FROM ENGLISH TO FORMAL SPECIFICATIONS

A thesis submitted to the University of Salford
for the degree of Doctor of Philosophy

June 1994

By

Farid MEZIANE

Department of Mathematics and Computer Science

Contents

List of Figures

Acknowledgements

I wish to express my deep appreciation and thanks to my supervisor Dr. Sunil Vadera for his advice and support during the course of this research and for his patience and understanding during the writing up process.

I am also grateful to British Aerospace Ltd for providing me with the case study.

DECLARATION

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Abstract

Specifications provide the foundation upon which a system can be formally developed. If a specification is wrong, then no matter what method of design is used, or what quality assurance procedures are in place, they will not result in a system that meets the requirements.

The specification of a system involves people of different profiles who favour different representations. At the beginning natural language is used because the specification document acts as a contract between the user and the developers. Most of the time, the only representation that users understand and agree on is natural language. At the other end, developers find natural language specifications ambiguous and incomplete and may therefore prefer formal specifications. The transition from informal specifications to formal ones is an error prone and time consuming process. This transition must be supported to ensure that the formal specifications are consistent with the informal ones.

In this research we propose an interactive approach for producing formal specifications from English specifications. The approach uses research

in the area of natural language understanding to analyse English specifications in order to detect ambiguities. The method used for analysing natural language text is based on McCord's approach. This method consists of translating natural language sentences into a logical form language representation. This helps to identify ambiguities present in natural language specifications and to identify the entities and relationships. These entities and relationships are used as a basis for producing VDM data types.

We also investigate the production of data type invariants for restricted sentences and the production of some common specifications.

We test our approach by implementing it in Prolog-2 and apply it to an independent case study.

1

Introduction and Motivation

1.1 Introduction

Computer applications are used in all aspects of life. They vary from a simple payroll program to a more complex rocket design system. Consequently, an enormous number of applications are being developed to satisfy the users' demands. During the last decade, the production of software and its maintenance has dominated the overall cost of computer systems. Boehm [12] states that in 1985 worldwide software costs were in excess of $140 billions and it has been predicated by Sommerville [64] that this annual cost will exceed $435 billions by 1995.

To cope with the growing number and complexity of the software projects being tackled, many methodologies have been developed, (e.g. [29, 32, 39, 49]). Each model presents a way of improving software development and

design. Some of these models are:

1. *Rapid prototyping* [29]. This approach is used for a better understanding of the user requirements. A working solution is quickly developed and implemented prior to the requirements phase. This system is then presented to the users for experimentation. The remarks and suggestions of the users are used to obtain a better understanding of their requirements. Once the requirements are known and well understood, a new version of the system is implemented.

2. *Exploratory programming* [32]. This approach is similar to rapid prototyping and aims to tackle the volatile nature of requirements. It is developed to evolve as the user's needs change over time. A first version of the system is developed to satisfy the known requirements. The initial version is modified on a continuing basis as the user requirements evolve. The software is delivered when it performs in a satisfactory and adequate way.

3. *Automated software synthesis* [9, 49]. This approach suggests the development of a formal specification and its automatic transformation into a program.

4. *System assembly from reusable software* [39]. The aim of this technique is to reduce the cost of software development. It suggests the reuse of components of software that already exist. Jones [39] has reported that of all the code written in 1983, less then 15% is new

and specific to particular applications. The remaining 85% is common to most applications. The aim of this model is to standardise this 85% of code and make it ready for use for software developers and programmers.

5. *The life cycle development methodology.* The life cycle development methodology, also called the waterfall model, was one of the first models suggested [60]. It is a process describing the different stages of software development from the original idea to the installation of the working system. Many versions of this model have been published and many different names were given to each phase. Nevertheless, most of the approaches agree on the following six phases:

   (a) Requirements analysis.

   (b) System specification.

   (c) Software design.

   (d) Implementation.

   (e) System testing.

   (f) Maintenance.

   The requirements analysis phase is mainly concerned with the definition of the requirements for an acceptable solution to the problem. All aspects relating to the organisation and system such as personal needs, computer availability, cost and security must be considered in

this phase. The next phase is the specification of the system. Specifications define what tasks the system is expected to perform. No details of how these tasks will be performed are given at this stage. In the design phase, each component of the system is described in detail. The subsystems are defined and documented. The interfaces between different subsystems and the data types used by each subsystem are designed. All algorithms and procedures defined in the design process should be translated into a programming language in the implementation phase. With the use of high level programming languages and the modern programming methodologies this phase is the easiest to develop and has been mastered better than the other phases [19]. However, all the code produced must be tested to determine the behaviour of the software in its operating environment. To ensure positive results, the test plan and data should be very detailed and every known case should be tested. At the end of this stage, the software is ready for use. The maintenance stage concerns the amendments and changes of the software after delivery. It will concern program modifications to satisfy new customer requirements, diagnosing and correcting errors, verifying program performance and updating the documentation.

A comparative study of these methodologies can be found in [22]. In the present work we concentrate on the life cycle development methodology. The life cycle development methodology presents many advantages

[22] but its main advantage is that it shows clearly the different stages of software development so that we can analyse each phase separately showing where designers' efforts should be placed and where the resources have been consumed.

Many studies have shown that the first stages of the software life cycle have been devoted only a small proportion of the resources allocated to the complete life cycle [17, 19]. These studies have also shown that a considerable number of errors occur in these early stages and most importantly, these errors are the most expensive to correct if detected later in the life cycle. It is therefore important to give more attention to the early phases to reduce these errors and to assure that an appropriate system is developed.

Hence, this study focuses on the requirements and specification stage. In section 1.2 of this chapter, we give a detailed definition of the requirements and specification stage. In section 1.3, we will list the problems encountered in the requirements phase. In section 1.4 we give the characteristics of a good requirements document. This motivates the objectives of our research which are given in section 1.5.

1.2 A Definition of the Requirements and Specification Phases

Most of the differences between the many versions of the life cycle development methodology occur in the earlier phases. Some authors define one phase which they refer to as requirements analysis and definition. This phase is then subdivided into subphases. For example, Sommerville [64] has subdivided this phase into three subphases where the output of each phase is a document:

1. Requirements definition is the first document produced. It states what the system is expected to provide. This document represents the contract between the client and the analyst and must be understood by both the client and the contractor. It is therefore written in natural language.

2. Requirements specification is the second document produced. This document is written in a more formal notation and sets in detail the services of the system. The requirements specification represents the contract between the analyst and the developer.

3. Software specification (also called design specification) is an abstract description of the software design. This document serves as a basis for the design and implementation of the software and is intended for the software designer.

Comer [19] gives a different approach. He divides the early phases into two subphases: requirements analysis and specification. These are similar to the requirements and specification phases described earlier in section 1.1.

Presland [57] defines the earlier stages as Comer except that the tasks to be performed by the new computer system are included in the requirements analysis. A document called requirement analysis, written in natural language is produced, and forms the beginning of the specification phase. Many other authors have followed this approach, with some differences in the requirements phase. Christel [16] divides the requirements into two classes, functional requirements and nonfunctional requirements. Functional requirements concern all the tasks that the new system is going to perform. They represent the basis for the development of the specifications. Nonfunctional requirements concern all the constraints of the new system. These include performance constraints, resources constraints, design constraints and hardware requirements.

Some other approaches define an earlier stage prior to the requirements stage called a feasibility study. This phase investigates if the user's needs are satisfiable by using existing software and hardware and if the solution is financially realisable.

If we analyse these different approaches, we find that the tasks performed are all the same and the order in which they are performed is identical; it is only the names and the allocation of each task to a phase which differs. At the beginning of the software development process, some tasks have to be performed to state clearly the objectives and foundations for the

development of a system. The role of these tasks is to investigate why the system is to be developed, what is expected from it and under which constraints it will be developed. These three tasks form the requirements phase. Once acceptable answers to these questions are found, the "what is expected from the system" part of the requirements will be elaborated in a more detailed manner to be used for the future development.

We distinguish two main tasks in these early phases. At some stage a document written in natural language is produced. This document contains all the tasks the new system is expected to perform. It should be understood by both the customer and the analyst and agreed with the customer. This first document is usually used as a contract between the customer and the supplier. The document is then handed to the specifier who will produce the specification document. In this work we will refer to the first document as the requirements specification, and to the second as system specification. In some cases, a system specification is only a formal version of the requirements specification. These two stages of the specification are necessary because the customer and the supplier have different views on what constitutes an ideal specification. Natural language specifications are attractive to the customer because they are the natural way of communication and in most cases it is the only representation the customer can understand and agree on. For the developers, natural language specifications are a major source of errors and incompleteness, and an ideal specification for them would be a formal one.

Since informal specifications are desirable by users, many authors have stated that informality will always exist in software development [7, 57]. Further, Balzer [7] has presented two advantages for informal specifications:

1. Informal specifications are more concise then formal specifications and focus both the specifier's and the user's attention.

2. Informal specifications are useful in maintaining a system.

Balzer defended the first advantage by stating that informal specifications are concise because only part of them is explicit, the rest is implicit and must be extracted from the context. So if more attention is focussed on the implicit information this will increase the readability and understandability of the specifications. As discussed by Presland [57], the second advantage is only valid if there is a computer system which can transform informal specifications to formal ones. The issue now, as presented by Balzer [7], is whether the informal form should exist only outside a computer system and the translation from informal specifications to formal ones done manually or should the informal specifications be part of the computer system and their translation done automatically?

Before answering this question, an analysis of the problems which may occur in the requirements and specification phases is necessary to exhibit which problems a computer system can solve.

1.3 Problems of Requirements

Christel [16] groups the problems of requirements into three categories:

- Problems of scope.
- Problems of understanding.
- Problems of volatility.

We describe each of these in the following sections.

1.3.1 Problems of Scope

The requirements process must begin with a full analysis of the organisation and the context in which the system is to be developed. This analysis should determine the boundaries of the new system to adhere completely with the users' or organisation's goals. Requirements which do not address these problems will run the risk of producing incomplete and probably unusable requirements. Requirements can also be overstated and concentrate on design activities. This will result in the production of requirements that are ambiguous and therefore may not be verifiable by the user. The financial situation of an organisation is also very important, since one cannot develop a system that an organisation cannot afford.

1.3.2 Problems of Understanding

The earlier stages of software development involve people of different backgrounds. This may include customers, sponsors, users, requirements analysts and specifiers. This makes communication difficult and causes misunderstandings. In fact, Christel [16] has reported the results of a study stating that

> "56% of errors in installed systems were due to poor communication between user and analyst in defining requirements and that these types of errors were the most expensive to correct using up to 82% of available staff time".

Problems of understanding can be separated into three issues:

- The different background of the communities.

  Information which is common and obvious for a group can be completely foreign and not understandable to another.

- The language used to express the requirements.

  The language used by a group can be too formal or too informal for the other groups.

- Structure of information.

  The large amount of information gathered during requirements analysis needs to be structured. The different communities may not understand the structure equally well.

1.3.3 Problems of Volatility

User needs evolve over time. It is not unusual that during the time it takes to develop a system, the user requirements have changed. The causes of these changes may vary from the increasing understanding of the user about the capabilities of a computer system to some unforeseen organisational or environmental pressures. If the changes are not accommodated, the original requirements set will become incomplete and inconsistent with the new situation or in the worst case useless.

After analysing the problems that may occur during the requirements analysis phase, the next section will look at the characteristics of well formed requirements.

1.4 An Adequate Approach to Requirements and Specifications

According to the IEEE guidelines on producing requirements [3], a good software requirements document should have the following characteristics:

1. *Unambiguous.* A requirement is unambiguous if it has only one interpretation.

2. *Complete.* A requirement is complete if it has the following qualities:

- It includes all requirements whether relating to functionality, performance, design constraints, attributes or external interfaces.
- Definition of the responses of the software to all realisable classes of input data in all realisable classes of situations.
- Full labelling of tables, figures and diagrams.

3. *Verifiable.* A requirements document is verifiable if there is a way to check that it satisfies the users' needs.

4. *Consistent.* A requirements document is consistent if no set of individual requirements described in it conflict.

5. *Modifiable.* A requirements document is modifiable if its structure and style are such that any necessary changes can be made easily, completely and consistently.

6. *Traceable.* A requirements document is traceable if any modifications can be traced to their origin.

7. *Usable during the operation and maintenance.* The requirements must address the needs of the operations and maintenance phase, including the eventual replacement of the software.

When gathering information about the user requirements, the analyst must be able to fully understand the user's needs. A better understanding of these needs will help to identify requirements that are incorrect, missing or incomplete and ambiguous. The final requirements document presented

to the specifier should be complete, consistent and error free. The specifier can then produce adequate specifications to enable the developers to design the correct system. During the last few years a lot of effort has been put in to the development of the specification stage of software design. The development of specification languages and formal methods is probably the most important step made to improve the first stages of the software development process. The benefits of formal methods for the development of systems are widely recognised [17, 55, 64]. Some of these advantages are [64, page 125]:

1. Formal methods can be used as a way of understanding the user requirements and the software design.

2. When using a formal method it is possible to prove that a program satisfies its specification.

3. Formal specifications may be automatically processed.

4. It is possible to use a formal specification as an aid to identifying possible test cases.

A formal development life cycle begins with a formal specification. Design steps can then be proved with respect to their specifications. This verification of design steps against their specification provides the primary benefit of formal methods – namely that design errors are detected earlier in the life cycle and are not propagated further down the life cycle.

However, formal methods have two major weaknesses. First, formal specifications may not be consistent with the requirements written in English. Second, formal methods are based on mathematical logic making them hard for some analysts to comprehend. There is therefore a need to reduce these weaknesses.

The basic problem is then the transformation of a requirement specification, expressed in natural language into a system specification written in a formal notation. Ultimately, this will make the use of formal methods much easier to the analyst but still complete and consistent for the designer. When developing the SAFE project [7], Balzer investigated the desirability and feasibility of such a system. At first site, such a system may present three disadvantages which are:

1. The informal specifications may be misunderstood by the computer.

2. The computer-based tool will decrease the reliability of the transformation to the formal specifications.

3. The required volume of interactions will abrogate the advantages of informality.

The first disadvantage was probably a big issue when Balzer was considering such a system because of the lack of powerful natural language understanding systems at that time (1978). Although no perfect system is yet developed for natural language understanding, many respectable systems have been developed and tested [47, 51, 69]. Balzer acknowledges that

a computer cannot match human performance in understanding informal specifications, but it operates much more methodically. If a computer detects several possible interpretations for a statement, it asks the user to choose the intended meaning. It can record and make explicit all assumptions when transforming the informal specifications. With a powerful natural language understanding system, this first disadvantage can be ignored. For the second disadvantage, the issue is again understanding rather then reliability. If the informal specifications exist outside the computer system, then we have to rely on a human to accurately transform the informal specifications and this transformation depends upon properly understanding the informal specifications. Balzer has stated that once the understanding is achieved, the restatement of the informal specification involves moving information from one place to another and changing its form. Experience has shown that these tasks are error prone and are better done by a computer tool. Therefore, reliability would be improved rather then reduced. Balzer did not see the third disadvantage as a major one for the SAFE project since the interaction involved was not high.

1.5 Thesis Aims and Contents

The general aim of this research is to investigate the link between the English specifications and the formal ones. We can summarise our approach as in

figure 1.1. The English specifications are taken as input. We use the natural language analysis part to process the text and to detect any ambiguities. The aim of this process is to associate a unique interpretation for each sentence. The meanings of the sentences are represented as logical forms (this representation will be discussed in the next chapter). Such logical forms then acts as a basis for producing data types. Further, based on the data types there are some common operations that are encountered in the specification of systems which can be produced automatically by the system. The formal specifications are produced in the Vienna Development Method (VDM) [37]. VDM is a formal language based on predicate logic. VDM is used to produce specifications of systems and provides proof obligations which enable a designer to establish the correctness of design steps.

As figure 1.1 shows, the approach we develop is an interactive requirements process. That is, we do not take the view that the initial English text is complete.

To summarise, the aims of our approach are:

- To identify ambiguities and incompleteness in natural language requirements documents.
- To aid the production of data types from natural language specification documents.
- To accept invariants as natural language sentences and then transform them into VDM data type invariants.

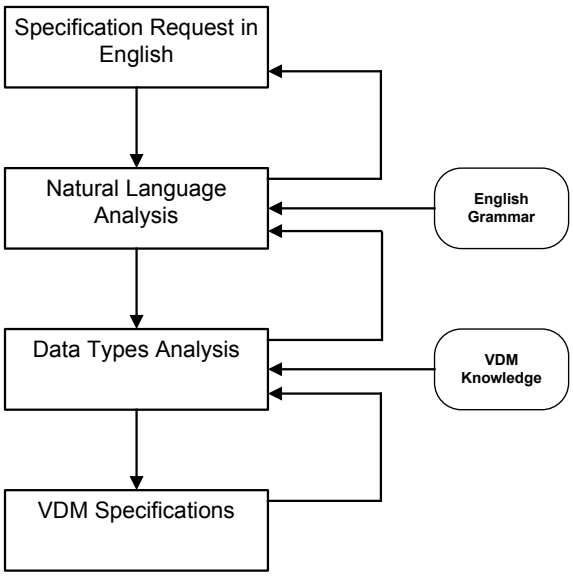

Figure 1.1: Overview of the approach

- To produce some common specifications in the VDM language.

The raison d'être of this thesis is to show how these aims are achieved. The following outlines the chapters of the thesis.

Chapter 2: The Logical Form Language and Logic Grammars

Our aims require the analysis of natural language specifications. Much work has already been done to analyse English. For example, Warren and Pereira's [69] system for interpreting natural language queries for a geographical data base and McCord's [47] work for a student data base both tackle the problems of quantification and ambiguities in English texts. For our work, we adopt McCord's approach to natural language processing [43, 44, 47]. In chapter 2 the syntax and semantic analysis of the natural

language text are described. The modular logic grammar formalism used to analyse texts is also presented.

Chapter 3: Using LFL to Analyse Natural Language Documents

The main disadvantage with natural language specifications is ambiguity. The same statement may be interpreted differently by different users. In the first part of chapter 3 a list of ambiguities one may encounter when analysing natural language texts is identified. The second part develops an approach to detect these ambiguities and presents the different possibilities that users have to correct them.

Chapter 4: Identifying the Data Types

Once the natural language text is analysed and the ambiguities resolved, the next step is to identify the data types. Chapter 4 develops an approach which first attempts to identify the entity relationship model. The entity relationship model is then used as a basis for the definition of the VDM data type.

Chapter 5: Invariants and the Specification of Operations

Data types usually have associated invariants. Invariants are truth-valued functions used to record restrictions on data types. In chapter 5 we concentrate on how to produce such invariants from natural language sentences. In addition we also show how some common specifications of operations are

automatically generated.

Chapter 6: A Case Study: A Flight Planning Data Base

In chapter 6, we demonstrate the approach developed on a realistic case study. We show how the approach works on a specification problem where the English specification was written without prior knowledge of this work. The different steps of the analysis are shown and the problems encountered are presented.

Chapter 7: Related Research

Many systems have been developed with the aim of improving the requirements phase. In chapter 7, we outline some alternative systems and compare our approach with them.

Chapter 8: Conclusion and Future Work

The last chapter contains the conclusions and suggestions for future work.

2

The Logical Form Language and Logic Grammars

2.1 Introduction

The aim of a natural language understanding system is to provide an unambiguous interpretation of natural language texts. To understand a language, the knowledge of words alone is not enough. One has to understand the meaning of the words, how they are pronounced and how the words combine to form sentences. In a way, we have to understand the structure of the language. Allen [2] has identified six types of knowledge that are essential to understand a language:

- Phonetic and phonological knowledge concern the sounds of words.
- Morphological knowledge concerns the construction of words.

- Syntactic knowledge concerns how words are combined to form sentences.
- Semantic knowledge concerns the meaning of sentences.
- Pragmatic knowledge concerns the study of the context which gives the meaning of sentences.
- World knowledge concerns the general knowledge a user needs to communicate and use a language.

An ideal natural language understanding system would be one that has all these types of knowledge. Most natural language processing systems use morphological knowledge, syntactic knowledge and semantic knowledge. The phonetic knowledge is only used by automatic speech-understanding systems. Pragmatic knowledge is used if a system is developed for a restricted domain. To date, it has been difficult to utilise world knowledge. It appears that only humans have a capability for remembering and applying general knowledge to understand events.

A main features of natural language understanding systems is the use of dictionaries to check the spellings and to determine the different classes of words. This task involves the storage of a large number of words. Morphological knowledge is used by some systems [57] to reduce the size of the dictionary. Generally, a word consists of a *root* to which a *suffix* or *prefix* is added. For example, the word friendly is composed of the root friend and the suffix -ly. Many other words can be derived from the root by adding

other prefixes or suffixes. When using morphological knowledge, only the roots are stored and the formation rules of words such as plural nouns, finite verbs, adjectives and adverbs are then defined. The syntactic and semantic components are the most important parts of a natural language understanding system. These two components are examined in more detail in the following sections.

2.2 Syntactic Analysis

In each language there are some rules that determine which strings of words produce structurally correct sentences. This set of rules is called the *grammar* of the language. The process of checking that a sentence abides by these rules is known as *syntax analysis* or *parsing*. Parsing is the process of assigning a grammatical structure to a sentence. For example, the sentence:

A company maintains a simple system.

is composed of the *noun phrase* a company and the *verb phrase* maintains a simple system. Further, the *noun phrase* is composed of the *determiner* the and the *noun* company. The *verb phrase* is composed of a *verb* followed by a *noun phrase*. *Phrase structure trees* [28] also called *syntax trees* [47] are used to represent this analysis. The syntax tree representing the above analysis is shown in figure 2.1.

Such grammar rules can be expressed formally. For example, we can use

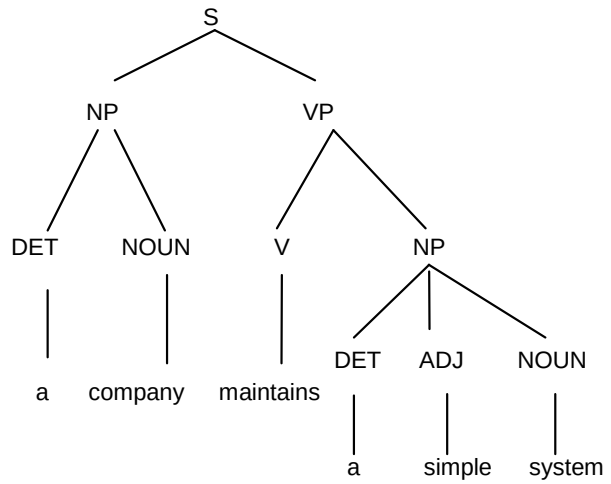

Figure 2.1: Syntax Tree

the Backus-Naur notation to define a grammar for the above example:

$$
\begin{aligned}
< sentence > &::= < noun\_phrase >< verb\_phrase > \\
< noun\_phrase > &::= < determiner >< noun > \\
< noun\_phrase > &::= < determiner >< adjective >< noun > \\
< verb\_phrase > &::= < verb >< noun\_phrase > \\
< determiner > &::= a \\
< noun > &::= company \mid system \\
< adjective > &::= simple \\
< verb > &::= maintains
\end{aligned}
$$

Another notation used to express natural language grammars is Chomsky's notation [15]. In the following parts of this chapter, this notation is adopted. The above grammar is equivalent to the following one expressed

in Chomsky's notation:

$$
\begin{aligned}
S &\rightarrow NP\text{-}VP \\
NP &\rightarrow DET\text{-}N \\
NP &\rightarrow DET\text{-}ADJ\text{-}N \\
VP &\rightarrow V\text{-}NP \\
DET &\rightarrow a \\
ADJ &\rightarrow simple \\
N &\rightarrow company \\
N &\rightarrow system \\
V &\rightarrow maintains
\end{aligned}
$$

The symbols in capitals denote *nonterminals*, also called *syntax categories*. We will use NP to denote a noun phrase, VP to denote a verb phrase, DET to denote a determiner, ADJ to denote an adjective and N to denote a noun. In this example, the left hand side of each rule consists of a single nonterminal. Such grammars are called *context free grammars* (CFGs). In CFGs a rule such as $NP \rightarrow DET\text{-}N$ defines that a noun phrase is composed of a determiner followed by a noun.

It is not possible to give a complete grammar for the English language. However, many authors have considered some grammars that are acceptable [24, 27, 54]. In the next section we will analyse the main syntax categories of the English grammar used in our approach and a simplified structure will be suggested for each category.

2.2.1 Syntax of Noun Phrases

The simplest form of a noun phrase consists of a single pronoun as in the sentence:

He passed the exam.

or a single proper noun as in:

John passed the exam.

In all other cases, noun phrases consist of a head noun, premodifiers and/or postmodifiers. The most common premodifiers of nouns are determiners. Determiners are a broad but closed class of words. They can be articles, quantifiers, possessives, demonstratives, ordinals and cardinals. An example illustrating this case is:

The student passed the exam.

Other modifiers of nouns are adjectives and other nouns. Adjectives are words that give attributes to nouns as in the sentence:

A *unique* identifier is assigned to each item.

As an example where a noun modifies another noun we have:

A company maintains a *stock* system.

In general, we can define the following grammar for a noun phrase:

$$NP \rightarrow PR$$

$$NP \rightarrow PN$$

$$NP \rightarrow [PreMods]\text{-}N\text{-}[PostMods]$$

Where *PR* stands for pronoun, *PN* stands for proper noun, *PreMods* stands for premodifiers and *PostMods* stands for postmodifiers. The square brackets denote optional items.

2.2.2 Syntax of Verb Phrases

A verb phrase is a group of words where the head word is a verb. Verbs are divided into two categories: auxiliary verbs and lexical verbs. Auxiliary verbs include *be, have, do* and the modal verbs such as *can* and *may*. Examples of sentences containing auxiliary verbs are:

The program *is* ready for use.

A program *can* be divided into several modules.

Lexical verbs are divided into two groups, transitive verbs and intransitive verbs. Transitive verbs need an object or a complement as in the following examples:

The student runs *the program.*

He was looking *for the book.*

Transitive verbs also allow the use of passive voice as in the example:

The program is run by the student.

Intransitive verbs do not need a complement as in:

The program crashed.

The above examples suggest that the structure of a verb phrase might be:

$$VP \rightarrow V$$

$$VP \rightarrow V\text{-}NP$$

$$VP \rightarrow V\text{-}PP$$

However a verb phrase may also take some other forms as in the examples:

(a) The student ran the program on the new computer.

(b) The teacher gave the student a high mark.

(c) The flight is planned from Blackpool to Doncaster.

In (a) the verb phrase has the structure:

$$VP \rightarrow V\text{-}NP\text{-}PP$$

In (b) the structure:

$$VP \rightarrow V\text{-}NP\text{-}NP$$

and in (c) the structure:

$$VP \rightarrow V\text{-}PP\text{-}PP$$

Therefore, the generalised structure of a verb phrase is:

$$VP \rightarrow V$$

$$VP \rightarrow V\text{-}NP$$

$$VP \rightarrow V\text{-}PP$$

$$VP \rightarrow V\text{-}NP\text{-}PP$$

$$VP \rightarrow V\text{-}NP\text{-}NP$$

$$VP \rightarrow V\text{-}PP\text{-}PP$$

2.2.3 Prepositional Phrases

Prepositional phrases are introduced by prepositions and consist of a preposition followed by a noun phrase. This is the definition used by many authors attempting to define or to deal with prepositions [47]. Unfortunately the reality is far away from this assumption. As shown by Jackendoff [34] the prepositional phrase does not take only the form:

$$PP \rightarrow P\text{-}NP$$

but also the following more complex forms:

1. $PP \rightarrow P$
2. $PP \rightarrow P\text{-}PP$
3. $PP \rightarrow P\text{-}NP\text{-}PP$

We discuss these forms in the following subsections.

Analysis of Intransitive Prepositions

In this category, prepositional phrases consist of a single preposition. These prepositions are called *intransitive prepositions*. That is, the prepositional

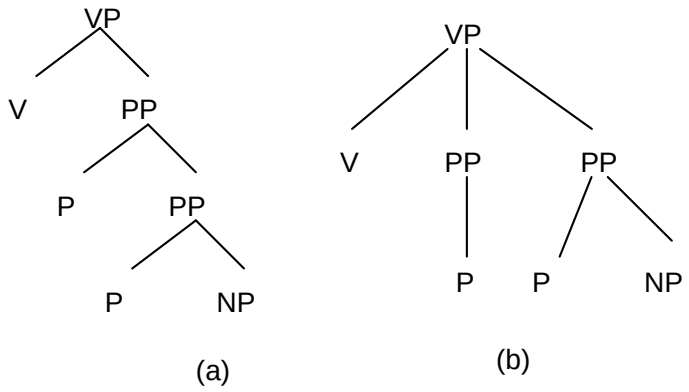

Figure 2.2: Parsing the P-P-NP string

phrase does not need a noun phrase as a complement. This group of prepositions contains adverbs such as *downstairs*, *afterwards* and *before* which, according to Jackendoff, are better identified as intransitive prepositions and particles of verb-particle combination such as *look up* and *give out*. Examples:

John put the clothes *on*.

John went *downstairs*.

Analysis of a String of the Form *P-P-NP*

In a string of the form *P-P-NP*, prepositional phrases are composed by a combination of two prepositions followed by a noun phrase. Given the existence of intransitive prepositions, a string of the form P-P-NP can be analysed in the two ways given in Figure 2.2.

In (a) there is a preposition whose complement is a prepositional phrase

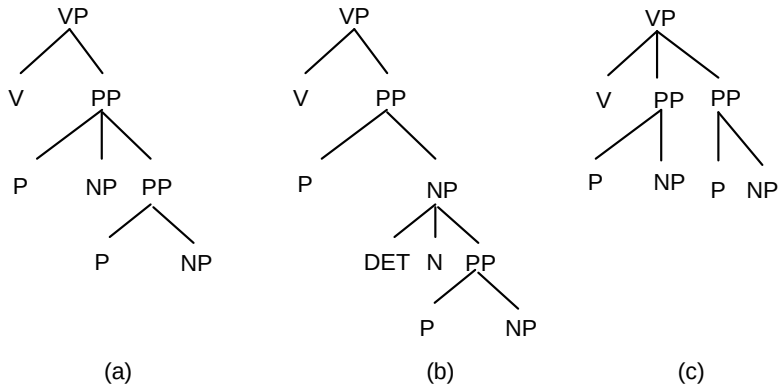

Figure 2.3: Parsing the P-NP-P-NP string

and in (b) there is an intransitive preposition followed by a normal prepositional phrase. These two cases are illustrated by the following examples:

(a) The challenger ran *out of* time.

(b) The man raced *away in* a red car.

Analysis of a String of the Form *P-NP-P-NP*

Given the existence of the above categories of prepositional phrases, three different structures can be obtained from a string of the form *P-NP-P-NP*. Figure 2.3 gives the three possible syntax trees.

In (a) all the verb phrase except the verb can behave as a single constituent. Example:

The flight is planned from Blackpool to Doncaster.

Structure (b) appears in sentences such as:

John went to the house in the woods.

Where the noun phrase of the first prepositional phrase (*the house in the woods* in this case) forms a constituent that can be used independently.
In structure (c), we have a prepositional phrase which modifies another prepositional phrase. An example illustrating this case is:

John went to the park with Mary.

These three interpretations are produced for each sentence containing the string *P*-*NP*-*P*-*NP*. The analyst is the one who decide which interpretation is suitable for a sentence.

From the different cases illustrated above, the general structure of a prepositional phrase is:

$$
\begin{aligned}
PP &\rightarrow P \\
PP &\rightarrow P\text{-}NP \\
PP &\rightarrow P\text{-}PP \\
PP &\rightarrow P\text{-}NP\text{-}PP
\end{aligned}
$$

2.2.4 Adjective and Adverb Phrases

Adjective phrases have adjectives as their first word. Adjectives can be premodified or postmodified. An adjective can be premodified only by adverbs

as in:

This problem is *extremely* difficult to resolve.

where the adjective *difficult* is modified by the adverb *extremely*. Three postmodifications may occur in an adjectival phrase: a prepositional phrase, an infinitive clause and a *that*-clause. Examples are respectively:

He is very happy *about his results.*

The student was happy to *find a job.*

The teacher was happy *that all students passed the exam.*

Adverbial phrases have an adverb as their first word. An adverb is the minimal form of an adverbial phrase, but it may be premodified. An adverb can be premodified only by an adverb as in:

He realised *very* quickly that he was going in the wrong direction.

where the adverb *quickly* is premodified by the adverb *very*.

2.2.5 Embedded Structures

Many sentences do not have the simple structure $S \rightarrow NP\text{-}VP$. More complex sentences can be built by allowing some sentences to include other sentences or parts of sentences. For example, the following sentence:

The book that John bought yesterday is with Mary.

can be seen as composed of the two clauses: "The book is with Mary" and "John bought a book yesterday" The second clause is used to modify the

noun phrase "The book". This type of clauses are called relative clauses. Relative clauses involve a sentence form used as a modifier in a noun phrase. Relative clauses are introduced by relative pronouns.

Conjunctions are another type of words that can relate clauses and sentences. For example, the sentence:

Each item of stock is assigned a unique identifier when it is introduced.

can be analysed as composed of the two clauses: "Each item of stock is assigned a unique identifier" and "The item is introduced". The two clauses are related by the conjunction when.

To cover the embedded structures, the following syntax rules can be included in the grammar:

$$
\begin{aligned}
NP &\rightarrow N\text{-}RC \\
RC &\rightarrow RP\text{-}S \\
S &\rightarrow CL\text{-}CL \\
CL &\rightarrow S \\
CL &\rightarrow CONJ\text{-}S
\end{aligned}
$$

where RC stands for relative clause, RP stands for relative pronoun, S stands for sentence, CL stands for clause and CONJ stands for conjunction.

Syntax analysis is the first step in a natural language understanding system. It can detect sentences that are structurally incorrect but cannot determine whether a sentence makes any sense. The next phase of natural language understanding systems is to produce interpretations for sentences.

This phase, called semantic analysis, is described next.

2.3 Semantic Analysis

The primary aim of the semantic analysis phase is to obtain the meaning of sentences. Hence, there is a need for a *meaning representation language* (MRL) to represent the results of semantic analysis. The MRL should have the following properties [28]:

- The statements in the MRL should be unambiguous.
- We should be able to tell which statements are valid.
- We should be able to derive mechanically new statements that follow from a given statement.

There are several MRLs for semantic analysis. Two common representations are case frames [2, 8] and logical form language (LFL) [47, 52]. Case frames are based on the view that a sentence has a deep structure consisting of a verb, which represents the central component, and one or more noun phrases. Each noun phrase is associated with the verb in a particular relationship. The relationships are called *cases*. These cases are filled as a sentence is analysed. Typical cases include [8]:

- *Agent*: The investigator of the event.

- *Counter-Agent*: the force or resistance against which the action is carried out.
- *Object*: the entity that moves or changes or whose position or existence is in consideration.
- *Result*: the entity that comes into existence as a result of the action.
- *Instrument*: the stimulus or immediate physical cause of an event.
- *Source*: the place from which something moves.
- *Goal*: the place to which something moves.
- *Experiencer*: The entity which receives or accepts or experiences or undergoes the effect of an action.

As an example consider the sentence:

John opened the door with the key.

John would be the *Agent* of the verb opened, the door would be the *Object*, and the key would be the *Instrument*.

For the present work, we use the LFL as an MRL. The main reason for choosing this representation is its closeness to the formal specification representation we are aiming to produce from the semantic interpretations.

In LFL the meanings of sentences are logical forms. The main predicates are word senses. Each predicate of LFL takes a fixed number of arguments. The arguments might be variables, constants or other logical forms. The formation rules for logical forms are as follows:

- If **P** is a predicate of LFL taking **n** arguments, and each of $x_1, \ldots, x_n$ is either a constant or a logical form or a variable then $P(x_1, \ldots, x_n)$ is a logical form.
- If **P** and **Q** are logical forms then **P** & **Q** is a logical form.
- If **P** is a logical form and **E** is a variable, then **P:E** (read **P** indexed by **E**) is a logical form.

The predicates and arguments are obtained from the different parts of the sentence. Each syntactic group has a representation in LFL. The semantic interpreter will combine these syntactic groups to produce the complete interpretation of the sentence. In the next subsections, we will show how the different syntactic groups can be represented in LFL.

2.3.1 Interpretation of Noun Phrases

In the LFL, a noun is generally represented as a 1-place predicate where the name of the predicate is obtained from the singular form of the noun. Examples:

stock is represented by $stock(X)$

company is represented by $company(X)$

where X is a variable and *stock* and *company* are predicates. In general, we adopt the Prolog convention that variables begin with a capital letter.

There are two exceptions to the representation of nouns by 1-place predicates. Some nouns, called relational nouns, take two arguments.
Example:

father is represented by $father(X, Y)$

which is interpreted as X is a father of Y. Depending on the context, some ordinary nouns may behave as relational nouns and therefore take two arguments as in the sentence:

The company maintains a description for each item of stock.

The noun *item* is related to the noun *stock* and is represented by $item(X, stock)$. The other exception is the representation of proper nouns which correspond to constants in LFL.

In LFL, noun phrases do not have isolated meanings of their own but only contribute to the meaning of the sentence in which they appear. As we have seen in section 2.2.1, the head noun of a noun phrase is usually modified by premodifiers and postmodifiers. In the following subsections we analyse the classes of noun modifiers.

Determiners

Determiners are part of a large class of modifiers called focalizers. Focalizers are words that need a focus to determine the meaning of a sentence. In most cases, determiners' senses, as predicates in LFL, have two arguments which are filled by logical forms. The first argument is called the *base* of the determiner and the second is called the *focus*. In general a determiner is

represented by:

$$determiner(Base, Focus)$$

Typically, the base comes from the remainder of the noun phrase in which the determiner appears, and the focus comes from some of the sisters of the noun phrase. The pair *(Base,Focus)* is called the *scope* of the determiner. An example of a sentence involving determiners and its logical forms is:

> The company maintains a system.
> *the(company(X),ex(system(Y), maintain(X,Y)))*

We can read this logical form as follows. The quantifier *the* has two arguments: the base $company(X)$ and the focus:

ex(system(Y), maintain(X,Y))).

This denotes that there is a system Y that is maintained by the company X. We will use logical forms like this one in the remainder of this thesis.

Adjectives

When interpreting adjectives, we distinguish between two categories: *extensional adjectives* and *intensional adjectives.* Intensional adjectives occur in a composed noun where it is not possible to dissociate the adjective from the other parts of the composed noun. Intensional adjectives take logical forms as arguments. For example:

> The pilot uses a moving map display.

is represented by:

the(pilot(X), the(moving(map(display(Y))), use(X,Y)))

which means that the moving map display is a single entity. Whereas if the sentence is interpreted as:

the(pilot(X), the(map(display(Y)) & moving(Y), use(X,Y)))

it means that map display is an entity which is modified by the adjective moving.

Extensional adjectives can be dissociated from the other parts of the composed noun and therefore behave as nouns in having one argument. For example:

A complex aircraft uses a radar.

is represented by:

ex(aircraft(X) & complex(X), ex(radar(Y), use(X,Y)))

2.3.2 Interpretation of Verbs Phrases

Depending on their category, verbs may be represented by predicates having nil, one, two or three arguments. The predicate's name is obtained from the infinitive form of the verb which is defined in the lexicon.

Examples:

- It snows.

  snow

- The program crashed.

  *the(program(X),***crash(X))**

- The student writes a program.

  *the(student(X),ex(program(Y),***write(X,Y)))**

- The police gave a reward to John.

  *the(police(X),the(reward(Y),***give(X,Y,john)))**

One particular property of verbs is the voice. Verbs can have an active or a passive voice. For example the passive version of the last sentence is:

The reward was given to John by the police.

This sentence has exactly the same meaning as when the verb is in the active voice. Therefore it should have the same interpretation. MLGs have this ability and produce exactly the same logical form for both sentences.

2.3.3 Interpretation of Prepositional Phrases

In section 2.2.3 we described the structure of prepositional phrases. In the following subsections we associate an interpretation for each case considered in that section. We use the same examples cited in the syntactic analysis to illustrate these interpretations.

Interpretation of a String of the Form *P-NP*

Considering the general form $PP \rightarrow P\text{-}NP$, a prepositional sense is a 2-place predicate. The first argument corresponds to the noun phrase associated with the preposition and the second corresponds to the phrase modified by the prepositional phrase. A prepositional phrase may modify a verb or a noun phrase. An example where a prepositional phrase modifies a noun phrase is:

The pilot uses an aircraft with a sophisticated system.

which is represented by:

the(pilot(X), ex(aircraft(Y),
ex(system(Z) & sophisticated(Z) & **with(Y,Z)**, *use(X,Y))))*

When a prepositional phrase modifies a verb, the second argument of the preposition predicate will be a logical form. For example the sentence:

The pilot detects the obstacles with a radar.

is represented by:

the(pilot(X),the(obstacle(Y),ex(radar(Z), **with(Z,detect(X,Y))))))**

Interpretation of Intransitive Prepositions

For this category of prepositional phrases, the interpretations adopted are:

1. If the preposition plays the role of an adverb, then it is treated like an adverb as in the following example:

   John went downstairs.

downstairs(go(john))

The preposition takes a single argument that represents the rest of the sentence. In the above logical form, go is used since it is the infinitive of the verb went.

2. If the preposition is a particle of a verb, then it is treated as part of the verb as in:

   John puts the clothes on.

   the(cloth(X), put_on(john,X))

Interpretation of a String of the Form *P-P-NP*

In the syntactic analysis, we showed that this string can be parsed in two different ways. In the case where there is a preposition which has a prepositional phrase as a complement, the prepositions are combined as if they form a single one as in:

The challenger ran out of time.

the(challenger(X),out_of(time,run(X)))

out of is a constituent that acts as a unit and cannot be separated. The previous sentence can be reformulated as:

Out of time ran the challenger.

but we cannot split the two prepositions.

When an intransitive preposition is followed by a normal prepositional phrase, the first preposition is considered as a particle of the verb it modifies.

For example:

The man raced away in a red car.

the(man(X), ex(car(Y) &red(Y), in(Y, race_away(X))))

Here the two prepositions can be split and we can say:

In a red car the man raced away.

Interpretation of a String of the Form *P-NP-P-NP*

The interpretation of the three different cases identified in the syntax phase is performed as follows. In the first case where the whole prepositional phrase can behave as a single constituent, the prepositions successively modify the verb phrase.
Example:

The flight is planned from Blackpool to Doncaster.

from(blackpool,to(doncaster,the(flight(X),be(X,plan(X)))))

In the second case, only the noun phrase of the first prepositional phrase behaves as a single constituent. Therefore the noun phrase is used as a complement of the first preposition. The following example illustrates this:

John went to the house in the woods.

the(wood(X),the(house(Y) & in(Y,X),to(Y,go(john))))

The third case is interpreted as two independent prepositional phrases modifying a verb phrase.
Example:

John went to the park with Mary.

the(park(X),to(X,with(mary,go(john))))

2.3.4 Interpretation of Adverbs

All adverbs take logical forms as arguments. Some adverbs take a single argument as in the example:

John sold the car yesterday.

which is interpreted as:

yesterday(the(car(X), sell(john,X))) .

Other adverbs take two arguments. This category of adverbs is part of the focalizer class. The adverbs have the same interpretation as determiners and need a base and a focus to determine their scope. They have the structure:

$$adverb(base, focus)$$

This case of adverbs is used by McCord to determine which part of the sentence is stressed. This aspect of McCord's work is not relevant for the current research. For more details see [46].

2.3.5 Interpretation of Conjunctions

A sentence can contain an infinite number of coordinating conjunctions. This makes the analysis of such sentences very difficult. An attempt to treat conjunctions by Dahl and McCord [20] has resulted in a system that analyses only very simple sentences containing a maximum of two conjuncts. For example the analysis of:

Each man ate an apple and a pear.

can result in the following logical form:

each(man(X), ex(apple(Y), eat(X,Y)) & ex(pear(Z),eat(X,Z)))

The embedding of coordinating conjunctions in a sentence makes the production of logical forms very difficult and its interpretation ambiguous as we will show in the next chapter.

However subordinating conjunctions are much easier to treat. In general they involve only two clauses that are related by one conjunction as in the example:

Each item of stock is assigned a unique identifier when it is introduced.

This is interpreted as:

all(item(X,stock), ex(identifier(Y) & unique(Y),
when(be(X,introduce(X)), be(X,assign(X,Y))))))

As we have hinted, handling coordinating conjunctions adequately in

general remains a difficult research problem [20]. Hence we resolve coordinating conjunctions manually by splitting the conjuncts into simple sentences.

2.3.6 Interpretation of Pronouns

Pronouns are another class of words which are difficult to deal with in general [33]. An example of how some pronoun references are resolved is shown in the following:

Bill owns a cat. He likes it.

This results in the following logical form:

ex(cat(X), own(bill,X) & like(bill,X))

Our current implementation also omits the resolution of pronoun references.

2.4 Logic Grammars

So far we have used CFGs to express natural language. It has been shown that the CFG formalism is not powerful enough to express natural language grammars [43, 53]. Another type of grammar is therefore necessary to efficiently analyse natural language. In the early seventies, Kowalski [40] and

Colmerauer [18] showed that programming in logic is possible, and that natural language grammars can be expressed easily and efficiently in predicate logic. This approach of expressing grammars, known as *logical grammars*, was quickly adopted and attracted many linguists.

In a logic grammar formalism, symbols are no longer restricted to be atomic symbols, but can be any logic term; so terminals and nonterminals can be augmented by an indefinite number of arguments. These extra arguments can be very useful when analysing natural language texts which have so many features. Arguments such as the number of a noun phrase or the tense of a verb phrase can be added to the grammars to give more consistency to the analysis.

Many systems have been developed using logic grammars. Most of them were subclasses of Colmerauer's metamorphosis grammars [18]. Among the developed systems, we can particularly cite Definite Clause Grammars (DCG) [53], Extraposition Grammars (XG) [51], Modifier Structure Grammars (MSG) [20] and Modular Logic Grammars (MLG) [44, 45, 46, 47]. In the next subsections we describe the DCG formalism and show how McCord improved it to obtain the MLG formalism that we use.

2.4.1 Definite Clause Grammars

The term DCGs was first used by Pereira and Warren [53], and refers to the fact that these grammars translate into Prolog definite clauses. The DCG formalism is seen as a natural extension of the context-free grammar. These extensions, according to Pereira and Warren [53] are:

1. DCGs provide for context-dependency in a grammar. This means that some sentences may depend on some particular context.

2. DCGs allow the building of arbitrary trees during the parsing process.

3. DCGs allows extra conditions to be added in the grammar rules.

In this section, we briefly describe the DCG formalism (for more details see [47, 53]). To show how grammars can be expressed in the DCG formalism, we start by defining a simple CFG which covers simple sentences such as:

A company maintains a stock

and

Each item has a unique identifier.

Each rule of a CFG has the form:

$$nt \rightarrow body.$$

where *nt* is a non terminal and *body* a sequence of nonterminals or terminal symbols separated by commas. Each rule is terminated by a full stop. For example, the following CFG defines a simple grammar:

$$
\begin{aligned}
\mathit{sentence} &\rightarrow \mathit{noun\_phrase}, \mathit{verb\_phrase}.\\
\mathit{noun\_phrase} &\rightarrow \mathit{determiner}, \mathit{adjective}, \mathit{noun}.\\
\mathit{noun\_phrase} &\rightarrow \mathit{determiner}, \mathit{noun}.\\
\mathit{verb\_phrase} &\rightarrow \mathit{verb}, \mathit{noun\_phrase}.\\
\mathit{determiner} &\rightarrow [a].\\
\mathit{determiner} &\rightarrow [\mathit{each}].\\
\mathit{noun} &\rightarrow [\mathit{company}].\\
\mathit{noun} &\rightarrow [\mathit{stock}].\\
\mathit{noun} &\rightarrow [\mathit{item}].\\
\mathit{noun} &\rightarrow [\mathit{identifier}].\\
\mathit{verb} &\rightarrow [\mathit{maintains}].\\
\mathit{verb} &\rightarrow [\mathit{has}].\\
\mathit{adjective} &\rightarrow [\mathit{unique}].
\end{aligned}
$$

The translation of these CFG rules into Prolog clauses is a relatively simple process. We associate to each nonterminal a 2-place predicate having the same name. The arguments of the predicate delimit the part of the string to which the nonterminal is applied. The predicate $\mathit{noun\_phrase}(S0, S1)$ means that there is a noun phrase between the positions $S0$ and $S1$ of the string of words. A terminal symbol is replaced by a 3-place predicate, *connect*. The predicate $\mathit{connect}(S1, T, S2)$ means that the terminal T lies between the positions $S1$ and $S2$. The previous CFG will then be translated

to the following Prolog clauses (see [47, 53] for details of the translation process):

$$\begin{aligned}
sentence(S0, S) &\ :\!- \ noun_phrase(S0, S1), verb_phrase(S1, S).\\
noun_phrase(S0, S) &\ :\!- \ determiner(S0, S1), adjective(S1, S2), noun(S2, S).\\
noun_phrase(S0, S) &\ :\!- \ determiner(S0, S1), noun(S1, S).\\
verb_phrase(S0, S) &\ :\!- \ verb(S0, S1), noun_phrase(S1, S).\\
determiner(S0, S) &\ :\!- \ connect(S0, a, S).\\
determiner(S0, S) &\ :\!- \ connect(S0, each, S).\\
noun(S0, S) &\ :\!- \ connect(S0, company, S).\\
noun(S0, S) &\ :\!- \ connect(S0, stock, S).\\
noun(S0, S) &\ :\!- \ connect(S0, item, S).\\
noun(S0, S) &\ :\!- \ connect(S0, identifier, S).\\
verb(S0, S) &\ :\!- \ connect(S0, maintains, S).\\
verb(S0, S) &\ :\!- \ connect(S0, has, S).\\
adjective(S0, S) &\ :\!- \ connect(S0, unique, S).
\end{aligned}$$

A grammar rule interpreter, which can automatically carry out this translation, is included in most versions of Prolog.

The DCG formalism was successful, to some extent, in translating simple sentences to logical form. However the DCG formalism has some problems when dealing with some particular kind of sentences. These problems include:

- *The problem of scoping with adverbs.*

  This problem deals with the use of adverbs as focalizers. A detailed study of these adverbs can be found in [46]

- *The definition of the priorities in the scope of the quantifiers.*

  For example consider the sentence:

  The company maintains a description for each item of stock.

  Because of its linear analysis the DCG formalism will produce the following interpretation:

  $$the(company(X), ex(description(Y), all1(item(Z), for(Z, maintain(X, Y)))))$$

  This interpretation suggests that there is one description for all items. Whereas the correct interpretation is:

  $$all1(item(X), the(company(Y), ex(description(Z), for(X, maintain(Y, Z)))))$$

  because the determiner each has a higher scoping than the determiner the.

- *The analysis of left-recursive constructions in noun phrases.* A sentence such as:

  John saw each boy's brother's teacher.

  cannot be correctly analysed by the DCG formalism because of its

linear analysis. The DCG will produce the following interpretation for the last noun phrase:

$$boy(X, brother(Y, teacher(Z)))$$

which is not the intended meaning. In the next section we show how sentences involving left-recursive constructions are analysed by an improved version of the DCG formalism.

These problems occur primarily because the DCG formalism does not separate syntax analysis from semantic analysis. To handle the above problems, we need to modify the DCG formalism. McCord suggested that this could be done by separating the analysis into a component that performs syntactic analysis and a semantic interpreter that can produce logical forms. The semantic interpreter deals with the construction of the logical forms, and the handling of all ambiguities which will result in the production of several logical forms for the same string of words. Some other advantages in having a separate semantic interpreter, such as having greater modularity in grammars, being able to construct syntactic structures and the usefulness of the syntactic structures for debugging large grammar, are cited by McCord in [47]. The new formalism is called *The Modular Logic Grammars* (MLG) and will be presented in the next subsection.

2.4.2 Modular Logic Grammars

The MLG formalism is an extension of the DCG formalism. All the ingredients of the DCG formalism are present in the MLG formalism. The DCG is extended in three ways:

1. *The declaration of strong nonterminals.*

   The easiest way to build syntactic structures automatically is to build syntax trees. However, in a large grammar there will be many syntactic rules that are auxiliary in nature. For example the recursive rules that find the postmodifiers of a verb. It would be undesirable for the application of every such rule to contribute a node to the analysis tree because:

   - The tree would be large.
   - The additional nodes in the tree would complicate the work of the semantic interpretation rules.

   To solve these problems, MLG distinguishes between *strong* nonterminals and *weak* nonterminals. A strong nonterminal is a nonterminal that contributes nodes to analysis trees. In the grammar there is a syntax rule that declares strong nonterminals. Every non-declared nonterminal is considered weak.

2. *The shift operator.*

   The shift operator is used "to shift" components of a noun phrase

involved in a left-recursive construction which mainly involves possessive noun phrases such as in:

John saw each boy's brother's teacher.

which should be interpreted as:

all1(teacher(X,brother(Y,boy(Z))), see(john,X))

When analysing the phrase each boy's brother's teacher the shift operator provides the information that brother should be on the left of boy and that teacher should be moved to the left of both nouns. This information is used by the semantic phase to get the correct interpretation in spite of using right-recursive grammars.

3. *Logical terminals.*

   In the MLG formalism, the body of a rule can contain a logical terminal. A logical terminal has the form:

   OP-LF

   where the term **LF** is a logical form and **OP** is a logical operator. The logical operator is used by the semantic interpreter to combine LF with the other logical forms obtained from the analysis of other parts of the sentence.

As with DCGs, MLGs are translated to Prolog clauses by adding some extra parameters holding the different information obtained from the analysis. Each nonterminal gets two extra arguments holding the *difference*

list [1] that represents the part of the sentence (a word string) analysed by the nonterminal. In addition, the nonterminals get some other arguments representing analysis structures such as the syntax tree and the associated logical operators. This translation is implemented by defining $\rightarrow$ as a macro in Prolog-2 and is described in appendix B.

At the end of the syntactic analysis a syntax tree is obtained. This syntax tree is then used by the semantic interpreter to produce logical forms.

2.5 An Illustration of the MLG Translation Process

In this chapter we have described the syntax and semantic interpretation phases that we have adopted. In section 2.3, we described the logical form language that we use as our meaning representation language and in section 2.4 we described logical grammars. In this section, we present an illustration of the approach to give a deeper appreciation of the transformations that are carried out. This illustration is given mainly for completeness and may be skipped without affecting the rest of the thesis.

Let us consider the following sentence:

The company maintains a description for each item of stock.

The syntax analysis phase produces two syntax trees for this sentence. One of these takes the form given in figure 2.4.

[1] A difference list is simply a representation of a list by the difference of two lists. See [65] for a tutorial account.

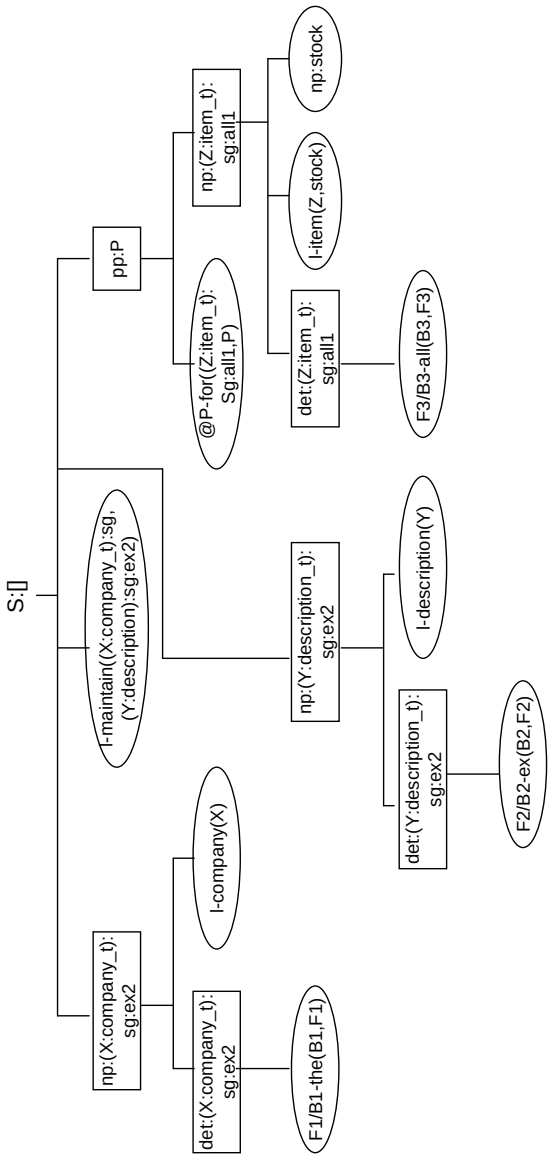

Figure 2.4: An MLG syntax tree

In general, the nodes of such a syntax tree are either *labels* or *semantic items* in the following form:

- *Label* is of the form *Pred*: A where *Pred* is the principal functor of a strong nonterminal NT and A is either the first argument of NT or [] if NT has no arguments.
- *Semantic items* are terms of the form OP-LF where OP is a *logical operator* (also called a *modification operator*) and LF is a logical form. Semantic items only appear as leaf nodes.

The aim of the semantic interpretation phase is to produce a logical form from such syntax trees. This is done by constructing augmented semantic items that take the form:

$$sem(Label, Op, LF)$$

where *Label* is a node label, OP a logical operator, and LF is a logical form. The augmented semantic item for a syntax tree is produced in the following recursive manner:

1. A terminal (or leaf) node OP-LF results in an augmented semantic item: $sem(terminal, OP, LF)$.
2. For a nonterminal (or non-leaf) node we first obtain an augmented semantic item for each of its daughters. These augmented semantic items are then used to *modify* the augmented semantic item:

$$sem(Label, id, t)$$

where *Label* is the label of the node, *id* is a special logical operator that is the identity operator, and t is simply the logical form *true*.

An important part of the second step of this procedure is the modification of an augmented semantic item by other items. The kind of modifications made depend on the logical operators and include:

- Modification by the identity operator id.
- Modification by the left conjoin operator l.
- Modification by the operator F/B.

We illustrate these in the following example and give all the operators in appendix D.

Example:

Let us consider the left subtree of figure 2.4 which represents the noun phrase The company. The semantic analyser first produces the augmented semantic items for each subtree of the noun phrase node. The augmented semantic items produced are:

$$sem(det, F1/B1, the(B1, F1))$$

$$sem(terminal, l, company(X))$$

The semantic analyser will then produce an augmented semantic item for the parent node of the daughters, which in this case is:

$$sem(np, id, t)$$

The augmented semantic items of the daughters will then modify the augmented semantic item of the parent node. The modification of

$$sem(np, id, t)$$

by

$$sem(terminal, l, company(X))$$

results in the augmented semantic item:

$$sem(np, l, company(X))$$

Since, by definition, the identity operator has no effect on the logical form.

The modification of the augmented semantic item

$$sem(np, l, company(X))$$

by

$$sem(det, F1/B1, the(B1, F1))$$

results in the following augmented semantic item:

$$sem(np,@P,the(company(X),P))$$

The operator $F1/B1$ uses the logical form $company(X)$ to fill the first argument of the logical form $the(B1, F1)$. The logical operator of the resulting augmented semantic item is $@P$. This operator will later use any logical form produced by the remaining subtrees of the sentence to fill the second argument of the logical form $the(company(X), P)$).

In addition to these modifications, the semantic interpretation phase may also involve reshaping. There are two kinds of movement in reshaping:

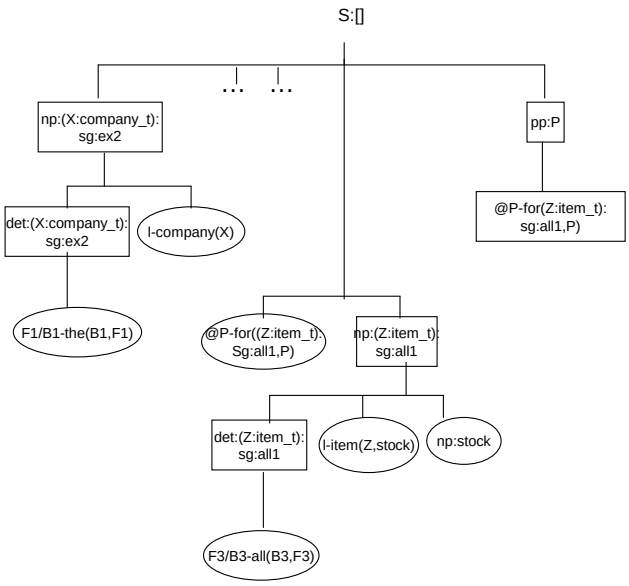

Figure 2.5: Raising a subtree

- *Raising* promotes daughters of a node to be left sisters of the node.
- *Reordering* rearranges subtrees within a given level.

As an example of these two movements, consider again the syntax tree given in figure 2.4. First we describe why raising is needed and then give the transformation that is performed. The noun phrase each item of stock is quantified by the determiner each which has a higher precedence than the quantifiers of the other noun phrases of the sentence. Hence, this noun phrase is promoted to be the left sister of the prepositional phrase. This puts the noun phrase at the same level as the other two noun phrases. Figure 2.5 gives the resulting tree.

Reordering is among subtrees at the same level. Here again, the noun

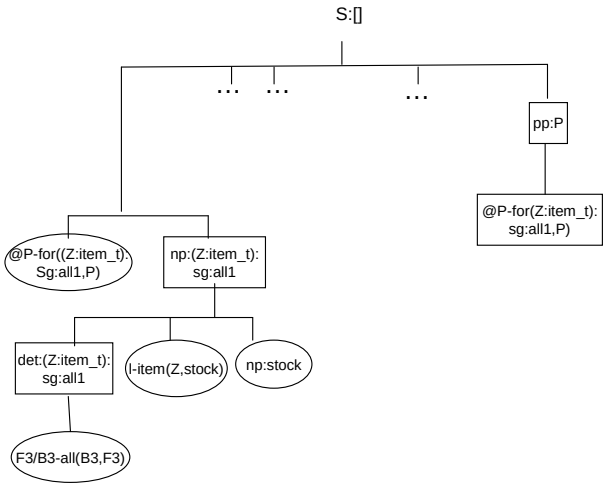

Figure 2.6: Reordering subtrees

phrase each item of stock is found to have a higher precedence quantifier. It is therefore put first in the list of augmented semantic items to be analysed. In terms of the syntax tree, figure 2.6 gives the reshaped tree.

In this section we have illustrated the kind of transformations made by the semantic interpretation phase. The appendices give the implementation which gives further details.

3

Using LFL to Analyse Natural Language Documents

3.1 Introduction

The main disadvantage in using natural language to write specifications is the potential for ambiguities. In any specification document written in natural language, it is essential to remove any ambiguity before proceeding to any further analysis. This leads to two major problems. The first problem is the detection of the ambiguities that are present and the second is the resolution of these ambiguities. Many studies have attempted to define the kind of problems that exist in natural language texts [35, 48, 57]. For example Meyer [48] has identified seven classes of deficiencies in natural language specification texts. Meyer called them "the seven sins of the

specifier". These seven sins are:

- *Noise*: The presence in the text of an element that does not carry information relevant to any feature of the problem.
- *Silence*: The existence of a feature of the problem that is not covered by any element of the text.
- *Overspecification*: The presence in the text of an element that corresponds not to a feature of the problem but to features of a possible solution.
- *Contradiction*: The presence in the text of two or more elements that define a feature of the system in an incompatible way.
- *Ambiguity*: The presence in the text of an element that makes it possible to interpret a feature of the problem in at least two different ways.
- *Forward reference*: The presence in the text of an element that uses features of the problem not defined until later in the text.
- *Wishful thinking*: The presence in the text of an element that defines a feature of the problem in such a way that a candidate solution cannot realistically be validated with respect to this feature.

It is claimed [57] that humans are bad at detecting ambiguities, but are very good at resolving the ambiguities. In this chapter we show how

ambiguities and incompleteness can be detected. These ambiguities and incompleteness are categorised as:

- Lexicographic ambiguities.
- Grammatical ambiguities.
- Textual cohesion

We consider each of these in the following sections.

3.2 Lexicographic Ambiguities

Many words in English have different meanings. The resolution of these ambiguities requires the selection of the exact definition for each word. The logical form language uses semantic types to resolve these ambiguities. These types can be used as a set of some general classes as defined by McCord [47, page 387], and Allen [2, page 195]. For example Allen has used the type hierarchy given in figure 3.1 to classify entities.

Objects in the world are classified into groups and each object has its specific characteristics. One of the most fundamental characteristics of any object is its type. The use of types can sometimes unambiguously identify these objects. As shown in figure 3.1, this decomposition is exhaustive for some entities, for example each physical object (PHYSOBJ) is either LIVING

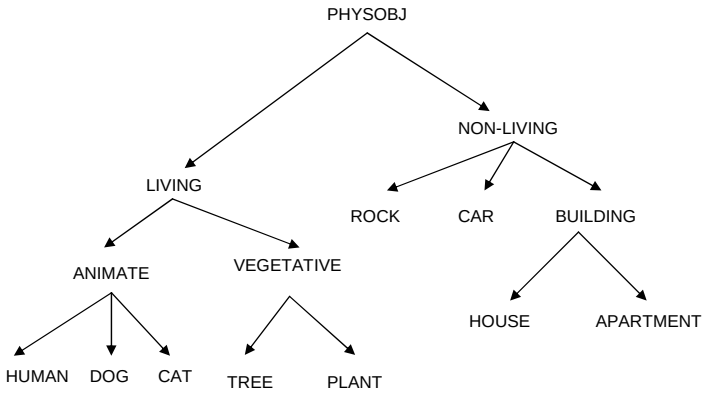

Figure 3.1: A type hierarchy for physical objects

or NONLIVING. Similarly every LIVING object is either ANIMATE or VEGETATIVE. But this decompositions is not exhaustive for other entities, for example there are more classes of ANIMATE objects then just HUMAN, CAT and DOG. Depending on the context and the kind of object that are manipulated, each organisation or system may need a different classification.

Another category of lexicographic ambiguities is when a word belongs to more then one syntactic category. For example "flies" may be the plural of the noun "fly" or the present, singular third person of the verb "to fly". These ambiguities are easily identified when the syntactic category of the word is identified. For example, in the sentence:

The pilot flies over the town.

Once the category of the word "flies" is identified as a verb, the ambiguity is resolved. When using LFL most of the lexicographic ambiguities can be resolved using this approach.

3.3 Grammatical Ambiguities

A grammatical ambiguity occurs when there is more than one possible parsing for a sentence or part of a sentence. The different parsings will lead to different interpretations and different meanings. The sentence:

The pilot draws the tracks of the route on the map.

has three different parsings which lead to the following interpretations.

1. The pilot draws (the tracks of the route) on the map.

   That is, the information is drawn on the map.

2. The pilot draws (the tracks of the route on the map).

   That is, the tracks are already on the map and the drawing is done somewhere else.

3. The pilot draws the tracks of (the route on the map).

   That is, the route is given on the map and its tracks are drawn somewhere else.

The location where the drawing takes place is different according to which interpretation we choose. Finding the proper place to attach the preposition phrase needs prior knowledge of the event. In general, a natural language understanding system, cannot decide which interpretation is meant. It can however highlight the ambiguity and produce all possible interpretations and it is the role of the analyst to choose the desired interpretation.

Ambiguities may also occur within parts of a sentence. This ambiguity occurs mainly in noun phrases. For example the sentence:

A complex aircraft uses a moving map display.

will have two different parsings, according to whether moving is interpreted as an adjective modifying the noun map display or as part of the noun moving map display.

This ambiguity may occur also in the interpretation of nominalisations – where the *ing form* of a verb plays the role of a noun – such as in the following sentence:

The shooting of the hunters is disgraceful.

If The shooting of the hunters is interpreted as a subject then the shooting is done by the hunters, but if it is interpreted as an object then the hunters have been shot.

3.4 Textual Cohesion

When writing texts, many techniques are used to ensure that the different parts of the text are connected correctly and to ensure a smooth transition from one idea to another. These techniques are called *textual cohesion*. The connected sentences will define the context in which they have to be interpreted. Therefore it will be difficult to take a sentence out of context

and try to interpret it. Each sentence is linked to the others. Jackson [35] has identified five types of textual cohesion.

1. Reference
2. Substitution
3. Ellipsis
4. Conjunction
5. Lexical cohesion

Unfortunately, textual cohesion is a source of ambiguities and incompleteness. In most specifications, the authors attempt to avoid the use of ambiguous textual cohesion. However, it is hard to find a realistic specification without ambiguities or incompleteness. In the next subsections we will look at each type of textual cohesion and when possible give examples of specifications containing these deficiencies.

3.4.1 References

References involve items that cannot have their own interpretation but make reference to something else for their interpretation. The interpretation of a reference requires the identification of the item from which it gets its interpretation. There are two types or references, *exophoric references* and

endophoric references. An exophoric reference is a reference to an item which is outside the considered text. An endophoric reference is a reference to an item within the text. In this research, we only concentrate on endophoric references.

There are three categories of endophoric references: personal, demonstrative and comparative. A personal reference is achieved by personal pronouns, possessive pronouns and possessive identifiers. According to Jackson [35], the third person pronouns are nearly always endophoric, but the first and second person pronouns may often be exophoric references. Sometimes a pronoun, particularly *it*, will refer not to a noun or a noun phrase, but to a longer stretch. An example of a pronoun reference can be seen in the following specification contained in [38, page 339]:

> "A graphics device displays images in a number of colours. It may be capable of depicting thousands of colours, a range from black to white, or possibly just two colours."

In the second sentence, *it* refers to the noun phrase "A graphic device". In this particular case the reference is contained in an independent sentence. In general *it* may refer to any noun phrase already mentioned in the text. It is only our background knowledge that enables us to find the item referenced. Without background knowledge it is generally very difficult to resolve a pronoun reference. In our approach a sentence containing a pronoun is considered ambiguous, the system therefore detects the ambiguity and asks the user to rephrase the sentence and resolve the reference.

A demonstrative reference is achieved by the demonstratives, the definite article *the*, and the adverbs *here, there, now* and *then*. For example, consider the following two sentences taken from the specification of the case study used in chapter 6:

> "The navigation system of a simple aircraft can be considered to comprise the pilot's map and route plan, a heading indicator and a pilot's visual sense. This system contrasts with that of a more complex aircraft."

In the second sentence *This* refers to The navigation system of a simple aircraft and *that* replaces the navigation system. The second sentence also refers to The navigation system of a complex aircraft which has not yet been described in the text. This kind of forward reference is called *a cataphoric* reference and a backward reference is called *an anaphoric* reference. Here again, the reference should be resolved before the natural language processor analyses the sentence.

A comparative reference can be either general, expressing the identity, similarity or difference between things; or particular, expressing a qualitative or quantitative comparison. For example, in:

> "The pilot of a simple aircraft with no sophisticated electronic navigation system, would be cleared to undertake *such* a flight in good visibility"

the *such* refers to A flight in a simple aircraft without a sophisticated electronic navigation system.

3.4.2 Substitutions

A substitution is defined by Jackson [35] as:

> " A grammatical relation, where one linguistic item substitutes for a longer one"

The substituted item is therefore interpretable only by reference to the original longer one. There are three types of substitutions: nominal, verbal and clausal. A nominal substitution occurs when a head noun of a noun phrase is replaced by *one* or *ones*, or the whole noun phrase is replaced by *the same*. A verbal substitution occurs when the verb *do* substitutes the lexical verb of the sentence. A clausal substitution occurs when the word *so* is used to substitute a positive clause or when *not* is used to substitute a negative clause as in the example [57, page 193]:

> "The program reads all client records and checks each record to determine if a premium is due notice or a cancellation (i.e., past due) notice should be issued and if so, prints the appropriate notice"

In this sentence the word *so* has been substituted for the clause "a premium notice or a cancellation notice should be issued."

3.4.3 Ellipsis

Ellipsis is a particular case of substitution. it occurs when certain words or phrases, which have been mentioned earlier in the text, are omitted. A structural gap therefore occurs and can only be filled by reference to a previous sentence. As for substitutions, there are three types of ellipsis: nominal, verbal and clausal. Nominal ellipsis involves the omission of the head of a noun phrase, sometimes together with some modifiers. Let us consider the following specification which is analysed in [37, page 200]:

> " A program is required to process a stream of telegrams. This stream is available as a sequence of letters, digits and blanks on some device and can be transferred in sections of predetermined size into a buffer area where it is to be processed."

In the first sentence, the concept of a stream of telegrams is introduced. In the second sentence, this concept is referred to only by this stream. It is well understood from the text that the author is talking about the stream of telegrams. In this ellipsis only part of the noun phrase is omitted. In this case ellipsis are easily recoverable if a list of entities is kept. The entity stream is already identified in the first sentence, when encountered in the second sentence it is related with the previous one. In the second part of the second sentence: can be transferred in sections of predetermined size, the subject stream is completely omitted. Here the omission is easily recoverable because it concerns the syntactic subject. In general this type

of ellipsis should be resolved before the analysis proceeds. This is done manually in the current implementation of our approach.

Verbal ellipsis involve the omission of the lexical verb from the verb phrase. For example:

> "Does a patient suffering from cancer always die? He may or he may not."

In the answer given in the second sentence the verb "to die" was omitted. The omitted verb phrase is recoverable from the previous verb phrase. Here again the resolution is done manually for the current implementation of the approach.

Clausal ellipsis occur when a large part of a clause is omitted as in the following example:

> "Some classifications that determine the types of objects are exhaustive. Every physical is either living or nonliving. Others are not exhaustive."

In the last sentence, *others* replaces "some classifications that determine the types of objects".

3.4.4 Conjunctions

Let us consider the following example [57, page 90]:

> "Packages which are being sent abroad and weigh less then 20 kg or are marked urgent are to be sent airmail"

In this kind of sentence, the scope of the conjunction is not defined. If we consider a domestic parcel weighing 10 kg and marked urgent, then according to the interpretation given to the above statement, it may or may not be sent by airmail. If the scope of the conjunction "and" includes both the following adjectival phrases then the parcel is not sent airmail. If the scope includes only the first adjectival phrase, then it is sent airmail. In general, the conjunctions "and", "or" and "not" are sources of ambiguity when more than one of them appear in a sentence. In our approach we have adopted to split conjunctions whenever possible. For example the sentence:

> The company maintains a description, a unit cost, a quantity in stock and a minimum reorder level for each item of stock.

is split into four small sentences which are:

- The company maintains a description for each item of stock.
- The company maintains a unit cost for each item of stock.
- The company maintains a quantity in stock for each item of stock.
- The company maintains a minimum reorder level for each item of stock.

Nevertheless, when the sentence is not ambiguous then there is no need for splitting as in the following example:

> Each item of stock is assigned a unique identifier when it is introduced.

3.4.5 Lexical Cohesion

Lexical cohesion refers to the replacement of a word by its synonym or a related word in successive sentences. The later occurrence of such words refer to and link up with previous occurrences. There are two types of lexical cohesion: reiteration and collocation. Collocations refer to the habitual company which words keep. For example the word *book* implies other words such as *page, title, shelf, library, ... etc.* The next sentence of the specification reported in subsection 3.4.3 is:

> "The words in the telegrams are separated by sequences of blanks and each telegram is delimited by the word "ZZZZ".

It is obvious that a telegram is composed of words but for a natural language processing system a link is missing between the telegram and the words. A sentence such as:

A telegram is composed of words.

may need to be added before the introduction of the previous sentence to resolve this incompleteness.

Reiteration may be of four types:

1. The same word may be repeated in successive, though not necessarily contiguous sentences. As we will see, this problem is solved by our approach because we keep a list of the entities encountered in the documents.

2. A synonym or near-synonym of a word may appear in a following sentence. This problem may be reduced by using semantic types. All words having the same meaning will be classified as being of the same semantic type.

3. A word may be replaced in a following sentence by another which is semantically superordinate to it. For example if we use the word "Porsche" in one sentence and in another sentence we refer to it by "The car". This kind of problem can be resolved by using the semantic types. The semantic type of "Porsche" will be for example "car" which is the same as the type defined for the word "car".

4. A word may be replaced in a following sentence by a "general word" which describes a general class of objects.

   The word "person" may replace a human, and the word "toy" may replace a toy that is mentioned before. Depending on the semantic types defined, this ambiguity may or may not be detected. For example, consider a sentence where the word "ball" is used and later referred to as "the toy". This ambiguity is resolved if ball and toy have the same semantic type.

To summarize, in this chapter we have shown the kind of ambiguities that could be present in natural language specifications and have given appropriate examples from specifications already present in the literature.

Where appropriate, we have also described how they are detected and resolved by our approach.

4

Identifying the Data Types

4.1 Introduction

In this chapter we describe how we identify and produce VDM data types from a natural language specification. Our approach proceeds in two stages:

1. We first aim to identify an entity relationship model (ERM)

2. Then we present a translation of the ERM to VDM.

In section 4.2 we show how the entities are identified. In section 4.3 we show how the relations are identified and finally in section 4.4 we show how the degrees of the relations are identified.

The first task in identifying an ERM is to obtain the list of entities in the specification and the relationships between the entities. There is no clear definition of what constitutes an entity. In SSADM for example [4],

an entity is defined as something of importance to the system about which information can be held. The same definition is also used by Bowers [14], who further suggests that entities can be:

- *Objects*: Person, car.
- *Events*: Birth, scoring a goal.
- *Activities*: Production, playing.
- *Associations*: Marriage (X is married with Y).

Grammatically speaking, the above list gives types which define entities that are related. They all belong to the same grammar category of *nouns*. Several authors have reported that nouns denote entities [4, 26].

Entities have relationships with other entities. In a bank system, for example, there is a relationship between the entities "Customer" and "Account". These two entities are related by an ownership relation which will probably be described in the various documents by the verb "have". We therefore believe that relations are described by verbs. This view is also supported by others [4, 26].

We base our identification process on the view that entities are denoted by nouns and relationships by verbs. For example, in the following sentences:

1. The pilot chooses the waypoints from the air.
2. A complex aircraft uses a radar.

The nouns suggest the entities:

$$\{pilot, waypoint, complex\ aircraft, radar, air\}$$

and the verbs the relations:

$$\{choose, use\}$$

However, as we mentioned in the previous chapter, noise exists in specifications and this can lead to the presence of irrelevant entities and relations in such lists. For example, in the following sentences:

1. An example route is planned for a flight from Blackpool to Doncaster.

2. The pilot may have unnecessarily flown through a storm.

In the first sentence, the verb planned does not describe any process but just a statement. In the second sentence, the entity storm may be not important to the specification of a flight database.

Bearing in mind that we aim to develop an interactive tool, this deficiency can be circumvented by requiring the analyst to remove those nouns and verbs that are not important when they are detected. We therefore expect that an analyst should filter such a list of entities and relations before proceeding.

We can, of course, obtain a list of nouns and verbs by simply scanning the text. However, this approach has several deficiencies and does not help one to:

1. Find the relationships between entities. For example in the sentence:

   A company maintains a description for each item of stock.

   although we can list the nouns as company, description, item and stock, we do not obtain any relationships between these entities.

2. Identify the degree of the relationships between the identified entities.

3. Extract compound nouns without ambiguity. For example consider the two sentences:

   - A computer-assisted flight planning system is used by a complex aircraft.
   - A pilot planning a risky flight needs special training.

   In the first sentence, "planning" is a part of the compound noun "computer-assisted flight planning system" whose form must be preserved. In the second sentence, planning is part of the participial verb phrase "planning a risky flight" and is not part of a compound noun.

The following subsections show how the use of logical form helps us to overcome these problems.

4.2 Identifying Entities

The nouns form the basic list of entities. Below, we see how entities can be extracted from sentences that contain simple and compound nouns.

4.2.1 Simple Nouns

Simple nouns are extracted from noun phrases containing just one noun. The sentence:

The aircraft may hit an obstacle.

contains two noun phrases: The aircraft and an obstacle. Each noun phrase is composed of a unique noun and each noun is extracted as an entity with its associated quantifier.

Relational nouns are also extracted in a similar fashion. For example the sentence:

The system of an aircraft comprises the plan of the pilot.

results in the entities:

$$\{system, aircraft, plan, pilot\}.$$

Proper nouns identify particular objects and therefore do not normally constitute entities. Hence in a sentence like:

An example route is planned for a flight from Blackpool to Doncaster.

Blackpool and Doncaster do not constitute entities.

4.2.2 Compound Nouns

Compound nouns are nouns which are composed of two or more nouns or a combination of nouns and adjectives. For example, the following sentences:

1. A complex aircraft uses a computer-assisted flight planning.
2. The flight planning software package calculates the route tracks.

will have respectively the following logical forms:

1. *ex(aircraft(X) & complex(X),*
   ex(computer-assisted(flight(planning(Y))),use(X,Y)))
2. *the(flight(planning(software(package(X)))),the(route(track(Y)),*
   calculate(X,Y)))

In the first sentence, the noun phrase a computer-assisted flight planning composed of four nouns. In the second, the noun phrase The flight planning software package is composed of four nouns. As we can see from the above logical forms we can easily extract the entities. The entities identified are: computer-assisted flight planning and flight planning software package.

Identifying entities by using just the head noun may, of course, lead to confusion. For example, in a specification of an aircraft system (see the case study in chapter 6), both the description of a simple aircraft and a complex aircraft may occur.

4.3 Identifying Relations

As we have seen in section 4.1, a natural way of identifying relationships is to use verbs and relational nouns.

4.3.1 Identifying Relationships within Relational Nouns

Relational nouns always define relationships between nouns. The sentences:

1. The company maintains a description for each item of stock.
2. The system of a simple aircraft comprises the plan of the pilot.

will produce respectively the following logical forms:

1. *all1*(**item(X,stock)**,*the(company(Y),ex(description(Z), for(X,maintain(Y,Z)))))*
2. *ex(aircraft(X)& simple(X),*__the(system(Y,X)__*, the(pilot(Z),* __the(plan(T,Z)__*, (comprise(Y,T)))))*

The logical forms show clearly the relations defined by relational nouns. In the first sentence there is a relation between item and stock and this is shown by **item(X,stock)**. In the second, there are two relations. The first relation is between simple aircraft and system and the second relation is between plan and pilot. These two relations are respectively shown by

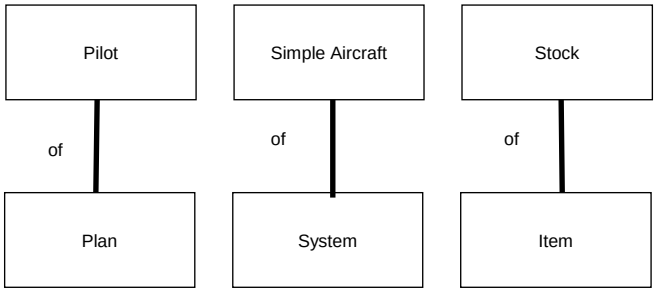

Figure 4.1: Relationships extracted

system(Y,Z) and **plan(T,Z)**. The direction of the relation is from the second entity to the first and we use of to name the relationship. The relations extracted for the above examples are shown in figure 4.1.

4.3.2 Identifying Relationships within Verb Phrases

Verbs generally refer to actions, events and processes [35]. In the case of transitive verbs, the verb defines a relation between two entities. Let us consider the sentences:

1. The pilot chooses the waypoints from the air.
2. The system of a simple aircraft is considered to comprise the plan of the pilot.

In the first sentence, the verb chooses relates the entities pilot and waypoints. This information is readily available from the logical form of the first sentence:

$$the(pilot(X), the(waypoint(Y), the(air(Z), from(Z, choose(X, Y)))))$$

where *choose* relates the variables X and Y which are defined in the logical form to be of type *pilot* and *waypoint*.

The second sentence has two verbs, making it a little more complex to analyse. The logical form produced for this example is:

$$ex(aircraft(X)\&simple(X), the(system(Y, X), the(pilot(Z), the(plan(T, Z),$$
$$be(Y, consider(Y, comprise(Y, T))))))$$

The verb comprise introduces the main action (which is represented in natural language as an infinite complement of the verb consider), and is therefore extracted as the relationship between the system of a simple aircraft and the plan of the pilot. The verb consider plays a subsidiary role and does not relate any entities. Again this information is ready to extract from the logical form. In cases where the logical form contains more then one verb, the inner verb phrase identifies the relationships between the entities. The relations extracted for the above sentences are given in figure 4.2.

The next step is to identify the degree of the relationships. The next subsection shows how the degree of a relation can sometimes be determined from the logical form of a sentence containing the relationship.

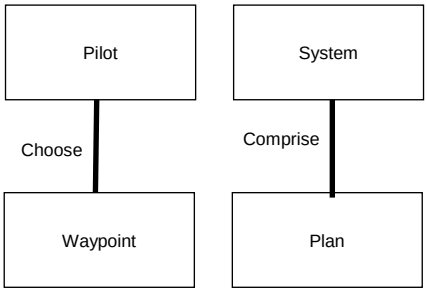

Figure 4.2: Verb Relations extracted

4.4 Quantification and the Determination of the Degree

Early attempts at natural language analysis assumed that quantifiers occurred explicitly in the text. Thus it was assumed that the presence of a universal quantifier was always indicated by words like "every" and "all", and the presence of an existential quantifier was indicated by words like "some" [31]. However, many sentences are implicitly quantified by articles.

In this section we first examine how such implicit quantifiers can be identified and then show how quantified LFL statements can sometimes aid the identification of the degree.

4.4.1 Identifying Implicit Quantifiers

Most studies of quantification identify quantifiers from the articles present in the sentences [2, 31]. Initial studies of quantification regarded both definite and indefinite articles as existential quantifiers, with some additional information in the case of definite articles [31]. More recent studies have shown various problems with this assumption and have shown how the indefinite article and the definite article can also lead to a universal quantifier.

Below we show the problems of identifying quantifiers from the articles "the" and "a" and our approach to these problems.

The Definite Article "the"

Russell [61] gives the following example to illustrate the meaning of the definite article "the":

The president of France is bald.

Russell argues that this should be interpreted as:

$$\exists X \cdot president\_of\_france(X) \wedge \neg(\exists Y \cdot president\_of\_france(Y) \wedge X \neq Y) \wedge bald(X)$$

The "additional information" given for the interpretation of the definite article is given in the statement that Y is the one and only president of France.

McCord recognises that this interpretation is inadequate in general but suggests that in some applications it is correct to translate "the" into the

unique existence. However, he does not give any guidance to when the usual existential quantifier should be used instead of the unique existential quantifier. In the case of obtaining the meaning of sentences in a requirements document, we cannot assume that one of these holds throughout the application. For instance, consider the sentences:

The students passed the exams.

The student passed the exams.

The first sentence does not suggest the unique existential quantifier, while the second does not suggest the normal existential quantifier.

As we will see later, obtaining appropriate quantifiers is an important prerequisite for our approach to identifying the degree of relationships between entities. Hence, we have attempted to improve upon McCord's approach to this problem.

In our approach, we do not simply translate "the" into the unique existence – instead we use the singularity or plurality of the noun to determine if it should be translated to the unique existence or normal existence. That is, if the quantified noun is singular, we adopt the unique existence, otherwise we interpret it as a normal existential quantifier. Again this information is available from the LF of the sentence.

We concede that there remain sentences for which these approaches remain inadequate. For instance, the following examples given by Hess [31] are not covered:

1. The unicorn is a mythical creature.

2. The lion is a dangerous animal.

In the first sentence we do not presuppose the existence of unicorns, but the sentence nevertheless makes perfect sense. This kind of sentences are unlikely to appear in requirements documents because specification of systems are normally about concepts and objects that exist. The second sentence shows that depending on the context, "the" could be interpreted as a universal quantifier, or a unique existential quantifier.

The Indefinite Article "a"

The use of the indefinite article as a quantifier is always a source of ambiguity [2]. The indefinite article can sometimes be translated to the existential quantifier and sometimes to a universal quantifier. In this subsection, we investigate when the indefinite article is interpreted as an existential quantifier and when it is interpreted as a universal quantifier.

According to Hess [31] the most important way to determine the quantification of a sentence is through the choice of the verb. For example, consider the following sentences:

1. A text editor *makes* modifications to a text file.

2. A text editor *is making* modifications to a text file.

3. A text editor *made* modifications to a text file.

4. A text editor *has made* modifications to a text file.

The present tense is used in example (1) to say that a text editor makes modifications to a text file in general. The main use of the present tense is to express habitual actions. In examples (2) to (4) we say that there is, or was, a case of a text editor making modifications to a text file. Therefore, Hess suggested that because the present tense is used in the first sentence, text editor must be universally quantified. Likewise, because of the tenses used in the other sentences, text editor must be existentially quantified in the remaining sentences.

In some cases the future is preferred over the present tense for general statements as in the following example:

A man who loves a woman will stroke her.

Dynamic verbs, such as to stroke, seem to call for the future tense, whereas static verbs such as to respect seem to go better with the present tense. Hence, Hess formulated the following rules:

- **Rule 1:**

  The subject of a sentence is existentially quantified if the VP is:

  1. in the past tense.
  2. in the progressive aspect, or
  3. in the perfective aspect.

- **Rule 2:**

  Otherwise the subject is universally quantified, in particular if it is:

  1. in the present tense or

2. in the future tense.

Once we have determined the quantification of the subject of the sentence, we have to do the same thing to the other components of the sentence. Let us consider the following examples:

1. A man who loves a woman is happy.

2. A man that loves a woman respects her.

Intuitively, we can see that woman should be existentially quantified in the first sentence and universally quantified in the second sentence. To observe the difference, Let us consider the logical forms of these sentences:

1. *all(man(X),ex(woman(Y)&love(X,Y),happy(X)))*

2. *all(man(X),all(woman(Y)&love(X,Y),respect(X,Y)))*

The main verb of the first sentence is happy and does not refer to the noun phrase woman. In the second sentence the main verb respects refers to the noun phrase woman. This is the reason why the noun phrase "woman" should be existentially quantified in the first sentence and universally quantified in the second. Hence, Hess suggested a third rule which is:

- **Rule 3:**

  In a restrictive noun phrase those arguments that are referred to by the main verb are universally quantified and those that are not referred to by the main verb are existentially quantified.

This rule now enables the correct interpretation of the above sentences. However, it does not hold for non-restrictive noun phrases. In particular, when a noun phrase appears at the right of a verb, the kind of sentences we have encountered suggest that the indefinite article should be interpreted as an existential quantifier. For example in the sentence:

A complex aircraft uses a radar.

The second indefinite article is interpreted as the existential quantifier and not as the universal quantifier.

There are two exceptions to the above rules which are analysed in the following cases:

- As an exception to rule 2, the past tense can express a universally quantified assertion, as in the following example:

  A student read books *when* I was young.

  This universal quantification is possible because the main verb (read) requires a spatial or temporal postmodifier(when).

- As an exception to rule 1, the progressive aspect can express universal quantification as in:

  John is always coming late

  This is only possible when the verb is modified by expressions such as "always", "in general", "regularly".

To cover these exceptions, we can suggest the following fourth rule which takes precedence over rules 1 and 2.

- **Rule 4:**

    1. The past tense can express a universally quantified assertion if the main verb requires a spatial or a temporal postmodifier.
    2. The progressive aspect can express a universal quantification if the verb is modified by expressions such as "always", "in general" and "regularly".

4.4.2 Obtaining the Degree from the Quantifiers

In the last subsection we saw how we could obtain quantified logical forms. In this subsection we show how the quantifiers associated with each entity can be used to determine the degree of a relationship between the entities. It is not always possible to determine the degree of a relation from the quantifiers. However, we describe how our system gives a default degree for some cases. Of course, the user is allowed to override the system determined degrees.

4.4.3 Identifying Many-to-One Relationships

Consider the following examples and their logical forms:

1. A complex aircraft uses the radar.

   all(aircraft(X) & complex(X), the(radar(Y), use(X,Y)))

2. A student passed the exam.

   ex(student(X), the(exam(Y), pass(X,Y)))

3. The students passed the exam.

   the(student(X), the(exam(Y), pass(X,Y)))

4. The company maintains a description for each item of stock.

   all1(item(X,stock), the(company(Y), ex(description(Z),

   for(X,maintain(Y,Z)))))

In the first example, The first entity in the relation is quantified by the universal quantifier and the second by the unique existential quantifier (the definite article quantifying a singular noun). Then, by definition, we have a many-to-one relationship from the first entity to the second. In the second and third examples, the first entity is quantified by the normal existential quantifier and the second by the unique existential quantifier. Based on our current experience, in such cases we interpret the existential quantifier as referring to more than one occurrence of the first entity. That is many occurrences of the first variable are related to only one occurrence of the second variable. Then by definition, we have a many-to-one relation between the entities student and exam.

In the fourth example, we consider the relation between item and decription. The entity item is quantified by the universal quantifier and the entity

description is quantified by the existential quantifier. Based on the different examples we have analysed, we infer a many-to-one relation between item and description.

Notice that the degrees given by default by this approach to the last three examples are not as strong as the one given to the first example. The analyst, is ofcourse, allowed to override the degrees identified by the system.

Some one-to-many relationships can also be detected by the system. For example consider the following sentences and their logical forms:

1. The company maintains a description for each item of stock.
   all1(item(X,stock), the(company(Y), ex(description(Z),
   for(X,maintain(Y,Z)))))

2. The student passed all exams.
   the(student(X), all2(exam(Y), pass(X,Y)))

Logical form language distinguishes between its different quantifiers by associating different predicates. For example LFL associates the predicate all1 for the determiner each, all2 for the determiner all and the predicate all for the interpretation of the indefinite article as a universal quantifier. These different predicates are used to determine the priorities between the quantifiers. These differences also help in the interpretation of the sentences. Hence, in the first sentence, the phrase each item of stock suggests that we are talking about one stock system that contains many items (i.e., a one-to-many relation between the entities stock and item).

Sentences where the first entity is singular and quantified by the definite article define one-to-many relationships. The second sentence is a typical example. An exception to this rule occurs when the second entity is also quantified by "the" and is singular. In this case, we infer a one-to-one relationship between the entities.

We have now given several cases in which we can identify the degree of a relationship. In other cases, when it is difficult to predict the degree of a relation, we let the user decide it.

At this stage we should have a list of entities, relations and the degrees of the relations and therefore the entity relationship model. The aim of our approach is to translate these entity relationship models to VDM data types. The translations are described in section 4.6. In the next section, we summarise the VDM notation that we use.

4.5 VDM Notation

VDM provides data types for sets, sequence, maps and composite objects. We describe each of these below.

Set Notation

We use the following set notation:

- We denote the empty set by $\{\,\}$.
- A binary predicate $a \in s$ is true iff a occurs in the set s.
- The set A**-set** denotes the set of all finite subsets of A.
- The function **card** s denotes the number of elements in the set s.
- The expression $s1 \cup s2$ denotes the union of the sets $s1$ and $s2$. Thus for example:
$$\{a, b\} \cup \{b, d, e\} = \{a, b, d, e\}.$$
- The expression $s1 \cap s2$ denotes the intersection of the sets $s1$ and $s2$. Thus, for example:
$$\{a, b, c\} \cap \{b, c, d\} = \{b, c\}.$$

Sequence Notation

A sequence is an ordered collection of elements. Sequences differ from sets in that duplicates and the order of elements are significant. We use the following sequence notation:

- The notation A^* denotes the set of all finite sequences whose elements are obtained from the set A.
- We denote the empty sequence by $[\,]$.
- The function **elems** s denotes the set of elements in the sequence s. For example:

$$\textbf{elems}\,[1,2,3,1] = \{1,2,3\}.$$

- The function **len** s denotes the length of the sequence s.
- The expression $s1 \frown s2$ denotes the concatenation of the sequences $s1$ and $s2$. For example:

$$[1,2,3] \frown [1,3,4] = [1,2,3,1,3,4].$$

Map Notation

A map defines a many to one mapping from one set, called the domain, to another called the range. The notation **dom** m returns the elements in the domain of a map, and **rng** m returns the elements in the range of a map. For example,

$\textbf{dom}\,\{a \mapsto 1, b \mapsto 2\} = \{a, b\}$ and $\textbf{rng}\,\{a \mapsto 1, b \mapsto 2\} = \{1, 2\}$.

We can add new maplets to a map by using the union operator. We can

overwrite a maplet by using the overwrite operator †. For example, given:

$$m = \{a \mapsto 1, b \mapsto 2, c \mapsto 3\}$$

we can overwrite the second maplet by:

$$m \dagger \{b \mapsto 3\}$$

to obtain a map:

$$m = \{a \mapsto 1, b \mapsto 3, c \mapsto 3\}$$

We can also delete a maplet by using the operator $\lhd$. For example, with the above map m:

$$\{b\} \lhd m = \{a \mapsto 1, c \mapsto 3\}$$

Composite Objects

Composite objects are record like objects that have fields. A composite object type can be defined by using the notation:

$$\begin{aligned} \mathit{Cobject} :: \; & \mathit{selector}_1 \; : \; \mathit{Type}_1 \\ & \mathit{selector}_2 \; : \; \mathit{Type}_2, \\ & ... \\ & \mathit{selector}_n \; : \; \mathit{Type}_n \end{aligned}$$

The values of fields can be selected by applying the field name to the composite object. For example, given a composite object named v of the above type *Cobject*, we can select the second field by:

$$selector_2(v).$$

Composite objects can be created by using an appropriate constructor function. These constructor functions are obtained by pre-fixing the name of the composite object type by *mk-*. For example, an object of type *Cobject* can be created by:

$$mk\text{-}Cobject(value_1, value_2, ...value_n)$$

where the arguments are of the appropriate type.

We can change a component of a composite object by using the μ function. For example, if we want to change the value of the second field of a composite object *co* of type *Cobject* we would write:

$$\mu(co, selector_2 \mapsto new\ \ value)$$

where *new value* is the required new value.

Function Definition

In VDM the symbol $\triangleq$ is used as the function definition symbol. VDM functions can be recursive and can use IF and CASE statements in the usual manner.

4.6 Production of VDM Data Types from Entity Relationship Models

In general, the entity relationship model produced by the above process will be quite complex. As an example appendix G contains the diagram obtained for a problem that we illustrate in chapter 6. As the diagram shows, we may have several sub-models. The diagrams may contain many-to-many relationships as well as one-to-one, many-to-one, and one-to-many relationships.

We can model one-to-one relationship as a one-to-one map in VDM. Many-to-many relationships can be modelled as a set of pairs. However, as with SSADM [4] we require the user to have resolved one-to-one and many-to-many relationships so that our entity relationship diagram only contains one-to-many and many-to-one relationships. It is shown in SSADM that many-to-many relations can always be decomposed into two or several one-to-many relations. One-to-one relations can be merged. We now examine how the remaining relationships can be modelled in VDM.

4.6.1 Modelling One-to-Many Relationships

Figure 4.3 gives a typical one-to-many relationship which defines that there are many items in a stock system. We can model such relationships in VDM

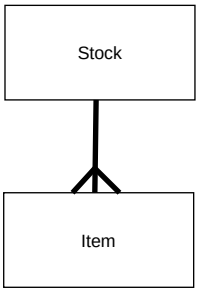

Figure 4.3: Modelling one-to-many relationships

in several different ways including the following three:

1. As a set of items: $Stock = Item$-**set**
2. As a sequence of items: $Stock = Item^*$
3. As a map from item identifiers to items: $Stock = Item\text{-}ID \xrightarrow[\text{m}]{} Item$

Notice that for the map data type we only use the many part of the relation. Based on the examples in the literature (e.g. [37, 68]), we prefer the third approach since VDM's map data type appears to be more natural for modelling this kind of relationship. However, when the user specifies that the order of the items is significant, the second option is adopted.

4.6.2 Modelling Many-to-One Relationships

Figure 4.4 gives a typical many-to-one relationship between the entities *teacher* and *course*. We can model this using a VDM map. For this example

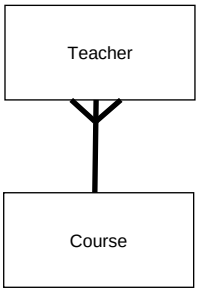

Figure 4.4: Modelling one-to-one relationships

the type would be:

$$\begin{aligned} State &:: sm &&: System \\ &\quad\ tm &&: Teachers \end{aligned}$$

$$System = Teacher\text{-}Id \xrightarrow[\text{m}]{} Course$$

$$Teachers = Teacher\text{-}Id \xrightarrow[\text{m}]{} Teacher\text{-}Infos$$

$$\begin{aligned} Teacher\text{-}Infos &:: Name &&: Text \\ &\quad\ Qualification &&: Text \\ &\quad\ ...etc \end{aligned}$$

With the following invariant:

$$inv\text{-}State(mk\text{-}State(sm, tm)) \quad \triangleq \quad \mathbf{dom}\ sm = \mathbf{dom}\ tm$$

In addition, we may also have the situation shown in figure 4.5 where there is a many-to-one mapping to leaf entities. Benyon [11] states that if an entity has no attributes apart from its identifier, has a one-to-many relationship with another entity and that the relation between the entities

is obligatory then it can probably become an attribute of the other entity. Hence, the attributes of an entity have a one-to-many relationships with that entity. In such situations we use a composite object to model such relationships. For the following example (based on a past undergraduate exam question) :

```
1- A company maintains a simple stock system.
2- The company maintains a description for each item of stock.
3- The  company  maintains a unit cost for each item of stock.
4- The company maintains a minimum reorder level for each item
   of stock.
5- The company  maintains a quantity in stock for each item of
   stock.
6- Each item of  stock is assigned a unique identifier when it
   is introduced.
```

Our approach results in the ER diagram of figure 4.5. The subtree for item is then translated to the following composite object type:

$$
\begin{array}{lll}
\mathit{Item} :: \mathit{description} & : & \mathit{Description\_t} \\
\mathit{unit\text{-}cost} & : & \mathit{Cost\_t} \\
\mathit{quantity} & : & \mathit{Quantity\_t} \\
\mathit{reorder\_level} & : & \mathit{Level\_t}
\end{array}
$$

where we obtain the names of the selectors from the nouns that identified the entities.

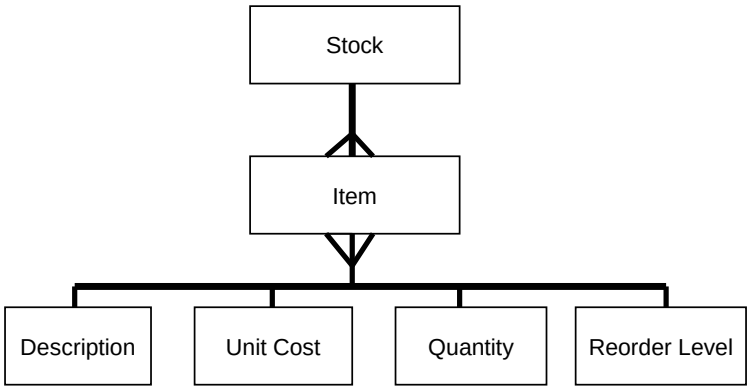

Figure 4.5: The Stock Case Study

5

Invariants and the Specification of Operations

5.1 Introduction

Data types usually have associated invariants. Invariants are truth-valued functions which can be used to record restrictions on data types. Invariants are very important in the definition of data types because it is only after their definition that we can proceed to specify functions and operations. For example, in a banking system the opening of some accounts are conditional upon the amount of money one has to invest. Let us say for example that the opening of a *diamond* reserve account requires a minimum of £5000. If we model an account as the following composite object:

$$
\begin{array}{lll}
\mathit{Diam\_Acc} :: & \mathit{Account\_Number} & : \mathit{Ident} \\
& \mathit{Customer\_Name} & : \mathit{Name} \\
& \mathit{Customer\_Address} & : \mathit{Address} \\
& \mathit{Balance} & : \mathit{Balance}
\end{array}
$$

The invariant that restricts the balance of the account to be greater than or equal to £5000 is defined as:

$$\mathit{inv\text{-}Diam\_Acc}(\mathit{mk\text{-}Diam\_Acc}(an, cn, ca, ba)) \quad \triangleq \quad ba \geq 5000$$

After the definition of the invariant, the *valid* objects of *Diam_Acc* are those that belong to the composite object type and also satisfy *inv-Diam_Acc*. The invariant cuts out those elements which do not arise in reality. This is the typical way in which invariants arise and also their simplest form.

Invariants can be complex; particularly when they are on data types that are defined recursively. For example, an ordered binary tree that represents a set of natural numbers can be defined as:

- has two (possibly nil) branches and a number at each node;
- all the numbers in the left branch of a node are less than the number in the node;
- all the numbers in the right branch of a node are greater than the number in the node.

The first line can be modelled by the following VDM data type [37]:

$Setrep = [Node]$

$Node :: lt : Setrep$

$v : \mathbf{N}$

$rt : Setrep$

The second and third lines express an invariant property. If *retrns* is a function which retrieves the set of numbers in a tree and is defined by:

$retrns : Setrep \rightarrow \mathbf{N\text{-}set}$

$retrns(bt) \triangleq$

cases bt **of**

nil $\rightarrow \{\}$

$mk\text{-}Node(lt, v, rt) \rightarrow retrns(lt) \cup retrns(rt) \cup \{v\}$

end

then we can define an invariant for the *Setrep* data type as:

$inv(mk\text{-}Node(lt, n, rt)) \triangleq$

$\forall ln \in retrns(lt) \cdot ln < n \wedge \forall rn \in retrns(rt) \cdot rn > n$

The function *retrns* has to be recursive to apply to all elements of the tree. As the above example shows, the representation of recursive invariants can be very complex. In general it is not possible to generate an invariant from an English sentence. This work is not intended to resolve the problem of invariants completely. However, in this chapter, we show that in some cases, it is possible to provide an invariant as an English sentence and then automatically produce a VDM form of this invariant. To produce this

invariant we proceed in two stages:

1. First a general form of this invariant is produced;

2. The invariant is then specialised to the particular data type that is obtained.

The invariant is provided as an English sentence. The first stage is the translation of this English sentence into an LFL expression. This transformation is done in the manner described in chapter 2. The LFL expression is then translated into a first order logic (FOL) expression. The latter transformation is done in two stages because LFL differs from first order logic. These differences are:

1. First FOL uses only two primary quantifiers, the *universal quantifier* and the *existential quantifier*. The existential quantifier also has a variant called the *unique existential quantifier*. LFL uses many quantifiers. A correspondence between LFL quantifiers and FOL quantifiers is therefore needed.

2. The second difference occurs in some relations introduced by some adjectives that we will call *relational adjectives*. Examples of such adjectives are *adjacent* and *different*. The action of adjacency or difference involves at least two elements. These relations are represented as unary relations in LFL. In FOL these elements need to be explicitly quantified.

The following sections describe the different stages needed for the production of the invariant.

5.2 The Production of the General Form

The first task in producing invariants is to identify the quantifiers. The quantifiers are obtained from the LFL expressions. As shown in chapter 2, LFL is a powerful tool for quantifying the different entities occuring in natural language sentences. In this section we start by analysing FOL quantifiers and then show how they can be obtained from LFL quantifiers.

5.2.1 Quantifiers in FOL

FOL quantifiers extend the power of the logical notation and are motivated as abbreviations. For example, the disjunction:

$$is\_prime(7) \vee is\_prime(8) \vee is\_prime(9)$$

can be written as:

$$\exists i \in \{7, 8, 9\} \cdot is\_prime(i)$$

This quantified expression can be read as:

> "There exists a value in the set {7,8,9} which satisfies the truth-valued function *is_prime*."

Vice versa, for finite sets, an existentially quantified expression can be expanded into a disjunction. Thus,

$$\exists i \in \{11, 12, 13\} \cdot is\_odd(i)$$

is true because:

$$is\_odd(11) \vee is\_odd(12) \vee is\_odd(13)$$

In the case of infinite sets, it is impossible to represent such disjunctions. The reason why quantifiers extend the expressive power of a logic is that the sets in the constraint of a quantified expression can be infinite.
Let us consider now the following disjunction:

$$is\_odd(11) \vee is\_odd(12) \vee is\_odd(14)$$

This is true and differs from the previous expression in that there is one and only one value in the set {11,12,14} that satisfies the function *is_odd*. When quantifying such expressions we use the unique existential quantifier $\exists!$. The above expression is then represented as:

$$\exists! \, i \in \{11, 12, 14\} \cdot is\_odd(i)$$

A conjunction such as:

$$is\_even(2) \wedge is\_even(4) \wedge is\_even(6)$$

can be written as a universally quantified expression:

$$\forall i \in \{2, 4, 6\} \cdot is\_even(i)$$

This expression can be read as:

> "All values in the set {2,4,6} satisfy the truth-valued function *is_even*."

5.2.2 The Transformation of LFL into FOL

As we have seen in chapter 4, the determiners *all*, *each* and *every* are always translated to the universal quantifier. The determiners *a* and *an*, are sometimes translated to the universal quantifier and sometimes to the existential quantifier. The determiner *the* when quantifying a plural noun, is translated to the existential quantifier and when quantifying a singular noun is translated to the unique existential quantifier. In the following subsection we present the transformation of LFL quantifiers into FOL quantifiers.

The Universal Quantifier

The standard universal quantifier in FOL has two operands.

$$\forall X \cdot P(X)$$

which specifies a single variable X to quantify over and the proposition P is supposed to hold for every value of X. The single form P is the scope of the quantifier. As defined in chapter 2, in LFL the scope of the quantifier is split into two parts, the base and the focus, so we write

$$all(P(X), Q(X))$$

which means that for all X: *if P holds than Q holds.*
Hence from this definition, we can translate the quantifier *all* to the following equivalent FOL expression:

$$\forall X \cdot P(X) \Rightarrow Q(X)$$

For example the following sentence:

Each man knows Bill.

has the following logical form representation:

all1(man(X),know(X,bill))

The translation of this LF into FOL is:

$$\forall X \cdot man(X) \Rightarrow know(X, bill)$$

After this first translation we need to transform the FOL expression into a typed FOL expression. Hence, if *man* is considered to be a type in the previous example, then $man(X)$ can be replaced by a constraint and the implication removed. The above FOL expression becomes the following typed FOL expression:

$$\forall X\colon man \cdot know(X, bill)$$

As another example, the following sentence:

Each overdraft account is closed

will have the following LF representation:

all1(account(X)&overdraft(X),close(X))

and will be translated into the following FOL expression:

$$\forall X \cdot account(X) \wedge overdraft(X) \;\Rightarrow\; close(X)$$

If *account* is used as a type for the variable X then the typed FOL expression is:

$$\forall X\colon account \cdot overdraft(X) \;\Rightarrow\; close(X)$$

As this example shows, in general the implication cannot always be removed.

The Existential Quantifier

In LFL the existential quantifier is represented by

$$ex(P(X), Q(X))$$

which means that there exists a variable X such that $P(X)$ *and* $Q(X)$ hold. Hence, we can translate such an LFL expression to the following FOL expression:

$$\exists X \cdot P(X) \wedge Q(X)$$

For example the sentence:

A company maintained a stock.

has the following representation in LFL:

ex(company(X),ex(stock(Y),maintain(X,Y))

and the translation of the LFL into FOL is:

$$\exists X \cdot company(X) \wedge \exists Y \cdot stock(Y) \wedge maintain(X, Y)$$

If *company* and *stock* are used as types for the variables X and Y then the $\wedge$ symbols can be removed so that we obtain the following typed FOL expression:

$$\exists X : company \cdot \exists Y : stock \cdot maintain(X, Y)$$

Expressions containing the unique existential quantifier are translated in the same manner.

5.2.3 Nested Quantifiers

Logical form expressions can also result in quantifiers nested deep in an expression. For example consider the following sentences, their logical forms and their typed FOL expressions:

- Each new student passed every test.

  all1(student(X)&new(X), all(test(Y), pass(X,Y)))

  $\forall X : student \cdot new(X) \Rightarrow \forall Y : test \cdot pass(X, Y)$

- A small company maintained a simple stock.

  ex(company(X)&small(X),ex(stock(Y)&simple(Y),maintain(X,Y)))

  $\exists X\colon company \cdot small(X) \wedge \exists Y\colon stock \cdot simple(Y) \wedge maintain(X, Y)$

Although such expressions are well-formed formulae of predicate calculus, a better representation of each of these is:

$$\forall X\colon student,\ Y\colon test \cdot new(X) \ \Rightarrow\ pass(X, Y)$$

$$\exists X\colon company,\ Y\colon stock \cdot small(X) \wedge simple(Y) \wedge maintain(X, Y)$$

This representation is widely used and preferred because it is more readable.

Hence, we use the following rules to obtain the desired representations.

$$\boxed{\text{Uni\_Nested}}\ \frac{\forall x\colon X \cdot p(x) \ \Rightarrow\ \forall y\colon Y \cdot g(x, y)}{\forall x\colon X, y\colon Y \cdot p(X) \ \Rightarrow\ g(x, y)}$$

$$\boxed{\text{Exi\_Nested}}\ \frac{\exists x\colon X \cdot p(x) \wedge \exists y\colon Y \cdot g(x, y)}{\exists x\colon X, y\colon Y \cdot p(x) \wedge g(x, y)}$$

Invariants are defined on all elements of the data type. Hence we assume that an invariant is a total predicate. Therefore, in the following proofs we assume that p is a total predicate. The proof for Uni_Nested is given in figure 5.1 and the proof for Exi_Nested is given in figure 5.2.

We use the natural deduction style of proof presentation used in Jones [37]. In this style, hypothesis are prefixed with a from and conclusions are prefixed with an infer. Subproofs are presented in an indented fashion. Lines are numbered in dewy decimal manner. Proof lines are justified by giving the inference rule used on the right together with the line numbers

from $\forall x\colon X \cdot p(x) \;\Rightarrow\; \forall y\colon Y \cdot g(x,y)$		
1 **from** $a\colon X$		
1.1 $p(a) \;\Rightarrow\; \forall y\colon Y \cdot g(a,y)$		$\forall$-E(h,h1)
1.2 **from** $b\colon Y$		
1.2.1 $\delta(p(a))$		p total
1.2.2 **from** $p(a)$		
1.2.2.1 $\forall y\colon Y \cdot g(a,y)$		$\Rightarrow$-E(1.1,h1.2.2)
infer $g(a,b)$		$\forall$-E(1.2.2.1,h1.2)
infer $p(a) \;\Rightarrow\; g(a,b)$		$\Rightarrow$-I(1.2.2,1.2.1)
infer $\forall y\colon Y \cdot p(a) \;\Rightarrow\; g(a,y)$		$\forall$-I(1.2)
infer $\forall x\colon X, y\colon Y \cdot p(x) \;\Rightarrow\; g(x,y)$		$\forall$-I(1)

Figure 5.1: Proof of Uni_Nested

from $\exists x\colon X \cdot p(x) \wedge \exists y\colon Y \cdot g(x,y)$		
1 **from** $a\colon X, p(a) \wedge \exists y \in Y \cdot g(a,y)$		
1.1 $p(a)$		$\wedge$-E(h1)
1.2 $\exists y\colon Y \cdot g(a,y)$		$\wedge$-E(h1)
1.3 **from** $b\colon Y, g(a,b)$		
1.3.1 $g(a,b)$		h1.3
1.3.2 $p(a) \wedge g(a,b)$		$\wedge$-I(1.1,1.3.1)
infer $\exists y\colon Y \cdot p(a) \wedge g(a,y)$		$\exists$-I(h1.3,1.3.2)
1.4 $\exists y\colon Y \cdot p(a) \wedge g(a,y)$		$\exists$-E(1.2,1.3)
infer $\exists x\colon X \cdot \exists y\colon Y \cdot p(x) \wedge g(x,y)$		$\exists$-I(h1,1.4)
infer $\exists x\colon X, y\colon Y \cdot p(x) \wedge g(x,y)$		$\exists$-E(h,1)

Figure 5.2: Proof of Exi_Nested

of the expressions to which the rule is applied. A line number with a prefix 'h' refers to the hypothesis of the line. The inference rules that we use are based on [37] and are given in appendix H.

5.2.4 The Transformation of Relational Adjectives

In this section we describe a pecular problem that occurs when translating relational adjectives. In this case we give the complete LFL representation that is produced (including the type of the variables). Consider the sentence:

All adjacent waypoints are different.

This results in the following LFL expression:

$$all(waypoint(X)\&adjacent(X\colon waypoint\_t),$$
$$be(X\colon waypoint\_t, different(X\colon waypoint\_t)))$$

Both $adjacent(X)$ and $different(X)$ are within the scope of the quantifier *all*. That is, all waypoints must have the given characteristics. In FOL this representation is not satisfactory since the relations of adjacency and difference are normally between two variables. An appropriate translation of the above LFL expression into a FOL expression would be:

$$\forall waypoint_1\colon waypoint\_t, waypoint_2\colon waypoint\_t \cdot$$
$$adjacent(waypoint_1, waypoint_2) \;\Rightarrow\; different(waypoint_1, waypoint_2)$$

Two different variables of the same type as the one defined in the LFL expression must be defined. These two variables will replace all the occurrences of the unique variable in the LFL expression. Whenever a relational adjective is encountered in LFL this kind of transformation is required.

The FOL expression represents the general case of the invariant. However, the invariants are defined for a particular problem, and therefore for

a particular data type modelling the problem. In the next section we will show how the invariant is specialised for a particular data type.

5.3 Specialisation of an Invariant

The general form of the invariant produced by the above process provides no context for its application. Since a VDM data type is required, our aim is to transform the general invariant so that it applies to a particular data type. The specialisation is performed in two stages. The first stage consists of the identification of the variables involved in the FOL expression. The second stage is to relate these variables to their occurrences in the definition of the data type. These variables are likely to match because the data type is defined from the specification text and the invariant is related to the text.

We will consider the data type defined for the stock problem in section 4.6.2 to illustrate these two points. The data type identified by our approach and system to model the stock problem is:

$$Stock\_t = Item\_ID \xrightarrow[\text{m}]{} Item\_t$$

$$\begin{array}{lll} Item\_t :: & description & : Description\_t \\ & unit\_cost & : Cost\_t \\ & quantity & : Quantity\_t \\ & minimum\_reorder\text{-}level & : Level\_t \end{array}$$

If we apply the following invariant to this data type:

The quantity in stock is greater than the minimum reorder level.

we obtain the following FOL expression which represents the general form of the invariant.

$$\forall quantity\text{:}\ quantity\_t \cdot \forall minimum\_reorder\_level\text{:}\ level\_t \cdot$$
$$is\_greater\_than(quantity, minimum\_reorder\_level)$$

The variables of this expression are: *quantity* with its associated type *quantity_t* and *minimum_reorder_level* with its type *level_t*. The next stage is to check if these variables can be associated with the defined data type. The matching is obtained either by using the types of the variables or their names if there are several variables of the same type.

The specialisation is carried out by first scanning the data types produced. We notice that the previous variables corresponds to the field selectors of the composite object of type *Item_t*. Hence, the general form of the invariant is applied to the set of composite objects of type *Item_t*. We define a variable of type *Item_t* and the invariant is now applied to the appropriate fields by using the field selectors. Hence, the form of the invariant becomes:

$$\forall item\text{:}\ Item\_t \cdot$$
$$is\_greater\_than(quantity(item), minimum\_reorder\_level(item))$$

The invariant is now restricted to the set of composite objects of type *Item_t*. Further, scanning shows that *Item_t* is used in defining the type *Stock_t*:

$$Stock\_t = Item\_ID \xrightarrow[\text{m}]{} Item\_t$$

The approach now aims to specialise the invariant so that it applies to *Stock_t*. Given a map *stock* of type *Stock_t*, we can obtain a composite object by applying the map to an index. Hence we can carry out the specialisation by replacing the composite object by $stock(i)$ in the above expression. In addition, the constraint part of the expression now quantifies over the domain of the map. The new form of the invariant is:

$$\forall i \in \mathbf{dom}\ stock \cdot$$
$$is\_greater\_than(quantity(stock(i)), minimum\_reorder\_level(stock(i)))$$

There is no data type containing the type *Stock_t*. We say that we have reached the top level of the data type, and we conclude that this is the final form of the invariant.

This example illustrates the transformation of a general invariant to an invariant that is applied to a map type. In general the invariant may have the following form:

$$\forall x\colon Xt, y\colon Yt \cdot p(x, y)$$

We can transform such invariants so that they apply to a map M of type: $Dom \xrightarrow[\text{m}]{} Ran$ where in general, *Ran* may be a composite object. Using the notation $s_i(c)$ to denote the selector s_i of the composite object c, we can present the transformation as the following rule:

$$\boxed{\text{SetToMap1}}\ \frac{\forall c\colon X \cdot p(s_1(c), s_2(c))}{\forall i \in \mathbf{dom}\ M \cdot p(s_1(M(i)), s_2(M(i)))}$$

In the case of sequences, given a sequence of composite objects S, the rule takes the form:

$$\text{SetToSeq1}\ \frac{\forall c{:}\, X \cdot p(s_1(c), s_2(c))}{\forall i \in \mathbf{inds}\, S \cdot p(s_1(S(i)), s_1(S(i)))}$$

Below we show that the transformations carried out are valid. The proofs are only produced for a sequence because the proofs for maps are similar. The rules need to be proved in both directions because we need to show that if an element is not a member of the considered set, than it cannot contribute to a relation with the sequence.

Given a composite object type D with selectors s_1 and s_2, we assume the following hypotheses:

1. $X{:}\, D\text{-}set$
2. $\forall x \in X \cdot p(s_1(x), s_2(x))$
3. $X'{:}\, D^*, X = \mathbf{elems}\, X'$

Substituting 3 in 2, we obtain:

$$\forall x \in \mathbf{elems}\, X' \cdot p(s_1(x), s_2(x))$$

Now proceeding from this gives the proof in figure 5.3 and figure 5.4 gives the proof in the other direction.

We have proved that the transformations carried out on the invariant are valid. Given simple English invariants, we can use these transformations to produce appropriate VDM invariants. We however, accept that there are invariants that are difficult to translate. For example the invariant described in section 5.1, is difficult to produce from a simple English sentence.

from $\forall x \in \textbf{elems}\, X' \cdot p(s_1(x), s_2(x))$		
1.	**from** $i \in \textbf{inds}\, X'$	
1.1	$X'(i) \in \textbf{elems}\, X'$	**elems**,h1
	infer $p(s_1(X'(i)), s_2(X'(i)))$	$\forall$-E(h,1.1)
infer $\forall i \in \textbf{inds}\, X' \cdot p(s_1(X'(i)), s_2(X'(i)))$		$\forall$-I(1)

Figure 5.3: Proof of SetToSeq1 (Top to Bottom)

from $\forall i \in \textbf{inds}\, X' \cdot p(s_1(X'(i)), s_2(X'(i)))$		
1.	**from** $x \in \textbf{elems}\, X'$	
1.1	$\exists k \in \textbf{inds}\, X' \cdot X'(k) = x$	**elems**,h1
1.2	**from** $j \in \textbf{inds}\, X', X'(j) = x$	
1.2.1	$p(s_1(X'(j)), s_2(X'(j)))$	$\forall$-E(h,h1.2)
	infer $p(s_1(x), s_2(x))$	=subs(h1.2,1.2.1)
	infer $p(s_1(x), s_2(x))$	$\exists$-E(1.2,1.1)
infer $\forall x \in \textbf{elems}\, X' \cdot p(s_1(x), s_2(x))$		$\forall$-I(1)

Figure 5.4: Proof of SetToSeq (Bottom to Top)

5.4 The Specification of Common Operations

The development of specifications in VDM can be very complex. It is not our intention to develop a tool that produces all possible specifications, this is beyond the current research. However, there are several specifications that are common across applications. These include the specification of operations that add items, delete items and list items that satisfy requirements. In this section we describe how such specifications are generated once the preceding step has identified the data types.

The general format of an operation specification in VDM is as follows:

OPER (*input*: *In_t*) *output*: *Out_t*

ext ...

pre ...*input*...

post ...*input*...*output*...

The first line, where *OPER* is the name of the operation, is called the signature of the operation. The signature is composed of the name of the operation, a list of input parameters and their types and a list of results and their types. The second line records those state variables to which an operation has external access. These state variables can be read only (rd) or read and write (wr) and the name of each variable is followed by its type. The pre-condition of an operation records assumptions about the arguments and state variables to which it is to be applied. The post-condition is an assertion that is required to hold after the operation is applied.

We can view this format as a template that needs to be filled to obtain a specification. In general, the template used is dependent on the operation required and the data type identified. Thus for adding an item to a map we provide a template which specifies that:

- there is one input argument (the item to be added), and one output argument (the identifier of the item added),
- a state variable with write access (the map),
- no precondition,
- a postcondition that records the requirement that the identifier of the

added item is new and that the map has been updated appropriately.

The information required for naming the arguments and the types are readily available as a result of the previous phase that identifies the types. Thus for the data type defined in section 5.3 and for convenience repeated here:

$$Stock\_t = Item\_ID \xrightarrow[\text{in}]{} Item\_t$$

$$\begin{array}{lllll} Item\_t & :: & description & : & Description\_t \\ & & unit\_cost & : & Cost\_t \\ & & quantity & : & Quantity\_t \\ & & reorder\text{-}level & : & Level\_t \end{array}$$

we describe how the required information is extracted knowing that we want to specify an operation that adds an item to a map.

- The name of the operation is obtained by the kind of function (add, delete, update), concatenated to the item on which the operation is carried out. In this example we want to specify an operation that adds an item to a stock map. Hence the operation is named: $ADD\text{-}ITEM$.

- There is only one input argument which should be of the same type as the range of the map. We use the first three letters of the name of the data type to generate a name for the argument.

- There is only one output argument and this should be of the same type as the domain of the map.

- There is a state variable with write access to the map (Stock).
- There is no precondition.
- As postcondition, we check that the identifier of the added item (the output argument) is not a member of the domain of the map and update the map by adding the new element (the input argument).

Hence, the template can be filled to obtain the following specification for adding an item into a stock system:

ADD-ITEM $(ite\colon Item\_t)\ r\colon Item\text{-}ID$

ext wr $stock\ :\ Stock\_t$

pre true

post $r \notin \mathbf{dom}\ \overleftarrow{stock} \wedge stock = \overleftarrow{stock} \cup \{r \mapsto ite\}$

where a hook over a state variable that has write access denotes the prior value of the variable. For example, $\overleftarrow{stock}$ denotes the map before the item is added.

Specifying the same operation for a sequence or a set does not differ too much from that for a map. The other category of common operations are updating and selection of elements. The specification of these operations differs from the previous ones in that the user has to supply extra information to specify the operations. For example to select a list of elements we need to know the condition for selection. This condition is supplied by the user as an English sentence which is translated into LFL. For the stock problem a condition such as:

Which items are to be reordered.

is first transformed to logical form:

wh(X&item(X),be(X,reorder(X)))

Then the condition $reorder(X)$ can be extracted from the focus of the logical form and used to construct the following post-condition.

$$r = \{iteide \mapsto stock(iteide) \mid iteide \in \mathbf{dom}\ stock \wedge reorder(stock(iteide))\}$$

As this example shows, there is no guarantee that the condition generated is fully defined. In this case the user has to define the function *reorder*.

If instead the user had typed:

The quantity in stock is less than or equal to the reorder level.

The system would have produced the logical form:

$$the(quantity(X, stock), the(reorder(level(Y)),$$
$$is\_less\_then\_or\_equal(X, Y)))$$

which leads to the post-condition:

$$r = \{iteide \mapsto stock(iteide) \mid iteide \in \mathbf{dom}\ stock \wedge$$
$$is\_less\_than\_or\_equal\_to(quantity(stock(iteide)), level(stock(iteide)))\}$$

Notice that in this example the user provided enough information for the system to generate a complete post-condition. In general, the template that generates the specification that lists specific items of a map specifies that:

- There is no input argument. The output argument is a map of the same type,

- a state variable with read access is used,
- there are no preconditions,
- a postcondition that ensures that the output variable consists of those items that belong to the map and which satisfy the given condition for selection.

Hence, the specification of the operation that lists the items of stock whose quantity in stock is less than or equal to the minimum reorder level is:

$LIST\text{-}ITEM \quad () \; r\text{:}\, Stock\_t$

ext rd $stock \; : \; Stock\_t$

pre true

post $r = \{iteide \mapsto stock(iteide) \mid iteide \in \mathbf{dom}\, stock \wedge$

$is\_less\_than\_or\_equal\_to(quantity(stock(iteide)), level(stock(iteide)))\}$

The template that generates the specifications of an operations that deletes an item from a map specifies that:

- There is one input argument which represents the identifier of the item to be removed from the map.
- A state variable with write access containing the map.
- A precondition that requires that the identifier of the item to be removed should be a member of the domain of the map.

- A postcondition that ensures that the new value of the state variable reflects the deletion made.

Hence the specification of an operation that removes an item from the stock is:

$DELETE\text{-}ITEM \quad (i\colon Item\_ID)$

ext wr $stock \ : \ Stock\_t$

pre $i \in \mathbf{dom}\, \overleftarrow{stock}$

post $stock = \{i\} \lhd \overleftarrow{stock}$

We also adopt a similar template based approach to obtain specifications for sequences. The specifications of operations for a sequence will be given in the case study presented in the next chapter.

6

A Case Study: A Flight Planning Data Base

6.1 Introduction

So far, the thesis has presented an approach based on natural language analysis that aims to:

1. Detect ambiguities and incompleteness in natural language requirements.

2. Produce an entity relationship diagram from the requirements.

3. Produce a VDM data type from the entity relationship model.

4. Produce formal specifications for some common operations.

The technique and approach developed have mainly been justified on theoretical grounds. In this chapter, we describe how the approach works in practice. The approach is implemented in Prolog-2 [23] and we use a case study to illustrate the kind of problem the approach is currently able to handle.

It would be easy to "cheat" and concoct a toy example for which our approach appears to work. Hence, we have used a case study that was written without prior knowledge of this work and was written independently from us. The case study was provided by the Department of Systems Computing of British Aerospace Ltd [30]. It concerns the planning of a route for a flight from one point to another. The route is planned as a number of waypoints. Each waypoint is identified by a number and a grid reference. Each grid reference contains the longitude and latitude of the waypoint. The original text is given at the end of this thesis.

It is worth repeating that the approach has the following phases:

1. Natural language analysis.

2. Identification of the entity relationship diagrams.

3. Development of the VDM data type.

4. Definition of invariants and the specification of operations.

In the next section we describe some pre-processing that is necessary. We then describe the manner in which the implemented system proceeds through each of these phases when it is presented with the case study.

6.2 Pre-Processing of the Specification Text

Bearing in mind that the current implementation does not handle conjunctions and pronoun references, we resolve these two problems manually. Each sentence containing a conjunction is decomposed into simple sentences and each pronoun reference is resolved before proceeding to the automatic analysis of the text.

For example, the following sentence of the original text:

As such the navigation system of a simple aircraft can be considered to comprise:

- the pilot's / navigator's map and route plan,
- a heading indicator,
- the pilot's / navigator's visual sense.

is decomposed into the following simple sentences:

1. The system of a simple aircraft can be considered to comprise the map of the pilot.
2. The system of a simple aircraft can be considered to comprise the plan of the pilot.
3. The system of a simple aircraft can be considered to comprise a heading indicator.

4. The system of a simple aircraft can be considered to comprise the visual sense of the pilot.

As another example, the following sentence:

> This information is used as input to a flight planning software package which calculates route tracks, distance between waypoints, heading for given wind conditions, non-violation of controlled airspace, etc.

is decomposed into:

1. The information is used as input to a flight planning software package.
2. The flight planning software package calculates the route tracks.
3. The flight planning software package calculates the distance between waypoints.
4. The flight planning software package calculates the heading for the wind conditions.
5. The flight planning software package calculates the non-violation of a controlled airspace.

Another problem that our system cannot deal with at the moment is the analysis of tables. In the current specification, some information is given in a tabular format. This table is replaced by text which describes the table. Once these transformations are done, the text is ready for analysis. The transformed text together with an indication of the transformations is given in appendix A.

6.3 Natural Language Analysis

The aim of the natural language analysis module is to detect sentences that are ambiguous or incomplete. These deficiencies are either detected during this phase or during the construction of the entity relationship diagrams. The text is analysed sentence by sentence. Some sentences may be ambiguous and therefore produce several interpretations. For example the following sentence:

The pilot draws the tracks of the route on the map.

produces three different logical forms which correspond to the following different meanings:

1. "The pilot draws (the tracks of the route) on the map". That is, the information is drawn on the map.

2. "The pilot draws (the tracks of the route on the map)". That is, the tracks are already on the map and the drawing is done somewhere else.

3. "The pilot draws the tracks of (the route on the map)". That is, the route is given on the map and its tracks are drawn somewhere.

The sentence:

An example route is planned for a flight from Blackpool to Doncaster

also produces three different logical forms which correspond to the three different interpretations of the string *P*-*NP*-*P*-*NP* described in section 2.3.3.

At the end of the analysis phase, each sentence is associated with a unique interpretation. That is, a unique logical form is associated with each English sentence. When several interpretations are produced for a sentence, the user is asked to remove the ambiguity by choosing the correct interpretation.

In this case study, out of the 41 sentences analysed:

- Twenty one are not ambiguous.
- Eleven sentences produced two interpretations because they contain a preposition.
- Nine sentences produced more than two interpretations.

Of the nine sentences that produced more than two interpretations, the system produces reasonable interpretations for seven of them. However, two of the sentences resulted in unexpected interpretations. These sentences are:

1. The planned tracks will assure the safe arrival of the aircraft over doncaster when they are flown in correct order by the aircraft.
2. The pilot chooses the waypoints from Blackpool to Doncaster in a complex aircraft.

The first sentence resulted in 32 interpretations and the second sentence resulted in 12 interpretations. If we analyse the first sentence, we find that it is composed of two conjuncts, three prepositional phrases and "by" which

is also interpreted as a preposition. The parser produces 32 different syntax trees and each syntax tree results in a logical form. The same scenario occurs in the second sentence where the presence of three prepositional phrases produces 12 syntax trees and each syntax tree is translated to a logical form. These two examples clearly demonstrate that the use of multiple prepositions can lead to sentences that are difficult to interpret by a natural language understanding system. Not surprisingly, such sentences are also difficult for a human reader. Indeed, style checkers, like the one employed by Grammatik discourage the use of multiple prepositions. In our approach we could encourage the user to reformulate those sentences that have more than five interpretations. The next phase concerns the identification of the entity relationship models.

6.4 Identification of the Entity Relationship Model

The identification of the entity relationship model is achieved in three stages. As a reminder, these three stages are:

1. Identification of the list of entities;
2. Identification of a list of relations involving the entities previously identified;
3. The different relations are combined to form the entity relationship

model.

Given the logical forms of the sentences, the system produces a list of entities. For the current case study, the system identifies 55 entities. The complete list of entities is given in appendix E. In general, this list may include "noisy" entities. Such noisy entities are detected and removed by the analyst by using problem dependent knowledge.

The next stage is the identification of the relations between the entities. In this case study, 52 relations are identified and the complete list of these relations is given in appendix F. These relations are binary relations. Some degrees are identified automatically by the system. For the present case study, the system successfully identifies the degrees of 37 relations. The remaining degrees are identified by the analyst. In general, during the identification of an entity relationship model, an entity may be related to many other entities. For example, in this case study the entity "System" of a simple aircraft is related to the entities: "Map", "Plan", "Heading Indicator" and "Visual Sense". By combining these relations we obtain the entity relationship diagram given in figure **??**. The complete entity relationship diagrams are shown in appendix G.

At this stage, the user may notice incomplete parts. For example, expected relations may not be identified by the system. In this case study the initial entity relationship model produced does not show a relation between the entity route and the entity waypoint. From the given text we understand that a route is composed of waypoints but there is no explicit sentence

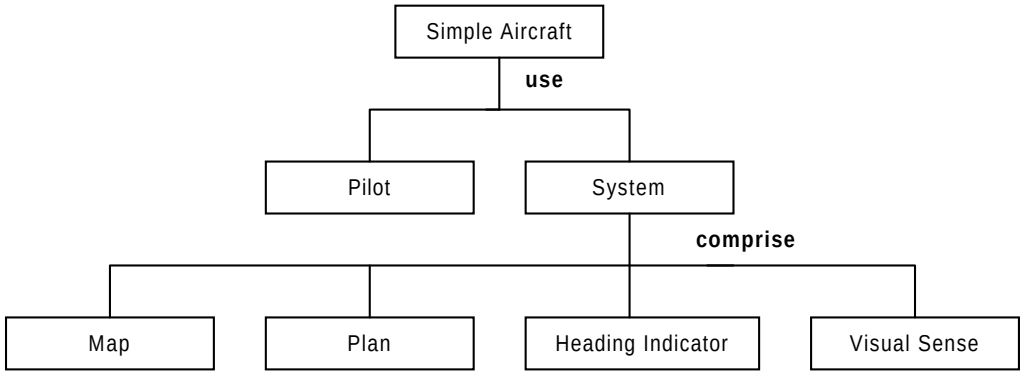

Figure 6.1: The ER Diagram of a simple aircraft

giving this relationship. Hence, we could add the following sentence:

The route is composed of waypoints

which would reduce the incompleteness of the specifications as well as the entity relationship model.

Another important observation is that the system produces an entity relationship model that is broad. Indeed as appendix G shows, there are several independent entity relationship models produced for the case study. We believe that this broad view, also encouraged by approaches like Multi-view [5], is an important benefit of our approach. For instance in this case study, the analyst has to consider which part of the system he wants to translate into formal specifications.

6.5 Identification of the VDM Data type and the Specification of Operations

In this case study suppose we are interested in modelling the route planning system. We need to select this diagram by giving the name of its root entity, "Route" in this case. The diagram is a one-to-many relationship between the entity "Route" and the entity "waypoint". This relation can be modelled either as a map from *waypoint_identifier* to *waypoint* or as a sequence of *waypoints*. Therefore there is a need to check if the order in which the waypoints are defined is important. Here, suppose that the user confirms that the order is important, then the "Route" is modelled as a sequence of waypoints and the following data type is produced:

$$Route\_t = Waypoint\_t^*$$

The system will now prompt the user to provide any invariant that is associated with the data type. Given the following sentence as an invariant:

All adjacent waypoints are different

The following invariant is produced:

$$\begin{aligned} &inv\text{-}Route\_t(route) \quad \triangleq \\ &\quad \forall i_1 \in \mathbf{inds}\ route \cdot \forall i_2 \in \mathbf{inds}\ route \cdot \\ &\qquad adjacent(i_1, i_2) \ \Rightarrow \ different(route(i_1), route(i_2)) \end{aligned}$$

The specification of the functions *adjacent* and *different* should be provided by the user. The last stage of the approach is the specification of the

operations. The templates defined result in the following specifications:

(a) Adding a Waypoint

ADD-WAYPOINT (*way*: *Waypoint_t*)

ext wr *route* : *Route_t*

pre true

post $route = \overleftarrow{route} \frown [way]$

(b) Deleting a Waypoint

$$
\begin{array}{l}
\mathit{DELETE\text{-}WAYPOINT} \quad (i\colon \mathbf{N}) \\
\textbf{ext wr } \mathit{route} \ : \ \mathit{Route\_t} \\
\textbf{pre } i \in \textbf{inds}\,(\overleftarrow{\mathit{route}}) \\
\textbf{post } \exists \mathit{route}1\colon \mathit{Route\_t} \cdot \exists \mathit{route}2\colon \mathit{Route\_t} \cdot \\
\qquad\qquad \overleftarrow{\mathit{route}} = \mathit{route}1 \frown [\mathit{route}(i)] \frown \mathit{route}2 \wedge \\
\qquad\quad \mathit{route} = \mathit{route}1 \frown \mathit{route}2
\end{array}
$$

(c) Updating a Waypoint

$$
\begin{array}{l}
\mathit{UPDATE\text{-}WAYPOINT} \quad (\mathit{way}\colon \mathit{Waypoint\_t}, i\colon \mathbf{N}) \\
\textbf{ext wr } \mathit{route} \ : \ \mathit{Route\_t} \\
\textbf{pre } i \in \textbf{inds}\,\overleftarrow{\mathit{route}} \\
\textbf{post } \mathit{route}(i) = \mathit{way} \wedge \forall j \in \textbf{inds}\,(\overleftarrow{\mathit{route}}) \cdot i \neq j \\
\qquad \Rightarrow \ \mathit{route}(j) = \overleftarrow{\mathit{route}}(j)
\end{array}
$$

6.6 Summary

The case study shows the kind of problems that the current implementation of the system can handle. The system has helped to:

- Identify sentences that are ambiguous. Out of 41 sentences, 20 sentences are considered to be ambiguous by the system. All sentences but two have produced expected interpretations.

- Identify incomplete aspects of the original text. In this case study only one incompleteness was identified, but in other examples many others may be present.

- Produce a list of entities and a list of relations of the case study. Fifty-five entities and 52 relations were identified.

- Identify the degrees of relations. The degrees of 37 relations were identified by the system. The user helped to identify the remaining.

- Associate an appropriate VDM data type with the route entity relationship diagram.

- Translate an invariant written in English into an acceptable data type invariant.

- Produce the specifications of the some operations.

In summary, the results obtained for the case study are encouraging and most of the objectives set for the current system have been illustrated by the case study.

7

Related Research

7.1 Introduction

A number of systems have been developed in order to improve and give a rational method for requirements engineering. In this chapter, we describe several approaches to requirements engineering and contrast them with our approach. We first begin by providing the context in which these approaches are based. Then we define the characteristics of the requirements engineering process that will form the ground for comparing the different approaches. The requirements engineering process is composed of three distinct activities that are [56]:

- requirements elicitation and capture,
- agreement,

- requirements representation.

The elicitation and capturing phase concerns the identification of all parties that are sources of requirements, gathering a "wish list" of each party and the documentation and refinement of the wish lists. The integration of the different wish lists may raise some conflicts. In the agreement phase, one attempts to remove the conflicts. In practice, an absolute agreement is rarely reached, and one talks about the degree of agreement that may be reached. At the beginning of the requirements process, each person involved has his own view of the system. Some requirements may be shared among the team, but others may not be shared. The aim of the agreement phase becomes one of increasing the common system requirements and to find a compromise for the remaining requirements so that a consistent set of requirements is identified. Even if a consistent set of requirements is not identified, this process aids our understandings of the problem. The third activity is the representation of the requirements. There are three categories of representations: informal representations such as arbitrary graphics and natural language, semi-formal languages such as SADT, JSD and ER diagrams and formal languages such as VDM and Z.

Most of the systems for requirements engineering concentrate on the representation phase. Some systems take the form of languages developed to represent the requirements. Examples of these systems are the PSL/PSA system [66], the structured analysis language (SA) [59] and the SREM project [1, 10].

Another category of systems attempt to automatically analyse natural language requirements. Among these systems are the SAFE project [7, 6], the SPAN system [57], Williams' work [70] and Comer's work [19].

Some other systems aim to transform semi-formal representations to formal ones [**?**, 25, 63]. Most recent systems use artificial intelligence techniques such as knowledge based systems to analyse requirements [13, 42, 36, 58].

Thus, there are several different approaches to requirements engineering, each addressing a different aspect of the requirements engineering problem. To give a reasonably broad view of the approaches, in the next section we summarise the following four systems:

- The PSL/PSA system.
- The SAFE project.
- The SPAN project.
- The Analyst Assist system.

Thus, the PSL/PSA system addresses issues of documentation and the circulation of information during the requirements phase. The SAFE project was among the first systems to suggest the use of a computer tool for the transformation of informal requirements into formal ones. The SPAN system deals with the analysis of natural language requirements documents. Finally, the Analyst Assist system raises issues of requirements elicitation.

In the last section we compare and contrast these four systems with the approach proposed in this thesis.

7.2 The PSL/PSA System

When developing a system, it is necessary to record the changes made as it evolves through the different phases. Teichroew [66] argued that in practice the emphasis in documentation is on describing the final form of the system to help its maintenance instead of documenting each phase. Teichroew stated that this deficiency occurs because the communications from one activity to another are accomplished either by the same person, by oral communication between the project developers or by notes which are discarded after their use. The Problem Statement Language/Problem Statement Analyser (PSL/PSA) system developed by Teichroew and Hearshey [66] therefore has the following aims:

1. The results of each activity in the system development process are recorded in a computer processable format as they are produced.

2. A computerised data base is used to maintain all the basic data about the system.

3. The computer is used to produce hard copy documentation when required.

PSL is a text language which is used for describing processes, data flow diagrams and the hierarchical decomposition of processes and data. PSL is based on a model of a general system. This model can then be specialised for a particular class of systems. The model of a general system states that a system consists of things which are called *objects* (not to be confused with the objects in Object Oriented Design). These objects may have *properties* and each of these properties may have *property values*. The objects may be connected and the connections are called *relationships*. The general model is specialised for an information system by allowing the use of only a limited number of predefined objects, properties and relationships.

The objective of PSL is to be able to express the information which commonly appears in system definition reports in a syntactically analysable form. The PSL statements are input into PSA which store all the statements in an internal form in a data base. The stored requirements can be modified, updated and deleted. PSA uses the stored information to generate a variety of reports and documents including:

- *Data dictionaries.*
- *Data flow diagrams.*
- *Data flow inconsistency reports.* These concern the information generated but not used or information used but not generated.
- *Data base modification reports.* These constitute a record of changes that have been made.

- *Reference reports.* These present the information in the data base in various formats. The reference report gives the summaries of all processes referencing an item of information.

- *Analysis reports.* These present a summary of information.

The PSL/PSA system has received a lot of attention and has undertaken further development. PSL/PSA was also used in more then 140 projects including numerous large systems [21]. Davis [21] reported that unfortunately the PSL/PSA system was often used for only design, mainly the decomposition of software into its actual subcomponents, rather than for requirements analysis. PSL/PSA provides no help with requirements elicitation or with the transformation of informal requirements to formal representations.

7.3 The SAFE Project

The SAFE project was one of the earliest systems to suggest the idea of analysing requirements written in natural language. The goal of the system was to create a formal operational specification from the informal input. The input text was manually transformed into a parenthesized form to avoid syntactic parsing problems.

To generate an operational specification the SAFE system first needs to identify a complete specification. In general, the sentences provided by

a user will be incomplete and will only contain part of the information required to construct an operational specification.

To obtain a complete specification, SAFE generates all possible completions for each sentence. This is done by using the information in the parsed sentences to identify additional information from a predefined domain model. In general this can result in several possible complete versions for each sentence. SAFE then associates one completion with each sentence so that it is able to generate the 'complete' specification. This is done by using a depth first searching strategy. That is, the first completion of the first sentence is considered; then the first completion of the second sentence is brought into consideration. If these are not consistent, then the second completion of the second sentence is considered and so on. If none of the completions of the second sentence are consistent with the first completion of the first sentence then SAFE backtracks and considers the second completion of the first sentence. This process continues until it manages to find consistent completions for all sentences.

The completions of the sentences are then used to produce specifications in a language called AP2. The AP2 language, which is developed by the authors, is based on structured English and has loops and conditions. Specifications written in the AP2 language behave as programs and are executable.

The successful use of the SAFE system is dependent on the predefined domain model and the process of obtaining the completions. In general, the completion process, can be computationally expensive since it employs a

blind search strategy. It is therefore not surprising that the largest specification analysed by SAFE consists of twelve sentences. Further, the authors do not give details of:

- how partial descriptions were detected,
- how the decisions to complete the partial descriptions are taken,
- the format of the knowledge base for the domain model,
- how the knowledge base is updated by the informal specifications,
- how it handles contradictory sentences.

In summary the SAFE project has not considered the problem of requirements elicitation but has tackled the problem of interpreting natural language requirements with an attempt to resolve partial descriptions and presented a tool to help the production of specifications from natural language specifications.

7.4 The SPAN System

The SPAN system was developed at the University of Liverpool by Mander and was reported in Presland's work [57]. This system was aimed at analysing English sentences provided by a user. The system is composed of three parts: a dictionary, a preparation program and an analysis program.

The dictionary contains the definitions of words. The preparation program is used to check if the root of a word is contained in the dictionary. The system is intended to minimise the number of words stored in the dictionary, so only roots of words are used as entries in the dictionary. Once the specifications are prepared, the analysis program will check if the sentences are grammatically correct with respect to the grammar developed and attempts to find ambiguities.

The primary mechanism for detecting ambiguities is to use specific rules based on experience. For example, one rule states that each sentence containing a preposition is ambiguous.

This research brought two new features compared to previous attempts. First, it used free format English text as input and produced the analysis without any context knowledge. This makes it a general approach and if the dictionary is complete enough, there is nothing to add if a new problem is tackled. Second, the SPAN system aimed to detect system structure by identifying the actions and entities present in a specification.

The approach used by SPAN is limited to the analysis of natural language requirements. The result of the analysis is the production of case frames for requirements documents. The approach has not considered the production of formal specification from the case frames and has not considered the problem of requirements elicitation.

7.5 The Analyst Assist

The Analyst Assist (AA) system developed by Loucoupoulos and Champion [41, 42] is aimed to:

- capture informal requirements and improve the transition from informal requirements capture to a semi-formal representation.
- specify and document requirements using the JSD method.
- validate the specification by prototyping and animating the specification.

The authors argue that knowledge of the application domain is very important to build the specification of a system. The analyst has to express clearly the domain's properties and constraints that, with some non-functional requirements will form the complete requirements specification. The approach developed is based on the behaviour of an experienced analyst. An experienced analyst is able to make analogical reasoning, manage the hypothesis, plan, set goals and strategies [67]. It has been proposed that these skills are supported by some knowledge bases which fall under the following categories:

- Organisation knowledge.
- Functional domain knowledge.
- Application domain knowledge.

- Development method knowledge.

The AA system is composed of a domain knowledge base, a user fact base, a JSD knowledge base and a JSD specification base. The user fact base is used to store concepts of the application before they are translated into a JSD specification. This data base is populated using a fact input tool. This tool is guided by an elicitation dialogue formulator which makes use of the domain base and the current state of the user fact base. The JSD specification base holds an evolving system specification represented in JSD. A JSD method advisor acts as an assistant to the translation of the concepts to a JSD specification and provides consistency checking on the evolving JSD specification.

Requirements elicitation in AA is based on the premise that an analyst should be able to define anything of the modelled application domain as a potential concept of interest and allow an analyst to reason with all information that is perceived to be relevant so that a system specification can be constructed.

AA provides three tools for fact gathering, a text analyser, a hierarchy editor and the elicitation dialogue formulator. The text analyser allows the user to highlight keywords and phrases from a document. Single words are stored in the user fact base as concepts and the phrases are used to record relations between concepts. These concepts include those identified by the analyst and those known from the system from the domain knowledge base. The concepts known to the system are described in terms of their

relationships with other concepts and are arranged in a concept hierarchy. The hierarchy editor allows an analyst to classify concepts by placing them directly in the hierarchy, thus by-passing the text editor. In situations where more information about a concept is needed, and cannot be obtained by using the previous tools, the elicitation dialogue formulator asks the user some questions to elicit the information.

The AA system has considered the problem of requirements elicitation. However, the AA system has not considered the analysis of informal requirements and the detection of ambiguities or the development of formal specifications.

7.6 Comparison and Contrast

So how do these four systems compare with our approach? First it should be clear that they all emphasise different aspects of the requirements process and tackle different problems. Bearing this in mind, we contrast the approaches by placing their contribution in the overall context of the requirements engineering process. That is, we place the contributions in the context of: requirements elicitation and capture, agreement, and requirements representation.

Requirements Elicitation

The Analyst Assist system brought a significant contribution to the requirements elicitation activity by using a domain knowledge base. This knowledge base guides the analyst through the requirements process prompting him with questions that will give important information and also helping him to obtain a better understanding of the organisation. Among the four systems we have analysed, AA is the only system which significantly contributes to this activity.

The primary contribution of the PSL/PSA system is its ability to keep track of the history of changes. Although this contribution may appear minor, in our opinion this is a very important aspect of requirements engineering and was a significant contribution given today's quality assurance standards [62].

Our own approach is not aimed primarily at the elicitation stage and has not duplicated these existing contributions. However, using our approach can help an analyst gain a better understanding of the overall system and as illustrated in the case study, can help to identify incompleteness.

Agreement

The aim of the agreement activity is to reach a consistent, unambiguous specification that is agreed by all parties. None of the reviewed systems appear to tackle this aspect of requirements engineering. Apart from prototyping, the subproblem of reaching agreement between the informal requirements and the semi-formal requirements or formal requirements have only been partially addressed by the SAFE system.

The primary contribution of SAFE was to suggest the possibility of analysing natural language text. However it had several deficiencies including:

- the absence of a syntax analyser,
- the method adopted to resolve partial descriptions is inadequate for large specifications.

The SPAN system also aimed to identify ambiguities in natural language specifications. Unlike SAFE, it did attempt to automatically parse sentences. However, it detects ambiguities by using specific rules based on experience. These rules are based on the syntax of the sentences and do not consider the meaning of the sentences.

Our approach improves upon these two systems. It is a more general way of detecting ambiguities. Unlike SPAN, we do not use specific rules but instead produce different logical forms for ambiguous sentences. This enables an analyst to resolve any ambiguities at the semantic level. Our

approach also produces entity relationship models and formal specifications which may be useful to an analyst.

Requirements Representation

The last activity of the requirements phase is the transformation of informal requirements to formal or semi-formal representations. The PSL/PSA system and the SPAN systems do not consider this issue.

The AA system transforms the informal requirements into a semi-formal specification represented in JSD. A text analyser allows an analyst to highlight keywords and phrases which may form the concepts and relations between the concepts. A JSD method advisor helps to produce a JSD specification from the concepts and the relationships identified.

Our approach proceeds in two stage. First it produces an entity relationships model by detecting the entities and relationships between them. Second the entity relationship model is translated into a VDM data type. Our approach is the only one to produce formal data types. Our approach also produces the specifications of some common operations in VDM. We view the specification of operations as templates that need to be filled. The information used to fill the templates is obtained from the definition of the data types and the nature of the operations.

The SAFE project also tried to produce specifications from English ones.

The language defined by the authors (AP2) results in executable specifications, and is not based on first order logic or any other mathematical formalism. It is a language which combines structured English with the characteristics of a programming language. The specification language used is therefore not like today's formal specification languages. In contrast, our approach generates specifications in VDM.

8

Conclusion and Future Work

8.1 Conclusion

Requirements specifications provide the foundation upon which a system should be developed. Failure to provide a specification that accurately reflects the requirements of a system will result in the development of the wrong system. Further, errors such as ambiguities, that occurr during the requirements phase are the most expensive to correct. The requirements analysis phase also involves groups of people with different backgrounds. Communication between these groups may be difficult and misunderstandings and ambiguities may arise. Different representations are preferred by both the users and the developers. Therefore much attention should be paid to the requirements phase in order to reduce the number of errors and

to ensure a correct transition from one representation to another. This research aims to support some activities carried out during the requirements specification phase. In particular, the objectives of the approach taken in this thesis are:

- To identify ambiguities in natural language documents.
- To produce entity relationship models from the analysed natural language specifications.
- To produce VDM data types together with their invariants.
- To produce some common specification in the VDM language.

Our approach utilises the results achieved in natural language understanding to analyse requirements written in natural language. We use McCord's approach to natural language processing. This approach proceeds in two phases. First, it produces an analysis based on the grammar rules of the language, and then uses a semantic analysis phase to produce statements in a meaning representation language called logical form language. We have successfully used this approach to detect ambiguities present in English text. An ambiguous sentence results in several interpretations and logical forms. An analyst is therefore made aware of ambiguous sentences and given the alternative possible interpretations. To utilise McCord's approach we needed to improve its interpretation of the definite articles and the indefinite articles. Our improvement enables these articles to be translated to more appropriate quantifiers.

The result of this analysis forms the basis for the development of the remaining parts of our approach. An entity relationship model is first produced which is then translated to a VDM data type. The first step in producing an entity relationship model is to identify a list of entities and the relationships between them. We base our identification process on the view that entities are described by nouns and relations by verbs. Our approach identifies a list of entities with their relations. The verbs are used as names for the relations. The quantifiers associated with the nouns help to determine the degrees of the relations. The list of relationships is filtered by an analyst to remove noisy entities. The analyst also has to provide the degrees for relations if they are not successfully identified by the system. The combination of the entities and relationships form the entity relationship model.

Our approach then translates the entity relationship model into a VDM data type. If the system has identified one-to-one or many-to-many relationships, then we require the simplifications of these relationships before we proceed to their translation into a VDM data type. One-to-many relations can be modelled by representing the many part by a set, a sequence, or a map. The last representation is preferred by many authors, so by default our system selects the map representation. However, if the analyst specifies that the order of the elements is important then the sequence representation is adopted. Many-to-one relationships are modelled as maps.

The definition of VDM data types is usually followed by the definition of their invariants. We have developed an approach that generates some

invariants which are provided as English sentences. A general form is produced as a first order logic expression. The system then specialises the general form so that it applies to the specified data type.

Once invariants are defined, our approach generates the specification of some common operations. The approach views the specification of an operation in VDM as a template that needs to be filled to obtain a specification. The template used is dependent on the operation required and the data types identified. Hence, a template used to generate an operation to add an item into a map differs from a template that generates the removal of an item from a sequence.

To test our approach, we have implemented it in Prolog-2 and tested it on a case study. The case study was written without prior knowledge of this work and was written independently from us. The case study was provided by British Aerospace Ltd and concerns the planning of a route for a flight. The case study brought out the kind of problem that the current implementation of the system can handle. The system helps to:

- Identify sentences that are ambiguous.
- Identify incomplete aspects of the original text.
- Produce a list of entities and relationships that gives a broad view.
- Identify the degrees of relations.
- Associate an appropriate VDM data type to the route entity relationship diagram.

- Translate an invariant written in English into an acceptable VDM data type invariant.
- Produce the specifications of some operations.

We have contrasted our approach with other contributions to requirements analysis. Most systems have contributed to the representation and tracking of requirements. Such systems enable an analyst to develop a specification in a consistent manner with the aid of existing (predefined) knowledge. In contrast, our contribution has been to provide an approach that could take written requirements as input and produce formal specifications in VDM.

In comparison to other systems that also take natural language text as input, we have developed a more general approach and have utilised natural language analysis techniques better. For example in comparison to Presland's work:

1. The production of the different interpretations of an ambiguous sentence using logical forms is a more general process. In Presland's work specific rules are defined to detect ambiguities at the syntactic level.
2. The use of the quantifiers helps to determine the degrees of relations.

In addition, Presland's work does not produce formal data types or tackle the problem of invariants.

To conclude, the approach proposed in this thesis is a significant contribution to research in the area of requirements analysis. It contributes

to two main activities. First, it helps to improve requirements and helps to identify ambiguities. Second our approach helps the transition from one specification representation to another. The first transition is from informal requirements to semi-formal ones (English to ER diagrams), the second from semi-formal representations to formal ones (ER diagrams to VDM specification). Based on our initial case studies, the approach proposed is very promising. However, to experiment with a broader range of specifications our implementation needs to be improved. In the next section we suggest some improvement to the current implementation.

8.2 Future Work

Several aspects of the current implementation of the approach can be improved:

1. To handle a wider range of sentences the natural language analyser should be improved so that it can handle pronoun identification and conjunction properly. Dahl's system for treating coordinations [20] and Paskiewicz's work on anaphoric pronouns [50] may provide a basis for solving these problems.

2. The resolution of conjunctions in the natural language analyser will also help with the production of exclusive relationships in the entity

relationship diagrams.

3. Our system needs to improve in order to handle a wider range of specifications. For example, specifications involving time have not been addressed by the approach presented in this thesis.

A

The Aircraft Problem

In this appendix, we present the input text to the natural language analyser. A key is associated to some sentences to show how they are obtained from the original text. The explanation of these keys is:

Ci : All the sentences referred by Ci are the result of the split of the the same conjunct.

T : The sentence is added to replace a table.

PR : A pronoun reference is encountered and resolved.

The text is:

```
1- An example route is planned for a flight from blackpool to
 doncaster.
2- The route is planned as a number of the discrete tracks
   between the intermediate waypoints.
3- The planned tracks will assure the safe arrival of the
```

aircraft over doncaster when they are flown in correct order by the aircraft.

4- The pilot may have unnecessarily flown through a storm.(C1)

5- The pilot may have unnecessarily flown through a controlled airspace.(C1)

6- The aircraft may hit an obstacle. (C2)

7- The aircraft may hit another aircraft.(C2)

8- The pilot of a simple aircraft without a sophisticated electronic navigation system would be cleared to undertake a risky flight.

9- The pilot chooses the visible waypoints from the air.

10- The pilot draws the tracks of the route on the map.

11- The pilot steers a heading giving the required tracks along the ground.

12- The pilot scans the ground for the visible features.(PR)

13- The pilot verifies the visible features against the map.

14- The system of a simple aircraft can be considered to comprise the map of the pilot.(C3)

15- The system of a simple aircraft can be considered to comprise the plan of the pilot.(C3)

16- The system of a simple aircraft can be considered to comprise a heading indicator.(C3)

17- The system of a simple aircraft can be considered to comprise the visual sense of the pilot.(C3)

18- The system of a simple aircraft contrasts with the system of a complex aircraft.

19- A complex aircraft uses a whole range of the electronic equipment to support the navigation.

20- A complex aircraft uses a computer-assisted flight planning.(C4)

21- A complex aircraft uses an inertial navigation system.(C4)

22- A complex aircraft uses a radar.(C4)

23- A complex aircraft uses a moving map display.(C4)

24- A complex aircraft uses a route display.(C4)

25- A complex aircraft uses a waypoint display.(C4)

26- A complex aircraft uses an autopilot.(C4)

27- The pilot chooses the waypoints from blackpool to doncaster in a complex aircraft.

28- The pilot identifies each waypoint with a number. (C5)

29- The pilot identifies each waypoint with a grid reference.(C5)

30- The grid reference contains the latitude.(T)

31- The grid reference contains the longitude.(T)

32- The information is used as input to a flight planning software package.

33- The flight planning software package calculates the route tracks.(C6)

34- The flight planning software package calculates the distance between waypoints.(C6)

35- The flight planning software package calculates the heading for the wind conditions.(C6)

36- The flight planning software package calculates the non-violation of a controlled airspace. (C6)

37- The derived information may be listed for the pilot to record on the map.(C7)

38- The derived information may be transferred to a cassette tape.(C7)

39- The cassette tape is used to load the navigation database on the aircraft.

40- The autopilot uses the data to fly the aircraft according to the plan of the pilot.

41- The route is composed of waypoints.(T)

B

The English Grammar

In this appendix, we present the implementation of the syntax analyser. In the first section, we describe the syntax analyser module. The syntax module is composed of six parts which are:

1. Definition of strong terminals.
2. Clause rules.
3. General rules of postmodifiers.
4. Verb modifiers.
5. Noun phrase rules.
6. Noun modifiers.

In the following subsections, we list each of these rules.

B.1 Definition of Strong Nonterminals

```
/*   definition of strong nonterminals */


strongterminal(s).
strongterminal(np).
strongterminal(det).
strongterminal(relclause1).
strongterminal(compclause1).
strongterminal(verbph).
strongterminal(avp).
strongterminal(pp).
strongterminal(subclause1).
strongterminal(sentcomp).
```

B.2 Clause Rules

This subsections contains the general rules of the English grammar used for the current implementation. In general, a grammar rule has the following form:

```
rule_head --> rule_body1:rule_body2.
```

The arrow operator is used to separate a *rule head* from a *rule body*. The colon operator is used to represent the left-to-right sequencing, and it can be read as "followed by". The mlgtrans module described below will transform these grammar rules into Prolog clauses. In general each rule of the grammar will be augmented by a number of parameters when translated to Prolog clauses. A term preceded by the + (which always represent a call to a terminal) means that the terminal should be removed from the list representing the sentence. A term preceded by the % symbol means that the clause is not augmented by parameters when translated to a Prolog clause. The complete list of rules used for the current implementation together with the module mlgtrans are listed below.

```
\*                          *\
\*    mlgtrans module       *\
\*                          *\

\* The general procedure  gtrans will transform a grammar rule
   into a Prolog term. The procedure gtrans will first transform
   the head of the grammar rule by calling the procedure
   mlgtransh, than it will call the procedure mlgtransb which
   transforms the body of the grammar rule.                    *\

gtrans(A-->B,Clause) :- !,
    mlgtransh(A,Head,Syn0,Syn,Mods1,Mods2,X,Y),
```

```
    mlgtransb(B,Body,Syn0,Syn,Mods1,Mods2,X,Y).

gtrans(Clause,Clause).

\* transforms the head of a grammar rule                    *\

mlgtransh(A,Head,syn(Pred:Arg1,Mods),Syn,Mods,[],X,Y) :-
  cons([Pred|Args],A) ,
  strongterminal(Pred) ,!,
  (Args=[Arg1|_],! ; Arg1=[]) ,
  append(Args,[Syn,X,Y],Args1) ,
  cons([Pred|Args1],Head).

mlgtransh(A,Head,Syn0,Syn,Mods1,Mods2,X,Y) :-
  cons([Pred|Args],A) ,
  append(Args,[Syn0,Syn,Mods1,Mods2,X,Y],Args1) ,
  cons([Pred|Args1],Head).

\* different rules to transform the body of a grammar rule *\

mlgtransb(B1:B2,(C1,C2),S0,S2,M,N,X,Y) :- ! ,
  mlgtransb(B1,C1,S0,S1,M,P,X,Z) ,
  mlgtransb(B2,C2,S1,S2,P,N,Z,Y).
mlgtransb(+T,true,S,S,M,M,[T|X],X) :- !.
```

```
mlgtransb(Op-LF,true,S,S,[(Op-LF)|M],M,X,X) :- !.

mlgtransb([],true,S,S,M,M,X,X) :- !.

mlgtransb(%P,P,S,S,M,M,X,X) :- !.

mlgtransb(!,!,X,X,X,X,X,X) :- !.

mlgtransb(NT,NT1,S,S,[Syn|M],M,X,Y) :-
  cons([Pred|Args],NT) ,
  strongterminal(Pred) ,!,
  append(Args,[Syn,X,Y],Args1) ,
  cons([Pred|Args1],NT1).

mlgtransb(NT,NT1,Syn0,Syn1,M,N,X,Y) :-
  cons([Pred|Args],NT) ,
  append(Args,[Syn0,Syn1,M,N,X,Y],Args1) ,
  cons([Pred|Args1],NT1).


mlgtransb(Lab^NT,NT1,Syn0,Syn,M,M,X,Y) :-
  cons([Pred|Args],NT) ,
  Syn0 = syn(Lab0,Mods0),
  ShiftSyn = syn(Lab0,[syn(Lab,Mods)|Mods]),
  append(Args,[ShiftSyn,Syn,Mods,[],X,Y],Args1) ,
  cons([Pred|Args1],NT1).


cons([Term],Term) :- atomic(Term),! .

cons(List,Term) :-  Term=..List.
```

```
append([],X,X).
append([X|Y],Z,[X|Z1]) :- append(Y,Z,Z1).

/*                                    */
/*        Clause   rules              */
/*                                    */

/* A sentence does not  always  start by a noun phrase.
   To deal with the different beginnings of a  sentence
   a general  procedure called topic is defined.  The
   procedure clause1 deals with the verb phrase of the
   sentence.                                          *\
s -->   topic(Topsubj,Qaux,Topic):
      clause1(Topsubj,Qaux,Topic).

topic(Topsubj,Qaux,hold(X)) --> np(X):
                                topic1(Topsubj,Qaux,X).

topic(Topsubj,Qaux,hold(X)) --> there(X):
                                topic1(Topsubj,Qaux,X).

topic(Topsubj,Qaux,[])      --> adverbial:
                                topic1(Topsubj,Qaux,Top).
```

```
topic(t,pre(V),hold(X))      --> +V:
                                 %finiteaux(V):
                                 np(X):
                                 ( B << F)-yesno(B,F).

topic1(Topsubj,Qaux,Top)      --> mood(Top,Mood):
                                   qaux(Mood,Topsubj,Qaux).

mood(_:_:wh,wh)               --> [].
mood(_,dcl)                   --> [].

qaux(dcl,_,[])                --> [].
qaux(wh,f,pre(V))             --> +V : %finiteaux(V).
qaux(wh,t,[])                 --> [] .

clause1(Topsubj,Qaux,T)       -->
                              subject(X:Num:_,Topsubj,T,T1):
                              vp(fin(_,Num,_),active,_,X:Num,Qaux,T1).

subject(X,t,hold(X),[])       --> [].
subject(X,f,T,T)              --> np(X).

vp(Infl,Voice,E,X,Qaux,T)     -->
                              v_premods(E):
```

```
                                vhead(Qaux,Infl,E,Y,Slots):
                                %voice(Voice,Infl,X,Y,Slots,Slots1):
                                %theme(X,Slots,Z):
                                postmods(vp,Slots1,State,E,Z,T,[]).

vhead(Qaux,Infl,E,X,Slots)   --> getverb(Qaux,V):
                                    %verbf(V,Vinf,Infl):
                                    verb(Vinf,Pred,E,X,Slots):
                                    l-Pred.

getverb([],V)                --> +V .
getverb(pre(V),V)            --> [].

voice(active,_,X,X,Slots,Slots)   :- !.
voice(passive,en,X,Y,Slots,Slots2):- choose(Slot:X,Slots,Slots1),
                                     pfill(Slot),

append(Slots1,[actor:Y],Slots2).

theme(X,Slots,Y) :- member(obj:Y,Slots),!.
theme(X,Slots,Y) :- member(pobj(_):Y,Slots),!.
theme(X,Slots,X).
```

B.3 General Rules for Postmodifiers

A word may have more than one postmodifier. The filling of the slots for modifiers is not always done sequentially. Some slots have higher priority than others and are filled first as in the following two examples:

John gave the book to Mary.

John gave Mary the book

The slot filling of the verb *give* is:

$$[obj, iobj]$$

In the first sentence the *object* "the book" appears before the *indirect object* "Mary" but in the second sentence, the indirect object appears before the direct object. The predicate associated with verb give is:

$$give(john, book, mary)$$

To fill the direct object before the indirect object we need to give the direct object a higher priority over the indirect object. McCord used the notion of *states* to determine the priority of fillers. A slot having a higher state is filled first. The degree of priority is determined by the number of the symbols # that is associated with the slot filling.

```
/*                                        */
/*    General rules for postmodifiers     */
/*                                        */
```

```
postmods(Cat,Slots,State,E,Y,T,T2) -->
                        %choose(Slot:X,Slots,Slots1):
                        filler(Slot,State,X,Y,T,T1):
                        %precede(State,State1):
                        postmods(Cat,Slots1,State1,E,Y,T1,T2).

postmods(Cat,Slots,State,E,Y,T,T1) -->
                        adjunct(Cat,State,E,Y):
                        %precede(State,State1):

postmods(Cat,Slots,State1,E,Y,T,T1).

postmods(_,Slots,_,_,_,T,T)        --> %satisfied(Slots).

choose(X,[X|L],L).
choose(X,[Y|L],[Y|L1]) :- choose(X,L,L1).

satisfied([Slot:X|T])  :-   opt(Slot), satisfied(T) .
satisfied([]):- !.

precede(0,_)   :- !.
precede(#X,#Y) :- precede(X,Y) .

/*  general filling  */
```

```
filler(Slot,State,X,Y,Topic,Topic) --> fill(Slot,State,X).

/*  virtual filling */
filler(Slot,State,X,_,hold(X),[])  --> vfill(Slot,State,X).

/* for the filling that needs the passing of */
/* the topics T1 and T2                      */

filler(Slot,State,X,Y,T,T1)  -->  tfill(Slot,State,X,Y,T,T1).
/*                                                    */
/*    The following rules are the verb modifiers      */
/*    They fill the arguments of the verb predicate   */

v_premods(_) --> [].
v_premods(E) --> avp(E).

\* rules for filling an object               *\
pfill(obj).
vfill(obj,0,X) --> [].
tfill(obj,# # 0,X,_,T,T) --> np(X).

\* rules for filling an indirect object      *\
pfill(iobj).
 fill(iobj,# 0,X) --> np(X).
```

```
 fill(iobj,# # 0,X)--> +to:np(X).
vfill(iobj,# # # 0,X) --> +to.

\* rules for filling a verb in a passive voice *\

  opt(actor).
 fill(actor,# # 0,X) --> +by : np(X).
vfill(actor,# # # 0,X) --> +by.
tfill(pass, # 0,X,Y,T,T) --> verbph(en,passive,X,Y,T).

\* rules for filling a prepositional phrase   *\

 fill(pobj(Prep),# # 0,X) --> +Prep :np(X).
vfill(pobj(Prep),# # # 0,X) --> +Prep.

\* rule for filling an infinite complement    *\

tfill(infcomp,# # 0,X,Y,T,[]) --> +to:verbph(inf,active,X,Y,T).

\* rule for filling a finite complement       *\

tfill(fincomp,# # 0,X,Y,T,[]) --> compclause1(X,T).
```

```
\* remaining rules                        *\

tfill(con(Conj),# # 0,X,_,T,[])-->+Conj:%conj(Conj,_,_,_):s.
tfill(auxcomp,# # 0,X,Y,T,T)   --> verbph(Infl,active,X,Y,T).
fill(indcmp,# 0,X,_,T,T)       --> np(X).
tfill(scomp, # # 0,X,_,T,T)    --> sentcomp(X).
tfill(sta, # # 0,X,_,T,T)      --> st(X).

st(X) --> +State:
          %sta(State,X,Pred,Op,L):
          Op-Pred.



adjunct(vp,# # 0,E,_) --> pp(P ,@P).
adjunct(vp,# # 0,E,_) --> avp(E).
adjunct(vp,# # 0,E,X) --> subclause1(X,P,@P).

subclause1(Sub:_,X,Op) --> +Conj:
                           %conj(Conj,Pred,Y,X):
                           Op-Pred:
                           verbph(Infl,Voice,Y,Sub,T).

subclause1(Sub:_,X,Op) --> +Conj:
```

```
                              %conj(Conj,Pred,Y,X):
                              +PNoun:
                              %defpron(PNoun,_,_,_,_):
                              Op-Pred:
                              verbph(Infl,Voice,Y,Sub,T).

sentcomp(X)              --> subst(X)-t:s.


verbph(Infl,Voice,X,Y,T) --> subst(X)-t:
                                 vp(Infl,Voice,E,Y,[],T).

compclause1(X:_,T)       --> subst(X)-t :
                                 binder(Topsubj):
                                 clause1(Topsubj,[],T).

avp(E)                   --> qualifiers(E):
                                 +Adv:
                                 %adv(Adv,Pred,Op,E):
                                 Op - Pred.


qualifiers(_) --> [].
binder(f)     --> +that.
binder(f)     --> +whether .
```

```
binder(t)     --> [].
```

B.4 Noun Phrase Rules

```
/*                                                    */
/*    The following rules define noun phrases         */
/*                                                    */


np(Feas)   --> propernoun(X0,Num):
               np2(X0,Num,def,[],Feas).

np(Feas)   --> pronoun(X0,Num,DType):
               np2(X0,Num,DType,[],Feas).

np(Feas)   --> det(DFeas):
               np1([],DFeas,Feas).

np1(X0,X1:Num:DType,Feas) --> n_premods(X1):
                              nhead(Num,X1,Slots):
                              %prefill(Slots,X0,Slots1):
                              np2(X1,Num,DType,Slots1,Feas).
```

```
np2(X0,Num,DType,_,Feas)  --> poss:

(np:X0:Num:DType)^np1(X0,_:_:DType,Feas).

np2(X,Num,DType,Slots,X:Num:DType) -->

postmods(np,Slots,State,Num,X,[],[]).


np(X)         --> n_premods(X):
                  n_compl(X).

np(X)         --> n_compl(X).

n_compl(Nsg) --> +Noun:
                  %nounf(Noun,Nsg,Num):
                  %noun(Nsg,Pred,X,Slots).

det(X:Num:DType)        --> +D : dt(D,B,F,P,X,Num,DType):
                            F/B-P.

propernoun(X:Type,Num) --> +Noun:
                            %propern(Noun,Noun0,Num,Type):
```

```
                          1 - (X = Noun0).

pronoun(X,Num,DType)   --> +PNoun :
                          %defpron(PNoun,Pred,X,Num,DType):
                          1 - Pred.

pronoun(X,Num,DType)   -->
                       +PNoun:
                       %indefpron(PNoun,Op-LF,Pred,X,Num,DType):
                       1-Pred.


nhead(Num,X,Slots)     --> +Noun:
                          %nounf(Noun,Nsg,Num):
                          %noun(Nsg,Pred,X,Slots):
                          1-Pred.

prefill(Slots,[],Slots).
prefill([(Slot:X)|Slots],S,Slots).


/*                                                    */
/*    The following rules are define the fillers for  */
/*    noun phrases                                    */
```

```
n_premods(X)           --> [].
n_premods(X)           --> adjph(X,_,_) : n_premods(X).

adjunct(np,_,_,X)      --> pp(X,r).
adjunct(np,_,_,X)      --> partvp(X).
adjunct(np,_,Num,X)    --> relclause1(X,Num).
adjunct(np,State,E,X) --> adjph(X,State,E).


pp(X,Op)               --> +Prep:
                           prep(Prep,Pred,Y,X):
                           Op-Pred:
                           np(Y).

partvp(X)              --> vp(ing,active,_,X,[],[]).

partvp(X)              --> vp(en,passive,_,X,[],[]).

relclause1(X,Num)      --> relpron(Topsubj).
                           clause1(Topsubj,[],hold(X:Num:wh)).

relpron(_)        --> +who.
```

```
relpron(_)       --> +that.
relpron(_)       --> +which.
relpron(f)       --> +whom.
relpron(f)       --> +[].


adjph(X,State,E) --> adjective(X,Slots,E):
                     postmods(adjph,Slots,State,E,X,[],[]).

adjective(X,Slots,E) --> +Adj:
                         %intenadj(Adj,_,P,Op,Rel,Slots):
                         Op-P.

adjective(X,Slots,E) --> +Adj:
                         %extenadj(Adj,X,P,Op,Rel,Slots):
                         Op-P.

tfill(nobj,# # 0,X,Y,T,[]) --> +of:np(X).
vfill(nobj,# # 0,X,Y,T,[]) --> +of.

opt(npobj(Prep)).

tfill(npobj(Prep),# # 0,X,Y,T,[]) --> +Prep:np(X).
vfill(npobj(Prep),_,X) --> +Prep.
```

```
tfill(nfincomp,# # 0,X,Y,T,[])    --> compclause1(X,T).

tfill(nouncomp,# # 0,X,Y,T,[])    --> np(X).
```

C

The Lexicon for the Case Study

C.1 Determiners

Determiners have the following format:

$$dt(D, B, F, P, X, Num, DType)$$

Where D denotes the determiner, B and F means that this determiners needs a base and a focus. P is the predicate associated with the determiner. X is the marker variable associated with the determiner and finally $DType$ is the type of the determiner. A list of the determiners used in our approach is given below.

```
dt(a,B,F,ex(B,F),_,sg,ex2) --> [].
dt(all,B,F,all2(B,F),_,pl,all2) --> [].
dt(an,B,F,ex(B,F),_,sg,ex2) --> [].
```

```
dt(another,B,F,another(B,F),_,_,ex2) --> [].
dt(any,B,F,all1(B,F),_,_,all1) --> [].
dt(each,B,F,all1(B,F),_,sg,all1) --> [].
dt(how,B,F,how_many(X,B&F),X:_,_,wh) --> + many .
dt(some,B,F,ex(B,F),_,_,ex2) --> [].
dt(the,B,F,the(B,F),_,_,ex1) --> [].
dt(this,B,F,the(B&point_to(B),F),_,_,ex1) --> [].
dt(these,B,F,the(B,F),_,pl,ex2) --> [].
dt(which,B,F,wh(X,B&F),X:_,_,wh) --> [].
```

C.2 Nouns

A noun has two entries to the lexicon:

1. The first entry has the following form:

   $$nounf(Noun1, Noun, Number)$$

   The argument *Noun*1 indicates the form under which the noun appears in the text. The argument *Noun*2 indicate the singular form of the noun. The argument *Number* indicates the number of the noun (singular or plural).

2. The second entry has the following form:

   $$noun(Noun, Pred, Type, Postmods)$$

The argument *Noun* is obtained from the previous entry. The argument *Pred* is the predicate associated with the noun. The third argument determines the Type of the noun and the fourth argument represents the list of the noun postmodifiers.

The entries used for the case study are:

```
nounf(air,air,sg).
nounf(aircraft,aircraft,sg).
nounf(airspace,airspace,sg).
nounf(arrival,arrival,sg).
nounf(autopilot,autopilot,sg).
nounf(conditions,condition,pl).
nounf(data,data,sg).
nounf(database,database,sg).
nounf(device,device,sg).
nounf(display,display,sg).
nounf(distance,distance,sg).
nounf(diversion,diversion,sg).
nounf(emergency,emergency,sg).
nounf(equipment,equipment,sg).
nounf(event,event,sg).
nounf(features,feature,sg).
nounf(flight,flight,sg).
```

```
nounf(ground,ground,sg).
nounf(indicator,indicator,sg).
nounf(information,information,_).
nounf(input,input,sg).
nounf(latitude,latitude,sg).
nounf(level,level,sg).
nounf(longitude,longitude,sg).
nounf(map,map,sg).
nounf(message,message,sg).
nounf(navigation,navigation,sg).
nounf('non-violation','non-violation',sg).
nounf(number,number,sg).
nounf(obstacle,obstacle,sg).
nounf(occurence,occurence,sg).
nounf(package,package,sg).
nounf(pilot,pilot,sg).
nounf(plan,plan,sg).
nounf(planning,planning,sg).
nounf(processing,processing,sg).
nounf(program,program,sg).
nounf(radar,radar,sg).
nounf(range,range,sg).
nounf(record,record,sg).
nounf(reference,reference,sg).
```

```
nounf(result,result,sg).
nounf(route,route,sg).
nounf('route-points','route-point',pl).
nounf(sens,sens,sg).
nounf(store,store,sg).
nounf(storm,storm,sg).
nounf(stream,stream,sg).
nounf(system,system,sg).
nounf(tape,tape,sg).
nounf(track,track,sg).
nounf(tracks,track,pl).
nounf(visibility,visibility,sg).
nounf(waypoint,waypoint,sg).
nounf(waypoints,waypoint,pl).
nounf(wind,wind,sg).

noun(air,air(X),X:air_t,[]).
noun(aircraft,aircraft(X),X:aircraft_t,[]).
noun(airspace,airspace(X),X:airspace_t,[]).
noun(arrival,arrival(X,P),X:arrival_t,[nobj:P]).
noun(autopilot,autopilot(X),X:autopilot_t,[]).
noun(condition,condition(X),X:condition_t,[]).
noun(data,data(X),X:data_t,[]).
noun(database,database(X),X:database_t,[]).
```

```
noun(device,device(X),X:device,[]).
noun(display,display(X),X:display_t,[]).
noun(distance,distance(X),X:distance_t,[]).
noun(diversion,diversion(X),X:diversion_t,[]).
noun(emergency,emergency(X),X:emergency_t,[]).
noun(equipment,equipment(X),X:equipment_t,[]).
noun(event,event(X,P),X:event_t,[nobj:P]).
noun(feature,feature(X),X:feature_t,[]).
noun(flight,flight(X),X:flight_t,[]).
noun(ground,ground(X),X:ground_t,[]).
noun(heading,heading(X),X:heading_t,[]).
noun(identifier,identifier(X),X:identifier_t,[]).
noun(indicator,indicator(X),X:indicator_t,[]).
noun(information,information(X),X:information_t,[]).
noun(input,input(X),X:input_t,[]).
noun(latitude,latitude(X),X:latitude_t,[]).
noun(longitude,longitude(X),X:longitude_t,[]).
noun(map,map(X,P),X:map_t,[nobj:P]).
noun(map,map(X),X:map_t,[]).
noun(navigation,navigation(X),X:navigation_t,[]).
noun('non-violation','non-violation'(X,P),X:violation_t,[nobj:P]).
noun(number,number(X,P),X:identifier_t,[nobj:P]).
noun(number,number(X),X:identifier_t,[]).
noun(obstacle,obstacle(X),X:obstacle_t,[]).
```

```
noun(package,package(X),X:package_t,[]).
noun(pilot,pilot(X,P),X:pilot_t,[nobj:P]).
noun(pilot,pilot(X),X:pilot_t,[]).
noun(plan,plan(X,P),X:plan_t,[nobj:P]).
noun(planning,planning(X),X:planning_t,[]).
noun(program,program(X),X:program_t,[]).
noun(radar,radar(X),X:radar_t,[]).
noun(range,range(X,P),X:range_t,[nobj:P]).
noun(record,record(X),X:record_t,[]).
noun(route,route(X),X:route_t,[]).
noun('route-point','route-point'(X),X:route-pt_t,[]).
noun(sens,sens(X,P),X:sens_t,[nobj:P]).
noun(store,store(X),X:store_t,[]).
noun(storm,storm(X),X:storm_t,[]).
noun(system,system(X,P),X:system_t,[nobj:P]).
noun(system,system(X),X:system_t,[]).
noun(tape,tape(X),X:tape_t,[]).
noun(track,track(X,P),X:track_t,[nobj:P]).
noun(track,track(X),X:track_t,[]).
noun(visibility,visibility(X),X:visibility_t,[]).
noun(waypoint,waypoint(X),X:waypoint_t,[]).
```

Proper nouns have the following entries:

$$propern(Noun, Noun0, Num, Type)$$

Where *Noun* is the form under which the proper noun appear in the text, *Noun*0 is the predicate associated with the proper noun. *Num* and *Type* determine respectively the number and the type of the proper noun. The entries for the case study are:

```
propern(blackpool,blackpool,_,_).
propern(doncaster,doncaster,_,_).
```

Definite pronouns have the following entries:

$$defpron(PNoun, Pred, X, Num, Type)$$

Where *PNoun* is the pronoun, *Pred* is the predicate associated with the pronoun. X is the marker variable and *Num* and *Type* are respectively the number and the type of the pronoun.

```
defpron(he,he(X),X:male,sg,def).
defpron(i,i(X),X:human,sg,pers1).
defpron(it,it(X),X:_,sg,def).
defpron(they,they(X),X:_,pl,pers3).
```

C.3 Verbs

Verbs have the following two entries in the lexicon :

1. The first entry has the form:

$$verbf(Form, Infinitive, Inflection)$$

Where *Form* is the form under which the verb appears in the text. The *Infinitive* is the infinitive form of the verb and *Inflexion* inflection of the verb.

2. The second entry has the form:

$$verb(Infinitive, Predicate, Type, Subject\_marker, Slots)$$

The first argument is obtained from the first entry. The argument, *Predicate* determines the predicate associated with the verb and *Type* is the type of the verb. The argument *subject_marker* determines the marker of the subject associated with the verb. *Slots* is the list of verb post modifiers.

The entries for the case study are:

```
verbf(arrive,arrive,fin(_,_,pres)).
verbf(are,be,fin(_,_,pres)).
verbf(assure,assure,inf).
verbf(be,be,fin(_,_,_)).
verbf(calculates,calculate,fin(pers3,sg,pres)).
verbf(choose,choose,fin(pers3,sg,pres)).
verbf(chooses,choose,fin(pers3,sg,pres)).
verbf(cleared,clear,fin(_,_,past)).
verbf(composed,compose,fin(pers3,pl,pres)).
verbf(comprise,comprise,inf).
verbf(considered,consider,fin(_,_,past)).
```

```
verbf(contains,contain,fin(pers3,sg,pres)).
verbf(contrasts,contrast,fin(pers3,sg,pres)).
verbf(draw,draw,inf).
verbf(draws,draw,fin(pers3,sg,pres)).
verbf(flown,fly,en).
verbf(fly,fly,inf).
verbf(hit,hit,fin(_,_,pres)).
verbf(identified,identify,en).
verbf(identifies,identify,fin(pers3,sg,pres)).
verbf(is,be,fin(pers3,sg,pres)).
verbf(listed,list,fin(_,_,past)).
verbf(load,load,inf).
verbf(plans,plan,fin(pers3,sg,past)).
verbf(planned,plan,fin(_,_,past)).
verbf(planning,plan,ing).
verbf(record,record,inf).
verbf(required,require,fin(_,_,past)).
verbf(scans,scan,fin(pers3,sg,pres)).
verbf(specified,specify,fin(_,_,past)).
verbf(steers,steer,fin(pers3,sg,pres)).
verbf(support,support,inf).
verbf(transferred,transfer,fin(_,_,past)).
verbf(undertake,undertake,inf).
verbf(updated,update,en).
```

```
verbf(used,use,en).
verbf(used,use,fin(_,_,past)).
verbf(uses,use,fin(pers3,sg,pres)).
verbf(verifies,verify,fin(pers3,sg,pres)).

verb(arrive,arrive_at(X,Y),_,X,[obj:Y]) --> +at.
verb(assure,assure(X,Y),_,X,[obj:Y]) --> [].
verb(be,be(X,Y),-,x,[obj:Y]) --> [].
verb(be,be(X,P,Q),_,X,[auxcomp:P,infcomp:Q]) --> [].
verb(be,be(X,P),_,X,[auxcomp:P]) --> [].
verb(be,be_passive(X,P),_,X,[pass:P]) --> [].
verb(be,is_available_as(X,Y),_,X,[obj:Y]) --> +available: +as.
verb(be,be(X,P,Q),_,X,[fincomp:P,infcomp:Q]) --> [].
verb(be,be(X),_,X,[sta:X]) --> [].
verb(calculate,calculate(X,Y),_,X,[obj:Y]) --> [].
verb(choose,choose(X,Y),_,X,[obj:Y]) --> [].
verb(clear,clear(X,P),_,X,[infcomp:P]) --> [].
verb(compose,compose_by(X,Y),_,X,[obj:Y]) --> +by.
verb(comprise,comprise(X,Y),_,X,[obj:Y]) --> [].
verb(consider,consider(X,P),_,X,[infcomp:P]) --> [].
verb(contain,contain(X,Y),_,X,[obj:Y]) --> [].
verb(contrast,contrast_with(X,Y),_,X,[obj:Y]) --> +with .
verb(draw,draw(X,Y),_,X,[obj:Y]) --> [].
verb(fly,fly(X,Y),_,X,[obj:Y]) --> [].
```

```
verb(fly,fly(X,Y,P),_,X,[obj:Y,pobj(in):P]) --> [].
verb(fly,fly_through(X,P),_,X,[obj:P]) --> +through .
verb(help,help(X,P),_,X,[obj:Y,infcomp:P]) --> [].
verb(hit,hit(X,Y),_,X,[obj:Y]) --> [].
verb(identify,identify(X,Y),_,X,[obj:Y]) --> [].
verb(list,list_for(X,Y,P),_,X,[obj:Y,infcomp:P]) --> +for .
verb(load,load(X,Y),_,X,[obj:Y]) --> [].
verb(plan,plan(X,Y),_,X,[obj:Y]) --> [].
verb(plan,plan(X,Y),_,X,[pobj(for):Y]) --> [].
verb(plan,plan_as(X,Y),_,X,[obj:Y]) --> +as.
verb(record,record(X,P),_,X,[scomp:P]) --> [].
verb(require,require(X,P),_,X,[infcomp:P]) --> [].
verb(record,record_on(X,P),_,X,[obj:P]) --> +on .
verb(scan,scan(X,Y),_,X,[obj:Y]) --> [].
verb(steer,steer(X,Y),_,X,[obj:Y]) --> [].
verb(support,support(X,Y),_,X,[obj:Y]) --> [].
verb(transfer,transfer(X),_,X,[]) --> [].
verb(undertake,undertake(X,Y),_,X,[obj:Y]) --> [].
verb(update,update(X,Y),_,X,[obj:Y]) --> [].
verb(use,use(X,P),_,X,[obj:P]) --> [] .
verb(use,use(X,P),_,X,[infcomp:P]) --> [] .
verb(use,use_as(X,Y),_,X,[obj:Y]) --> +as .
verb(use,use(X,Y,P),_,X,[obj:Y,infcomp:P]) --> [].
verb(verify,verify(X,Y),_,X,[obj:Y]) --> [].
```

C.4 Prepositions, Adverbs and Adjectives

Prepositions have the following entries:

$$prep(Prep, Pred, X, Y)$$

Where *Prep* is the preposition, *Pred* is the predicate associated with the preposition. X and Y are respectively the first and second argument of the preposition. The entries for the case study are:

```
prep(according,according_to(X,P),X,P) --> +to.
prep(against,against(X,P),X,P) --> [].
prep(along,along(X,P),X,P) --> [].
prep(by,by(X,P),X,P) --> [].
prep(between,between(X,P),X,P) --> [].
prep(during,during(X,P),X,P) --> [].
prep(for,for(X,P),X,P) --> [].
prep(from,from(X,Q),X,Q) --> [].
prep(in,in(P,Q),P,Q) --> [].
prep(on,on(P,Q),P,Q) --> [].
prep(over,over(X,P),X,P) --> [].
prep(to,to(X,Q),X,Q) --> [].
```

```
prep(through,through(P,Q),_,_) --> [].
prep(with,with(P,Q),P,Q) --> [].
prep(without,without(P,Q),P,Q) --> [].
```

Adverbs have the following entries:

$$adv(Adv, Pred, Op, Type)$$

Where *Adv* is the adverb, *Pred* is the predicate associated with the adverb. *Op* is the logical operator associated with the adverb and *Type* is the type of the adverb. The entries for the case study are:

```
adv(can,can(P),@P,_).
adv(may,may(P),@P,_).
adv(not,not(P),@P,negation).
adv(still,still(P),@P,_).
adv(unnecessarily,unnecessarily(P),@P,_).
adv(would,would(P),@P,_).
```

Intensional adjectives have the following entries:

```
intenadj(Adj,X,Pred,@X,Type,Mods)
```

Where *Adj* is the adjective, X is the variable marker. *Pred* id the predicate associated with the adjective. $@X$ is the logical operator associated with the adjective, note here it is fixed because an intensional adjective always fill a slot of a noun. *Type* is the type of the adjective and *Mods* is the list of the adjective postmodifiers. The entries for the case study are:

```
intenadj(adjacent,X,adjacent(X),@X,binary,[]).
intenadj('computer-assisted',X,'computer-assisted'(X),@X,nr,[]).
intenadj(discrete,X,discrete(X), @X,nr,[]).
intenadj(electronic,X,electronic(X),@X,nr,[]).
intenadj(flight,X,flight(X),@X,nr,[]).
intenadj(grid,X,grid(X),@X,nr,[]).
intenadj(heading,X,heading(X),@X,nr,[]).
intenadj(inertial,X,inertial(X),@X,nr,[]).
intenadj(information,X,information(X),@X,nr,[]).
intenadj(intermediate,X,intermediate(X),@X,nr,[]).
intenadj(map,X,map(X),@X,nr,[]).
intenadj(minimum,X,minimum(X),@X,nr,[]).
intenadj(moving,X,moving(X),@X,nr,[]).
intenadj(navigation,X,navigation(X),@X,nr,[]).
intenadj(navigated,X,navigated(X),@X,nr,[]).
intenadj(planning,X,planning(X),@X,nr,[]).
intenadj(route,X,route(X),@X,nr,[]).
intenadj(software,X,software(X),@X,nr,[]).
intenadj(sophisticated,X,sophisticated(X),@X,nr,[]).
intenadj(visual,X,visual(X),@X,nr,[]).
intenadj(waypoint,X,waypoint(X),@X,nr,[]).
intenadj(wind,X,wind(X),@X,nr,[]).
```

Extensional adjectives have the following entries:

```
intenadj(Adj,X,Pred,r,Type,Mods)
```

Where *Adj* is the adjective, *X* is the variable marker. *Pred* id the predicate associated with the adjective. *r* is the logical operator associated with the adjective, note here it is fixed because an intensional adjective always right conjoined to a noun. *Type* is the type of the adjective and *Mods* is the list of the adjective postmodifiers. The entries for the case study are:

```
extenadj(complex,X,complex(X),r,nr,[]).
extenadj(controlled,X,controlled(X),r,nr,[]).
extenadj(derived,X,derived(X),r,nr,[]).
extenadj(electronic,X,electronic(X),r,nr,[]).
extenadj(example,X,example(X),r,nr,[]).
extenadj(good,X,good(X),r,nr,[]).
extenadj(planned,X,planned(X),r,nr,[]).
extenadj(required,X,required(X),r,nr,[]).
extenadj(risky,X,risky(X),r,nr,[]).
extenadj(safe,X,safe(X),r,nr,[]).
extenadj(simple,X,simple(X),r,nr,[]).
extenadj(unique,X,unique(X),r,nr,[]).
extenadj(visible,X,visible(X),r,nr,[]).
extenadj(whole,X,whole(X),r,nr,[]).
extenadj(wrong,X,wrong(X),r,nr,[]).
```

Conjunctions have the following entries:

$$conj(Conj, Pred, X, Y)$$

Where *Conj* is the conjunction, *Pred* is the predicate associated with the conjunction. X and Y are respectively the first and second argument of the conjunction.

```
conj(and,Q\&P,P,Q).
conj(but,but(P,Q),P,Q).
conj(when,when(P,Q),P,Q).
sta(different,X,different(X),@X,[]).
```

D

Semantic Analysis

```
/*                          */
/* semantic interpretation  */
/* ------------------------ */

/* A call to synsem will associate an augmented  semantic item
  (ASI) to a syntactic element.  This operation is done in
  three stages. First a recursive  interpretation of the
  daughters is  performed. Then  synsem  calls  the  procedure
  reorder  which  will get a permutation of the ASIs. A set of
  ASI is reordered if the surface    order  does  not to  the
  intended  logical  order dictated by the quantifiers. The
  last  call  of synsem is made to the procedure  modlist
  which  will  combine the  elements of the ASIs with one
  another through the processes of modification and reshaping */
```

```
synsem(syn(Label,Mods),Sems2,Sems3) :-
                        synsemlist(Mods,Sems),
                        reorder(Sems,Sems1),

modlist(Sems1,sem(Label,id,t),Sem,Sems2,[Sem|Sems3]).

synsemlist([syn(Label,Mods0)|Mods],Sems1) :-
                        synsem(syn(Label,Mods0),Sems1,Sems2),
                        synsemlist(Mods,Sems2).
synsemlist([(Op-LF)|Mods],[sem(terminal,Op,LF)|Sems]) :-
                       synsemlist(Mods,Sems).
synsemlist([],[]).

/*                     */
/* reorder procedure   */
/* -----------------   */


reorder([X|L],R1) :-
                        reorder(L,R),
                        insert(X,R,R1).
reorder([],[]).
```

```
insert(X,[Y|L],[Y|L1]) :-
                        prec(X,PX),
                        prec(Y,PY),
                        ( PY > PX),!,
                        insert(X,L,L1).
insert(X,L,[X|L]) .

prec(Sem,N) :- dtype(Sem,DT),dtprec(DT,N),!.
prec(Sem,1).

dtype(sem(np:_:_:DT,_,_),DT) :- !.
dtype(sem(det:_:_:DT,_,_),DT):- !.
dtype(sem(_,_,t),t) :- ! .
dtype(sem(_,_,dcl(_,_)),dcl) :- !.
dtype(sem(_,_,yesno(_,_)),yesno) :- !.
dtype(sem(Cat:_,_,_),Cat).
dtype(sem(terminal,_,_),terminal).

dtprec(t,10).
dtprec(wh,10).
dtprec(ex1,4).
dtprec(dcl,6).
dtprec(yesno,6).
dtprec(all1,6).
```

```
dtprec(def,6).
dtprec(all2,4).
dtprec(ex2,4).
dtprec(terminal,2).
dtprec(avp,3).
dtprec(pp,4).
dtprec(subclause1,3).

modlist([Sem|Sems],Sem0,Sem2,Sems1,Sems3):-
                        modlist(Sems,Sem0,Sem1,Sems2,Sems3),
                        modify(Sem,Sem1,Sem2,Sems1,Sems2).
modlist([],Sem,Sem,Sems,Sems).
modify(Sem,Sem1,Sem1,[Sem2|Sems],Sems) :-
                        raise(Sem,Sem1,Sem2),!.
modify(sem(_,Op,LF),sem(Label,Op1,LF1),sem(Label,Op2,LF2),Sems,Sems) :-

mod(focal(B,P,Op)-B,Sem1,Sem2) :- !,
                        mod(Op-P,Sem1,Sem2).
mod(Sem,id-_,Sem) :-!.
mod((B1<F1)-P1,focal(P1,P2,Op)-F1,focal(B1,P2,Op)-t) :-!.
mod(((B:E)<F)-P , @ E-F,focal(B,P,l)-t) :- ! .
mod((B<F)-P,Op-F,focal(B,P,Op)-t) :- !.
mod(@P-Q,focal(B,P,Op)-B,Op-Q) :- !.
mod(Op-P,focal(B,Q,Op1)-P1,focal(B,Q,Op2)-P2) :- !,
```

```
                        mod(Op-P,Op1-P1 , Op2-P2).
mod(id-P,Sem,Sem).
mod(l-P,Op-Q,Op-(P&Q)).
mod(r-P,Op-Q,Op-(Q&P)).
mod(P/Q-R,Op-Q,@P-R).
mod(@P-Q,Op-P,Op-Q).
mod(subst(P)-t,Op-P,l-t).
mod((P<<Q)-R,Op-Q,focal(P,R,Op)-t) :- !.
mod((P<<Q)-R,focal(B,Q,Op)-B,focal(P,S,Op)-t) :-!.

raise(Sem,Sem1,Sem) :-
                        label(Sem,L),
                        label(Sem1,L1),
                        (L = np,(L1 = np ; L1 = pp ) ;
                          not(L = terminal) , L1 = verbph ;
                    L = verbph , L1 = subclause ).
raise(sem(Label,l,poss),sem(np:_,_,_),sem(Label,P/Q,the(Q,P))).
label(sem(L:_,_,_),L).
```

E

List of Entities for the Case Study

The produced entities have the following form:

(*entity*(*List-of-modifiers*): *Type*)

Where *entity* is the entity, *List-ofmodifiers* is the list of the entity modifiers. This list contains the adjectives and nouns that modifies the entity. *Type* is the type of the entity. The list of entities produced for the case study is:

```
(route(example):route_t)
(flight:flight_t)
(route:route_t)
(track(discrete):track_t)
(number:identifier_t)
(waypoint(intermediate):waypoint_t)
(waypoint:waypoint_t)
(track(planned):track_t)
```

```
(arrival(safe):arrival_t)
(aircraft:aircraft_t)
(airspace(controlled):airspace_t)
(obstacle:obstacle_t)
(system(sophisticated,electronic,navigation):system_t)
(aircraft(simple):aircraft_t)
(flight(risky):flight_t)
(waypoint(visible):waypoint_t)
(air:air_t)
(track:track_t)
(map:map_t)
(track(required):track_t)
(heading:heading_t)
(ground:ground_t)
(feature(visible):feature_t)
(system:system_t)
(plan:plan_t)
(indicator(heading):indicator_t)
(sense(visual):sens_t)
(aircraft(complex):aircraft_t)
(equipment(electronic):equipment_t)
(range(whole):range_t)
(navigation:navigation_t)
(planning(computer-assisted,flight):planning_t)
```

```
(system(inertial,navigation):system_t)
(radar:radar_t)
(display(moving,map):display_t)
(display(route):display_t)
(display(waypoint):display_t)
(autopilot:autopilot_t)
(number:identifier_t)
(reference(grid):reference_t)
(latitude:latitude_t)
(longitude:longitude_t)
(information:information_t)
(package(flight,planning,software):package_t)
(track(route):track_t)
(distance:distance_t)
(heading:heading_t)
(condition(wind):condition_t)
(airspace(controlled):airspace_t)
(non-violation:violation_t)
(information(derived):information_t)
(tape(cassette):tape_t)
(database(navigation):database_t)
(data:data_t)
(pilot:pilot_t)
```

F

List of Relations for the Case Study

Each relation has the following format:

$$d(entity1, entity2, degree), relation\_name$$

where *entity*1 and *entity*2 are the entities that are related, *degree* is the degree of the relation and *relation_name* is the name of the relation.

```
d(track(discrete):track_t,number:identifier_t,n:1),of
d(route:route_t,number:identifier_t,1:1),plan_as
d(route:route_t,waypoint:waypoint_t,1:n),compose_by
d(aircraft:aircraft_t,arrival(safe):arrival_t,1:1),of
d(track(planned):track_t,arrival(safe):arrival_t,n:1),assure
d(aircraft:aircraft_t,track(planned):track_t,1:n),fly
d(pilot:pilot_t,storm:storm_t,1:1),fly_through
d(pilot:pilot_t,airspace(controlled):airspace_t,1:1),fly_through
d(aircraft:aircraft_t,obstacle:obstacle_t,1:1),hit
```

```
d(aircraft1:aircraft_t,aircraft2:aircraft_t,1:n),hit
d(aircraft(simple):aircraft_t,pilot:pilot_t,1:1),of
d(pilot:pilot_t,flight(risky):flight_t,1:1),undertake
d(pilot:pilot_t,waypoint(visible):waypoint_t,1:n),choose
d(route:route_t,track:track_t,1:n),of
d(pilot:pilot_t,track:track_t,1:n),draw
d(pilot:pilot_t,heading:heading_t,1:1),steer
d(feature(visible):feature_t,ground:ground_t,n:1),scan
d(pilot:pilot_t,feature(visible):feature_t,1:n),verify
d(aircraft(simple):aircraft_t,system:system_t,1:1),of
d(pilot:pilot_t,map:map_t,1:1),of
d(pilot:pilot_t,plan:plan_t,1:1),of
d(system:system_t,plan:plan_t,1:1),comprise
d(system:system_t,indicator(heading):indicator_t,1:1),comprise
d(pilot:pilot_t,sens(visual):sens_t,1:1),of
d(system:system_t,sens(visual):sens_t,1:1),comprise
d(system1:system_t,system2:system_t,1:1),contrast_with
d(equipment(electronic):equipment_t,range(whole):range_t,1:n),of
d(aircraft(complex):aircraft_t,range(whole):range_t,1:1),use
d(aircraft(complex):aircraft_t,
  planning(computer-assisted,flight):planning_t,n:1),use
d(aircraft(complex):aircraft_t,
  system(inertial,navigation):system_t,n:1),use
d(aircraft(complex):aircraft_t,radar:radar_t,n:1),use
```

```
d(aircraft(complex):aircraft_t,
  display(moving,map):display_t,n:1),use
d(aircraft(complex):aircraft_t,
  display(route):display_t,n:1),use
d(aircraft(complex):aircraft_t,
  display(waypoint):display_t,n:1),use
d(aircraft(complex):aircraft_t,autopilot:autopilot_t,n:1),use
d(pilot:pilot_t,waypoint:waypoint_t,1:n),choose
d(waypoint:waypoint_t,number:identifier_t,1:1),identify
d(waypoint:waypoint_t,reference(grid):reference_t,n:1),identify
d(reference(grid):reference_t,latitude:latitude_t,1:1),contain
d(reference(grid):reference_t,longitude:longitude_t,1:1),contain
d(package(flight,planning,software):package_t,
  track(route):track_t,1:n),calculate
d(package(flight,planning,software):package_t,
  distance:distance_t,1:1),calculate
d(condition(wind):condition_t,
  heading:heading_t,1:1),calculate
d(airspace(controlled):airspace_t,non-violation:violation_t,1:1),of
d(package(flight,planning,software):package_t,
  non-violation:violation_t,1:1),calculate
d(information(derived):information_t,pilot:pilot_t,1:1),list_for
d(tape(cassette):tape_t,database(navigation):database_t,1:1),load
d(autopilot:autopilot_t,data:data_t,1:1),use
```

G

E-R Diagrams for the Case Study

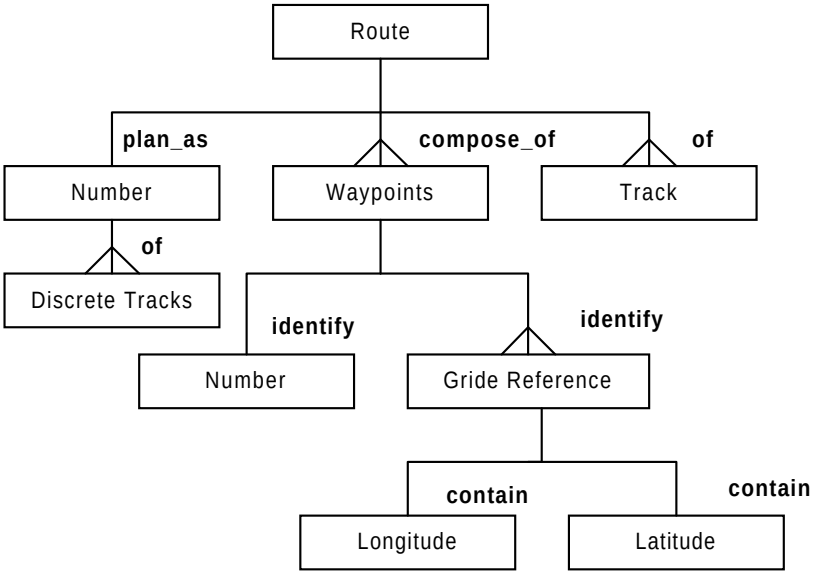

Figure G.1: The ER Diagram of route planing system

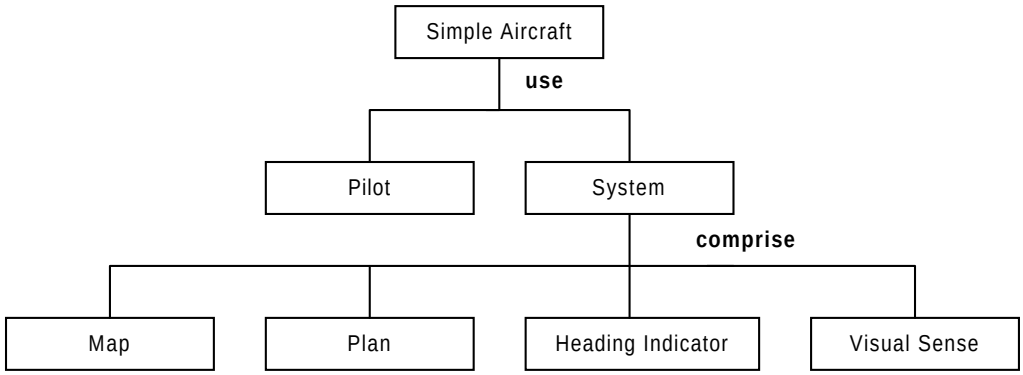

Figure G.2: The ER Diagram of a simple aircraft

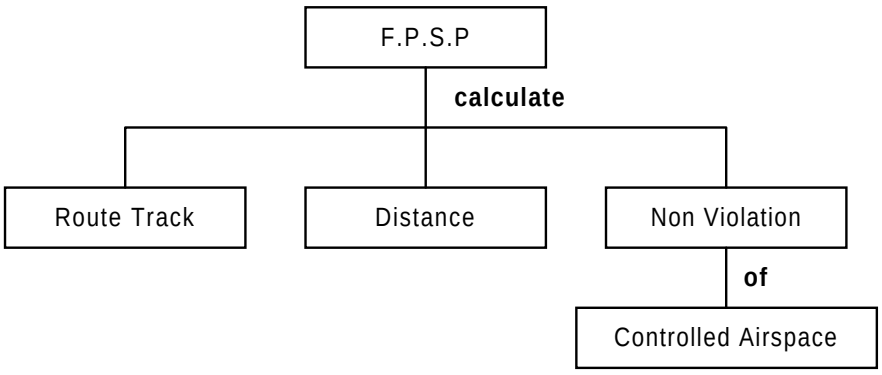

Figure G.3: The ER Diagram of the flight planning software package

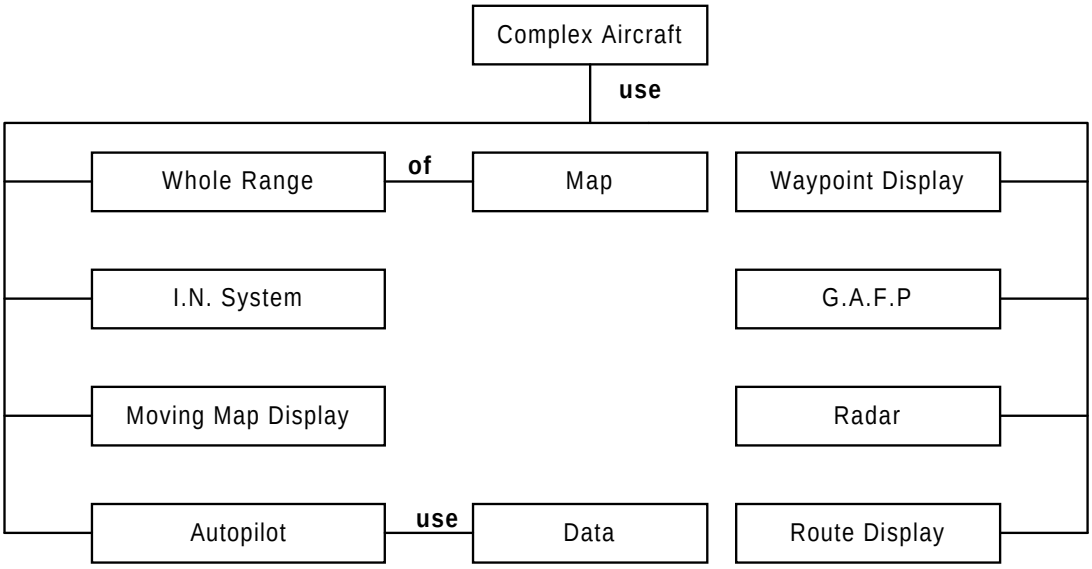

Figure G.4: The ER Diagram of a complex aircraft

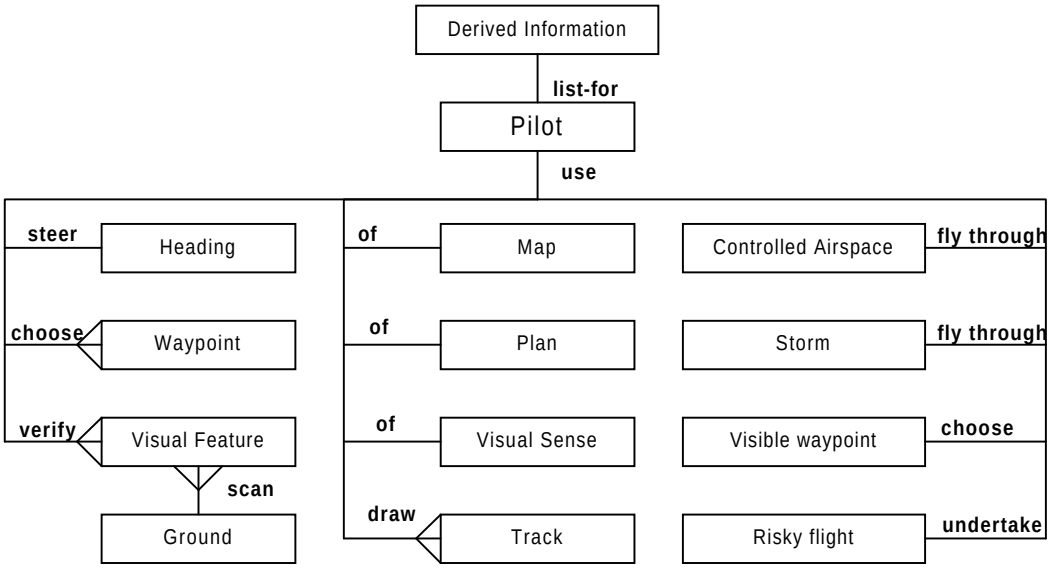

Figure G.5: The ER Diagram of the pilot

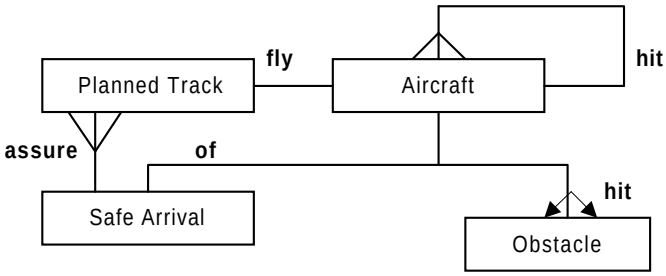

Figure G.6: The ER Diagram of the tracks

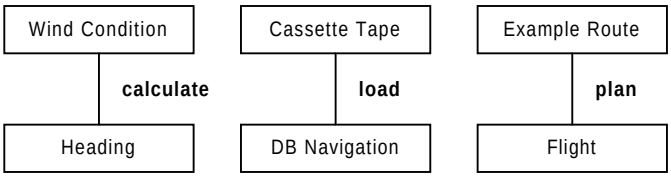

Figure G.7: The remaining diagrams

H

Rules of Logic

$$\boxed{\wedge\text{-}I}\ \frac{E_i}{E_1 \vee \ldots \vee E_n}$$

$$\boxed{\Rightarrow\text{-}I}\ \frac{E_1 \vdash E_2; \delta(E_1)}{E_1 \Rightarrow E_2}$$

$$\boxed{\exists\text{-}I}\ \frac{s \in X; E(s/x)}{\exists x \in X \cdot E(x)}$$

$$\boxed{\forall\text{-}I}\ \frac{x \in X E(x) \vdash}{\forall x \in X \cdot E(x)}$$

$$\boxed{\wedge\text{-}E}\ \frac{E_1 \wedge \ldots \wedge E_n}{E_i}$$

$$\boxed{\Rightarrow\text{-}E}\ \frac{E_1 \ \Rightarrow \ E_2; E_1}{E_2}$$

$$\boxed{\exists\text{-}E}\ \frac{\exists x \in X \cdot E(x); y \in X, E(y/x) \ \vdash \ E_1}{E_1}$$

$$\boxed{\forall\text{-}E}\ \frac{\forall x \in X \cdot E(X)}{E(s/x)}$$

Bibliography

[1] M.W. Alford. A requirements engineering methodology for real-time processing requirements. *IEEE Transactions on Software Engineering*, SE-3(1):60–69, 1977.

[2] J. Allen. *Natural language understanding*. The Benjamin/Cummings Publishing Company, Inc, 1987.

[3] ANSI/IEEE. IEEE guide to software requirements specifications, 1984.

[4] C. Ashworth and M. Goodland. *SSADM: A practical approach*. McGraw-Hill Book company, 1990.

[5] D.E. Avison and A.T. Wood-Harper. *MULTIVIEW: An exploration in information systems development*. Blackweel Scientific Publication, 2 edition, 1990.

[6] R. Balzer. A 15 year perspective on automatic programming. *IEEE Transactions on Software Engineering*, SE-11(11):1257–1268, 1985.

[7] R. Balzer, N. Goldman, and D. Wile. Informality in program specification. *IEEE Transactions on Software Engineering*, SE-4(2):94–103, 1978.

[8] A. Barr and E. A. Feigenbaum, editors. *Understanding Natural Language*, volume 1 of *The Handbook of A.I.* Pitman, 1981.

[9] D.R. Barstow. *Knowledge-Based program construction.* North Holland, 1979.

[10] T.E. Bell, D.C. Bixler, and M.E. Dyer. An extendable approach to computer-aided software requirements. *IEEE Transactions on Software Engineering*, SE-3(1):49–60, 1977.

[11] D. Benyon. *Information and data modelling.* Blackwell Scientific Publications, 1990.

[12] B.W. Boehem. Improving software productivity. *IEEE Computer*, 20(9):43–58, 1987.

[13] A. Borgida, S. Greenspan, and J. Mylopoulos. Knowledge representation as the basis for requirements specifications. *IEEE Computer*, 18(4):82–90, 1985.

[14] D.S. Bowers. *From data to data base.* Van Nostrand reinhold (U.K) Co. Ltd, 1988.

[15] N. Chomsky. *Aspects of the theory of syntax.* Cambridge, MA:MIT Press, 1965.

[16] M.G. Christel and K.C. Kang. Issues in requirements elicitation. Technical Report CMU/SEI-92-TR-12 ESC-TR-92-012, Software Engineering institute, Carnegie Mellon University Pittsburgh, Pennsylvania 15213, September 1992.

[17] B. Cohen. Justification of formal methods for system specification. *Software Engineering Journal*, 4(1):26–35, January 1989.

[18] A. Colmerauer. Metamorphosis grammars. In L. Bolc, editor, *Natural Language Communication With Computers*. New York Springer-Verlag, 1978.

[19] J.R. Comer. *An experimental natural language processor for generating data type specifications*. PhD thesis, Texas A & M University, 1979.

[20] V. Dahl and M. McCord. Treating coordination in logic grammars. *American Journal of Computational linguistics*, 9(2):69–91, 1983.

[21] A.M. Davis. *Software requirements objects, functions and states*. Prentice Hall International, Inc, 1993.

[22] A.M. Davis, E.H. Bersoff, and E.R. Comer. A strategy for comparing alternative software development life cycle models. *IEEE Transactions on Software Engineering*, SE-14(10):1453–1461, 1988.

[23] T. Dodd. *An Advanced Logic Programming Language- Prolog-2 User Guide*, volume 1. Intellect, 1990.

[24] D.R. Dowty, L. Karttunen, and A.M. Zwicky. *Natural language parsing.* Cambridge University Press, 1985.

[25] R.B. France. Semantically extended data flow diagrams: A formal specification tool. *IEEE Transactions on Software Engineering*, 18(4):329–346, 1992.

[26] C. Gane and T. Sarson. *Structured System Analysis.* Prentice-hall Software series, 1979.

[27] Gazdar, Klein, Pullum, and Sag. *Generalized phrase structure grammar.* Basil Blackwell, 1985.

[28] G. Gazdar and C. Mellish. *Natural language processing in PROLOG An introduction to computational linguistics.* Addison-Wesley Publishing Company, 1989.

[29] H. Gomma and D. Scott. Prototyping as a tool in the specification of user requirements. In *5th IEEE international Conference on software engineering*, pages 333–342, 1981.

[30] B. Hepworth. An introduction to Z. Technical Report BAe-WIT-RP-GEN-SWE-152, Systems Computing Department, British Aerospace Ltd, February 1988.

[31] M. Hess. How does natural language quantify? In *Second Conference of the European Chapter of the association for Computational Linguistics*, pages 8–15, 1985.

[32] E. Hirsch. Evolutionary acquisition of command and control systems. *Program Manager*, pages 18–22, Nov-Dec 1985.

[33] J. Hobbs. Resolving pronouns references. *LINGUA*, 44(4):311–338, 1978.

[34] R.S. Jackendoff. The base rules for prepositional phrases. In P. Keparsky Stephane R. Anderson, editor, *A festschrift for Morris Hall New York*, pages 345–356. Holt, Rinehart and Winston, 1973.

[35] H. Jackson. *Analysing English.* Pergman Press, 1982.

[36] W.L. Johnson, M.S. Feather, and D.R. Harris. Representation and presentation of requirements knowledge. *IEEE Transactions on Software Engineering*, 18(10):853–869, 1992.

[37] C.B. Jones. *Systematic Software development using VDM.* Prentice Hall International, 1990.

[38] C.B. Jones and R.C. Shaw. *Case studies in systematic software development.* Prentice Hall International, 1990.

[39] T. C. Jones. Reusability in programming: A survey of the state of the art. *IEEE Transactions on Software Engineering*, SE-10(5):488–494, 1984.

[40] R. A. Kowalski. *Logic for problem solving.* North-Holland, 1979.

[41] P. Loucopoulos and R.E.M. Champion. Knowledge-based support for requirements engineering. *Information and software technology*, 31(3):123–135, April 1989.

[42] P. Loucopoulos and R.E.M. Champion. Concept acquisition and analysis for requirement specification. *Software Engineering Journal*, 5(2):116–124, March 1990.

[43] M. McCord. Slot grammars. *American Journal of Computational Linguistics*, 6(1):31–43, 1980.

[44] M. McCord. Using slot and modifiers in logic grammars. *Artificial Intelligence*, 18:327–367, 1982.

[45] M. McCord. Modular logic grammars. In *Proceedings of the 23rd Annual Meeting of The Association for Computational Linguistics*, pages 104–117, Chicago, 1985.

[46] M. McCord. Focalizers, the scoping problem, and semantic interpretation rules in logic grammars. In Michael van Canengham and David H. D. Warren, editors, *Logic programming and its applications*, pages 223–243. Alex publishing corporation Norwood, New Jersey, 1986.

[47] M. McCord. Natural language processing in Prolog. In Walker Adrian, editor, *A logical approach to expert systems and natural language processing Knowledge systems and PROLOG*, pages 391–402. Addison-Wesley Publishing company, 1990.

[48] B. Meyer. On formalism in specifications. *IEEE Software*, pages 6–26, January 1985.

[49] H. Partsh and R. Steinbruggen. Program transformation systems. *ACM Computer surveys*, 15(3):199–236, 1983.

[50] T.M. Paskiewicz. Anaphoric pronouns. Master's thesis, UMIST, 1989.

[51] F.C.N. Pereira. Extraposition grammars. *American Journal of Computational Linguistics*, 7(4):243–256, 1980.

[52] F.C.N. Pereira and S.M. Shieber. *PROLOG and natural language analysis*. CSLI, 1987.

[53] F.C.N. Pereira and D.H.D Warren. Definite clause grammars for language analysis: A survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13:231–278, 1980.

[54] J.D. Phillips and H.S. Thompson. GPSGP a parser for generalized phrase structure grammars. *Linguistics*, 23:245–261, 1985.

[55] P.Nico, J.V. Katwijk, and T. Hans. Applications and benefits of formal methods in software development. *Software Engineering Journal*, 7(5):335–346, September 1992.

[56] K. Pohl. The three dimensions of requirements engineering. Informatik V, RWTH-Aachen, Ahornstr. 55, 5100 Aachen,Holland.

[57] S.G. Presland. *The analysis of natural language requirements documents*. PhD thesis, University of Liverpool, 1986.

[58] H.B. Reubenstein and R.C. Walters. The requirements apprentice: automated assistance for requirements acquisition. *IEEE Transactions on Software Engineering*, 17(3):226–240, 1991.

[59] D.T. Ross and JR K. E Schoman. Structures analysis for requirements definition. *IEEE Transactions on Software Engineering*, SE-3(1):6–15, 1977.

[60] W. W. Royce. Managing the development of large software systems: concepts and techniques. In *Proceedings of WESCON*, pages 1–9, Los Angeles, 1970.

[61] B. Russell. On denoting. In *Mind*, pages 479–493. NS,14, 1905.

[62] G. G. Schulmeyer and J. I McManus, editors. *Handbook of software qualite assurance*. Van Nostrand Reinhold, New York, 1987.

[63] L.T. Semmens, R.B. France, and T.W.G Docker. Integrated structured analysis and formal specification tools. *The Computer Journal*, 35(6):600–610, 1992.

[64] I. Sommerville. *Software Engineering*. Addison-Wesley Publishing Company, 1992.

[65] L. Sterling and E. Shapiro. *The art of Prolog*. MIT Press, 1986.

[66] D. Teichroew and E.A. Hershey. PSL/PSA: A computer-aided technique for structured documentation and analysis of information processing system. *IEEE Transactions on Software Engineering*, SE-3(1):41–48, 1977.

[67] N.P. Vitalari and G.W. Dickson. Problem solving for effective systems analysis: an experimental exploration. *CACM*, 26(11):948–956, 1983.

[68] A. Walshe. NDB: The formal specification and rigorous design of a single-user database system. In Prentice Hall International Series in Computer Science, editor, *Case Studies in Systematic Software Development*, pages 11–45. Jones Cliff B and Shaw R.C., 1990.

[69] D.H.D. Warren and F.C.N. Pereira. An efficient easily adaptable system for interpreting natural language queries. *American Journal of Computational Linguistics*, 8(3-4):110–122, 1982.

[70] E.E. Williams. Computer interpretation of narrative descriptions in conceptual data-modelling. Master's thesis, UMIST, 1987.