

We believe that a corpus of rhetorical relations and corresponding discourse connectives, in the format required for a document structure, would be very
valuable for further work in this field. We briefly describe how such a corpus might be constructed and used in Section 6.2.

2.6 Summary

We have reviewed several fields of language technology in this chapter. We began with a survey of NLG, focusing on two reference architectures which describe general approaches to the problem. While they differ in their details, they share many features. They begin by converting some form of internal knowledge base to semantic and/or syntactic data, which are then transformed into an intermediate form describing the output document, and then finally realised into a document itself.

A rhetorical structure is a simple representation of the relationships between different parts of a document. It is expressed as a tree; spans of text from the document become leaves, while internal nodes contain rhetorical relations like “concession” or “evidence” which describe the relationships between the spans of text, or between each other. Nodes can be either nuclei or satellites, depending on their parent relation. Some relations take one nucleus and one satellite, while others have only nuclei.

The theory of document structure attempts to capture the elements of a document which are relevant to its meaning. A document structure consists of the rhetorical structure of a given text. At every node, it adds four variables: indentation, position (relative to its sibling nodes), indentation and connective (used to express the rhetorical relation). Document structures can be generated from a rhetorical structure using constraint programming. A few constraints describe the relationships of the four document variables to each other, to variables at different nodes, and to properties of the rhetorical structure.
This procedure generates many document structures from a single rhetorical structure, some of which produce more readable output than others. In an attempt to model this effect, six different defects are introduced. They represent undesirable features in a document structure, but do not affect its correctness, or that of corresponding realisation. The candidate document structures can be ranked in order of the number of defects they contain.

There are many methods of summarisation. Our main interest is in a method by Marcu (2000), which operates on a rhetorical structure. It assigns a weight to each node in the structure, based on its type (nucleus or satellite), and those of its ancestors. With these weights, one can generate a summary of a text by simply selecting nodes in by their weights in descending order, until the desired summary length has been reached.

The main corpus of interest to us is by Carlson and Marcu (2001), containing rhetorical structures of articles in the Penn Treebank Marcus et al. (1993). There are, however, some issues revolving around different treatments of discourse connectives.

Having reviewed all the necessary concepts in natural language processing, we now turn our attention to constraint programming.
Chapter 3

Constraint programming

This chapter provides an overview of constraint programming, which is a programming paradigm based around the idea of expressing constraints between variables, rather than simply assigning values to them. We use it to express the relationships between document and rhetorical structures, and in doing so perform part of the natural language generation task. Our main focus is the subfield of constraint logic programming over finite domains.

We begin by introducing the main concepts behind constraint programming in Section 3.1. We describe these ideas much more formally in Section 3.2. Next we examine the subfield of constraint logic programming in Section 3.3, and survey various techniques which can improve the performance of constraint logic programs over finite domains, in Section 3.4. We then examine the role of constraint programming in natural language processing in Section 3.5, and conclude by summarising the main points of the chapter in Section 3.6.
3.1 Introduction to constraint programming

The central idea behind constraint programming Marriott and Stuckey (1998) is that we specify constraints between variables, rather than simply assigning values, as in a traditional programming language. Simple constraints take the form of mathematical equalities and inequalities, like the following:

\[
\begin{align*}
A &= 1 & (3.1) \\
B &\geq 2 & (3.2) \\
B &< 10 & (3.3) \\
C &\geq 4 & (3.4) \\
C &\leq 8 & (3.5) \\
C &= B + 3 & (3.6) \\
D &> A \times C & (3.7)
\end{align*}
\]

Using constraints allows us to write code once which can be used in multiple ways. For example, in (3.6) if we set either \(B\) or \(C\) to have a certain value, the other will automatically take the appropriate value. This is similar in some ways to a traditional logic programming language, but generally these do not allow multiple modes of operation on mathematical operations. Consider the Prolog equivalent of 3.6, which will give an error if we attempt to call it when \(B\) is not bound to a number:

```
C is B + 3
```

Even when we have decided beforehand which variables are inputs and which are outputs, this approach has benefits. It enables a declarative style to problem
solving, where we simply specify the constraints on the variables, without having to decide when and how they should be evaluated.

### 3.2 Formal definition of constraint programming

We now give a formal definition of constraint programming based on that of Cruz (2006).

Variables in constraint programming are essentially the same as variables in a mathematical sense. A domain $D(x)$ for some variable $x$ denotes the set of values which the variable may take.

A constraint can be represented by a pair $(s, \rho)$. Here $s$ is a tuple containing an arbitrary number of variables and $\rho$ is a relation between these variables, denoting which values they could take in order to satisfy the constraint.

A constraint satisfaction problem is a tuple $(X, D, C)$ where $X$ is a tuple containing the constrained variables, $D$ is the Cartesian product of the domain of each variable, and $C$ is the set of constraints over the variables.

In most cases, the sets over which the constraints operate will be too large to directly manipulate. They are Cartesian products of large numbers of variables, each of which may itself have a large domain. In order to solve CSPs, we need to use *narrowing functions*.

For a given constraint $C$, a narrowing function maps one subset of the domains of the variables associated with $C$ to another:

$$NF_C : 2^D \Rightarrow 2^D \quad (3.8)$$

Here $D$ denotes the Cartesian product of the domain of each variable involved in $C$. There are additional restrictions on a narrowing function:
The first two conditions, which are known as the narrowing and correctness properties, mean that a narrowing function can only remove values from the domains, never adding them. Moreover, it will only delete values which do not lead to solutions of the constraint. Note that it may also leave some values which cannot satisfy the constraint.

The last two conditions are known as monotonicity and idempotency. If one domain is contained in another, then narrowing the smaller domain should again be smaller than (or the same size) as the narrowed larger domain. Idempotency means that repeatedly applying the function to the same domain has no additional effect after the first application.

Clearly, we can use a narrowing function to prune unused values from the domains of constrained variables. In order to formalise this process, we introduce the notion of a fixed point.

Given a narrowing function $NF$, a fixed point $F$ is a domain such that $NF(F) = F$. Given the four properties of a narrowing function above, we can see that the union of all fixed points of some domain $A$ is given by $NF(A)$. Firstly, we show that the union of all fixed points of $A$ is contained within $NF(A)$.

If some value $x$ is in the union of all fixed points of $A$, we have:
\[ \exists A, x \in A, A_i \subseteq A, NF(A_i) = A_i \]  
(3.13)

By the monotonicity property of \( NF \), we have:

\[ A_i \subseteq A \Rightarrow NF(A_i) \subseteq NF(A) \Rightarrow x \in NF(A) \]  
(3.14)

Therefore, every point \( x \) in the union of all fixed points of \( A \) is in \( NF(A) \).

Now, observe that \( NF(A) \) is itself a fixed point by the idempotency property:

\[ NF(NF(A)) = NF(A) \]  
(3.15)

Thus \( NF(A) \) is equal to the union of all fixed points of \( A \). Having established this property, we can now use the constraint propagation algorithm shown in Figure 3.2. Given a set of narrowing functions \( Q \) and a subdomain \( A \), the function applies each narrowing function repeatedly until it finds a fixed point. Note that Relevant\(_{NF} \) refers to those values of the domain which may be pruned by the narrowing function \( NF \).

This algorithm is guaranteed to terminate because of the fixed point properties of a narrowing function satisfying the conditions specified above.

While there are many possible narrowing functions, the only one which concerns us is known as bounds consistency. This means that we keep track of the maximum and minimum values which each constrained variable can take, updating them as new constraints are added. If any variable ever has an empty domain, we know that there is no solution to the system of constraints.
function prune(Q, A)
    S ← ∅
    While Q ≠ ∅ do
        choose NF ∈ Q
        A' ← NF(A)
        if A' ≠ ∅ then return ∅
        P ← NF' ∈ S : ∃x ∈ Relevant_{NF}, A[x] ≠ A'[x]
        Q ← Q ∪ P
        S ← S \ P
        if A' = A then
            Q ← Q \ NF
            S ← S ∪ NF
        end if
        A ← A'
    end while
    return A
end function

Figure 3.1: Example code for constraint propagation (Cruz, 2006)
3.2. **FORMAL DEFINITION OF CONSTRAINT PROGRAMMING**

Consider the following system of constraints:

\[
\begin{align*}
1 \leq X & \leq 3 \\
1 \leq Y & \leq 5 \\
Z \leftrightarrow Y & < 3 \\
Z \rightarrow X & = 3 \\
Y & > X
\end{align*}
\]

The bidirectional arrow in Equation 3.19 is equivalent to listing the constraint twice, once with the arrow in either direction. Observe that Equations 3.18 and 3.19 are reified constraints, which attach the result of comparisons to a boolean value, which is \(Z\) in both of these cases. We will examine these in more detail in Subsection 3.3.1. Thus \(Z\) must take the value 0 or 1, so the first three constraints will set the domains as follows:

\[
D(X) = [1..3], \quad D(Y) = [1..5], \quad D(Z) = [0..1]
\]

For constraints involving multiple variables, we have propagation rules, which specify how the domains of the constrained variables are affected by each other. Equation 3.20 will cause propagation. As \(Y > X\), the minimum value \(Y\) may take must be greater than the minimum value \(X\) takes. Because we do not know whether \(Z\) is zero or one, Equations 3.18 and 3.19 will not cause any propagation at this stage. Thus, the domains of each variable will be as follows:

\[
D(X) = [1..3], \quad D(Y) = [2..5], \quad D(Z) = [0..1]
\]
We can see that the narrowing property holds as long as we never increase the bounds on any given interval. To prove the correctness, monotonicity and idempotency properties, we would need to prove them for each individual operation, such as addition, multiplication, greater than and so on, which we will not attempt here.

### 3.3 Constraint logic programming

We are particularly concerned with constraint logic programming over finite domains (Van Hentenryck, 1989) — that is, constraint programming implemented on top of a logic programming language, using variables which can only take a finite number of integer values.

#### 3.3.1 Reified constraints

In addition to the standard arithmetical operations, constraint logic programming allows reified constraints, which allow us to attach a Boolean variable to the result of a constraint. Reified constraints allow many logic-related operations which one might ordinarily expect to require choicepoints or if-then-else constructs to be performed purely using constraints.

However, reified constraints often do not set the bounds of the variables that they constrain as tightly as one might like. Consider the following constraints:

\[ \begin{align*}
A & \rightarrow B = 3 \\
\neg A & \rightarrow B = 4
\end{align*} \]
One might expect these constraints to set $D(B) = [3..4]$, but in fact they will not constrain $B$ at all until $A$ is constrained to either zero or one. This can lead to a lot of unnecessary computation in the labelling process, as values of $B$ for which no solution is possible are tested.

In view of this, it is necessary to use redundant constraints in order to improve the labelling efficiency. Redundant constraints are constraints which do not alter the set of possible solutions to the constraints already stored, but point out to the solver which values cannot lead to solutions, thereby pruning unproductive regions from the search space.

In this case, we would simply add the constraint

$$B \in [3..4]$$ (3.25)

### 3.3.2 Labelling

While maintaining bounds consistency can often show when a constraint program does not have a solution, this is not always the case. Moreover, it cannot actually find solutions except in trivial cases. To determine all solutions, we must label the constrained variables. Labelling is essentially an exhaustive search through the domains of each constrained variable.

Labelling is a very simple process in its basic form. We simply force a constrained variable to take one of the values in its domain, and then recursively label the remainder of the constrained variables until all have a specific value. When there is more than one possible value for a constrained variable, one value is chosen, and a choicepoint is created, allowing another value to be assigned upon backtracking, if the first value chosen does not allow any solutions, or if additional solutions are required.
CHAPTER 3. CONSTRAINT PROGRAMMING

```prolog
label([V|Vs]):-
    indomain(V),
    label(Vs).
label([]).

indomain(V):-
    mindomain(V, M),
    V = M.
indomain(V):-
    mindomain(V, M),
    V >= M + 1,
    indomain(V).
```

Figure 3.2: Example code for labelling (Van Hentenryck, 1989, Marriott and Stuckey, 1998)

Figure 3.3.2 shows sample code to accomplish this. Note that we will use Prolog-style code in this section, as we are dealing with constraint logic programming. This code relies on the `indomain` predicate, which binds its parameter to one of the possible values in its domain. In order to accomplish this, we use a simpler predicate, `mindomain`, which returns the smallest value in a variable’s domain; we presume that this is built into the CLP system. Using this predicate, we find all values in a domain by finding the lowest value in the domain, and then recursively finding all values that are at least one greater than this value.

For example, we could label the system of constraints 3.16-3.20, as shown in Figure 3.3. We begin with $X$, assigning it the first value in its range, which is one. This does not cause any additional propagation, so we continue by assigning $Y$ the first value in its range, two. This forces $Z$ to take the value zero, due to Equation 3.19.

However, Equation 3.20 then states that $X$ should be three, which is outside its current domain. Therefore, there is no solution using these assignments. We backtrack to the assignment of $Y$, and try the next value, three. This causes $Z$
3.3. **CONSTRAINT LOGIC PROGRAMMING**

\[
\begin{align*}
D(X) &= [1..3], D(Y) = [2..5], D(Z) = [0..1] \\
D(X) &= [1..1], D(Y) = [2..5], D(Z) = [0..1]
\end{align*}
\]

\[
\begin{align*}
D(X) &= [1..1], D(Y) = [2..2], D(Z) = \phi & D(X) &= [1..1], D(Y) = [3..3], D(Z) = [0..0] \\
\text{Failure due to empty domain} & \quad \text{Success - all domains have exactly one value}
\end{align*}
\]

Figure 3.3: Example of labelling

to take the value zero. Now Equation 3.20 is satisfied, and all domains now have contain exactly one value, so we have found a solution.

This process can be implemented in the constraint logic programming language itself, and varied in many ways in order to improve its efficiency.

Often there are many solutions to a given constraint satisfaction problem, and we wish to find the best one — the one which minimises some goal variable. In order to do this, we use the labelling process as described above, and record the values of each variable after the labelling is complete. We then add a constraint forcing the goal variable to take a value lower than that just achieved, and backtrack. Whenever we reach a solution with a new, lower value of the goal variable, we store the results and backtrack, until no further solutions are possible, at which point we are guaranteed to have found the optimal solution.

Searching for a solution is an expensive process, and is generally only done once, after all of the constraints have been added to the constraint store. A constraint logic program should not contain choicepoints based on the values of constrained variables, except in the labelling process.

It is desirable to reduce the domains of all the constrained variables as much as possible prior to the labelling process, in order to reduce the number of variable assignments which must be tested. Bounds consistency is still enforced during the labelling process, so labelling one variable can alter the domains of others.
CHAPTER 3. CONSTRAINT PROGRAMMING

labelFF(Vs):-
    selectFF(Vs, Vff, Vsothers),
    indomain(V),
    labelFF(Vsothers).

labelFF([]).

Figure 3.4: Example code for first-fail labelling

Therefore, the variables which have most effect on others should be labelled first, where possible.

3.4 Improving search performance

Many techniques have been developed to improve the performance of constraint logic programs. In this section, we describe some which are relevant to our work.

First-fail labelling In order to reduce the amount of backtracking involved in the labelling process, we can make use of first fail labelling (Haralick and Elliott, 1980). Using this method, we always choose to label the variable with the smallest current domain size. This is intended to reduce the amount of branching which occurs, and prune large regions of the search space as early as possible.

A simple implementation of this is given in Figure 3.4. This relies on the selectFF predicate, which splits a list of variables into the one with the smallest domain, and all the others. If this predicate were changed to simply select the first variable in the list, then this labelling procedure would be equivalent to the standard one given earlier in Figure 3.3.

For example, in the system of constraints describe earlier, we would begin with $Z$ when using first-fail labelling. We would first set $Z$ to zero. This causes $Y$ to be constrained to be three or more. This does not affect the domain of $X$. 
3.4. IMPROVING SEARCH PERFORMANCE

\[ D(X) = [1..3], D(Y) = [2..5], D(Z) = [0..1] \]
\[ D(X) = [1..3], D(Y) = [3..5], D(Z) = [0..0] \]
\[ D(X) = [1..1], D(Y) = [3..5], D(Z) = [0..0] \]
\[ D(X) = [1..1], D(Y) = [3..3], D(Z) = [0..0] \]

Success - all domains have exactly one value

Figure 3.5: Example of first-fail labelling

Now we have two variables each with three values in their domains, so we choose the first variable, X. We set it to one, which again causes no further propagation. Finally, we set Y to three, which does not cause any domain to become empty, so it gives us our solution. This process is shown in Figure 3.5.

In this case, both standard and first-fail labelling take three assignments to find a solution. First-fail labelling is not guaranteed to perform better than the standard variety, but it often does.

**Iterating through the goal variable:** One technique which can improve the efficiency of minimisation search is to iterate through the variable to be minimised, starting from zero, attempting to find a solution at each value.

While it may seem inefficient, this procedure can sometimes be faster than a simple search, because the goal variable is constrained to just one value for each call. Moreover, as soon as one of the calls succeeds, we are guaranteed to have the minimum value of the goal variable. Using a simple minimisation procedure, it is generally not immediately clear if a given solution is in fact the minimum, requiring further computation. Sample code for this algorithm is show in Figure 3.4.

**Limited-discrepancy search:** Limited-discrepancy searching requires a heuristic, which guesses the best value for each labelling choice (Harvey and Ginsberg,
minimiseI(Vs, Min, Max, G):-
  Min < Max,
  ( label(Vs)
    ->
    G = Min
  ;
  Min1 is Min + 1,
  minimiseI(Vs, Min1, Max, G)
).

Figure 3.6: Example code for iterative minimisation

1995). Any choice which differs from the heuristic is called a discrepancy. Using
this method, we search as before, but with the number of discrepancies limited to
some upper bound. This has the effect of reducing the search space, potentially
rejecting many possible solutions, but also allowing for a much faster search.

Unfortunately, it may be unclear how many discrepancies to allow when using
limited-discrepancy searching. If we allow too many discrepancies, the pruning
effect is reduced or eliminated, making limited-discrepancy search resemble the
normal searching procedure, while if we allow too few, we may not find a solution,
or only a very restricted, sub-optimal set of solutions. There is no obvious method
of selecting an appropriate number of discrepancies.

**Optimistic partitioning:** Optimistic partitioning (Prestwich and Mudambi, 1995) requires no further information except the total range of the search space. We split
the search space in two and search for a solution in the first half of the space. If one
is found, we partition again, using this solution as the new upper bound. If there is
no solution in the first half, we search again, in the upper half of the search space.
If there is a solution in the upper half, we partition again, using the midpoint and
3.4. IMPROVING SEARCH PERFORMANCE

minimise_OP(VsF, Min0, Max0, Vs0, Vs) :-
    Mid = (Min0+Max0)//2,
    (Min0 <= G, G <= Mid,
     label(VsF, Vs1, G), indomain(G)
    ->
     Max = G-1,
     minimise_OP(Data, Min0, Max, Vs1, Vs)
    ;
    (Mid+1 <= G, G <= Max0,
     label(VsF, Vs1, G), indomain(G)
    ->
     Min = Mid+1, Max = G-1,
     minimise_OP(Data, Min, Max, Vs1, Vs)
    ;
     Vs = Vs0
     )
    )
).

Figure 3.7: Example implementation of optimistic partitioning (Marriott and Stuckey, 1998)

the solution in the second half of the search space as the lower and upper bounds, respectively.

We show how to implement this method in Figure 3.4.

In either case, we store the current solution as the best one thus far, and if at any time we fail to find a solution in the given range, then we know that the previous best solution is the overall minimum.

Restricting the number of assignments: A final, crude method of improving performance is to simply restrict the total number of assignments which can be made. Once we have reached this limit, we can simply return the best solution found up to that point. While it is obviously preferable to find the best solution,
the search space is exponential in the number of constrained variables, so there may be cases in which this problem is intractable.

3.4.1 Caching search states

Some constraint programs can be considerably sped up by caching previously visited search states (Smith, 2005).

Usually in a constraint satisfaction or optimisation problem, once a search state has been shown to not lead to a solution, it is of no further use; we will not reach the same state again. In some cases, however, two states which are reached by different search paths can be equivalent.

Two states are considered to be equivalent if they require the same search to complete. The remaining, non-assigned variables must be the same, and have the same domains, and any assignments which can be made to these variables must be the same in both cases. The result of a search in an equivalent state will be the same as that of any other search in the same state.

Smith shows how permutation problems, in particular, have many equivalent states, and can benefit from caching. A permutation problem is defined as a CSP with the same number of values as variables, where every variable must have a different value. In some permutation problems, the satisfiability at any given point on the search tree only depends on the variables and values which have been used, not the actual pairs of values and variables.

In one example given, two assignments of the first $k$ variables can be considered equivalent if the sets of the first $k - 1$ values and $k - 1$ variables are the same, and the $k$th variable takes the same value. In this case, at any point on the search tree where we have made $k$ assignments, we can cache the sets of the first $k - 1$ values and variables which have been used, and the $k$th variable, with its value. We then follow the search to determine whether or not this assignment leads to a
solution. If not, we record this in the cache, and if any subsequent search leads to the same state, we can terminate searching along this branch with no further effort.

Smith also applies caching to an optimisation problem. In this case, we need to consider the cost associated with a solution, rather than merely its existence.

The overall cost of an assignment is the sum of the cost incurred up to any given point, and the cost of the remaining assignments after that point, and for caching to be applicable, the two costs must be independent. Therefore, if we have any two search paths which use the same values and variables up to a given point, and wish to extend one of them to find the optimal solution, the cheaper one is guaranteed to come from the cheaper of the partially-completed paths.

This also allows us to cache search states. For every partially completed path, we again store the sets of variables and values, along with the cost of the path. When we attempt to find another solution which uses the same set of variables and assignments, we check to see if an equivalent path has been cached. If it has, we proceed only if our current path is cheaper than the cached path. If so, we replace the path in the cache with the current path.

### 3.5 Constraint programming in natural language processing

In this section, we review prior applications of constraint programming in natural language processing. While logic programming, from which much constraint programming is derived, was originally invented for parsing natural language, the fields have largely diverged since then. There is no work spanning both fields that we are aware of which is directly relevant to us, except those which have already been covered in Chapter 2.
These works are the theory of document structuring by Power (2000), Power et al. (2003). They formulate NLG, particularly the problem of choosing discourse markers, as a constraint satisfaction problem, and use constraint logic programming to solve it.

Reiter and Dale (2000) claim that several NLG tasks are amenable to representation as constraint satisfaction problems, but do not actually formulate the problem or build a system to solve it. Indeed, they identify the area as one which has not yet been thoroughly researched.

There have been several other attempts to apply constraint programming to problems in natural language processing, but we are unaware of any which directly relate to our work.

For example, Duchier (2000) has surveyed the usage of constraint programming in natural language processing applications. The work consists mostly of tutorial material on constraint programming, interspersed with applications to natural language processing.

It covers German determiner-adjective-noun agreement, dominance constraints and parsing dependency grammars, which is perhaps the most common application of constraint programming in this area. Dependency grammars use different tree structures to the traditional phrase structure grammars. The construction of one of these trees is defined as a constraint satisfaction problem; solving the CSP gives a parse tree for a given sentence.

Another sample work also deals with property grammars (Blache, 2004). It seeks to represent constructions such as nouns and verbs with three components: label, form and properties. Of these, properties are represented as constraints, and describe features such as number or gender agreement and ordering. As with the other material in this section, it does not directly deal with the natural language-related problems that we are interested.
In a third paper, Saint-Dizier (1989) designs a new programming language, Dislog, which extends Prolog. The new language is used to model an aspect of Government and Binding theory (Chomsky, 1981, Black, 1999), which represents a universal grammar which is hypothesised to make up a large proportion of the grammar of all languages.

None of these applications are closely related to our work, and all are fairly typical of the applications of constraint programming in natural language processing. They also focus almost exclusively on the linguistic issues of their respective problems, ignoring details of the constraint solving system.

### 3.6 Summary

We have overviewed the field of constraint programming, focussing particularly on those fields which are relevant to our work: constraint logic programming over finite domains.

We described the basic operation of such a constraint programming system and gave formal definitions of its major components. We then examined constraint logic programming, along with several techniques which may be used to optimise its performance. Finally, we examined prior work on constraint programming applications to natural language processing, finding very little material which is directly relevant to us.

Having reviewed all the necessary concepts in language technology and constraint programming, we now turn our attention to our contributions.
In this chapter we describe our document structuring system. We begin with an overview of the architecture of the system. Then, in Section 4.2, we explain how we reimplemented a document structurer as described by Power et al. (2003), with enhanced constraint handling. We describe how to realise the document structures this generates as text in Section 4.3, and then describe some additional features which we added to the system: a set of layout constraints, in Section 4.4, and summarisation capabilities, in Section 4.5. We also describe some ways in which our system’s performance can be improved in Section 4.6. We describe the ECLiPSe Prolog implementation of our system in 4.7. Finally, we compare our system to Reiter and Dale’s reference architecture in Section 4.8 and to RAGS in Section 4.9.

4.1 Overview

Our architecture uses a three-stage pipeline, as shown in Figure 4.1. It is similar to the reference architecture of Reiter and Dale (2000).
We start with a rhetorical structure tree as input. The internal nodes of the tree are rhetorical relations, and the leaves are spans of text. In the first stage of the system, we use constraint logic programming to define the possible values which can be taken the variables at each level of the document structure. These constraints are given in terms of the input rhetorical structure, and other document variables, in a similar manner to the system described in Power et al. (2003).

In addition to their document structure variables, we allow summarisation, so that some of the information in the rhetorical structure may be omitted from the output. This adds a fifth document structure variable to leaf nodes, indicating whether they are enabled or not. We also include layout constraints, which give the exact position (row and column) of every word and punctuation mark in the structure, in terms of the basic document variables. These two additions enable us to control the size of the output. This allows us to vary our output for different audiences or display systems, for example.

The document structure produced by the first stage is generally incomplete. All of the document structure variables are constrained so as to satisfy any possible realisation of the rhetorical structure. However, except in trivial cases, there are multiple valid document structures which correspond to a given rhetorical structure, so merely constraining the document structure variables in this way does not actually give us a solution.
Because of this, we need to assign values to the document structure which satisfy the constraints. This process, known as labelling, was described in Subsection 3.3.2. It constitutes the second stage of our system. The process of labelling the basic document structure variables will also assign the correct values to the position variables, without any need to directly manipulate them.

With multiple candidate solutions, we need some method of choosing between them. Following Power et al. (2003), we can choose the solution with the lowest defect count. Additionally, because of our addition of layout constraints and the capacity for summarisation, we can consider the size of the output document and the weights of the nodes on the rhetorical structure which have been excluded from it.

The labelling process can be modified to minimise a certain goal variable; this was also explained in Subsection 3.3.2. Depending on our aims, we could choose this goal variable to be the defect count, the summarisation score, the size, or some combination of these.

The final stage is the syntactic realiser, which generates an actual document from a fully-labelled document structure. While the position of every document unit is already determined as a consequence of the labelling, there is still some work to be done. In particular, we need to determine which units should commence with a capital letter, and what punctuation to use when terminating units.

### 4.2 Implementing a document structurer

We implemented a document structurer in ECLiPSe Prolog, using a constraint model based on that of Power et al. (2003). However, our program differs in that it simultaneously evaluates both the required constraints to create the document structure, and the constraints required to find the defects. It produces as output
both a document structure and a count of its defects. While generating a document structure, our system maintains a count of the defects. If the structure has already generated more defects than some previous version, we can abandon the current structure, significantly reducing the total search space. By contrast, the document structurer of Power et al. (2003) produces a document structure which must then be evaluated. In order to find the document structure with the lowest defect count, their system must generate all possible document structures.

Moreover, our document structure contains positioning information for each word in the output, including the lines on which each word is to be displayed. This information allows for very fine-grained control over the amount of space which is occupied by a rendered document.

We have also made some modifications to the document structuring process of Power et al. — we do not make a distinction between homomorphic and isomorphic document structures, and treat indentation differently. We believe that these modifications make the theory simpler and more consistent for implementation purposes. We begin by describing these alterations.

### 4.2.1 Modifications to the document structuring process

**Homomorphic and isomorphic document structures**

Power et al. (2003) characterise document structures as being either isomorphic or homomorphic to the rhetorical structure from which they were created. A document structure and rhetorical structure are isomorphic if every node on the document structure has a strictly higher level than the levels of the nodes corresponding to its children on the rhetorical structure. If any node has the same level as the nodes corresponding to its children on the rhetorical structure, then the
structures are homomorphic. Note that this differs from the normal mathematical definition of homomorphic functions between trees.

An example of this is given above, where Figure 4.4 is homomorphic to the rhetorical structure in Figure 4.2, while Figure 4.3 is isomorphic to it.

In both our discussions and implementation, we find it more convenient to use the same tree structure for rhetorical and document structures, even if parent and child nodes on the document structure have the same level. We do, however, acknowledge that it can useful to generate document structures with parents and children of the same level; the distinction is largely one of presentation, rather than content. Indeed, it is quite possible for a rhetorical structure to be of higher depth than the number of allowed levels in the document structure, thereby preventing a document structure from being isomorphic to the rhetorical structure.

The reason we have for taking this position is that the document structuring and realisation of a parent, nucleus and satellite works in the same way, whether the children are of the same level as the parent, or a lower one. In the above example, there is no special behaviour which makes the case where the children are text-sentences any different from the cases where they are text-clauses or text-phrases.
Although one of the main ingredients is penicillin the medicine has no significant side-effects. However, some people might suffer a mild allergic reaction.

```
concession
  Level: 3
  Position: 0
  Indentation: 0
  Connective: however
```

```
“some people might suffer
a mild allergic reaction”
  Level: 2
  Position: 0
  Indentation: 0
  Connective: although
```

```
“the medicine has no
significant side-effects”
  Level: 0
  Position: 1
  Indentation: 0

“one of the main ingredients
is penicillin”
  Level: 0
  Position: 0
  Indentation:
```

Figure 4.3: Document structure which is isomorphic to its rhetorical structure

The positions of the nucleus and satellite are given relative to each other, ignoring the parent relation, even when the parent is of the same level. Similarly, the connective is a property of the parent, describing the relation between its children, and is attached to one of them. One might expect these properties to change if the parent node was actually a sibling of the nucleus and satellite, but they remain the same. Therefore, we believe that it is more accurate to always consider a parent node to be at a higher level than its children on the document structure tree.

This will simplify matters considerably when we deal with structure realisation and summarisation, which become unduly complicated were we to consider relations to be siblings of their component facts.
Some people might suffer a mild allergic reaction, although although one of the main ingredients is penicillin, the medicine has no significant side-effects.

```
concession
Level: 3
Position: 0
Indentation: 0
Connective: although
```

```
“some people might suffer a mild allergic reaction”
Level: 2
Position: 0
Indentation: 0
Connective: although
```

```
“the medicine has no significant side-effects”
Level: 1
Position: 1
Indentation: 0
```

```
“one of the main ingredients is penicillin”
Level: 1
Position: 0
Indentation: 0
```

Figure 4.4: Document structure which is homomorphic to its rhetorical structure

**Indentation**

Power et al. (2003) define indentation as one of the four variables in the document structure. This is a non-negative integer, representing the number of times a particular unit should be indented, relative to the root node. However, we contend that the most important feature is not the total amount of indentation in an element, but rather whether it is indented relative to its parent.

If an element is indented relative to its parent, it must be realised on its own line and with its own bullet point, whereas if its parent is at the same level, it can be part of a larger structure, which is all indented. Consider the two rhetorical structures and realisations in Figure 4.2.1.
The FDA approves Elixir+, and
• the FDA bans Elixir

(a) Bulleted List Format

The FDA approves Elixir+, but the FDA bans Elixir.

(b) Inline List Format

In both cases, the elementary units have level zero and indentation one, meaning that they are realised as phrases which are terminated with a comma, and are indented once from the base. However, in the first case they are on the same line, while in the second they are rendered on separate lines.

The reason for this is that in the first case, they are both part of a concession, which is a list element, while in the second, they are list elements themselves. It is clear that when list elements are indented, they should each be realised on a separate line, but this is not actually spelled out in the document structure. While this can be inferred from the relative indentations of the elementary units to their parents, but this adds unwanted complexity to the implementation, as well as violating the principle of modularity of the document units.
Elixir is safe to use since

- the medicine has been thoroughly tested and,
- it has no significant side effects.

Accordingly, we consider the indentation to be a binary variable, indicating whether a given document unit is indented relative to its parent. Note that the actual amount of indentation at any given node is just the sum of the indentation variables of it and each of its ancestors.

Other than these two modifications, our basic document structuring variables are the same as those that of Power et al. (2003), which we described in Section 2.3. We now give our constraint model for this process.

### 4.2.2 Constraint model

The level, indentation, position and connective values at a node $X$ are denoted by $L(X)$, $I(X)$, $P(X)$ and $C(X)$, respectively. We also use a Boolean function $E(X)$, which is true if $X$ is an elementary unit, and false otherwise. The root level is specified as an input to the document structuring process.

The root node is always known as $R$. Given an arbitrary nucleus-satellite relation, we define the parent node as $P$, the nucleus $N$, and the satellite $S$. In the case of a multinuclear relation, we simply number the children starting from 1, and refer to the $n^{th}$ child as $C_n$. The node to which the connective is attached is labelled $A$, while the other node (for a nucleus-satellite relation) is labelled $NA$. Note that this can be either the nucleus or satellite, depending on the connective in question. We give an example of these labels in Figure 4.6, which correspond to the text in Figure 4.5.
Figure 4.6: Different labels which can be used to refer to nodes

The position variable (denoted by \( Pos \)) specifies the order in which a nucleus and satellite of the same relation will be ordered. Initially, they are constrained so that they can occur in either place. Clearly, they cannot have the same position, so this must be added as an extra constraint. The connective chosen to represent the parent relation may restrict this ordering, as we describe later.

\[
Pos(N) \in 0..1 \\
Pos(S) \in 0..1
\] (4.1) (4.2)
4.2. IMPLEMENTING A DOCUMENT STRUCTURER

Because multinuclear relations can have many children, we do not permit them to be reordered. Otherwise, a multinuclear relation with \( n \) children would have \( n! \) possible orderings, which would clearly result in an intractable number of document structures.

\[
\text{Pos}(N) \neq P(S) \tag{4.3}
\]

\[
\text{Pos}(C_n) = n \tag{4.4}
\]

The level variable indicates what sort of linguistic structure is generated from the document unit, such as a phrase, sentence or paragraph. All siblings have the same level. For nucleus-satellite relations, a unit must have a level less than or equal to its parent. As with the position variable, the level may be further affected by the choice of connective.

Also, the level of all non-root nodes must be strictly less than the root level, which represents the level of the entire document. This level is given as an input to the document structuring process, so that if we wanted the document to consist of a single paragraph, for example, we would specify a root level of three. Allowing sub-nodes of the root to also be paragraphs will cause the document to consist of multiple paragraphs, contrary to our requirements, so we force all non-root nodes to take strictly lower values than the root. Therefore, we have the following constraints:

\[
L(N) = L(S) \tag{4.5}
\]

\[
L(N) \leq L(P) \tag{4.6}
\]

\[
L(N) < L(R) \tag{4.7}
\]
As with nucleus-satellite relations, all children of a multinuclear relation have the same level, which must be strictly less than that of the root node. If the children are indented, then they may have a greater level than the parent, up to one less than the level of the entire document. If they are not indented, then they can be at most the same level as their parent.

\[ L(C_n) = L(C_0) \forall n \]  
\[ L(C_0) < L(R) \]  
\[ \neg I(C) \Rightarrow L(C_0) \in 0..L(P) \]

Recall that we define the indentation variable as the difference in indentation between a unit and its parent. Therefore, it can only be zero or one. In the case of a nucleus-satellite relation, we do not permit indentation at all, so it is always zero.

\[ I(N) = 0 \]  
\[ I(S) = 0 \]

For a multinuclear relation, we have a choice as to whether we indent its children or not. Note that all the children share the same indentation.

\[ I(C_n) = I(C_0) \forall n \]  
\[ I(C_0) \in 0..1 \]
When visiting each node, we constrain the level, indentation and position variables of that node’s children. Because of this, and the fact that leaf nodes have no connective attached, there is no further work to do when visiting a leaf node. However, it means that we have to treat the root node separately for these three variables.

The position variable is irrelevant for the root, because it only specifies the position relative to sibling nodes. The level of the root node is given beforehand, and the indentation should always be zero, as there is little point in indenting an entire document.

\[
\begin{align*}
\text{Pos}(R) & = 1 \\
\text{I}(R) & = 0
\end{align*}
\]  

(4.16)  

(4.17)

The connective variable is constrained in two ways. Firstly, for all internal nodes, the connective variable itself depends on the rhetorical relation of that node. Each relation has a set of connectives which realise it. (Recall that rhetorical relations are only present in internal nodes. Therefore, the connective variable is unused for leaf nodes.)

Secondly, for nucleus-satellite relations, the level and position of a node’s children are constrained according to the type of connective used. The node which is affected is the one to which the connective is attached, which we denote by \( A \). This node may be either the nucleus or satellite, depending on the Locus value of the particular connective being used.

\[
C(P) \in \text{Coordinators} \implies L(A) = 0
\]  

(4.18)
Note that, unlike the other document variables, the connective is constrained at the root node in exactly the same way it is for any other node, depending only on the type of the root (nucleus-satellite, multinuclear or elementary).

This mapping from a rhetorical structure to a document structure involves a large number of independent choices. As the rhetorical structure grows, the number of corresponding document structures grows exponentially. We would like to choose only those with less than a fixed number of defects, or perhaps the one having the fewest defects. Generating all possible document structures, to only choose one, turns out to be extremely wasteful.

Our constraint programming model constrains the count of defects while we are generating document structures. This allows us to stop generating a structure as soon as it has more defects than the upper limit (for minimisation this limit is defined by the number of defects in the best answer so far). Moreover, because a partially generated document structure may in fact lead to several document structures, each of which will have at least as many defects as the partially generated structure, we can prune entire branches from the search tree.

We accomplish this by expressing the rules for the defects as constraints. Each constraint is evaluated at every node on the document structure tree, indicating whether or not the defect occurs at that point. For defects which involve two different nodes, we count the defect only once. Where a defect involves the relationship between a node and its children, we note the defect at the parent. Single-sentence paragraphs may also occur on leaf nodes, without any effects from other nodes, in which case they are noted there, and repeated connectives are noted where the connective occurs for the second time.

\[
C(P) \in \text{Subordinators} \; \Rightarrow \; \text{Pos}(A) = 1 \tag{4.19}
\]
\[
C(P) \in \text{Parentheticals} \; \Rightarrow \; \text{Pos}(A) = 1 \land L(A) > 0 \tag{4.20}
\]
4.2. **IMPLEMENTING A DOCUMENT STRUCTURER**

We simply sum them all to obtain the total count of defects found so far. The constraints are stored for each node, along with the other document structure parameters. We now examine the constraints for each defect type.

The nucleus-before-satellite defect is trivial, given the positions of both nucleus and satellite:

\[ P(N) < P(S) \Rightarrow \text{NucleusBeforeSatellite}(P) \]  \hspace{1cm} (4.21)

The left-branching defect is somewhat more complicated. It requires that there be an elementary child on the right, and a non-elementary child on the left. Given that we are working with the nucleus and satellite, rather than left and right children, we need to consider two cases: when the nucleus is first, and when the satellite is.

\[
\begin{align*}
E(N) \land \neg E(S) \land P(S) &< P(N) \lor \\
E(S) \land \neg E(N) \land P(N) &< P(S) \\
\Rightarrow \text{LeftBranching}(P)
\end{align*}
\]

An oversimple clause is relatively easy to detect. Both the nucleus and satellite to be elementary, and the whole relation needs to be of text-clause level, or 1.

\[ E(N) \land E(S) \land L(N) = 1 \Rightarrow \text{OverSimpleClause}(P) \]  \hspace{1cm} (4.22)

A single sentence paragraph occurs when a structure is of paragraph or higher level, and its children are of clause or lower, meaning that they do not form separate sentences. It can also occur when an elementary unit (which we denote by \( E \)) is realised at a paragraph or higher level. Note that this is the only defect which can occur at an elementary structure. The others involve relations between a node and its children, or, in the case of a repeated connective, require a connective
which is not used for elementary units. Here we use $C$ to represent an arbitrary child of the relation. We could use $C_1$ for multinuclear relations, or $N$ for nucleus-satellite relations; note that the levels of all sibling nodes are the same in both cases.

$$L(P) \geq 3 \land L(C) < 2 \lor E(E) \land L(E) \geq 3 \Rightarrow \text{SingleSentenceParagraph}(P) \quad (4.23)$$

It is trivial to determine whether a lost rhetorical occurs; the parent and children are of the same level. As before, $C$ denotes any child node of the nucleus.

$$L(N) = L(C) \Rightarrow \text{LostRhetoricalGrouping} \quad (4.24)$$

The repeated connective defect requires some slightly more complicated programming. Rather than involving only a node and its parents, it occurs when any ancestor of a node uses the same connective. Therefore, at each node we keep a list of all the connectives used by that node’s ancestors, which we denote by $AC(P)$. Using this, it is easy to determine whether a repeated connective has occurred:

$$C(P) \in AC(P) \Rightarrow \text{RepeatedConnective}(P) \quad (4.25)$$

Observe that the above defects are all only defined for parent nodes. All of the defects except the repeated connective involve a node and its children, or siblings. Any occurrence of a defect should only be scored once; we choose to represent it at the parent node, as it makes the constraints simpler. The repeated connective defect requires a connective, which is only present in nodes which have children, so again we only record it for parent nodes.
4.3 Realisation

A document structure is meant to provide all the information necessary to generate the corresponding document, with the exception of minor formatting details which do not affect the document’s meaning Power et al. (2003). We have modified the details of the structure slightly, but the overall principle still holds. However, the document structure is clearly not a document in itself, and the process of turning it into a document, structure realisation, is non-trivial. We have developed our own algorithms for structure realisation, which we describe in this section.

Power et al. (2003) briefly describe the tasks which must be performed after document structuring. The first is syntactic realisation, which involves the conversion of semantic representations at the leaf nodes of the document structure into text. This would be the more complicated of the two processes. However, they do not describe it, or the format their elementary propositions might take; their paper is exclusively concerned with document structuring.

We also do not deal with the problem of converting semantic to textual representations. However, because of our interests in layout (see Section 4.4), we
need to produce fully rendered documents as output, so we use text for our elementary propositions. (We consider how our system might be extended to deal with abstract, semantic representations, thereby becoming a fully-fledged NLG system, in Section 6.2.)

The second and final step is called **formatting**, and involves generating the appropriate text in a form suitable for display. Power et al. (2003) mention it only in passing. As an example of what this module would do, they suggest that it would determine whether paragraphs should be realised by a vertical gap, or a tab character on a new line. We presume that issues such as capitalisation and termination of sentences would be dealt with by this module, as they are not included in the document structuring process; doing so would lose some of the abstraction of the document structure from an actual text. However, this is not explicitly stated; Power et al. do not give any details on the actual operation of this task.

This process is referred to as **surface realisation** in RAGS (Mellish et al., 2004), and includes elements of both **structure realisation** and **linguistic realisation** from the consensus architecture of Reiter and Dale (2000).

Because we are producing complete documents, we do need to consider this procedure. In particular, we must correctly capitalise the first letter in a sentence, terminate document units with the appropriate punctuation and indent units which require it.

At a first glance, it may appear that this problem is trivial. The level parameter in the document structure specifies whether a given unit is a phrase, clause or sentence, for example, which are terminated by a comma, semicolon and full stop, respectively. Similarly, sentences and higher levels begin with a capital letter, while lower level units do not.
4.3. REALISATION

The FDA approves Elixir, but the FDA bans Elixir+.

(a) Document structure

#the FDA approves Elixir, but the FDA bans Elixir+.
The FDA approves Elixir, but the FDA bans Elixir+.

(b) Possible realisations

Figure 4.7: A first, incorrect attempt at structure realisation, compared to the correct version

However, consider the document structure and realisations in Figure 4.7. Both elementary units are phrases, but the first is the beginning of the entire paragraph. Therefore, it begins with a capital letter, when clauses in general do not. Similarly, the second phrase is the last element in the paragraph, so it is terminated with a full stop rather than a comma. Note, however, that the first element is still terminated with a comma, and the second one begins with a lowercase letter. Both elements behave partially like phrases, and partially like higher-level elements.

Clearly, we need to consider more than the level of the current element when generating capitalisation and punctuation. While the approach we take may seem to be more complicated than necessary, its advantage is that it absorbs the entire realisation task into that of labelling the document structure. Thus, all the variables associated with realisation — down to the positioning of individual words
— are expressed in terms of the basic document structure variables. Once we determine the unique document structure, all that needs to be done is to actually display the words in their designated locations.

Generating the correct punctuation is relatively simple: when an element is the rightmost rendered child of some relation, we do not generate any punctuation after that element, leaving it to the higher-level ancestor node.

Note that the rightmost rendered element may or may not be the rightmost element in the rhetorical structure. For nucleus-satellite relations, we always list the nucleus first in a rhetorical structure, but the satellite is generally put first in real text. However, we do not reorder multinuclear relations.

Recall from 2.3 that the Position variable in the document structure specifies the relative positions of the nucleus and satellite. The left position is designated as 0, and right as 1. Therefore, determining whether a node of a nucleus-satellite relation is the rightmost child is trivial:

\[ RC(X) = Pos(X) \] (4.26)

Correct capitalisation is a little more involved. It involves knowledge of two separate nodes simultaneously. Consider Figure 4.8; the paragraph-level concession relation forces us to capitalise the first letter in the phrase-level elementary relation, but the letter which is to be capitalised is determined by the phrase-level relation, so we cannot simply render the character at the earlier node.

In order to deal with capitalisation, we define the variable StartLevel, abbreviated as \( SL \). This exists at every node, representing the level at which the start of the node is realised, as distinct from the level of the node itself. When rendering high-level nodes with children, we need to pass the level of the higher level node down to the child which is to be rendered first.
concession
Level: 3
Position: 0
Indentation: 0
Connective: however

“the FDA approves Elixir”  “the FDA bans Elixir+”
Level: 2  Level: 2
Position: 0  Position: 1
Indentation: 0  Indentation: 1

(a) Document structure

#The FDA approves Elixir. However, The FDA bans Elixir+. The FDA approves Elixir. However, the FDA bans Elixir+.

(b) Possible realisations

Figure 4.8: A second attempt at structure realisation, compared to the correct version
There is a further complication: connectives occur at the start of elementary units, and therefore they may be capitalised instead of the elementary unit. We resolve this difficulty by realising the connective at the level that the child to which it is attached would otherwise have taken. The child element is realised at phrase level.

Therefore, we need to consider both the effects of the connective, and of higher level ancestor nodes when deciding what level we should use to begin an element. For nucleus-satellite relations, we need to know if the connective is attached to the leftmost child. We represent this situation by a Boolean variable, \( CL \). This requires us to use the \( Locus \) variable of the discourse connective, which determines whether the connective is attached to the satellite or nucleus. This was described in Subsection 2.3.2.

\[
CL(P) \leftrightarrow (P(N) = 0 \land Locus = nucleus) \lor (Pos(S) = 0 \land Locus = satellite)
\]

We define the variable \( SL_0 \) for every node as the starting level which is passed down to it by the parent node. It may be used at that particular node, or passed down further, depending on whether the current node actually provides the start of the text which realises it. At each node, we calculate \( SL \) for that node, and \( SL_0 \) for all of its immediate children. Because the root node has no parent, it uses its own level as the starting level also:

\[
SL_0(R) = L(R)
\]

For a nucleus-satellite node \( P \) with nucleus \( N \) and satellite \( S \), the connective should be realised at \( SL_0 \), the level passed down from the parent node, and the right child should be passed its actual level. If the connective is attached to the
right child, then it should be realised at the level of that child, and we should pass
the current starting level down to the left child. In either case, the child to which
the connective is attached should be realised at level zero, because it will begin
partway through a sentence, after the connective.

\[
\begin{align*}
CL(P) & \rightarrow SL(P) = SL_0(P) \land SL_0(NC) = L(C) \\
-CL(P) & \rightarrow SL(P) = L(C) \land SL_0(NC) = SL_0(P) \\
SL_0(C) & = 0
\end{align*}
\] (4.29)

Now consider a multinuclear relation \( P \), which may have more than two
children. We denote the first of these by \( F \) and subsequent children by \( S_n \), up
to the final child, \( C \). For multinuclear relations, we always render the connective
before the last element, which simplifies the logic somewhat.

Because the connective will be placed before the last child, it should be
realised at the level of the children. The starting level of the entire structure should
be passed down to the first element, which will be the first part of the structure to
be rendered. The middle elements, if they exist, should be realised at their own
level; note that the constraint of sibling inequality (see 2.3) forces the actual level
of each child to be the same. Finally, the last element, to which the connective
is attached, should be begun at level zero, because the connective will be realised
immediately before it, in the same sentence.

\[
\begin{align*}
SL(P) & = L(C_i) \\
SL_0(C_1) & = SL_0(P) \\
SL_0(C_i) & = L(C_1), \ 1 < i < n
\end{align*}
\] (4.32)

(4.33)

(4.34)


\[ SL_0(C_n) = 0 \quad (4.35) \]

Determining the starting level for an elementary relation is trivial, as the work has already been done by the parent relations in setting \( SL_0 \).

\[ SL(X) = SL_0(X) \quad (4.36) \]

4.4 Layout constraints

In this section, we present a constraint-based system which augments a document structure with layout information, down to the row and column of every character. This information is useful, because it allows us to determine exactly how much space a given document structure will take up, enabling us to better use the space that we do have. It is the first step in controlling the document structuring process in order to produce documents reduced to fit any given size, starting from a large rhetorical structure.

There are several reasons why we might wish to perform this task. We might wish to generate different versions of a document, depending on the level of detail required by the user or application. It may be desirable to optimise our generation for display on different displays. For example, we might render the full version of a document for display on a PC, but reduce it for use on a PDA.

A simple approach to this problem would be to estimate the size of the output, purely using the lengths of the text spans in the rhetorical structure. If, for example, we use 80 characters per line, we could estimate the amount of space taken by a document unit by dividing the number of characters in it by 80. We could make this estimate more accurate by reducing the line length by half the average word length, to account for the unused space at the right margin when a
word is wrapped to the next line, and to half of its normal length on the last line of
indented or paragraph-level document units, which are terminated by new lines.

However, we want to account for the effects of document structuring decisions.
Our document structures are often quite small, and the effects of document struc-
turing decisions on the size of the output can be significant. This applies to those
involving the aforementioned indented and paragraph-level units in particular. In
order to accurately capture the effects of these decisions, we need to model the
size of the document at a lower level, accounting for each word and line break.

It should be noted that by realising a document structure, we would generate
all of this positioning information anyway, even if we did not explicitly use it. The
difference between the system which we describe here and a conventional method
of realisation is that we are calculating this positioning information using con-
straints, as part of the document structuring process. In contrast, a conventional
system would perform this task later, on a fully specified document structure.
Because of this, we can use partial information about the layout to guide the
document structuring process and optimise the result to fit into a certain size.

While document structuring decisions can have affect the size of the output,
you are not sufficient to reduce the size of the output dramatically, such as by
50% or more. In order to make these sort of reductions in the size of the output,
we need to omit parts of the rhetorical structure from the output. We deal with
this process in Section 4.5.

We now examine the constraint model which allows us to perform this task.

### 4.4.1 Positioning information

We model our page of text as a grid of width $W$ characters, which can extend
indefinitely. A location on this page is given by the row $R$ and column $C$. We store
this information for the start and end of every rhetorical element, and every word
Although the FDA bans Elixir, the FDA approves Elixir+.

---

Figure 4.9: An example of text positioning information

contained inside them. For example, in Figure 4.9, the first rhetorical element begins at (1, 1) and ends at (1, 30), and the second begins at (1, 31) and ends at (2, 17).

Note that this presumes that every character is of equal width. While this assumption clearly does not hold for most typefaces, it leads to a somewhat clearer constraint model, and does not significantly affect the system’s performance. We consider how our layout model would have to be changed to account for variable-width typefaces in Section 6.2.

In addition to the row and column, we use the raw position $P$. It subsumes both the row and column in one variable, as follows:

$$ P = R \times W + C $$

Given either a raw position or both the row and column, we can work out the other components of the tuple. The reason for including the raw position is that the redundant constraints are simpler and more effective on a single variable.

There are basic constraints which hold for all positions, which simply describe the layout of the page:

$$ R \geq 0 $$

$$ C \geq 0 $$
4.4. LAYOUT CONSTRAINTS

\[ C < W \]
\[ P \geq 0 \]

4.4.2 Word positioning

Using these descriptions of the position of some text, we determine the position of every word which will be realised from a document structure. We can find the order in which the document elements occur using the position variables, and equate the starting position of any given node with the finishing position of its preceding node.

This will order each node in the rhetorical structure into a document. We also need to calculate the space taken by individual document units. In general, they are made up of multiple words. In order to account for line-breaking, we must determine the location of every word.

When inserting a word of length \( L \), we start with \( P_0 \), the position at which the last character was rendered. We wish to determine \( P_s \), which is where this word begins (which will differ from \( P_0 \) if we need to insert a line break before rendering the word), and \( P_f \), the end of the word. We must therefore determine if a line break occurs, and then leave the right amount of space for the word.

We need to account for line breaks. If a word would end past the width of the page, it must be rendered on the next line instead. We may also need to add spacing at the beginning of a line due to indentation. We define the amount of leading spaces due to indentation as \( N \). We represent the number of spaces in one level of indentation by \( IS \). This gives us the following:

\[
N(X) = IS \times \sum_{P \in AN(X)} I(P) \quad (4.38)
\]
Note that $AN(X)$ denotes all the ancestors of $X$. Recall that we define indentation as a binary value, indicating only whether a given node is indented relative to its parent. Therefore, we must sum the indentation values for all ancestors of a given node to find out how far it is indented relative to the root.

In the following, $W$ is the line width, $T$ is the length of any trailing spaces required. Note that each position variable is split into row and column variables, as before. Some constraints are better expressed in terms of the raw positions, while others are described using the row or column variables.

\begin{align*}
C_0 < N & \Rightarrow C_s = N \land R_s = R_0 \quad (4.39) \\
C_0 \geq N \land C_0 + L < W & \Rightarrow C_s = C_0 \land R_s = R_0 \quad (4.40) \\
C_0 \geq N \land C_0 + L \geq W & \Rightarrow C_s = N \land R_s = R_0 + 1 \quad (4.41) \\
C_f &= C_s + L + T \quad (4.42) \\
R_f &= R_s \quad (4.43) \\
P_s \geq P_0 & \quad (4.44) \\
P_s \leq P_0 + W & \quad (4.45) \\
P_f \geq P_0 + L + 1 & \quad (4.46) \\
P_f \leq P_0 + L + W & \quad (4.47)
\end{align*}

4.4.3 Redundant constraints

The above constraints will exactly determine the position of each word, but only after all of the document structure variables have been assigned. We would like to have as much knowledge as possible about the positions and lengths of document
elements while the labelling is in progress, to allow us to prune branches from the
search tree which are going to result in unacceptable realisations.

We need to determine upper and lower bounds on the amount of space which
a given unit could take up in the output, which we denote by \( \ell(X) \). Trivially, if
there are \( C(X) \) characters in the text for the elementary node \( X \), and it is enabled,
it must take at least that many characters in the output. Hence we have

\[
\ell(X) \geq C(X) \tag{4.49}
\]

We also want to determine the maximum possible length of the realisation.
Firstly, observe that two consecutive, wrapped lines of text of column width \( W \),
not including any indentation prior to the line, must contain at least \( W \) characters
between them. If they were to contain less, then the contents of the second line
would be included on the first rather than being wrapped. Note that we presume
here that the first line begins at the leftmost position, except for any indentation.

Now an elementary unit does not contain any new lines except those required
for wrapping. In general, it may start at any point along a line, then continue over
zero or more groups of two lines, and possibly have one extra line at the end.
Therefore, if the text is \( N \) lines in total, it must contain at least \( \left\lfloor \frac{N-1}{2} \right\rfloor \) groups of
two lines, each of which contains at least \( W \) characters.

Inverting this, we see that a block of text of \( W \left\lfloor \frac{N-1}{2} \right\rfloor \) characters can take up at
most \( N \) lines. Rewriting in terms of an arbitrary number of characters \( C \), it follows
that the text must be realised in at most \( 2 \left\lceil \frac{C}{W} \right\rceil + 3 \) lines. Observe that these lines
are complete lines in the output, including any indentation, unlike earlier where
we only included the part of the lines which could contain text. We denote the
total column width, including indentation, as \( W_0 \). Hence we have:

\[
\ell(X) \leq \left( 2 \left\lceil \frac{C}{W} \right\rceil + 3 \right) W_0 \tag{4.50}
\]
\[ -I(X) \land L(X) \leq 2 \lor RC(X) \Rightarrow \ell(X) \leq \left( 2 \left\lfloor \frac{C}{W} \right\rfloor + 2 \right) W_0 \quad (4.51) \]

We have an alternate upper bound on the length of an elementary unit. In any wrapped line, the amount of blank space at the end of the line is at most the length of the longest word in the text, plus one (to account for the space added afterwards when wrapping). We denote this quantity by \( L_m \); observe that it is trivial to calculate, and needs only to be done once per elementary unit.

Therefore, an elementary structure of \( C \) characters can take up at most \( Q \) lines, where

\[ Q = \left\lceil \frac{C}{W_0 - N - L_m} \right\rceil \quad (4.52) \]

Each of these lines will contain at most \( L_m + N \) unused characters. There will also be a blank line at the end of the text if the elementary unit is indented, or of level greater than 2, and is not the rightmost child of its parent. Hence we have the final length constraints:

\[ \ell(X) \leq C + Q(L_m + N) + W \quad (4.53) \]

\[ (-I(X) \lor L(X) \leq 2) \land RC(X) \Rightarrow \ell(X) \leq C + Q(L_m + N) + W \quad (4.54) \]

We expect that this second bound will be tighter in most situations. However, this will not always be the case, so we have implemented both bounds.
4.5 Summarisation

The document structure as defined by Power et al. (2003) does not leave us a great deal of room to affect the amount of space taken by its realisations. Ultimately, any document structure corresponding to a given rhetorical structure will present almost the same text, with relatively small differences in connectives and spacing.

In order to allow more control over the document size, we must allow elements of the document structure to be omitted from the output. This is accomplished by adding a fifth variable, Enabled, to every leaf node of the document structure which indicates whether or not the node will be displayed in the output. We denote this variable by $E(X)$ for some node $X$.

Prior to generating the document structure, we traverse the tree and record the importance of each leaf node, using Marcu’s summarisation scoring system (Marcu, 2000), which we described in Section 2.4.

When labelling, we have a choice as to whether each node is included in the output or not. We might wish to minimise the summarisation score while fitting the text within a given space, for example. Alternatively, if we want to force all nodes to be included, we can simply constrain the summarisation score $S = 0$ prior to generating the document structure.

Removing elements from the document structure creates some complications in the realisation process. We will describe these, and the modifications which we have made to resolve these issues.

Firstly, a connective will only make sense if it has two children (or at least two, in the case of a multinuclear relation) whose relationship it can describe. A connective with one or zero children should not be rendered. Therefore, we use the Enabled variable for non-leaf nodes also. However, unlike the other document structure variables, there are no choices to be made for this variable at non-leaf nodes; its value is completely determined by those of its descendant nodes.
For multi-level rhetorical structure trees, a relation may be required between its descendants, even if one or both of its immediate children are disabled; consider the example above. In order to correctly handle this situation, we introduce a new variable, \textit{DescendantEnabled} (abbreviated as \textit{DE}), which is true if a node, or any of its descendants, are enabled. It is defined in terms of the \textit{Enabled} variable.

For an elementary node \(X\), there are no descendants, so it is trivial to calculate:

\[
DE(X) = E(X)
\]  
\text{(4.55)}

Nucleus-satellite nodes require us to consider both of the children. Note that we do not need to consider whether the nucleus-satellite node itself is enabled, because this also depends on the children, as we show later. We retain our notation from the previous section: the parent, nucleus and satellite are denoted \(P\), \(N\) and \(S\), respectively.

\[
DE(P) = DE(N) \lor DE(S)
\]  
\text{(4.56)}

At multinuclear nodes, we again need to consider whether any of the children are enabled, but can ignore the node itself.

\[
DE(P) = \left( \sum_n DE(C_n) \geq 1 \right)
\]  
\text{(4.57)}

As the above three equations show, all \textit{DescendantEnabled} variables are set purely as a consequence of the \textit{Enabled} variables, and require no additional labelling. Using this variable, we can correctly determine the value of \textit{Enabled} at non-leaf nodes. These nodes are enabled when two or more of their children have some descendant enabled. For nucleus-satellite relations, this simply means that both children have a descendant enabled:
4.5. SUMMARISATION

\[ E(P) = DE(N) \land DE(S) \]  

(4.58)

For a multinuclear relation, there may be many children, so we need to count them to determine how many have descendants enabled.

\[ E(P) = \left( \sum_n DE(C_n) \geq 2 \right) \]  

(4.59)

4.5.1 Modifications to layout constraints

Obviously, disabling elements will alter the layout constraints. However, the modifications are relatively simple. If an element is disabled, its length is set to zero. Combined with the other constraints, this will automatically set the following:

\[ P_s = P_f = P_0 \]  

(4.60)

This in turn, will set the beginning of the following element to the end of the previous element.

All of the constraints discussed in section 4.4 still hold, with only minor modifications. The upper bounds on length remain the same, while the lower bounds and equalities must be modified to only hold where \( E(X) \) is true.

4.5.2 Modifications to structure realisation

Allowing some elements to be disabled complicates the process of structure realisation, which we described in Subsection 4.3.

Recall that we do not display a terminator after the rightmost child of any node, because an ancestor node will add a superceding terminator of its own. Previously, we could tell which child of a nucleus-satellite node was on the right by simply
looking at their relative positions. However, if we can disable one node, then the other will be the rightmost child by default.

\[
RC(N) \iff P(N) = 1 \vee \neg DE(S) \quad (4.61)
\]

\[
RC(S) \iff P(S) = 1 \vee \neg DE(N) \quad (4.62)
\]

(If both children are disabled, then this will identify both as the rightmost children. This does not matter, because neither child will be displayed, so the variables will not be used at all.)

We also need to deal with the effects of the connective on the starting levels of the children. If only one branch of a nucleus-satellite relation is enabled, then the connective will not be displayed. Therefore, we need to pass down the starting level from the parent. If neither branch is enabled, then the starting levels will not be used. These constraints replace those given in Equations 4.29-4.31:

\[ CL(P) \land E(P) \rightarrow SL(P) = SL_0(P) \land SL_0(NC) = L(C) \]  
\[ \neg CL(P) \land E(P) \rightarrow SL(P) = L(C) \land SL_0(NC) = SL_0(P) \]  
\[ E(P) \rightarrow SL_0(C) = 0 \]  
\[ DE(N) \land \neg DE(S) \rightarrow SL(NC) = SL(N) \]  
\[ DE(S) \land \neg DE(N) \rightarrow SL(NC) = SL(S) \]  

(4.63)  
(4.64)  
(4.65)  
(4.66)  
(4.67)

Multinuclear relations are somewhat simpler to deal with, as their order is predetermined. We define a new variable \( R(i) \), which denotes the number of enabled children to be realised from the \( i \)th child onwards (including that element). We use the same starting level variable \( SL \) to indicate whether an element should be
realised at its own level, a higher one (if it is the first element in a higher-level structure) or a lower one (if a connective precedes it).

\[ RC(C_i) \leftrightarrow (E(C_i) \land R(i) = 1) \]  
\[ E(C_i) \rightarrow SL_0(C_{i+1}) = L(C_i) \]  
\[ \neg E(C_i) \rightarrow SL_0(C_{i+1}) = SL_0(C_i) \]  
\[ (RC(C_i) \land E(P)) \rightarrow SL(C_i) = 0 \]  
\[ \neg(RC(C_i) \land E(P)) \rightarrow SL(C_i) = 0 \]

Elementary relations present no extra complications; they are either fully enabled or fully disabled. Therefore, we use the constraints from section 4.3 with no further modifications.

4.6 Improving performance

Expressing the entire model in a constraint-based form allows us to search for a solution with the fewest defects more quickly than a non constraint-based implementation. However, the search space is still very large, so we have implemented several other techniques to improve the searching.

Several of these techniques can be implemented simply by using the code given in Section 3.4, with no further effort required: first-fail labelling, optimistic partitioning and iteration. Other methods need to be tailored to suit this particular constraint satisfaction problem. We describe this process below.
4.6.1 Labelling order

Because labelling one type of variable will affect the domain of others, the order in which variables are labelled can make a significant difference. For example, if we set the connective to a coordinating type, then the levels of any child nodes must be zero, while the converse does not necessarily hold. Therefore, at least in this case, it will be more efficient to label the connective before the level. The relationships between the variables are relatively complicated, so it is unclear what the best labelling order is without conducting some experiments. We test two methods for ordering variables. One approach is to label the document structure by traversing the tree, depth-first. We make one pass for each variable type. For example, we might first label all the Indentation variables, then all the level variables, and so on.

4.6.2 Limited-discrepancy search

As described in Section 3.4, limited-discrepancy search is a method of reducing the search space when labelling. It requires a heuristic which guesses a value for each labelling decision. The search is then restricted in the number of assignments it can make which differ from the value suggested by the heuristic.

Every constraint problem using this method requires its own heuristic. We now describe ours. If the level variable is not already set, we choose the second largest possible value for it. We are attempting to avoid two nodes having the same level, thereby incurring a lost rhetorical grouping defect. The larger the value, the more values will be available for descendant nodes to use, but the largest possible value will generally be the same as its parent.
4.6. IMPROVING PERFORMANCE

We always choose to indent multinuclear relations, thereby allowing children to take any available level, and place the satellite first, avoiding a nucleus before satellite defect.

Subordinating connectives place the least restrictive constraints on the connected nodes, followed by parenthetical and then coordinating connectives. In particular, we do not wish to use a coordinating connective, as it forces the nucleus and satellite (and hence any child nodes they may have) to be of level zero, and will incur a lost rhetorical grouping defect on all further child nodes. Hence, we choose a subordinating connective if one is available; otherwise a parenthetical one, and finally a coordinating connective.

4.6.3 Caching search states

In order to improve the performance when summarising, we have implemented caching of search states, as described in Subsection 3.4.1. The two examples given by Smith (2005) were both permutation problems, where there are the same number of possible values as variables, and each variable must take a different value. Clearly, document structuring does not fall into this category, as neither of the above conditions hold.

However, while permutation problems are particularly suited to caching, it can be applied to other problems as well. For an optimisation problem like ours, there are two requirements for caching to work. Firstly, we need there to be large numbers of equivalent states, and secondly, we must be able to calculate our objective function as a sum of the current and future scores, which need to be independent.

Unfortunately, neither of these conditions hold exactly in our implementation of document structuring. However, we can modify the process somewhat, so that they are almost met. This means that we may sometimes prune search states
which would result in slightly better solutions than our current best. Nonetheless, we expect to find a solution which is good, if sub-optimal, and which will use considerably less assignments than would be required for a non-cached search.

In our document structures, we have three separate scores: summarisation score, length and defect count. The summarisation score meets the conditions for caching; it is a sum of the scores at each leaf node, each of which is independent of all the others.

We can separate length as the sum of the score up to a point, and the score beyond it, but the text spans realised before a point may not be continuous. If this is the case, we cannot exactly determine its length until the spans interspersing it are labelled, because of unpredictable line breaking behaviour and connective choices.

Because of this, we need to label the nodes in the order in which they will appear in the output. In order to achieve this, we insist on a fixed ordering: a nucleus must always be realised before the corresponding satellite. While this may seem like a considerable restriction, it has less effect than it might appear at first glance. It does not affect multinuclear relations at all. For nucleus-satellite relations, only one type of connective out of three allows the nucleus to appear after the satellite.

The requirements for caching are not met in the case of defect counts. Choices made early in the document structuring process can cause defects later on. For example, setting the Level variable to a small value at a high node on the document structure may mean that there are not enough distinct values for the lower nodes to use without incurring a lost rhetorical grouping defect.

Other defects may be incurred in a similar way. We may choose a connective at a high node in the rhetorical structure, and then be forced to repeat that connective lower down, because the alternative connectives cannot be used at the lower level.
This may arise if the alternative connective is parenthetical, and the lower level node is at phrase-level, or if it is coordinating, and the lower level is not at phrase-level. Likewise, choices of level and connective made at higher points on the rhetorical structure may cause single-sentence paragraphs and over-simple clauses later on.

In all of these cases, the defects are forced to occur by early labelling choices, but they will not be counted in the score until the labelling process reaches the lower nodes. This means that another search, which does not force these defects, will not appear to be any better, so the first, defect-inducing search may be cached and used instead.

These problems do not affect the other two defects, which have been made irrelevant by our choice to force satellites to occur after nuclei. Clearly, a nucleus before satellite defect is now impossible. Moreover, while left-branching clauses may still occur when the satellite branches more than the nucleus, we are unable to avoid them by reordering, so for the purpose of optimisation, they are now irrelevant.

We do not see any way of avoiding these problems without considerably changing the defect system. However, we believe that they are relatively minor — we expect that the conditions for them to occur will be fairly uncommon. There are many possible solutions to a document structuring problem, so there is virtually no chance that our caching will prevent a solution from being found. Rather, we expect to sometimes reject a search path which will lead to an optimal solution.

Nonetheless, given that the defects are only a very rough approximation to a true score of the readability of a document, a sub-optimal solution can still be good enough. At the very least, this method is superior to our earliest implementation of summarisation, which ignores the defect count entirely.
**4.7 Prolog implementation**

We have implemented the system described in this chapter in ECLiPSe Prolog. In this section we briefly explain how our system works. As explained in Section 4.1 and shown in Figure 4.7, it can be divided into three parts — the constraining, labelling and realisation stages.

**4.7.1 Constraining**

The input to our program is a rhetorical structure. We traverse this structure depth-first, defining a node on the document structure for each node in the rhetorical structure.

Each of these nodes has five main variables (position, indentation, level, connective and enabled). These are constrained in terms of each other and the levels of their parents, as described by Power et al. (2003) and Section 4.2. We also define and constrain several other variables such as the defects and positioning information. These are dependent on the five basic variables, so that we only need to assign values to the main variables and all other values will be set automatically. The constraints to which these variables are subject were described in Sections 4.2 and 4.3.
4.7.2 Labelling and Optimisation

After traversing the rhetorical structure and constraining the variables, the document structure is partially specified, matching all document structures which correspond to the input rhetorical structure. In non-trivial cases there will be more than one such document structure, so we need to choose which one to use.

This is done by labelling, the process of determining which values the constrained variables should take. In this process, we choose a variable and assign it a value from its domain. This can affect other variables which are constrained in terms of it, so their domains must be updated. This process continues and may affect the domain of any other variable. Once the propagation is complete, we choose another variable and continue. If there is any variable for which no value can be found, then there is no solution, and the system must backtrack to an earlier variable and choose a different value. Constraint propagation is already built into ECLiPSe Prolog, and we have not attempted to modify it.

When we are trying to optimise some quantity, such as the defect count, the process becomes slightly more complicated. When we find a solution, we must store it and attempt to find another, better solutions. This procedure is accomplished using the same backtracking method as when we cannot find a solution. There are many possible solutions to a given document structuring problem, and therefore the process can be quite slow. In order to reduce the number of possible solutions which must be considered, we can prune the structure based on the domain of the goal variable. Observe that the final document structure will have a larger or equal defect count, summarisation score and length than any partially labelled section of it. Thus, if at any point during the labelling process the lower bound on our final score is worse than the optimum found so far, we can discard this solution and attempt to find a different one.
This is done by adding redundant constraints to the program, as described in Subsections 3.3.1 and 4.4.3. These are constraints which do not affect the actual solution to the program, but reduce the domains of variables to eliminate values which will not be used in finding a solution, thereby reducing the search space. The redundant constraints are contained in the constraining stage, and are the same regardless of which labelling scheme is used.

The order in which variables are chosen on the document structure, and the order in which values are assigned to them also has a large effect on labelling performance, as we will see in the following chapter. Because of this, we have a number of different, interchangeable subsystems for labelling the document structure and choosing the optimal solution. Most of these procedures were described in more detail, along with pseudocode, in Section 3.4.

Our most basic labelling scheme involves labelling a document structure with five depth-first traversals of the structure, once for each of the main variables. We call ECLiPSe’s built-in minimisation predicate with our labelling predicate as an argument. (The minimisation procedure is explained in Subsection 3.3.2 and in more detail in Marriott and Stuckey (1998).) This predicate then calls the labelling predicate multiple times to find the optimal solution.

The next method is first-fail labelling, in which we collect all the basic five document structure variables from each level on the tree into a list, and choose the variable with the smallest domain. Again, we use this in conjunction with ECLiPSe’s minimisation procedure.

Another search strategy involves iteration through the score variable. We simply set it to zero and search for a solution. We increment the goal variable by one and repeat until a solution is found. This gives us the lowest-cost solution without using the minimisation procedure described earlier.
4.7. **PROLOG IMPLEMENTATION**

Optimistic partitioning is somewhat similar to iteration in that it finds the optimal solution by repeatedly searching for a solution with different constraints on the score variable. However, rather than iterating through all possible values of the score variable, it uses a divide-and-conquer approach.

Limited-discrepancy search proceeds identically to the basic labelling method, with two exceptions. For every labelling choice, we first choose the value predicted by the heuristic, which serves to guide the search towards values which are more likely to be optimal. Secondly, when we use a different value to that suggested by the heuristic, we count a discrepancy, and we terminate the search if there are more discrepancies than the predetermined maximum.

Our final labelling strategy involves caching. Again, it is similar to the basic tree-traversal method. At each point we compare the portion of the rhetorical structure which we have converted into a document structure thus far. If our score is higher (indicating a less desirable result) than a previously cached document structure, we abandon the search. If not, we cache this result and proceed. Because there are three different scores whose relative importance may differ, we only reject a structure if all three of its scores are equal to or higher than one in the cache. Thus the cache may contain multiple document structures corresponding to the same rhetorical structure.

Once the final document structure has been determined using any of these labelling schemes, all of the syntactic choices and positioning data are also set, so the realisation process is trivial — we simply display each word in its designated position.
In order to show how our system relates to other work in NLG, we compare our system to the reference architectures of Reiter and Dale (2000), and RAGS (Mellish et al., 2004).

Reiter and Dale (2000) give a list of seven tasks which constitute natural language generation. They also describe a pipeline architecture of three modules which perform these tasks, each of which is further divided into two or three subtasks, which we show in Figure 4.11. They also categorise their subtasks into those which deal with content, and those dealing with structure. In general, we implement far more of the latter than the former. We examined the consensus architecture in Subsection 2.1.1, and now point out key similarities and differences with our architecture.
4.8. **COMPARISON TO THE CONSENSUS ARCHITECTURE**

The input to the first stage in our system is a rhetorical structure, which is actually closer to the output of the consensus architecture than its input. In view of this, it should be unsurprising that much of the processing involved in this stage is omitted from our system; the rhetorical structures that we take as input already contain the required information.

By examining the differences between rhetorical structures and the document plans of the consensus architecture, we can see which functionality from the first stage that we do implement. A document plan is similar to a rhetorical structure, but differs in that it generally contains some realisation-related information, making it partway between a rhetorical structure and a document structure. Accordingly, our document structurer determines the positioning of the different document units, as well as the level at which they should be realised.

While the consensus architecture suggests that the document plan could specify the arrangement of messages at high levels, but leave lower level structuring decisions to the microplanning stage, we perform all these tasks together. Our document planning and microplanning system is divided into two parts, but along somewhat different lines than high and low levels, as we explain below.

In the consensus architecture, the document planning stage consists of two submodules: content determination and document structuring.

*Content determination* is the process of determining what information is to be realised in the output document. According to Reiter and Dale, this task can include selecting the most relevant information, summarising it and personalising it according to the user. We have not implemented all of these steps, however. We do not have any facility for personalising the data, or any reasoning system to determine the most salient facts to be generated. We use no specific domain knowledge; in fact, our system is completely domain-independent.
The only task which we implement which could be considered as being part of content determination is the optional summarisation. The rhetorical structure contains information about the relative importance of its component text spans, in the types of nodes (nuclei or satellites), and their depths. We use this data to construct a summary. While the information it contains is quite basic, a rhetorical structure is sufficient to construct a functional summarisation system for real documents (Marcu, 2000), and we believe that this would hold in our case as well.

The content determination process takes place later in our pipeline than in the consensus architecture. The decisions about whether to include parts of the rhetorical structure in the output are made during the labelling phase, which is our second module, roughly corresponding to microplanning. It may seem a little strange to decide what to display at the same time as we decide where to display it, or even afterwards. The reason for this is that we want to maintain as much control over the size of the output as possible. We want to account for small-scale effects such as line-breaking and indentation of items in lists, and so we need to make our summarisation decisions at the same time as we perform our document layout. As we discussed in Section 4.5, if we were willing to estimate the length occupied by a text span in the output, we could perform the summarisation earlier. This would improve our system’s speed and make it more consistent with other architectures, at the cost of reducing the accuracy of our length estimates, and hence potentially fitting less information into our space than we could otherwise.

The other task in the first module of the consensus architecture is that of document structuring. This involves several subtasks: grouping messages by theme, determining which discourse relations hold between which messages, ordering messages and determining which messages correspond to which type of document unit (such as sections and paragraphs). The latter two tasks may also be done later
in the pipeline, and the last may be split between two subsystems, with higher-
level structures (like the aforementioned sections and paragraphs) assigned earlier,
while lower level decisions about sentences and phrases left until later.

This task is the main focus of our system. Our document structuring process
performs most of the tasks described above, with two exceptions. We do not
need to group messages by theme, or determine which discourse relations hold
between them; the rhetorical structures which we use as input already contain this
information.

We base our document structuring process on that of Power et al. (2003). Like
the document structuring process in the consensus architecture, it involves the
choices of connectives, ordering and correspondence of messages to document
units. Our document structurer also adds two extra decisions to these — the
possibility of indentation, as per Power et al. (2003), and the choice to exclude
individual units.

However, our overall document structuring process performs considerably
more tasks than it would in the reference architecture, incorporating other func-
tionality which Reiter and Dale perform in separate modules. In addition to doc-
ument structuring (in the sense used by Reiter and Dale), our document structurer
performs the task of aggregation, and even calculates the row and column of each
word in the output while accounting for line breaks, which is arguably part of the
structure realisation task in the reference architecture.

Reiter and Dale’s architecture has three stages of structure-related tasks: doc-
ument structuring, aggregation and realisation. We have only two modules, docu-
ment structuring and realisation. However, our document structurer can be further
divided into two processes, constraining and labelling. The first of these stages
involves constraining the document structure variables in terms of the rhetorical
structure and each other, while the second is the labelling process, where we actually assign a value to each of the variables.

The first of these tasks roughly corresponds to document planning, and the second to microplanning. The consensus architecture leaves the boundary between these two tasks rather vague, but ours is very clear: we make no decisions in the first stage, but rather create the most general document structure possible, which matches every correct solution.

All the necessary relationships between different variables are specified (for example, that a child node should have the same or a lower level than its parent). However, those variables do not take specific values, but are only constrained to certain ranges, which will change if other variables on which they depend are further constrained. This process does not choose between the potential document structures, and leaves the document structure in a highly indeterminate state (unless there is only one possible document structure, which can only happen for a rhetorical structure with a single node).

While many of the details differ between our system and Reiter and Dale’s consensus architecture in this area, they match up quite well at a high level. The consensus architecture specifies that the document plan, as produced by the first stage, may correspond to many potential output documents, while the text specification produced by the second stage exactly describes only one document. This is exactly the same distinction that exists between constraining and labelling processes. The constraining step expresses the common details between all valid document structures, given the input rhetorical structure.

This very strict division is the norm in constraint programming over finite domains. It arises because labelling is a very expensive process, so it is only done once, after every variable has been constrained as much as possible, to minimise the number of assignments which have to be made.
We believe it is beneficial to separate the processes of deciding which solutions are acceptable from that of choosing the one which we wish to use. Aside from improving efficiency, separating the two processes allows for a modular, extensible system; incidentally, this is one of the aims of RAGS, to which we compare our system in the next section.

It allows us to separate our description of the solutions to the document structuring problem from the process of actually finding a solution. This makes it quite easy to change our criteria for finding the solution, or to use different algorithms for finding an optimal solution in an attempt to improve efficiency, without having to modify the actual document structuring process. We believe that similar considerations mean that this approach could be useful in other areas of NLG involving choices between a number of feasible options, particularly lexicalisation.

The next stage in Reiter and Dale’s system is microplanning. This stage produces a text specification, which is quite similar our definition of a document structure.

Lexicalisation is the process of choosing which words to use to represent the semantic content. For the most part, we presume that this task has already been performed. In an end-to-end NLG system, we would begin with semantic representations rather than text spans as our basic units, and these would have to be converted into text. We do touch on this task, however, when we choose which connectives to use to represent our rhetorical relations.

Referring expression generation means deciding how to refer to a specific entity. This including introducing a new entity, and referring to one which has already previously described, often using a pronoun. We have not implemented this capability at all, because it requires semantic knowledge of the text being generated (Reiter and Dale, 2000, p. 144). An end-to-end system would have this semantic information, but if we were to attempt to derive it from our text spans, we
would inevitably introduce inaccuracies from the original data, as well as adding unnecessary complexity to our architecture.

We describe how such capabilities could be added to our system, and what complications this might cause, in 6.2.

Aggregation involves taking phrase specifications and determining how they should be arranged into sentences, and where this is not specified by the document structurer, the order in which they should occur. In our system, however, the entire aggregation process is subsumed in the document structuring task. We determine which propositions are grouped together into sentences, and which are realised on their own when we assign levels to the various nodes on the document structure tree. We also determine the order in which elements should be rendered, down to the character level, all during document structuring. Aggregation corresponds mostly to the labelling part of the document structuring process, which selects and fully specifies a document structure from the set of all valid structures, based on various criteria such as defect counts and length.

The final module in the consensus architecture is surface realisation, which involves creating an actual document from the text specification. Linguistic realisation refers to the conversion of the NLG system’s internal representation of the document into a textual form. Reiter and Dale presume an intermediate level of representation for basic facts, between the semantic content and the output text.

In an end-to-end NLG system, the main part of linguistic realisation would be realising this lexicalised content into spans of text. Because we do not deal with semantic representations, we do not perform this task, either. However, Reiter and Dale (2000) also consider the generation of correct capitalisation and termination of sentences to be a part of linguistic realisation, which is the only part that we deal with.
4.9. **COMPARISON TO RAGS**

The other realisation task is that of *structure realisation*, which is the process of generating markers specific to the format in which the output will be rendered, such as HTML tags. We output our structures in plain text, in order to maintain control over the size of the output, so we have no need for structure realisation.

We believe that our system fits fairly well into Reiter and Dale’s architecture. There are many aspects of the consensus architecture which are not represented in our system, as we are mostly concerned with structure-related tasks, and a few inconsistencies with between the division between their document structuring and aggregation modules and our constraining and labelling stages inside the document structurer. However, overall there is quite a close correspondence between the two approaches.

### 4.9 Comparison to RAGS

In contrast to Reiter and Dale’s reference architecture, RAGS is defined primarily in terms of its data types (Mellish et al., 2004). There are six of these types: conceptual, rhetorical, document, semantic, syntactic and quote representations. Of these, we only utilise three in any detail: rhetorical, document and quote. The bulk of our system, the document structurer, can be thought of as a mechanism for converting rhetorical into document representations. We now briefly describe each data type and its correspondence to our system.

*Conceptual representations* are the inputs to the NLG process. In our system, we use rhetorical structures as our input, so clearly these are equivalent to rhetorical representations. In an end-to-end NLG system, however, we would expect our input to be in a less-structured form, which would require some processing to convert into a form like a rhetorical structure.
As the name suggests, rhetorical representations are equivalent to our rhetorical structures. In RAGS, rhetorical representations are defined as trees with rhetorical relations as internal nodes and some other elements at the leaves, just as in our model.

Power et al. (2003) define their rhetorical structures as having semantic, rather than textual, representations at their leaf nodes. However, they do not actually specify any method of representing the semantic data, nor for converting it into text. In fact, their implementation uses textual representations for leaf nodes. We simply use blocks of text, which may be considered as quote representations, as our leaf nodes.

Document representations are trees which describe the layout of the document. They are explicitly defined to be the same as document structures (Power et al., 2003). Consequently, they are almost identical to the document structures which we use.

We do not deal with semantic or syntactic representations; we are not dealing with the problem of representing and manipulating semantic information in this work. However, we describe how our system might be extended to use such representations in Section 6.2.

Quote representations are simply blocks of text. As stated earlier, we use these as the leaves of rhetorical structures; the output is also in this form.

Mellish et al. (2004) also list a collection of functional modules which transform data between the various representations. From this perspective, most of our system can be viewed as an implementation of the document structuring and aggregation modules.

Overall, our system represents only a subset of the data types and modules described in RAGS, but there is a fairly close match in those portions which we do use.
4.10 Extensions

We believe that multinuclear relations should be rendered inline where there are only two children, and in a list when there are many children. There could perhaps be some overlap where either option is appropriate. An additional defect could be added where either of these conditions is violated.

The nucleus before satellite defect is only discussed in the context of sentences, which we believe overlooks an important exception to it. In our opinion, this defect is not valid for higher level structures; in fact, the opposite may be the case at these levels.

At a chapter or section level, we believe that it is better practice to state the most important information first. For example, consider an evidence relation, which contains an assertion for a nucleus, and evidence for that assertion as the satellite. If a document had this relation as the highest level, we might expect that the claim would be presented first, followed by the evidence (Mann, 1999). If this were not the case, then the reader might well question the purpose of the reading several chapters of evidence for some as-yet unknown claim. Similarly, a concession relation contains some general rule as the nucleus, and an exception to this rule as the satellite. If a chapter were to be devoted to each of the nucleus and satellite, we would expect the nucleus to come first.

In view of these reservations, we suggest that this defect should be only applied in this form for sentence and lower level structures. It should be ignored at paragraphs, and reversed for section and higher level structures, so that placing the satellite before the nucleus constitutes a defect at these levels.
4.11 Summary

We have described our document structuring system in this chapter. We began by describing our reimplementation of the document structuring process as given by Power et al. (2003). Next, we explained in detail the process by which a document structure can be converted into an actual document.

We then described several enhancements to the basic document structuring process. Firstly, we introduced constraints which determine the row and position of every word in the output. In addition to this, we implemented summarisation, so that our system can remove elements from the output. Together, these two abilities give us very fine-grained control over the size of the output. Because document structuring is a computationally intensive process, we described several techniques from constraint programming which we use to improve the efficiency of our system.

We also compared our system to two reference specifications for NLG systems: the reference architecture of Reiter and Dale (2000) and RAGS (Mellish et al., 2004), and suggested several extensions which we could make to the system.

Having described our system in detail, we now turn our attention to its empirical performance.
Chapter 5

Experiments

The interactions between the variables in a document structure, combined with differences in tree structures, node types (nucleus-satellite or multinuclear) and connective types (coordinating, subordinating and parenthetical), make it infeasible to determine the complexity of the document structurer analytically. Therefore, we need to use empirical data in order to evaluate the performance of our system, which is the subject of this chapter.

In this chapter we give the results of various experiments we performed in order to test the efficiency of our system. We begin by describing the corpus used to conduct these experiments, and showing some sample outputs. Next, in Section 5.3, we give the results of our basic document structuring, first using an elementary minimisation procedure, and then more sophisticated constraint programming techniques. Finally, we examine the effects of extending the document structurer to control the amount of space occupied by the output.
CHAPTER 5. EXPERIMENTS

5.1 Corpus

Our testing data come from Marcu's rhetorical structure corpus (Marcu, 2000). We have chosen a particular article from his corpus, and rendered the individual rhetorical structures which make up this article.

Following Power et al. (2003), we presume that all rhetorical relations should be marked with a discourse connective. This is clearly not the case in normal written text, but is a reasonable stance to take for NLG, because it is not yet well understood when discourse connectives should be included, and when they can be left implicit Power et al. (2003), Scott and de Souza (1990). However, this corpus is generated from human-written text from the Wall Street Journal, so this assumption does not hold for it. Discourse markers are often left implicit, and where they are present, they are left in the text spans contained in the leaf nodes of the rhetorical structure, rather than being part of the relation they describe.

A list of cue phrases which suggest particular rhetorical relations is provided by Carlson and Marcu (2001). Most of these correspond to discourse connectives which might be used in document structuring. However, this list is incomplete; not all rhetorical relations have corresponding cue phrases. Moreover, they do not describe these phrases in the format which we require, with syntactic types and locus (see Subsection 2.3.2).

Without such data we do not believe that it is worthwhile testing on a large number of rhetorical structures. This is because the number and type of discourse connectives which correspond to a given rhetorical relation will have a considerable effect on performance of a document structurer.

Clearly, the number of discourse connectives directly affects the number of document structures which can be realised from a given rhetorical structure: at each internal node on the rhetorical structure tree, we can choose from any connective, subject to various other constraints, so any increase in the number of
possible connectives will cause an exponential increase in the number of candidate
document structures.

Moreover, coordinating discourse connectives restrict the level of child nodes
to be level zero, while the other syntactic types allow different levels, but force
the satellite to occur before the nucleus, as described in Subsection 2.3.2. This
means that some connective types will allow more possible document structures
than others. We expect that the largest number of structures will be generated
using parenthetical connectives, followed by subordinating and coordinating con-
nectives, respectively, but lack the data to make an extensive comparison.

In the absence of a corpus of discourse connectives, we have generated random
placeholder connectives for the rhetorical relations used in Marcu’s corpus. There
are about three connectives for every relation, with random syntactic types.

Because these connectives are only placeholders, we have no sensible text to
include with them. In addition, the leaf nodes of the rhetorical structure sometimes
include connectives which are realised along with those we generate, and the
rhetorical structures are not always aligned to clause or sentence boundaries.
Sometimes the ordering of rhetorical elements is reversed, causing problems such
as pronouns referring to objects which have not been introduced yet — a failure
in referring expression generation, which we described in Subsection 2.1.1.

As a result of these issues, the output produced using this technique is quite
ugly and difficult to read. This is why we have used other sources for our
examples in previous chapters. However, these structures are adequate for testing
our system’s capabilities.
5.2 Sample output

With the summarisation capability disabled, our system produces the same output as the system of Power et al. (2003), with only minor differences in formatting.

For example, given a simple rhetorical structure shown in Figure 5.1 with a column width of 40, we obtain the nine texts in Figure 5.2.

Enabling summarisation exponentially increases the number of possible solutions. Even with this near-trivial example there are 28 possible solutions with summarization. If we restrict the output to a single line and minimise the sum of the defect count and summarisation score, the optimal solution is given in Figure 5.4.

For testing the efficiency of our system, we use the corpus of Marcu (2000). Unfortunately, this corpus is not built with our requirements in mind. The leaves of their rhetorical structures contain connectives and punctuation, while we represent this information at higher levels. Moreover, we do not have a corpus of connectives corresponding to the rhetorical relations used here, so we have simply used placeholders with random connective types and loci.

This should not pose a problem, as we are only using this corpus for testing efficiency, as opposed to the quality of the output. A sample document rendered in this fashion is shown in Figure 5.4.
5.3. Basic document structuring

5.3.1 Labelling order

As described in Section 4.6.1, one of our methods of labelling the document structure variables is to iterate through the document structure tree once for each variable type, labelling every variable of that type as we go. There are four variable types, which results in 24 possible orderings.
We expect that the order in which the variables are assigned will have a considerable effect on the performance of the document structurer. For example, assigning a Connective variable to a discourse connective of coordinating type forces the level of its children to be zero, while assigning the children to zero merely requires the connective to be of subordinating or coordinating type, potentially leaving several options.

It is unclear which ordering will produce the best results, so we test all 24 possibilities. The numbers given in Table 5.1 represent the number of variable assignments required to find the optimal solution, divided by the number of variable assignments required by the most efficient variable ordering for the same structure. For each of the possible orderings, we show the minimum, maximum
5.3. **BASIC DOCUMENT STRUCTURING**

<table>
<thead>
<tr>
<th>Ordering</th>
<th>Max</th>
<th>Min</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indentation,Level,Connective,Position</td>
<td>57.1</td>
<td>1.00</td>
<td>7.52</td>
</tr>
<tr>
<td>Indentation,Level,Position,Connective</td>
<td>1.78</td>
<td>1.00</td>
<td>1.28</td>
</tr>
<tr>
<td>Indentation,Connective,Level,Position</td>
<td>46.9</td>
<td>1.00</td>
<td>6.24</td>
</tr>
<tr>
<td>Indentation,Connective,Position,Level</td>
<td>19.0</td>
<td>1.00</td>
<td>3.15</td>
</tr>
<tr>
<td>Indentation,Position,Level,Connective</td>
<td>2.79</td>
<td>1.00</td>
<td>1.38</td>
</tr>
<tr>
<td>Indentation,Position,Connective,Level</td>
<td>2.65</td>
<td>1.00</td>
<td>1.47</td>
</tr>
<tr>
<td>Level,Indentation,Connective,Position</td>
<td>57.2</td>
<td>1.00</td>
<td>7.52</td>
</tr>
<tr>
<td>Level,Indentation,Position,Connective</td>
<td>1.75</td>
<td>1.00</td>
<td>1.29</td>
</tr>
<tr>
<td>Level,Connective,Indentation,Position</td>
<td>116</td>
<td>1.00</td>
<td>14.04</td>
</tr>
<tr>
<td>Level,Connective,Position,Indentation</td>
<td>57.1</td>
<td>1.00</td>
<td>7.50</td>
</tr>
<tr>
<td>Level,Position,Indentation,Connective</td>
<td>1.75</td>
<td>1.00</td>
<td>1.27</td>
</tr>
<tr>
<td>Level,Position,Connective,Indentation</td>
<td>1.75</td>
<td>1.00</td>
<td>1.27</td>
</tr>
<tr>
<td>Connective,Indentation,Level,Position</td>
<td>65.0</td>
<td>1.00</td>
<td>8.38</td>
</tr>
<tr>
<td>Connective,Indentation,Position,Level</td>
<td>37.1</td>
<td>1.00</td>
<td>5.38</td>
</tr>
<tr>
<td>Connective,Level,Indentation,Position</td>
<td>74.1</td>
<td>1.00</td>
<td>9.20</td>
</tr>
<tr>
<td>Connective,Level,Position,Indentation</td>
<td>46.9</td>
<td>1.00</td>
<td>6.18</td>
</tr>
<tr>
<td>Connective,Position,Indentation,Level</td>
<td>19.0</td>
<td>1.00</td>
<td>3.24</td>
</tr>
<tr>
<td>Connective,Position,Level,Indentation</td>
<td>19.0</td>
<td>1.00</td>
<td>3.08</td>
</tr>
<tr>
<td>Position,Indentation,Level,Connective</td>
<td>4.56</td>
<td>1.00</td>
<td>1.65</td>
</tr>
<tr>
<td>Position,Indentation,Connective,Level</td>
<td>3.82</td>
<td>1.00</td>
<td>1.73</td>
</tr>
<tr>
<td>Position,Level,Indentation,Connective</td>
<td>2.40</td>
<td>1.00</td>
<td>1.31</td>
</tr>
<tr>
<td>Position,Level,Connective,Indentation</td>
<td>2.40</td>
<td>1.00</td>
<td>1.31</td>
</tr>
<tr>
<td>Position,Connective,Level,Indentation</td>
<td>3.08</td>
<td>1.00</td>
<td>1.47</td>
</tr>
<tr>
<td>Position,Connective,Level,Indentation</td>
<td>3.08</td>
<td>1.00</td>
<td>1.39</td>
</tr>
</tbody>
</table>

Table 5.1: Relative number of assignments required to find the optimal solution with different labelling orders.

and average number of assignments required over the eleven testing rhetorical structures.

As the table shows, the order in which variables are labelled has a large impact on the efficiency of the search, by a factor of about 4 on our testing data. The fastest method seems to be to label the *Level* and *Position* variables first, and then either of the other two. For the next table, we used the “Level,Position,Indentation,Connective” labelling strategy.
### Table 5.2: Comparison of number of valuations required by different searching strategies, using a tree-traversal system for choosing labelling variable.

<table>
<thead>
<tr>
<th>Size</th>
<th>Min Defects</th>
<th>First</th>
<th>Basic</th>
<th>Iterating</th>
<th>OP</th>
<th>LD(3)</th>
<th>LD(10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
<td>15 (2)</td>
<td>29</td>
<td>15</td>
<td>30</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>28</td>
<td>2</td>
<td>699 (20)</td>
<td>100000 (3)</td>
<td>—</td>
<td>100000 (4)</td>
<td>21008 (9)</td>
<td>100000</td>
</tr>
<tr>
<td>45</td>
<td>?</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>20666 (17)</td>
<td>100000 (11)</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>220 (12)</td>
<td>1167</td>
<td>392</td>
<td>1062</td>
<td>1677 (5)</td>
<td>1115</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>50 (6)</td>
<td>197</td>
<td>50</td>
<td>150</td>
<td>209</td>
<td>232</td>
</tr>
<tr>
<td>13</td>
<td>6</td>
<td>194 (9)</td>
<td>2626</td>
<td>2617</td>
<td>2953</td>
<td>455</td>
<td>1504</td>
</tr>
<tr>
<td>31</td>
<td>3</td>
<td>2310 (26)</td>
<td>100000 (8)</td>
<td>—</td>
<td>100000 (12)</td>
<td>100000 (6)</td>
<td>100000 (6)</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>15 (1)</td>
<td>20</td>
<td>15</td>
<td>30</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>136 (9)</td>
<td>520</td>
<td>65</td>
<td>436</td>
<td>350</td>
<td>345</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>40 (5)</td>
<td>143</td>
<td>40</td>
<td>120</td>
<td>91</td>
<td>91</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>25 (4)</td>
<td>53</td>
<td>49</td>
<td>74</td>
<td>28</td>
<td>28</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>35 (6)</td>
<td>85</td>
<td>117</td>
<td>151</td>
<td>46</td>
<td>46</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>45 (7)</td>
<td>189</td>
<td>45</td>
<td>135</td>
<td>93</td>
<td>93</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
<td>596 (16)</td>
<td>42410</td>
<td>40363</td>
<td>41634</td>
<td>9155</td>
<td>39894</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>15 (2)</td>
<td>29</td>
<td>15</td>
<td>30</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>214 (13)</td>
<td>1137</td>
<td>75</td>
<td>697</td>
<td>92</td>
<td>92</td>
</tr>
</tbody>
</table>

The minimum column shows 1.00 for each measurement. This means that for each ordering we have at least one structure for which it is optimal.

### 5.3.2 Search strategies

We have tested all of the search strategies discussed in the previous section, using both the tree-traversal and first-fail methods of choosing which variable to label. We forced all searches to terminate after 100,000 assignments and return the best solution found at that point, in order to prevent inordinately long execution times. Results are shown in Tables 5.2 and 5.3. We show the number of nodes in the rhetorical structure, **Size**, as well as the minimal number of defects possible (if known), **Min Defects**. The remaining columns give the number of assignments required to find the best solution, except the column **First** which gives the number of assignments to find the first solution with simple search. **Basic** is the simple minimisation search, **Iterating** iterates the defect count upwards from 0, **OP** uses...
optimistic partitioning, and $\text{LD}(n)$ is limited discrepancy search with a max discrepancy of $n$. If the best solution found is not optimal it is shown in parentheses after the number of assignments figure.

Because the total number of solutions to be checked is exponential in the size of the structure, the problem can easily become intractable for large structures. Therefore, we limit the number of assignments which can be made to 100,000, and simply fail to find any solutions after this point, and return the best solution which has been found thus far. If no solution has been found before this, we label the test with a dash.

Note that we have not normalised these results, as it is sometimes unclear which is the best solution; some strategies may perform faster than others but return a non-optimal solution, and some are terminated early for the sake of tractability.

For the search strategy, limited-discrepancy search and iterating through the defect variable (using tree-traversal labelling) seem to require the least number of assignments in order to find a solution, although the best search strategy varies considerably, depending on the structure. However, if we reach the maximum number of assignments without finding an optimal solution, iteration cannot provide a sub-optimal solution, because by definition, the first solution it finds is the optimal one. For this reason, limited-discrepancy may be a better choice when realising large structures.

Limited-discrepancy search often does not find a solution when used with first-fail labelling. We believe that this is because first-fail labelling will choose labelling variables from all over the document structure. Therefore, once a discrepancy has been incurred, the next variable to be labelled may be from an entirely different part of the document structure. This may happen several times,
<table>
<thead>
<tr>
<th>Size</th>
<th>Min Defects</th>
<th>First</th>
<th>Basic</th>
<th>Iterating</th>
<th>OP</th>
<th>LD(3)</th>
<th>LD(10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
<td>12 (2)</td>
<td>14</td>
<td>12</td>
<td>24</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>28</td>
<td>2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>45</td>
<td>7</td>
<td>189 (21)</td>
<td>100000 (9)</td>
<td>—</td>
<td>100000 (9)</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>68 (7)</td>
<td>196</td>
<td>74</td>
<td>220</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>40 (4)</td>
<td>70</td>
<td>40</td>
<td>86</td>
<td>—</td>
<td>48</td>
</tr>
<tr>
<td>13</td>
<td>6</td>
<td>5079 (7)</td>
<td>9170</td>
<td>5923</td>
<td>10895</td>
<td>—</td>
<td>1009 (7)</td>
</tr>
<tr>
<td>31</td>
<td>3</td>
<td>1490 (14)</td>
<td>80592</td>
<td>40318</td>
<td>72914</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>12 (1)</td>
<td>14</td>
<td>12</td>
<td>24</td>
<td>14</td>
<td>16</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>52 (7)</td>
<td>110</td>
<td>52</td>
<td>216</td>
<td>—</td>
<td>105</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>32 (5)</td>
<td>65</td>
<td>32</td>
<td>96</td>
<td>—</td>
<td>35</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>20</td>
<td>84</td>
<td>120</td>
<td>120</td>
<td>—</td>
<td>87</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>49</td>
<td>297</td>
<td>481</td>
<td>393</td>
<td>—</td>
<td>287</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>38 (7)</td>
<td>160</td>
<td>67</td>
<td>168</td>
<td>—</td>
<td>55</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
<td>30357 (13)</td>
<td>46594</td>
<td>250</td>
<td>38881</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>12 (2)</td>
<td>22</td>
<td>12</td>
<td>24</td>
<td>18 (1)</td>
<td>18</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>56087 (10)</td>
<td>91411</td>
<td>60</td>
<td>77427</td>
<td>—</td>
<td>232</td>
</tr>
</tbody>
</table>

Table 5.3: Comparison of number of valuations required by different searching strategies, using a first-fail method for choosing the labelling variable.

causing several discrepancies to occur, before the choices which have been made can affect new ones.

By contrast, when using tree-traversal labelling, once a discrepancy has been incurred, the next variables to be labelled will come from the same node, or its children, which will be constrained by the choice which has just been made.

Tree-traversal labelling seems to outperform first-fail labelling in most other cases too, but much less dramatically.

Basic search performs more poorly than the other methods in most cases, as we might expect. Optimistic partitioning represents an improvement on this, but not as much as limited-discrepancy or iterative searching.

Finding the first solution is generally quite fast, but the first solution is almost never optimal. It is also interesting to note that even merely finding the first solution is often slower than finding the optimal limited-discrepancy solution, indicating that our heuristic is improving performance considerably.
Overall, the iterating approach followed by a limited discrepancy search when this fails to find the solution appears to be the most robust combination. By comparison, the prototype system demonstrating the work in Power et al. (2003) generates all solutions and then chooses the best one. This approach will not scale to large rhetorical structures, which is why we have not attempted to compare the systems directly.

5.4 Summarisation

We now turn our attention to the summarisation and layout capabilities of our system. Enabling summarisation adds extra variables to the document structure, making the labelling process more complicated. Therefore, we need to test the performance of our system with these extra features. The results are shown in Table 5.4.

We use the same corpus as in the previous section. For each rhetorical structure, we first generate a document structure with summarisation disabled, in order to find its length. We then generate a document structure from the same rhetorical structure, forcing it to have a length at most one-third of the original structure. This can be accomplished by simply adding the appropriate constraint prior to the document structuring process. We report three numbers for each test, which are labelled A, D and S. These are the number of assignments required, the defect score, and the summarisation score, respectively.

This procedure is used because simply minimising the length or summarisation variables will result in every node being either disabled or enabled, respectively. Moreover, this process is a reasonable approximation to a likely real-world application of a natural language generation system which can control the size of its output: conveying as much information as possible within a given space.
We test three different approaches scoring the document structure while including the capacity for summarisation, as it is unclear how to score a document structure containing both defects and missing branches. We also allow caching of search states using the final scoring method.

The first is to simply minimise the summarisation score, which is labelled $S$ in the results. We also label the other document structure variables during this process, although they have little effect on the summarisation score. They do have some impact, however, because we restrict the amount of space which can be occupied in the output.

Therefore, any choices which we can make which will reduce the amount of space taken up in the output can allow us to include more nodes in the given space, and thus reduce the summarisation score. In particular, displaying multinuclear items inline, rather than indented with each on a separate line, will reduce the space taken up.

Because this method ignores the defect count and simply attempts to pack as much information as possible into the given space, it can lead to highly undesirable document structures, as discussed in Subsection 2.3.4, although it will be economical in terms of space.

The second method is to minimise the defect count using similar techniques to the previous section, while ignoring the summarisation variable. After this process is finished, we separately minimise the summarisation score. This ignores the effect of the document structure variables on the layout. The layout is entirely indeterminate when they are set, because we do not yet know whether any given node will be included or not in the output. Nonetheless, it is likely to produce more attractive output than the first method. This method is denoted $D$ then $S$ in the results.
The final scoring method is to label all variables together, minimising the sum of the defect count and summarisation score. The relative weighting of the defect count and summarisation score is rather arbitrary, as is the choice to add them in the first place. It is clear, however, that the quality of the document should be some increasing function of both variables. We discuss how we might obtain a better measure of the quality of a document in Section 6.2. We perform this method twice, once using a straightforward minimisation procedure, and once using caching. These are labelled $D+S$ and $D+S$ cached in the table of results.

Note that, as in the previous section, we terminate searches after 100,000 assignments for the sake of tractability, returning the best solution, if any, which has been found up to that point. When we minimise the defect count first, then the summarisation score, each search is allowed 100,000 assignments, so the total is 200,000 in one case.

The best method overall seems to be minimising the defects and summarisation scores separately. This is often faster than minimising the summarisation score on its own, and usually produces the same summarisation score with considerably lower defect scores. On a couple of occasions, minimising the defect count and then the summarisation score results in a lower sum of the two than directly minimising that sum, because the latter process stops after 100,000 assignments with a sub-optimal solution.

Minimising the sum of the two scores tends to require many more assignments than minimising each separately. Caching can improve this somewhat, with one notable exception, but it tends to still be slower than separate minimisation processes, and it has other drawbacks. It tends to result in higher defect or summarisation scores, and can sometimes fail to produce a solution at all.

This is because of inaccuracies in our caching process. We presume that two different states in the labelling process are equivalent if they have labelled the
Table 5.4: Performance of different summarisation methods. Here A represents the number of assignments required, D is the defect count and S is the summarisation score.
same part of the structure. If one has incurred more defects than the previous
best structure up to that point, then the new attempt is discarded. This can lead to
problems if the previous best attempt does not lead to a solution. For example, if a
level has been set to zero at that point, but lower on the tree there is a parenthetical
connective, which requires its children to take a level greater than zero, then there
will be no solution. Normally the labelling process would backtrack to before the
decision to set a level of zero, but the caching process may prune this branch of
the tree.
Chapter 6

Conclusion

In this chapter, we review the contribution which we have made in this thesis, and identify several possibilities for further work in this area.

6.1 Contribution

Natural language generation (NLG) is a significant domain of inquiry in language technology. It also represents a relatively untouched area of application for constraint programming.

We have focussed on document structure (Power et al., 2003), which provides both a theory describing the relationship between the layout and meaning of a document, and a means of NLG. We have reimplemented the document structurer, while improving upon it in several ways. Firstly, our system evaluates defects as part of the document structuring process, allowing it to minimise the defect count while generating the structure. This allows us to prune branches of the search space which lead to high defect counts without evaluating them all, as opposed to the original method of generating all possible document structures and then choosing the best.
We have also implemented several more advanced constraint programming techniques, further improving the performance of the system. Finally, we have taken advantage of the greater performance to add summarisation to the document structuring process, allowing us to reduce the amount of text contained in the output in order to fit into a given space. In order to accomplish this, we have created a much more detailed constraint model of the document, which contains the row and column index of every word.

With the exception of the previous work by Power et al. (2003), we believe that this work is the first application of constraint programming to NLG. Moreover, this work represents the first attempt to examine the performance of an NLG system from the perspective of constraint programming. We have used several constraint programming techniques to attempt to improve the performance of our system, and tested them in order to quantify this improvement.

Our testing indicates that limited-discrepancy search using tree-traversal labelling is generally the fastest of the search strategies, but it may return sub-optimal solutions at times. Other techniques represent various trade-offs between completeness, likelihood of finding a solution, and the score of the solution. Any of these represent a significant improvement in the performance of document structuring for large inputs.

The results for summarisation were less promising. The best method seems to be to minimise the defect count and then summarise the resulting document; performing both tasks simultaneously seems to add too much complexity to the model to be useful on larger documents.

Overall, we believe that our system is much more practical than the one described by Power et al. (2003). Our system can render structures of around size 30-40, whereas the largest example provided with the previous system contained 11 nodes, which sometimes caused it to fail due to a lack of memory.
6.2. **FURTHER WORK**

This means that our system can generate documents of approximately one page in length. While this is a definite improvement, and suitable for many NLG tasks, we would still like to be able to generate longer documents than this. This would require a modification of the document structuring process. We might break a rhetorical structure up into parts which must each be realised at a fixed level, such as a section or paragraph. Then a document structure can be generated from each of these parts independently, keeping the overall complexity at a manageable level.

We now consider other directions we might take in order to make further progress in this area.

### 6.2 Further work

We have tested our system on a relatively small corpus, which may not be very representative of the documents which we would like to generate using our system. We would prefer to perform larger-scale empirical testing, but this would require a larger corpus containing rhetorical structures, relations and discourse connectives. The thesis of Oates (2001) may be suitable for this task. Alternatively, we could construct one using available tools.

We could analyse a large corpus of rhetorical structures, probably that of Carlson et al. (2002). For any given rhetorical relation, we can look for the most commonly occurring words at the beginning of discourse units involved in the relation; these are likely to be connectives associated with that relation. It should be possible to determine the locus and syntactic type of the relation by observing the circumstances in which it occurs.

As noted previously, our system is only concerned with structuring side of NLG. In order to create an end-to-end NLG system, we would need to deal
with many content-related issues, which we described in Section 2.1. The input
to such a system would not be a rhetorical structure, but some more abstract
representation, which would allow us to use a wider range of summarisation
techniques, which may perform better than our current method.

Alternatively, we could implement our systems as a RAGS module, which
would allow it to be used as a subsystem in an NLG pipeline alongside other
mechanisms for dealing with different parts of the NLG process.

Another important improvement we could make would be to account for
variable-width characters. This would require us to modify our layout constraints
somewhat; currently, the length of a word is simply the number of characters
in it, and terminators are presumed to be of length 1. These would have to be
changed to the sum of the width of each character in the word, and the length of the
terminating character, respectively. This modification would add a small amount
of uncertainty about the length of a document structure prior to labelling. Different
terminating characters might have different widths, and changes in capitalisation
definitely would. However, these differences would still be within relatively tight
bounds.

This would not affect the performance of our system in any significant way,
as we know what each character will be in advance. Most characters come from
the text spans at the leaves of the rhetorical structure, which are fixed at the start
of the document structuring process. The remainder comes from the connectives
and punctuation, which are decided when the document structure is labelled.

We could also extend the formatting model to render of text together with
diagrams, or deal with pagination. However, we would need to simplify our model
somewhat in order for the problem to remain tractable, or at least deal with each
of these issues separately.
Another area with room for further work is the scoring mechanism. As noted by Power et al. (2003), the defects are rather arbitrary. There are several modifications that we might make to their set of defects, which we discussed in Section 4.10.

As an alternative to this method of imagining new defects, which remains quite arbitrary, we could also attempt some form of usability testing. Given a collection of generated documents, we could survey potential readers as to the quality of each document. We could then use data mining on the document structures and the readers’ scores to attempt to determine which features have most impact on the readability of the document, and generate a new scoring mechanism based on those features.

In conclusion, we have applied constraint programming to the problem of document structuring. This problem represents part of the larger challenge of natural language generation, and involves many interacting choices, making constraint programming a natural choice for solving it. However, this work represents the first application of advanced constraint programming techniques in order to solve this problem efficiently. We have tested the performance of our system experimentally, because the interaction of the various constraints is too complex to analyse directly. This has shown a distinct improvement over the previous method of document structuring, and has opened up a variety of new topics for investigation.
Bibliography


Author/s:
MARSHALL, ROBERT

Title:
Improving the efficiency and capabilities of document structuring

Date:
2007

Citation:

Publication Status:
Unpublished

Persistent Link:
http://hdl.handle.net/11343/39295

File Description:
Improving the efficiency and capabilities of document structuring

Terms and Conditions:
Terms and Conditions: Copyright in works deposited in Minerva Access is retained by the copyright owner. The work may not be altered without permission from the copyright owner. Readers may only download, print and save electronic copies of whole works for their own personal non-commercial use. Any use that exceeds these limits requires permission from the copyright owner. Attribution is essential when quoting or paraphrasing from these works.