Elicitation of Software Requirements: The Role of Natural Language Processing

Abu Turab Alam
Ph. D Thesis

Submitted to the University of Surrey in partial fulfilment of requirements for the degree of Doctor of Philosophy

Department of Mathematical and Computing Sciences
University of Surrey
Guildford
Surrey
Feb 1991
Abstract

The engineering of a software system depends crucially upon the requirements specification of the system. The specification of requirements is a complex and interactive process involving an analyst and a client in a requirements definition activity. The principal medium for this activity is natural language, and we observe that special terms or jargon are used to abbreviate the communication between an analyst and the client. The information available to an analyst during this communication is inherently ambiguous and incomplete and often defined by the client without context.

We emphasise the all-pervasive use of natural language during the requirements definition activity. Natural language is used from the very start of a project and used throughout requirements acquisition, expression and analysis for software specification. Furthermore, a substantial amount of relevant information about the client's system is also available in natural language.

An analyst performs various tasks to elicit and understand software requirements. We identify a number of techniques to expedite these tasks for an analyst. These techniques have their origins in three different fields: knowledge engineering (for system knowledge acquisition); information science (searching for key concepts underlying the user's domain directly from its text); and natural language studies (schema for formalizing the user's domain knowledge).

The main advantage of our framework is that it does not constrain (in the form of arbitrary method constructs) the thinking processes of an analyst. Instead, our framework emphasises the functional behaviour of natural language in a specific domain and allows the analyst to elicit and understand the requirements themselves in natural language.
Acknowledgements

This research project has been sponsored by the Government of Pakistan under a HIGH LEVEL SCIENCE & TECHNOLOGY MANPOWER Scheme. I would like to acknowledge the financial support which I received under this scheme. In addition my thanks go to my parent organisation, PCSIR Labs, for nominating me and granting me leave to pursue this research.

I would like to express my thanks to Dr. Khurshid Ahmad who showed a special interest in this project and supervised me during the difficult periods of its progress. I am also thankful to the IKBS group members, Dr. J. Segal, Mr. P. Holmes-Higgin, Miss. H. Fulford, Mr. S. R. Abidi, and the staff of the Computing Unit, for giving me their attention when and where it was needed.

I would like to express my gratitude to Mr. Bob Malcolm, and Mr. L. Russell of SEMA CAP Limited in the UK for supporting me in the technical aspects of this research. Thanks also to the 13 analysts of various organisations who answered the survey. A general acknowledgment goes to all those contributors in this field whose work I have benefited from immensely.

My thanks go to Prof. S. A. Schuman whose help was much appreciated during the initial and final stages of this project. Dr. R. D. Knott deserves my thanks for helping me to know what is required to program in PROLOG. I am also thankful to Prof. R. Shail and the staff of the Department of Mathematical and Computing Sciences for providing me with the facilities necessary for completing this research, especially to Mark Ellis, who has been highly cooperative. I am grateful also to the Department for providing additional financial support.

My wife Rafat and sons, Mashood and Sharjeel, deserve my thanks for their understanding and for bearing with me during my anguished moments when writing this thesis. Thanks to my mother for letting me proceed for these studies.
Contents

Chapter 1: Introduction
1.1 Software Development: Background .................................................... 1
1.2 Requirements Specification Processes .................................................. 8
1.3 Overview and Structure of the Thesis .................................................... 12

Chapter 2: Background
2.1 Current RA practices: Notational Techniques and Methods ............. 15
2.2 Novel Techniques and Intelligent Front-ends ........................................ 26
2.3 Natural Language Semantics ................................................................. 34
2.4 Knowledge Acquisition and Info. Retrieval Strategies ......................... 40
2.5 Concluding Remarks ............................................................................. 49

Chapter 3: RA Domain Knowledge
3.1 Knowledge Acquisition: an interview based approach in RA .............. 51
3.2 Experiential Knowledge in RA .............................................................. 52
3.3 Conclusion Section 3.2 ......................................................................... 61
3.4 Knowledge Verification: A questionnaire based approach .................. 63
3.5 Conclusion Section 3.4 ......................................................................... 73
3.6 CASE Tools Features ............................................................................ 74
3.7 Concluding Remarks ............................................................................. 75

Chapter 4: A Toolkit for Requirements Definition
4.1 Elicitation & Understanding in NLP Framework ................................ 77
4.2 Linguistic Cues Probing for Elicitation ................................................ 81
4.3 Text Selection and Text Processing ...................................................... 85
4.4 A Linguistic Toolkit ............................................................................... 96
4.5 Concluding Remarks ............................................................................. 102

Chapter 5: Summary and Conclusion
5.1 Requirements Definition: A Cognitive Activity ................................. 103
5.2 Capturing Requirements ...................................................................... 105
5.3 NLP in Requirements Definition .......................................................... 106
5.4 Future Work ......................................................................................... 110

Appendix A Natural Language Syntax and Semantics .............................. 112
Appendix B Experts Interview Transcript .................................................. 126
Appendix C Analysts Survey ..................................................................... 145
CHAPTER 1: INTRODUCTION AND OVERVIEW

The development of software systems according to the needs of end users is generally recognised as an expensive and risky business. Timely and cost-effective construction of a software system requires careful management and sensitive control of the development process. This development process also depends crucially on the definition of software requirements (i.e. requirements definition). This thesis introduces the use of natural language processing techniques for software requirements definition.

We first discuss software development models to introduce the requirements definition activity (Section 1.1). The notions of requirements specification processes are discussed next in an engineering perspective (Section 1.2). Lastly, the structure and conclusion of this thesis are discussed (Section 1.3).

1.1 Software Development: Background

There are a number of reasons which necessitate the use of an engineering approach for software development. For example, software costs alone account for 80% of the average total cost of a computer system (Lowery & Duran 1989). Large software projects usually suffer from budget over-runs. Once these expensive systems are delivered to the clients, it turns out that the software does not perform according to the users' expectations. However, in order to manage the software development activity in a more systematic manner, it is perhaps necessary to use a model dealing with the software development process, preferably one which is explicit and precise (Wolff 1989). There follows a brief review of some of the popular models, including the 'waterfall model', the 'prototyping model, and the 'spiral model' for software development.

The lifecycle models: The early attempts to turn software development into an engineering discipline were by Royce (1970). Royce has argued that software development logically proceeds along an extended 'what-to-how' spectrum. A number of models subscribe to the so-called lifecycle paradigm [1] of software development with slight variations, e.g. cascade or waterfall model, b-model and so on (see Birrell & Ould 1985 for details). In the so-called 'cascade model', it is suggested that software should be developed through separate successive phases. A typical sequence in this model is:

---

1 The term 'lifecycle' in software engineering is perhaps adopted for software due to its similarity to the life-cycle of an 'in-vivo' system: 'inception-birth-maturity-death' cycle.
1. Project Inception: defining objectives and constraints on the project
2. Planning: work breakdown, cost estimation etc.
3. Specification of Requirements for the System: technical and functional investigation, test criteria, quality management planning etc.
4. Design: system modelling at some high level of abstraction
5. Coding: the production of an executable program
6. Testing and Integration
7. System Release

In the lifecycle model, it is possible (in theory) to return to the previous stage (except the first) and rework it. From another point of view, this model can be regarded as equivalent to a top-down design approach, which emphasises a more or less fixed progression in design activities: from a largest abstracted component to a more detailed design activity.

*The Prototyping Model:* There are many variants of prototyping model available in software engineering literature (see details in Hekmatpour & Ince 1986). The model is based on the understanding that a prospective user of a software system is rarely able to define his/her requirements fully in one operation. The users often find it difficult to define what they want in abstract terms or indeed verbally. This model also provides guidance for system designers during system development. It is expected that the use of this model will provide answers to questions like: how the system should be structured for easy modification? what level of performance must be maintained in each version delivered to the client? and so on. There are three main variants of prototyping models: 'Throw-it-away', 'Evolutionary', and 'Incremental' (Wolff 1989:135).

*Throw-it-Away Prototyping:* In the 'throw-it-away' variant, all prototypes are discarded; the delivered system is the last system in sequence.

*Incremental Prototyping:* In the Incremental Model, the software development is carried out in different software versions and the functionality of the system is progressively refined and increased.

*Evolutionary Prototyping:* The Evolutionary Model allows the same software development approach as used in the Incremental Model, except the Evolutionary Model allows deletion and changes from stage to stage in different software versions, whereas the Incremental Model allows only additions.

*The Spiral Model:* This model emphasises a 'risk-driven' approach to software development (Boehm 1988). A project is developed in a series of *cycles or rounds:* each cycle is of '360 degrees' divided into four quadrants: definition; design; implementation; and validation (see Figure 1a, the steps in each quadrant are described in
Figure 1b). In Figure 1b, we also compare the spiral model with the various phases of the cascade or waterfall model. The comparison shows similarities between the tasks of the two models.

![Diagram of Spiral and Waterfall Models](image)

Figure 1 (a) Cyclic Behaviour of the Spiral Model which contains in each of its quadrants a cycle as defined in (b) within the context of quadrant name; (b) Spiral Model and Waterfall Model.

The software development models as defined above emphasise the task-oriented nature of software development. The most important task (see shaded area in Figure 1b) is the task of defining software requirements—the requirements definition activity.

**The Requirements definition activity**

*Requirements definition* is often regarded as a separate phase in software development models. For example, in the spiral model the requirements definition activity is separately defined for every quadrant of software development: 'define objectives for the cycle; identify constraints'. Similarly, in the cascade or waterfall model this activity is termed as the 'inception and system definition' phase of a software development cycle. However, in the prototyping models the situation appears different. For instance, in the Evolutionary Model, this activity is distributed all over the software development cycle. The above discussions show that requirements definition activity is essentially incorporated in all the software development models. There are three points to note in the context of requirements definition activity: firstly, the constraints on requirements definition; secondly, what issues are related to this activity; and thirdly, what results are expected after this activity.
Chapter 1

Introduction and Overview

Constraints on Software Requirements Definition Activity
There are three major constraints attached to the requirements definition activity. Also, they appear to be independent of any software development models discussed above.

The first important constraint is time. Software projects are normally scheduled to be completed in a limited period of time. It is perhaps important to note that the total time provided for producing a software (requirements) specification, for a typical lifecycle software development model, has been estimated to be between 10-30 percent of the total time provided for the whole project (Birrell & Ould 1985:17). Boehm (1988), however, when doing a 'value chain' analysis (due to Porter 1980) for software development projects in his company, estimated that only around 4 percent (without re-work estimation[2]) of time or effort is devoted to the requirements definition activity, as compared to preliminary design (8%), detailed design (11%), code and unit test (8%), and integration and test (7%).

The second constraint relates to the knowledge of the application domain. It is essential for an analyst to have a working knowledge of the application domain in order to determine the functionality of the proposed system. One factor which is crucial for the analyst to achieve a better understanding of the domain is the client terminology or the special language of the domain (e.g. legal language, process control language etc.). The incorrect use or ignorance of the domain terminology can produce misunderstanding and can cause major re-working of the software requirements definition. The terminological 'barrier' would be felt more acutely in application domains which are novel and are handled for the first time.

The third constraint relates to the usage of the (currently) available tools and techniques. Here, there are three main factors which can influence the results of the requirements definition activity (i) the relevance of the tools and technique to the problem, (ii) how skilfully the analyst can use the tools and techniques, (iii) the relevant technique's limitations for fully representing the domain knowledge. These factors collectively or individually can bias the domain knowledge during requirements definition and has been criticised as 'technology-driven' rather than 'problem-oriented' (Finkelstein & Potts 1985).

---

2 The re-work makes this figure to 9 percent, which is near to lower bound of Birrell and Ould (1985) time or effort estimate for requirements definition activity of a software development project.
The Main Issues of Requirements Definition Activity

One of the main concerns during the software requirements definition phase is generally considered to be the changes made by the client in their requirements. This gives rise to new management issues, often represented by the terms such as 'change management' or 'change control' management, during system development (see Section 3.1). As long as the software is in its initial stages of development, these changes are manageable. However, when the project comes near to its completion, these changes are difficult to be incorporated in the system.

The other main issue is that there appears to be no technique which can effectively deal with requirements definition activity objectively (i.e. the power of expression is limited to the constructs provided in the techniques). Our industry-based surveys indicate that an analyst does not rely on a single technique or method for requirements acquisition, expression, or analysis. Various other issues which are considered important during requirements definition include 'software performance', 'test criteria', 'software sophistication' and so on (see Chapter 3 for details).

The Requirement definition Activity Results

The main deliverable of the requirements definition activity is the software requirements specification document. There is some consensus on the standard form of the specification document (see STARTS Guide 1987[3], and ANSI/IEEE standards 1984 for details). These standards principally deal with the contents of the specification document and discuss what the document must contain and where, and what is not required. It appears that no standards are prescribed as to what methods or techniques should be used to represent the functionality of a proposed system.

A main purpose of the requirements definition activity is to understand the client's expectations of a proposed system. The task thus requires an analyst to understand the domain in which the software is going to work. It is essential to understand the basic functionality of the system after the requirements definition stage (at least up to the point of convincing the client that a computer solution is feasible).

---

3 The 'STARTS Guide' is a guide to methods and software tools for the construction of large real-time systems. This document was prepared by the collaborative efforts of over 50 experts from 21 organisations under the support of the Department of Trade and Industry and National Computing Centre of the United Kingdom.
Software Development Techniques and AI

There are a number of software techniques and methods which purport to help in the requirements definition activity. Some of the early techniques and methods (see details in Chapter 2) include: Data flow diagrams (Demarco 1978); Structure charts (Yourdon & Constantine 1979); Entity-relationship diagrams (Chen 1976); Jackson System Development (JSD) method (Cameron 1986, Jackson 1982); Structure Analysis and Design Techniques: SADT (Ross 1977); COntrol Requirements Expression: CORE (Kramer 1988, Mullery 1979). Our surveys (see Chapter 3) indicate that despite the popularity of these techniques and methods, they are not a panacea for solving various problems in requirements definition. Particularly they leave much to be desired for early parts of requirements definition where an analyst starts building his or her understanding for a proposed system using his or her problem-solving skills. Given the dominance of the Artificial Intelligence (AI), in the late 70's and early 80's, the use of AI claimed that these problem-solving skills can be simulated for solving crucial problems in various fields including the software development activity.

Artificial Intelligence is a branch of computer science dedicated to the development of computer systems which perform activities normally associated with intelligence in humans (Barr & Feigenbaum 1981). AI, therefore, borrows methods, tools and techniques from disciplines such as Psychology, Linguistics, Philosophy and Computer Science (Sowa 1984). Early attempts to involve AI in computer programs was based on (algorithmic but) general problem-solving techniques. However, despite some interesting progress, this strategy produced no (significant) breakthrough (Waterman 1985). It was soon realised that this approach is not adequate for specific tasks. The focus of AI research was shifted to capture specific problem-solving abilities of (human) experts, and its subsequent encoding in a 'knowledge-base' for special-purpose computer programs. These programs were called expert systems. An expert system approach then emphasised as useful for solving problems in a specific domain with a specialise problem-solving knowledge (see Jackson 1990, Waterman 1985 for details).

For software engineering, the overall objective of AI is to provide "intelligent computer-based assistance for all parts of the software lifecycle" (Lowery & Duran 1989:245). Even from an AI perspective, there are many issues associated with requirements analysis. The central one is of 'requirements acquisition', where an intelligent system has to develop a "coherent internal representation from an initial set of disorganized statements" (ibid 1989:297).

In the context of knowledge based software engineering (KBSE), most of the
present systems are based on the knowledge of the popular software development
methods (e.g. CORE, JSD, SADT). In KBSE systems it appear that the methods'
knowledge (i.e. its commitments for software development activities such as
requirements modelling, software design, implementation and testing) is central to a
knowledge based front-end for an analyst. For example, 'The Analyst' project
(Stephens & Whitehead 1985) and the 'TARA' project (Kramer 1988) are based on the
CORE method knowledge; and in the ASPiS project (Hughes et al. 1988) the SADT
method knowledge is used. The utility of these systems in real world situations has not
been clearly reported.

The 'Programmer's Apprentice' project (Rich & Waters 1988), at the MIT, has led
to the development of two demonstration systems which deal with the requirements
definition activity for software development: the Knowledge-Based Editors in 'Emacs'
(KBEEmacs) and the Requirements Apprentice (RA). The report discusses the problems
of developing an automated requirements assistant. These problems are generally
related to informal communication between a client and an analyst. These problems
include:

"1. Special terms or jargon are used to abbreviate the communication.
2. Informal communication is inherently ambiguous and incomplete so context
   needs to be used to solve ambiguity and fill in gaps.
3. Statements that are true in the abstract are sometimes false when considered in
detail.
4. Different aspects of the description may be in direct contradiction with each
   other."(Lowery & Duran 1989:298)

These results confirm our belief that the available requirements definition tools and
techniques are not fulfilling the demand of various aspects requirements definition
activities. This conclusion also shows that for an automated requirements assistance,
the researchers have started showing their concern about users jargon, context and
ambiguity when dealing with the client's software requirements. In order to tackle
these problems during requirements definition it is perhaps necessary to consider them
in a different framework. A framework which can provide an appropriate linguistic
treatment for the 'user jargon', 'context', and 'ambiguity of client's requirements and
which can help an analyst to understand the application domain quickly.

1.2 Requirements Specification Processes
According to the STARTS guide (1987), the software specification activity essentially
comprises four requirement definition processes: 'acquisition', 'expression',
'analysis', and specification'. These terms are defined by the STARTS guide as:
"• **Requirements Acquisition:** the acquisition, or capture, of information directly from the purchaser organisation.
• **Requirements expression:** the expression of purchaser statements in a notation which elucidates their implications, prompts further questions, correlates different aspects and facilitates detailed analysis.
• **Requirements Analysis:** the analysis of the expressed requirements for internal consistency, completeness and precision. The analysis may also examine alternative design options, allocation of functions to system elements and costs versus benefits.
• **Requirements Specification:** the consolidation of all relevant information gleaned during the above processes into a well structured format to serve as the definitive statement of what the required system should do and the basis for further system design." (1987:195)

STARTS also argues that the software "requirements evolve at an uneven pace and tend to generate further requirements from the definition processes" (ibid 1987:195). Thus, the progress of requirements definition can not be linearly defined in terms of progression from requirements acquisition to software specification. The (interactive) complexity of these requirements definition processes is shown in the following figure:

In order to understand the complexity of requirements specification processes, it is perhaps necessary break down the activities of these processes into small parts. We would like to suggest that these interactive processes can be grouped into two overlapping activities: requirements elicitation and requirements understanding and specification. The requirements elicitation activity will involve:

and the activity for requirements 'understanding and specification' \(^4\) will involve:

Our emphasis on this simplified view for requirements specification processes, in fact, highlights two significant aspects of the interaction that take place between an

---

\(^4\) Although we have described understanding and specification as one activity, there are two activities involved here: *understanding and specification*. However, there are three reasons of this simplification: (i) the whole purpose of the requirements definition activity is to specify system requirements, hence, from an analyst's point of view this may be one activity but it must be mentioned separately; (ii) during specification it is quite probable that more understanding of the system requirements can be achieved; (iii) we intend to present elicitation and understanding as language based skills of an analyst and propose techniques to mimic them for requirements acquisition, expression and (partly for) analysis.
Chapter 1

Introduction and Overview

analyst and a client:

(i) A variety of skills are required on part of the analyst in this interaction (e.g. memory, representation, imagery, learning and so on).
(ii) The medium in which much of the interaction takes place is natural language.

The natural places to study these disciplines are the fields of Psychology, Philosophy, and Linguistics, especially, in the context of language 'acquisition', 'comprehension' and 'production' mechanisms (Clark & Clark 1977). Moreover, the framework where these disciplines are applied for various computer applications is natural language processing (NLP). We believe that the techniques used in NLP for natural language analysis can also be used for requirements elicitation and understanding. The above observation has led us to specify a toolkit which takes into account the behaviour and skills of the analyst, specially, for performing certain knowledge acquisition and information retrieval activities during requirements definition.

The knowledge acquisition techniques for an expert system have a close behavioural similarity to requirements acquisition, expression and analysis. We believe some of the knowledge acquisition techniques can be used for requirements elicitation purposes. We discuss knowledge acquisition techniques separately.

The client produces most of the documents in natural language for internal and external communication in an organisation. Interviews for the software requirements acquisition are transcribed in natural language. A vast amount of information for a proposed system is available in natural language. We believe that the application of the available information retrieval strategies in a system, specially tuned for retrieving the information needed to specify a system, can help an analyst considerably during requirements definition stages. Figure 2 shows our view of the requirements definition activity, and the individual tasks with the disciplines which support them. These disciplines are introduced separately in the following few paragraphs.
Knowledge Acquisition Techniques: Buchanan et al. (1983) define knowledge acquisition as "the transfer and transformation of potential problem-solving expertise from some knowledge source to a program." Knowledge acquisition is the process by which 'facts, rules, patterns, heuristics, and operations' used by humans to solve problems, in the particular domain are elicited (Janardan & Salvendy 1988:119). Researchers in the field of expert systems have produced many techniques for knowledge acquisition such as 'Protocol Analysis', 'Ontological Analysis', 'Structured Analysis of Knowledge', 'Repertory Grid Analysis' and 'Interviewing Techniques' for knowledge elicitation purposes.

These knowledge acquisition techniques are based on psychological theories and observations of the behaviour of an expert. These techniques are regarded as useful in the context of the expert's knowledge structures, thinking processes, heuristics etc. of an (human) expert. We believe that these techniques can be considered for acquiring system knowledge for a conventional system.
Information Retrieval Strategies: An information retrieval system (IRS) deals with the representation, storage of and access to information from documents (Salton & McGill 1983). The information stored in IRS is in natural languages, either in the form of (full) documents or document excerpts and abstracts. Information retrieval strategies in IRS are generally based on (statistical) text analysis and automatic indexing (ibid 1983:52). These strategies are used in conjunction with various other natural language processing techniques such as parsing, text comprehension in a typical IRS. The output of an IRS, in response to a search request, consists of sets of references or definitions. These definitions and references are intended to provide information on items of potential interest to a IRS user.

During requirements definition, an analyst requires various items of knowledge of a proposed system. These knowledge items, according to the STARTS Guide and IEEE standards for software specification, include 'functions or processes', 'performance', 'interface', and 'design constraints' of a proposed system. In a real world situation, an analyst has to seek definitions or explanations, and is involved in identifying and elaborating the system objects. The sources of such knowledge available to the analyst typically include interview transcripts, organisational letters, reports, or documents related to the organisation. A typical use of an IRS can be envisaged for an analyst who analyses a variety of documents during requirements elicitation and domain understanding. It is, therefore, necessary for such a dedicated IRS to recognise the specific needs of an analyst, and hence to know how to deal with the requirements definition activity.

Natural Language Processing (NLP): NLP is an umbrella term for a diverse range of computer programs which simulate topics such as language acquisition, comprehension, conceptualisation and discourse analysis. Notice the terminological similarities between various areas in natural language processing to the terms used above for software specification processes. For example, language 'acquisition in NLP' is analogous to requirements elicitation, similarly the term 'comprehension' (and conceptualisation) in NLP is cognate for understanding in requirements definition activities. The usage of these terms for requirements definition activities, in fact, will allow us to use the results of the research activities in the fields of linguistics, philosophy, psychology and cognitive sciences.

In common with other sub-branches of AI, knowledge encoding and decoding are the main issues in natural language processing. This give rise to many schemata for discourse representation; namely, 'frames', 'scripts', 'episodes', 'plans' and 'semantic
networks"[5]. Many question-answering systems were made with help of these schemata, which provide a significant insight into the syntactic and semantic issues involved in natural languages. In 1984, John F. Sowa proposed conceptual graphs as a meaning representation schemata in his seminal book on "Conceptual Structures: information processing in mind and machine." Sowa talks about Philosophy, Psychology, and linguistic theories in this book before suggesting using his conceptual structures for meaning representation. We believe that the research work of 'psycholinguistics'[6], particularly in language 'acquisition', 'comprehension', and 'production', has a profound influence on the notion of conceptual structures in Sowa's book. This provided us with the inspiration to use conceptual graphs in the field of requirements definition activities.

1.3 Overview and Structure of The Thesis
We have confined our research activities to an early stage of software development generally known as requirements definition phase. The work in this thesis is focused on requirements acquisition, expression and (partly) analysis. Our research activity was initially focused on building an expert system for requirements definition stage. After having found that an expert system approach is not feasible for requirements definition phase, we have focused our attention on the simulation of certain (cognitive) skills essential for an analyst to perform requirements definition activities. The techniques we propose in the above simulation involve: knowledge acquisition techniques (for system knowledge identification and acquisition); information retrieval strategies (searching for key concepts underlying the user's domain directly from text); and a natural language semantic schema (for formalising the user's domain knowledge) to be used in subsequent analysis. The following three points have emerged in the course of this research:

(A) An expert system approach for solving problems in the early stages of software development is not feasible. This we believe to be particularly true for requirements acquisition, expression, analysis of software requirements. Principally, this is because the problem-solving activity for requirements definition is not (entirely) 'rule-driven' as compared to other disciplines such as Medicine or Geology. Also, there is more than one domain involved for

5 References and details can be found in Rosenberg (1980).

6 Psycho is from the Greek word psyche, the word psycho means mind or soul and is used in English as prefix in order to form other words refer to things connected with mind or with mental processes [(OXFORD 1986), (COLLINS 1987)]. Language behaviour in humans is an old topic in Psychology. The term 'psycholinguistics' first began to be used in the early 1950s, and is generally used to indicate a concern with linguistic methods for describing the outputs of language users.
requirements definition activity: application domain knowledge, analyst's experiential knowledge, RA tools and techniques knowledge. The problem-solving techniques available in the last two domains are intentionally general purpose (because these problem-solving techniques are not meant for one application). Our attempt to build an expert system were among the first which led us to document this knowledge with expert interviews, which was subsequently confirmed by the industry and vendors (see Chapter 3).

(B) Natural language is the principal medium used to transfer knowledge about a software system from a client to an analyst (or vice versa). Though the client and the analyst apparently speak same language, a language barrier exists due to the difference in semantics of their languages: the client speaks and understands the language of his/her domain of expertise, and the analyst is computer literate. In order to break the language barrier, we believe that some special techniques are needed. A possible solution may be to use a linguistic framework for software requirements acquisition, expression and (partly for) analysis. This approach will enable an analyst to elicit requirements and understand the domain in a manner which preserves most of the information available from the client.

(C) To provide an engineering approach to the above mentioned linguistic framework for an analyst we require three things: (i) a language based semantic schema; (ii) information retrieval strategies for extracting the relevant information directly from the application domain text; and (iii) knowledge acquisition techniques for acquiring the relevant knowledge of a proposed system. This framework will then be useful in eliciting the client's requirements and understanding the application domain directly from the text available for requirement definition activities. We believe that from such an approach an analyst can understand the application domain quickly.

In order to provide a language based toolkit for an analyst which exclusively operates within the early stages of software development and we have developed a prototype which contains various natural language processing facilities. This prototype is implemented in Quintus Prolog running under a Unix [7] environment on a SUN SPARC STATION. Briefly, our system is capable of identifying the application domain vocabulary, extracting 'relevant' sentences which include the identified terms; and, subsequently 'animates' these sentences using Sowa conceptual graphs.

---

7 UNIX is a registered trademark of AT&T Bell Laboratories.
Structure of the Thesis
This introductory chapter has established the background to this thesis and highlighted various topics in requirements definition, AI, and natural language processing fields. A detail discussion on various topics in these fields can be found in literature survey (Chapter 2). This chapter principally deals with various techniques and methods used in software development and requirements definition, and also includes topics such as knowledge acquisition techniques, information retrieval strategies, and representation schemata for natural language semantics. Chapter 3 describes the knowledge acquired from two expert analysts and the verification of this knowledge by a group of analysts for software requirements definition activity. Chapter 4 is devoted to our contribution to the subject of requirements definition, where a consolidated language based approach is discussed for requirements elicitation and understanding. Some results of the analyst's toolkit are discussed in this chapter. The conclusion of this research is presented in Chapter 5 with comments on future work.
Chapter 2: Background

Four topics are discussed in this chapter: Section 2.1 deals with requirements analysis practices covering some analysis techniques and methods well known to an analyst during software requirements definition activities; Section 2.2 deals with novel tools in requirements definition; relevant aspects of natural language semantics are discussed in Section 2.3; knowledge acquisition techniques and information retrieval strategies are discussed in Section 2.4. Concluding remarks for this chapter are presented in Section 2.5.

2.1 Current RA practices: Techniques and Methods

There are various so-called classical techniques and methods available for the software requirements definition activity. This section introduces their important features, aiming to find out their relevance to software requirements acquisition and expression. Our study supports Finkelstein and Potts' argument that "most work on methods and tools has been technology-driven rather than problem-oriented ..." (1985:4). In particular, existing techniques for the requirements definition activity appear less appropriate in dealing with early stages of requirements definition, especially for requirements acquisition. Conventional software developments methods approach requirements analysis in a manner which appears more suitable to the software design (or pre-design) activity. This can constrain an analyst's attempt to understand the application domain objectively or globally.

In following sections (Section 2.1.1 and 2.1.2), we investigate the objectivity of techniques and methods to software requirements definition respectively.

2.1.1 Notational Techniques

A formal notation is a means of recording information about a system (STARTS 1987:22). In this section, we discuss six well known notational techniques in the domain of requirements definition. These techniques are used to convey a functional understanding of the proposed system and play an important role in requirements definition whether used manually or semi-automatically as a computer aided tool or under the guidance of a method. The techniques include: Data flow diagrams (Demarco 1978); Structure charts (Yourdon & Constantine 1979); Entity-relationship diagrams (Chen 1976); Finite state diagrams (Birrell & Ould 1985, Salter 1976); Decision tables or decision matrices (Chvalovsky 1983, Hurley 1983, Metzner & Barnes 1977); Data dictionaries (Demarco 1978). In the following we briefly outline each technique and comment on the efficacy of the technique for the requirements definition activity.
(i) DATA FLOW DIAGRAMS (DFD)
The Data flow diagrams are intended to decompose the system into its constituent parts. The constituent parts highlight the component processes of the system that make up the whole, and establish interfaces among the components. In the case of a large target system, several successive decompositions may be required. This is accomplished by lower-level data flow diagrams for fine details. All one-level DFDs are combined into a levelled DFD set.

This approach appears to be popular in the software development community, and automatic or semi-automatic software development tools are frequently available (see Section 3.6). There are, however, three potential drawbacks of data flow diagrams: (a) this approach is generally used to model static behaviour of a software system (i.e. control constructs are not available in this technique); (b) due to the cluttering effects, only the person who has created the diagrams can correctly interpret them; (c) due to the nature of this technique, all domain dependent system objects must be represented either as processes, data, files, sources, or sinks; this can constrain an analyst's thinking.

(ii) STRUCTURE CHARTS
A structure chart represents a hierarchy of modules in a system, which call one another by passing data and control between them. Structure charts are generally known as a convenient means of converting a system knowledge already presented in data flow diagrams into a series of instructions, which can be executed hypothetically or through an automatic aid.

In the context of requirements definition, structure charts provide a higher level of abstraction. This representation is similar to Jackson's structured diagrams, where simple tree structures are converted into more informative but less clear graphical structures. Structure charts present the dynamic behaviour of a proposed system at module level, but they are inappropriate for expressing the static properties of a system.

(iii) ENTITY-RELATIONSHIP (E-R) DIAGRAMS
Entity-relationship models or diagrams are the representation schemata generally used in the context of 'data modelling'. They describe both the types of data (as names of entities) and the relationships between such types, where various classes of relation (i.e. one-to-one, one-to-many and so on) may be characterised. These appear to provide a good picture of the structure of system data as understood by the analyst; inputs and outputs of a system can also be identified as entities in this technique. However, the main drawback is that E-R diagrams do not express system functionality.
or processes.

(iv) STATE-TRANSITION SYSTEMS
Under this heading, the most frequently discussed technique is that of Finite State Machines (FSM). Here, the analysis involves identifying the states of a proposed system, which are then transformed into new states after some activation or input. The FSM formalism has been extensively used by Salter (1976), whose main concern was to specify data processing subsystem requirements. Chow (1978), and Braun and Givone (1981) have used this formalism basically for (mechanically) checking the correctness of client requirements and for optimising performance.

Another variant of state-transition systems is called Petri Nets. Here, the notion of analysis is similar to FSM. For example, the reader is referred to Alford & Davis (1981) – where Petri Nets used in conjunction with SREM method and to Balkovich & Engelberg (1976) – where Petri-Nets is used in the investigation of data processing performance requirements.

Specification in the form of state-transition systems requires considerable understanding of the client domain in order to check the system mechanically for correctness or for optimisation, which seems difficult at the requirements definition stages.

(v) DECISION TABLES
A decision table represents the conditional logic of processes in a system. This is a practical technique to analyse the logic where actions are taken depending upon the occurrence of a particular combination of circumstances. Again, however, complete domain knowledge is required to tabulate system rules in the form of if-conditions and then-actions.

(vi) DATA DICTIONARIES
A data dictionary is an important concept in requirements analysis, in that it serves to store the facts and data about the system which emerge from the above mentioned techniques. However, data dictionaries do not introduce any constructs of their own for requirements elicitation and understanding.

In conclusion, the above mentioned techniques involve the use of specific (design-oriented) constructs for requirements elicitation and understanding (e.g. the DFD technique envisages the use of Processes, Dataflow etc., and the E-R diagramming
technique models a proposed system in terms of entities, attributes, and relations). These techniques appear to be limited in whether they can help in elicitation and understanding the control knowledge say in DFD, and the process knowledge in E-R diagrams.

The next section deals with three methods available in the domain of requirements definition. These methods are discussed for their software development processes or activities in SADT notation.

2.1.2 Classical Requirements Definition Methods

Under this heading, we explore three methods designed to support the software requirements definition activity. A substantial body of literature is available on these methods showing their significance, relevance to various stages of software development, and suitability to software type (see Birrell & Ould 1985, STARTS Guide 1987 for details). For a comparative study of these methods we have used a unified diagrammatic description, particularly to understand their significance to software requirements definition activity.

We start our discussion with SADT (Structured Analysis and Design Techniques) and its ability to communicate, then proceed with the meta-level description of well known requirements definition methods, such as JSD (Jackson System Development) and CORE (COnrol Requirements Expression). The important aspects of each method are high-lighted using the SADT notations. For example, SADT and JSD can be regarded as two ends of a spectrum. Methods like SADT emphasise functional decomposition of a proposed system into activity diagrams, whereas the JSD method "eliminates the need for requirements [acquisition and expression]. Instead, the analyst models the relevant aspects of the real world" around a proposed system (Finkelstein & Potts 1985:3). The CORE method lies in between the two in that it provides functional decomposition like JSD's Entity Action Steps, and Combined Action Diagrams similar to SADT activity diagrams (Finkelstein & Potts 1985:35-45).

2.1.1 The SADT Method

Ross (1977), the originator of Structured Analysis and Design Techniques (SADT) stresses the need for a hierarchical decompositions of the application domain in software specification and design. This top-down decomposition with structured synthesis is the basic theme of a Structured Analysis . This aim of SADT is not limited to requirements definition nor even to system problems. The end product of a definition in SADT is supposed to be a working model of a well-structured scheme. Thus a requirements definition in SADT, as proposed by Ross, had its roots in a
"universal, standard pattern or process which appears to permeate all of software engineering and problem-solving in general" (ibid 1977:19). This pattern can be discussed in terms of: stating purpose; formulating concepts; seeking mechanisms of implementing concept and purpose; using adequate notation for expressing mechanism; and consistent usage of rules for notation.

There are forty graphical notations (or labels) used in SA language. These notations are used for various behavioural representations of a system in SADT (e.g. interchange, uniqueness, interface to parent diagrams etc.). However, the system's constructs which are represented by these notations are mainly concerned with system 'activity or process', 'input', 'output', 'mechanism', and 'control'. The following diagram shows how these constructs are used in a typical representation:

[Diagram: An activity box]

Eight years later, Ross (1985) presented an elaboration of his graphical notation for a structured analysis box, where an activity has been broadly modelled as: "... input, control, output, and mechanism. Input is transformed into output under control. The mechanism is what carries out the transformation. An algebraic analogy would be A*X+B=Y, where X (as argument) is input, A and B (as parameters) are control, and Y (as result) is output; the arithmetic itself is mechanisms (the "*" and "+" being specific operators), and the formula as a whole is the box. Note that both inputs and controls (collectively called entries) participate in the transformation to output." (1985:26)

[Diagram: The Structured Analysis box]

Any task in SADT can, in principle, be structurally decomposed into sub-activities
to handle the complexity of the application domain. According to Ross, activities should be arranged on single sheet of paper, with no more than six boxes.

Over the past 13 years, SADT notations appear to have influenced a number of researchers in the requirements definition field. For instance, Greenspan (1984) has designed a 'Requirements Modelling Language' (RML), which is based on SADT models of software specification. This author has claimed that "RML is designed to be compatible with SADT so that it is relatively straight-forward ... to proceed from an SADT model to a RML model." (Greenspan 1984:10) However, we have noticed in Greenspan's work that the requirements acquisition activity has not been considered before RML modelling. This system modelling in RML basically starts from a system functional specification then proceeds to an SADT specification and to RML modelling.

Various other tools now support SADT notations for requirements definition, some of them are mentioned in Ross (1985). More recently Hughes et al. (1988) have used the SADT formalism for software specification in an ESPRIT sponsored project.

In the following we have used the SADT 'word-and-arrow' notations to illustrate the principal features of JSD and CORE methods. However, it is not our intention to advocate SADT as a specification technique for requirements definition: we merely use it as a consistent diagrammatic notation.

2.1.2 The JSD Method
Cameron (1986) has claimed that the Jackson System Development (JSD) method is applicable to most of the software development stages (e.g., requirements, design, implementation, etc.). A JSD specification consists mainly of a distributed network of processes that communicate by message-passing and by read-only inspection of each other's data. Specification is developed middle-out from an initial set of 'model processes.' These model-processes define a set of events, which limit the scope of the system and in turn help to define its semantics. These model-processes form the basis for defining data and outputs (Cameron 1986:222). There are three main phases in the JSD method:

1. The Model phase in which the model processes are selected and defined
2. The Network phase in which the rest of the specification is developed
3. The Implementation phase in which the processes and their data are fitted on to the available processors and memory.
JSD has been discussed extensively in the literature (Birrell & Ould 1985, Cameron 1986, Jackson 1983). In the above figure we have highlighted some key activities in the context of requirements analysis in JSD (i.e. Figure 5 shows these activities in the form of boxes), where we intend to discuss only the Process Modelling (box 1, 2, and 3) and Network phases (Box 4, 5, and 6) of JSD. Note that the other phase, 'The Implementation phase', is particularly relevant to system design and implementation.

The modelling phase of JSD deals with 'entities' and their associated 'actions' (e.g. 'user', 'sampling rates' as entities and 'capture', 'process', as actions). The 'Entities Actions Identification' step (in Figure 5) provides a complete listing of atomic and instantaneous actions, which is also useful in capturing the scope of the system and is an important part of process modelling. Since JSD does not start with the functional requirements of a proposed system, in the second step, 'Subject matter purification step' the entities related to the functions of a system are deleted. The output from this stage provides a confined domain of the real world in which the system will work. The entire life-span of each entity at process level provides the process modelling in JSD, for which Cameron (1986:237) has argued that: "one major purpose of the modelling phase is to establish a basis for understanding and discussing the outputs of the system. That basis consists of the events in the JSD model, their orderings, and it is used not just to define the system outputs but also the data stored by the system and all the terms used in discussions with the users." Each process model then takes the form of a structured diagram showing the actions of the entities in terms of sequences, iterations, and selections in a subsequent time ordering.

The subsequent steps, in the 'network phase', produce elaborated System Structured Diagrams (SSD). These show the processing of messages communicated with the real world. The network diagrams consist of sequential processes, to which are then added the required system timing and functions. These steps are shown in the above diagram as steps 4, 5 and 6.

JSD is regarded as 'middle-out' (Cameron 1986:238) or 'not a top-down' (Birrell & Ould 1985:92) method of software development. Jackson's view is that top-down development implies hierarchical decomposition, which can be a good approach to systems which are already understood. However, as Jackson argues, top-down development is the worst possible way of proceeding where you are forced to make decisions at the time of greatest ignorance.

One of the interesting features in JSD is that of portraying the structure of the application domain through the structure of the system specification. Contrast this view
with the functional decomposition view of the specification, where the systems are
specified as functions 'mapping' from their inputs to outputs, which defines only the
external behaviour of the system (cf. STARTS 1987:179). The second major aim of
JSD modelling is to tackle the downstream maintenance of the system specification
which frequently arises due to changes in the functional requirements of the system.

The JSD approach "breaks down for data and relationships that cannot reasonably
be defined in terms of histories of events, for example, for a database describing
general compounds and their relationships. Except for correcting errors, such data are
never changed. Data only need to be changed when something has happened (i.e., an
event) that makes the current version inaccurate. The restriction to systems whose
databases (or equivalent) evolve is not severe. Still, some static portions of an
otherwise evolving database may not be amenable to JSD approach" (Cameron
1986:228).

2.1.3 The CORE Method
The COntrol Requirements Expression (CORE) method is one of the few methods that
suggests "step by step techniques for deriving definition of the [software]
requirements" (STARTS 1987:236). The CORE method consists of a sequence of
steps which are intended to elucidate the user's view of the functional architecture of the
proposed system and its operational environment (Kramer, et al. 1987). CORE offers
refinement of the specification through an iterative cycle of requirements acquisition
consisting of collecting relevant system information, formalisation by proposing data-
process relationship, and verification by proving data-process relationships.

CORE starts its requirements definition by first considering a 'viewpoint hierarchy'
of the system. A 'viewpoint' essentially reflects the perceptions of the system, for
example, by a system manager, a quality manager, a specialist, an end-user, an
operator and so on. This viewpoint hierarchy is quite similar to the entity structure
(Process Model) of JSD. However, the hierarchical structure in CORE is based on
different 'viewpoints' (i.e. decompositions) of a system as compared to 'entities' in
JSD. Each viewpoint is identified by a specific level number. These level numbers are
further used in the identification of nodes in the structure.
In the next step (see Box No. 2, Figure 6) of the CORE method, the analyst produces the viewpoints through data flow tables by considering sources, inputs, actions, outputs and destinations. Internal consistency checks are carried out in this step for each table by asking questions such as: "Does each input have a source? Does each output have a destination viewpoint? Are all actions for the viewpoint shown?" (Birrell & Ould 1985:75).

The third step in CORE's specification is to check the external consistency by considering the system as a whole. The questions asked at this stage are: "Do all sources and destinations match? Are the tables at this level consistent with those at the level above? Are all data names used on the tables consistent?" (Birrell & Ould 1985:75)

The individual relationships between viewpoints, as 'threads' (i.e. data-process relationships), are proposed in step 4 (see Box 4, Figure 6). The notations used in this step are similar to those of SADT, where the (functional) requirements of the proposed system emerge as process activities. By combining the individual thread and operational views, CORE tries to prove data-process relationships in step 5 (See Box 5, Figure 6). Here the analyst is required to check the diagrams for reliability, cost, integrity, error handling and so on, and to modify the diagrams if necessary.

In this section, we have discussed a number of techniques and methods and their specific constructs for the requirements definition activity. These methods and
techniques are mostly concerned with definition of system requirements in terms of specific constructs such as input, output, process, entity or viewpoint, which are not specific to the domain terminology used by the client. There is always a chance of misrepresenting the domain terminology in these constructs. We believe that the use of these techniques and methods do not guarantee an unambiguous representation of the user intentions for the system. Moreover, when these techniques and methods express software behaviour in their specific constructs, it can be hard for the client to validate his or her requirements objectively, unless the client has substantial experience of the usage of these techniques and methods.
2.2 Novel Techniques and Intelligent Front-ends
The foremost use of artificial intelligence in computer programs which solve problems in a specific domain (i.e. an expert system) can imply that the task of 'requirements definition activity' as a 'problem' and its successful 'specification' as solution is an achievable task in an expert system embodied with the (problem-solving) knowledge of a successful (human) expert in the field of requirements definition. This 'expert system' hypothesis appears plausible at first sight, and indeed there are a number of 'expert systems' which claim to provide assistance in definition of software requirements. To test this hypothesis we examine four systems: Kramer et al.'s TARA system (1988), Loucopoulos & Champion's Requirements Specification tool (1988), Hughes et al.'s ASPiS system (1988), and Ohnishi et al.'s Requirements Model (1985) in Subsections 2.2.1 to 2.2.4 respectively.

Our discussions will show that these systems are essentially 'front-ends' either for a well established method or they provide an intelligent front-end for storing and retrieving systems's objects for specification in a restricted language. For instance, TARA acts as a front-end or advisor for the CORE method, and ASPiS for SADT, and Ohnishi's essentially provides a restricted language (natural-cum-programming language in Japanese) for requirements statements. It has been noted in the literature that the boundary between requirements definition and design is "fuzzy" (STARTS 1987), this is evidently found true in most of the novel approaches in software development. We believe, however, that a systematic approach to requirements definition activities will reduce the design efforts for software development.

2.2.1 Tool Assistance for Requirements Analysis: the TARA System
The main reason for software project failure is misunderstanding at the specification stage. This reason has led Jeff Kramer and others (1988) to devise an intelligent system on the top of an analysis tool which was designed to support a method[1]. The two aspects of an expert analyst's expertise (experiential knowledge) were reported to have been incorporated in this project are as follows:

- **Method Guidance** knowledge is stored in the form of a *normative model* for active (run-time) advice and a *remedial model* to provide remedial strategies to support the method steps.

---

1 This analyst tool is called ANALYST (Stephens & Whitehead 1985) and supported by System Designers Plc UK. ANALYST is an interactive software tool which supports the CORE (Mullery 1979) method. This tools is designed to provide a basic set of clerical activities for storing and presenting CORE specifications graphically.
• **Animation** to show the dynamic behaviour of the system at the specification level following the method knowledge.

*Method Guidance:* Considering the role of methods during requirements definition, Kramer et al. pointed out that "many methods are little more than collections of representational techniques" (1988:86). These techniques tend to provide the practitioner with notations for representing the specification and perhaps some procedures for validating them, however, most of them do not impose set procedures for guiding an analyst during requirements specification.

Like other application-specific decision-support systems, a decision-support system for software requirements definition will enable a (novice) user to undertake the task of requirements specification correctly and efficiently. This efficiency and correctness will be based on knowledge of experts who have a history of performing the task well. Perhaps in such cases the experts have the intuitive experiential knowledge, and also the knowledge of usage of the tools and techniques. If the tools are designed to perform a task in a prescriptive (or procedural) fashion then it appears that the knowledge of how to use the tools can be encoded as "rules".

The CORE method is generally known as a prescriptive method. Kramer et al. consider the method knowledge for guidance in two different situations. The normative model consists of a description of the steps needed in order to formulate a specification using the method criteria. However, when the user deviates from the normative approach during specification, the second model should in theory detect the abnormalities and give advice on remedial actions. This model thus needs heuristic information from the methods expert in such situations.

*Animation:* Kramer et al. have used the term 'animation' "to indicate dynamic behaviour of the specified system by walking through a specification fragment to follow some scenario of interest". According to these authors, this form of animation allows the user to choose alternative decision paths based on the current state of animation. They further argue that this form of animation is useful for browsing through a specification in the form of computer-aided walkthrough without knowing beforehand "which are the actions of interest". However, this technique is essentially used to cover the validation aspect of requirements specification. A possible sketch for the TARA system is given in Figure 7.
2.2.2 Requirements Specification Tool (RST)

The Requirements Specification Tool (Loucopoulos & Champion 1988) was an ALVEY sponsored project involving a number of different U.K. organisations (Data Logic, UMIST, Scicon, MJSL, ARE and Istel). This project strongly argues that method and domain knowledge is necessary for providing intelligent machine assistance in all stages of software development. Loucopoulos and Champion propose a knowledge-based approach to requirements definition which will lead to an efficient and effective use of tools that aim to assist the analyst.

The main criteria by which a method has been selected for this project, as argued by Loucopoulos and Champion, have been those of "life-cycle coverage, model representations, object system type and philosophy" (1988:180).

These authors have developed a scheme for recording knowledge about any method's model in terms of the method's primitives. These primitives are defined in terms of objects, relationships and functions of the method constructs. An object is a basic component of a model which has an existence in its own right. A relationship is an association between two objects in a model representation. A function is a mapping from one or more domains to a range, or is confined to mapping from pair of objects to the Boolean values of true or false. The selection criteria for these primitives were based around the idea of semantic primitives used in studies of natural language understanding (see Wilks 1977, Schank & Abelson 1977 for details).

It has been recognized in the RST project that the role of domain knowledge in all stages of software development is of great importance. During the formalisation of
requirements specification; this role was emphasised even more in the context of consistency, completeness and the validation of client needs. In the RST project, the domain knowledge and application-dependent facts are coded in the form of conceptual graphs due to Sowa (1984). As to the strength of this suggested approach in requirements definition, Loucopoulos and Champion argue that "It is believed this type of exploratory concept identification and classification is close to the way that analysts work" (1988:185). In conclusion, the authors point out that "a single unifying representing formalism" (1988:186) used in their project has several advantages over the informality which characterises much of the initial requirements elicitation process.

This requirements elicitation approach in RST project uses a blend of (linguistic) semantics and knowledge representation schema (conceptual graphs). The relationship definitions within this approach allow both the domain knowledge and existing application knowledge to be represented in terms of the semantics of the domain rather than the semantics imposed by any design method. An overview of the RST approach, using (JSD) method and domain knowledge, is presented in Figure 8.

![Diagram](image)

Figure 8: Overview of the Requirements Specification Tool approach

### 2.2.3 ASPiS project

The main aim of the ASPiS project is to show the relevance of AI to conventional software engineering techniques, particularly those which are used in the early stages of software development (Hughes et al. 1988). After emphasising the importance of the
analysis phase in software development, Hughes et al. argue that most of the existing tools are aimed at the programming phase of software development. The current version of the ASPIS system supports a single specification method based on Structured Analysis diagrams (Ross 1977), the Entity-Relationship model for data (Chen 1976) and an ad-hoc formalism for non-functional requirements. Other features of the system include multiple viewpoints for the system, consistency checking between various descriptions and a protocol between designer and the analyst by means of marks attached to documents. It is claimed by the authors that the method knowledge is organised in such a way that it can be substituted by any other method knowledge.

The basic architecture of the ASPIS system comprises of four distinct user assistants, namely Analysis Assistant, Prototyping Assistant, Reuse Assistant and Design Assistant (Figure 9). These modules provide a graphical user interface to knowledge bases, where method knowledge, domain knowledge and the method dictated specification language knowledge are stored. In the following, we shall focus more closely on the Analysis Assistant, which appears more relevant to our work.

The main goal of the Analysis Assistant (AA) is to offer a computer based support environment for making the requirements analysis and specification tasks easier for a non-expert analyst. The basic objective of AA is to provide the (expert) analyst's problem-solving capabilities for a novice user. Hughes et al. claim that the AA seeks to provide intelligent assistance during requirements definition by making deductions based on the expertise embodied in the AA knowledge base. The AA provides many kinds of advice based on the method knowledge embedded in the system:

- guidance through the various phases;
- suggestions on the sequence of steps and on the criteria to be observed;
- monitoring of the correctness or the adequacy of the intermediate analysis results;
- automatic generation of requirements documents whenever a human decision is not needed or is implicit within the development context.

The knowledge representation scheme used by Hughes et al. is based on a 'hybrid scheme' comprising frame systems (Minsky 1975) and the object-oriented paradigm[2].

---

2 Borgida et al. (1985) suggest that an object-oriented approach has its origins in AI knowledge representation techniques. Here, certain advantages are achieved such as inheritance, class and sub-class operations, attributes characterisation etc. These features are applied in programming languages such as Simula and Smalltalk.
2.2.4 Requirements Model of Ohnishi et al.

Ohnishi, et al. (1985) have outlined a requirements definition environment based on a requirements model. These authors have addressed a number of software development activities in this project. The environment is primarily designed for a systems analyst, although some provisions are included in the system for a system designer to help him/her to understand the required system. The basic aim of Ohnishi, et al.'s requirements model is to represent the functional behaviour of a proposed system in such a manner so as to check for ambiguity in the user requirements. The authors claim that their environment is also effective and helpful in producing user documentation for the required system.

User requirements statements are first explored for a 'surface structure' by the analyst using the Japanese Requirements Description Language (JRDL). The analyst first identifies any string as an 'entity name' from a requirements statement. JRDL has no 'entity type' but has many reserved verbs. This provides a limited syntax for the description of requirements statements. However, to deal with the semantics of a user description as well as to avoid ambiguities in requirements statements, the following restrictions are also incorporated in the JRDL descriptions:
1. A requirements statement must be in a form of a simple sentence
2. Demonstrative pronouns cannot be used
3. Omission of cases is not permissible

A JRDL description is further processed to explore the 'deep structure' of the user statement with the help of a Conceptual Description Language (CRDL). CRDL uses the user statements to specify 'entity' to 'entity type' relationships, entity to case relationships and also identifies verbs among them. A verb of CRDL corresponds to one viewpoint of data structure: typically, DFLOW corresponds to viewpoints of data flow, and AND-SUB to functional structure. A total of 16 verbs are used to describe a requirement statement in a frame-like manner. These verbs correspond closely to the file handling and other constructs of a programming language (e.g. UPDATE: file update; GEN: file creation; LT: less than; and so on). Having identified the requirements as one viewpoint, the analyst can now view his/her description among other data structures: from 'data flow', to 'data structure', 'functional structure' and so on. This helps him/her to check the proposed system description for functional completeness. The supporting environment decides whether all the essential functions are defined or not by checking the existence of file processing verbs corresponding to the essential functions of a proposed system.

In Figure 10 we describe Ohnishi et al.'s system in detail and show the inputs to and outputs from the components of their system. It is worth noting that the Ohnishi requirements model uses the constructs of programming languages to express software requirements in order to process them for consistency and completeness or to specify a design of a system leading to executable system code. This restricted language approach to requirements definition can constrain an analyst's efforts to understand the domain objectively. It is to note in the Ohnishi et al. requirements model and in its subsequent implementation that the authors have considered the so-called 'Ripple Effect Analyzer' and 'Prototype Generators' (see Box 3 and 4 in Figure 10) as important activities for requirements definition. However, the requirements acquisition activity is not considered in this model and the system development starts with the initial user requirements.
The so-called novel tools, considered as intelligent front-ends for methods, do not adequately handle the elicitation aspect of requirements definition. Most of the above mentioned novel tools are method-knowledge dependent and use a limited number of constructs for requirements definition; hence they pose the same problems as various notational techniques for requirements elicitation and (domain) understanding.
2.3 Natural Language Semantics

In the context of requirements definition, the use of linguistic theories or Natural Language Processing (NLP) is fairly rare. However, the problem of transforming a text into a well-structured data base has received much attention in NLP research. Practical methods of representing the content of a text as a network of linguistic or philosophical units of words or concepts for natural language processing were first introduced by Quillian (1968). These networks of linguistic units with relations are commonly known as 'semantic networks'. According to Greene (1988), the basic idea of the semantic knowledge of a language is that all the words in the lexicon can be defined in terms of sets of semantic features. For example, Katz and Fodor (1963) proposed a theory "in which words are defined in terms of features like animate, inanimate, human, animal, physical object, activity" (Greene 1988:59). In this section, we discuss two formalisms (generally called semantic networks) which claim to deal with semantic features of word meanings. We introduce the first semantic network formalism as the notions of natural language semantics in Section 2.3.1. The second semantic network formalism, generally called 'Conceptual Graphs', is discussed in Section 2.3.2.

We believe that for unambiguous (domain) understanding of a proposed system a common framework is required. Natural language (and a schema based processing) can provide a common framework where it is generally believed that the 'understanding' is always based on the assumption that the speaker and listener share some mutual knowledge and beliefs (Greene 1988:26). The mutual knowledge between an analyst and a client is the common language they speak during requirements definition of a proposed system. However, the client's language and his or her domain terminology (i.e. the so-called user's jargon) can create a problem for an analyst during requirements understanding. Appropriate use of client domain terminology in a conceptual schema and its reasoning strategies can help an analyst to produce unambiguous requirement statements for system development.

2.3.1 Semantic Networks

Semantic Networks are generally used to represent an 'intention' or 'meaning' of a scenario with emphasis on "associative and other non-deductive processes" (Cercone 1980:127). According to Jackson (1989) a semantic network is a form of meaning representation network commonly used to structure a more general kind of information. Note here that non-deductivity and associativity in concepts are the main source of ambiguities in sentences. The use of a semantic network will help to resolve these ambiguities with the help of arcs and labels of a graph as well as formally in a quantifier-
free normal form in predicate calculus (Cercone 1980). Basically, the node-and-arc notations are used to distinguish between storage locations as 'type nodes' and pointers to these storage locations as 'token nodes' of a concept in an artificial memory organisation (Quillian 1968).

Sowa (1984), in his description of semantic networks, distinguishes two types of concepts. First is the so-called 'concrete concept' type, like CAT and TOMATO, which maps directly onto our perceptions. The second type corresponds to those concepts which we do not directly perceive, e.g. 'abstract' concepts like PRICE, FUNCTION, and JUSTICE. The inherent difficulty in defining such abstract concepts can be resolved with the help of concrete concepts (as well as with other abstract concepts). A semantic network is then essentially a collection of concepts, which embodies the relationships of one concept to other concepts. The following figure introduces the organisation of a semantic network used to represent a sentence:

Figure 11: Fragments of a semantic network showing an instance of giving as give "John gives Mary a copy of the book War and Peace" (adopted from Jackson 1990)

The representational and the organisational aspects of semantic networks are not strictly formalised. Over the past 40 years many versions have been proposed and implemented and because of this diversity, the terminology and notations associated with semantic networks vary widely (cf. Sowa, 1984, Cercone, 1980, Jackson, 1990). However, Sowa (1989:1012) argues that there are certain themes which are common to most semantic nets:

(i) nodes in the net represent concepts of entities, attributes, events, and state;
(ii) different nodes of the same concept type refer to different individuals of that type, unless they are marked with a name, identifier, or co-reference link to indicate the same individual;
(iii) arcs in the net, called conceptual relations, represent relationships that hold between
the concept nodes (labels on the arcs specify the relation types);
(iv) some conceptual relations represent linguistic cases, such as agent, object, recipient,
and instrument (others represent spatial, temporal, logical, and inter-sentential
connectives);
(v) concept types are organized in a hierarchy according to levels of generality, such as
ENTITY, LIVING-THING, ANIMAL, CARNIVORE, FELINE, CAT; and
(vi) relationships that hold for all concepts of a given type are inherited through the
hierarchy by all subtypes.

Sowa has also pointed out that this diversity of notation and terminology in network
formalism is due to a number of issues: "philosophical questions of meaning; methods
for representing all the quantifiers and operators of symbolic logic; techniques for
manipulating the networks and drawing inferences; notation and terminology that differ
from one author to another" (ibid 1989:1012).

2.3.2 Conceptual Graphs

Conceptual Graphs evolved as a semantic representational schema for natural language
(Sowa 1983). Sowa claims that conceptual graphs are a form of "knowledge
representation language" based on "linguistics, psychology, and philosophy" (Sowa
1984:69). As a graphical language it is simple and comprises only two types of lexical
entries, concept nodes and relation nodes. The concept nodes represent entities,
attributes, states, and events of the world. The relation nodes show the concept to
concept relationship.

Figure 12 shows a conceptual graph which may be read as "a monkey eating a
walnut with a spoon made out of the walnut's shell" (Sowa 1984:78), where the boxes
represent concepts of entities (monkey, walnut, spoon, shell) and a concept of an action
(an instance of eating). The circles represent conceptual relations: a 'monkey' is the
agent of 'eating', the object eaten is a 'walnut', the instrument of eating is a 'spoon';
and the material, of which the spoon is made, is a 'shell', which forms part of the same
walnut that is being eaten:

![Figure 12 A conceptual graph](image)

The equivalent linear notation, say for output to a line printer, uses square brackets
for concepts like [MONKEY] or [EAT] and round parentheses for conceptual relations
like (AGNT) or (OBJ):

\[
\text{[EAT]} - \\
\text{(AGNT)} \rightarrow \text{[MONKEY]} \\
\text{(OBJ)} \rightarrow \text{[WALNUT:*x]} \\
\text{(INST)} \rightarrow \text{[SPOON]} \rightarrow \text{[MATR]} \rightarrow \text{[SHELL]} \leftarrow \text{(PART)} \leftarrow \text{[WALNUT:*x]}. \\
\]

The hyphen after the concept [EAT] shows a concept where most arcs are combined and usually chosen as head (of a linear representation of a conceptual graph) for simplicity, indicates that the relations connected to [EAT] are continued on subsequent lines. If the graph has the form of a chain, it can be drawn on a single line; if it is a tree, it may be drawn on multiple lines with indentation and punctuation to show a tree structure; but if it contains a cycle, some concept node on the cycle must be repeated, and a variable symbol must be used to show cross references. In case of linear representation of Figure 12, the concept [WALNUT] was repeated, and the variable *x shows that both occurrences of [WALNUT] refer to the same entity.

There are few a restrictions generally observed in Sowa's conceptual graph representation. The first concerns the drawing of arrows. Two relational nodes cannot be connected directly. The direction of the arrows must help to read a graph back to a sentence representing the same meaning as the original. Secondly, the conceptual relations may have any number of arcs, and most of the common ones are dyadic as shown above. However, a few are monadic, such as the past tense marker (PAST) or the negation (NEG) as in:

\[
\text{NOT} \\
\text{PERSON:John} \rightarrow \text{AGNT} \rightarrow \text{GO} \rightarrow \text{INST} \rightarrow \text{BUS} \\
\text{PAST} \rightarrow \text{DEST} \rightarrow \text{CITY: New York} \\
\]

(a) "John did not go to New York by bus".

Others, like between (BETW), are triadic as in:

\[
\text{BETW} \\
\text{BRICK} \rightarrow \text{BETW} \rightarrow \text{SPACE} \\
\text{BRICK} \\
\]

(b) "a space is between a brick and a brick".
Certain conceptual graphs are assumed to be 'canonical' in conceptual graphs (Sowa 1984). Canonical graphs define a 'sensible' relationship between concepts by imposing certain restrictions on the concepts. For example, "a cat sat on the mat" is a 'sensible' human experience, represented by the canonical graph:

\[ \text{[ANIMATE]} \rightarrow (\text{SIT}) \rightarrow \text{[INANIMATE]} \].

This reflects a 'sensible' relation between two concepts in which one is 'animate' and other is 'inanimate', as against a 'non-sensible' experience:

\[ \text{[INANIMATE]} \rightarrow (\text{SIT}) \rightarrow \text{[ANIMATE]} \].

Human experience disallows such a relationship in which an inanimate object is sitting on an animate object (i.e. "the mat sat on the cat"). The canonical graphs are therefore referred to as basic memory structures by Sowa; they present declarative information in conceptual graph theory and are generally helpful in processing new information.

New graphs may become canonical or be canonized by any of the following three process (ibid 1984:91):

* **Perception.** Any conceptual graph constructed by the assembler in matching a sensory icon is canonical.

* **Formation rules.** New canonical graphs may be derived from other canonical graphs by the rules copy, restrict, join and simplify.

* **Insight.** Arbitrary conceptual graphs may be assumed to be canonical.

These canonical graphs have a variety of uses in natural language processing systems. For example, conceptual relations can be determined by a canonical graph. The canonical graphs represent the expected configuration of concepts and relations (Sowa 1984:224). Sowa also argues that canonical graphs can act as basic units in generating a bigger meaning structure with the help of the above mentioned 'formation rules.'

Conceptual graphs can present the semantics of a sentence elegantly, but this representation is not enough to make human-like inferences unless the knowledge is arranged in an 'active network form.' The requirements for such an active network are dealt with by Sowa (1984) through the notion of 'Actors', which are active elements in conceptual graphs that provide a mechanism for message passing. These active
networks can be used procedurally when they are executed, and declaratively when they are treated as a description of functional or relational dependencies. In this formalism the 'actor nodes', when attached to a conceptual graph, form a dataflow graph, where special control markers are used to trigger the actors to compute referents for the generic concepts. This technique was used by Sowa to develop a database query system (see Sowa, 1976, 1983, and 1984 for more details).
2.4 Knowledge Acquisition and Information Retrieval Strategies

The tasks performed by an analyst during requirements definition are comparable to the tasks which are performed during the knowledge acquisition phase for an expert system. Similarly, the needs fulfilled by a modern information system using textual databases can be exploited for an analyst who is seeking various definitions and explanations during requirements definition in a specific domain.

This section introduces various techniques in the fields of knowledge acquisition and information retrieval. In cases where these techniques are successfully used in their respective fields, we would hope to be able to exploit them in the context of requirements definition. However, before discussing knowledge acquisition techniques (Section 2.4.1) and information retrieval strategies (Section 2.4.2) we would like to highlight the specific aims and environments of the knowledge acquisition activity for an expert system as compared to the specific aims and environments of a requirements definition activity for a conventional system.

In requirements elicitation, the aim is to model the real world or a slice of the real world relevant to the required system (Greenspan 1985). In an expert system, an expert's knowledge is generally the target area for acquisition, formalisation and processing (Buchanan et al. 1983). In conventional system development, the current practices and existing systems are normally analysed as constraints for the proposed system: the software requirements in these cases are examined for consistency, completeness and provability. Whereas, the knowledge acquisition techniques are generally focused to exploit the current practices of a domain expert for an expert system.

In the context of expert system development, the role of a knowledge engineer during knowledge acquisition does not (generally) allow him or her to investigate the validity of a solution, or question the expert's problem-solving capabilities. This can be contrasted with the role of an analyst developing a conventional system, where he or she is required to seek an ideal solution. Despite the above dissimilarities in the aims and in environments, the problem-solving capabilities of a so-called 'knowledge worker'\[1\] within a conventional system are not dissimilar to an expert's problem-solving capabilities in an expert system. Despite differences in the degree of experiential knowledge or expertise, we can regard the knowledge of a common worker in an organisation as a subset of an expert's knowledge.

---

1 The term 'knowledge worker' is used here in an organisational context, where every worker in an organisation is referred as knowledge worker. The precise term of course is 'end-user'; however, individual end-user knowledge is likely to be only a 'fragment' of expert knowledge.
In this section we identify a number of knowledge acquisition techniques which are used to develop an expert system (e.g. protocol analysis, ontological analysis, structured analysis of knowledge, and various interviewing techniques). These techniques can be used for conventional system particularly when we aim to produce a software specification. Similarly, the information retrieval strategies use a variety of techniques for measuring the importance of a term through various statistical inferencing, and for discriminating between terms by exploiting criteria such as association and synonyms. These techniques can be used for an analyst who is seeking machine assistance during requirements definition.

2.4.1 Knowledge Acquisition

Buchanan et al. (1983) define knowledge acquisition as "the transfer and transformation of potential problem-solving expertise from some knowledge source [an expert] to a program [in our case 'a specification']." Researchers in the field of expert systems have developed many techniques for knowledge acquisition.

Knowledge acquisition/elicitation is the process by which 'facts, rules, patterns, heuristics, and operations' used by humans to solve problems in the particular domain are elicited (Janardan & Salvendy 1988:119). 'Knowledge acquisition' is a generic term, as it is neutral with respect to how the transfer of knowledge is achieved. For example, it could be achieved by a computer program, that learns to associate symptom sets with diagnostic categories by processing a large body of case data. The term 'knowledge elicitation', on the other hand, often applies to the transfer of knowledge, which is normally accomplished by a series of interviews between domain expert and a knowledge engineer. Jackson argues "the term [elicitation] could also be applied to the interaction between an expert and a program ..." (Jackson 1990:219). However, in the following discussion we will not distinguish between the two terms.

During requirements acquisition for conventional systems, an analyst transforms the user(s) unspecified but working model of the required system into a model on paper, which can be criticized or improved subsequently, and later used in the specification of a proposed system. Knowledge as such, which is acquired in knowledge acquisition/elicitation (for an expert system and used to solve a problem) or in requirements elicitation (for system understanding and specification) have a close semantic correspondence. In the following discussion, the two terms may be taken as equivalent.
We have argued above that a common worker's knowledge in an organisation can be regarded as a subset of an expert's knowledge. To explore this knowledge, there are at least four acquisition techniques in the domain of knowledge engineering for an expert system:

* Protocol Analysis
* Ontological Analysis
* Structured Analysis of Knowledge
* Interviewing Techniques

These are discussed briefly in the following subsections, with the intention of considering their relevance to the requirements definition stage of software development.

**PROTOCOL ANALYSIS**
Protocol analysis is generally used by a knowledge engineer to find the protocols of a problem-solving abilities of an expert (Diederich et al. 1988). Generally there are three variants available to protocol analyses. The first variant is due to Newell & Simon (1972), the second is described in Kuipers & Kassirer (1983), and the third is mentioned in Diederich et al. (1988). All the three techniques have two distinct phases: first, a domain description is achieved through its implicit structure of the different knowledge states; secondly, the problem solving strategies are identified between different knowledge states.

In the approach discussed by Newell and Simon (1972), first of all it is necessary to identify *phrases* with a high information content, and then such phrases are grouped into different areas of knowledge. Hayward (1988) suggested that the knowledge level or areas are the 'domain level', 'inference level', 'task level', 'logical level', and 'strategic level'. This is equivalent to Kuipers and Kassirer (1983, 1984) approach, where a structural description of the problem domain is first achieved. Once these areas have been defined, it is necessary to show the 'qualitative simulation' of transitions between knowledge states during problem solving. This is achieved by establishing the interrelationship between states, and the criteria for passing information from one area, or state of belief, to another.

Diederich et al. (1988) have used protocol analysis technique in KRITON, the knowledge acquisition system for eliciting the expert's procedural knowledge. In KRITON, the protocol analysis is accomplished in five steps. First, the transcribed conversation is partitioned into segments on the basis of the expert's speech gaps
during recording. The second step is the semantic analysis of the segments, creating propositions for each segment. In the third step, the appropriateness of the selected operators and arguments is validated. Next, a so-called 'knowledge-base matching' is attempted, to instantiate variables inside the propositions (where variables are inserted for pronouns). In the last step, propositions are arranged according to their appearance in the natural language protocol.

The aim underlying a protocol analysis is to structure the knowledge of the domain in order to systematize the knowledge acquisition process. This is similar to the techniques available in methods like JSD or CORE in the context of requirements analysis, where the entities are identified on the basis of their actions and evolve in structures by the process of refinement.

The methods and 'tool systems' based on protocol analysis include: 'Three-phase Method' by Grover (1983), DELPHI method (Jagannathan & Elmaghraby 1985), Crawford Slip Method (Rusk & Krone 1984). The 'tool systems' which use this technique as a tool include PURDUE (Gaines & Boose 1988), and KRITON (Diederich et al. 1988).

**ONTOLOGICAL ANALYSIS**

"An ontology is a collection of abstract and concrete objects, relationships and transformations that represent the physical and cognitive entities necessary for accomplishing a task" (Alexander et al. 1988:26). Ontological analysis is based on a taxonomy of knowledge types which are generally classified as *static*, *dynamic* and *epistemic* knowledge. This technique is used for the preliminary analysis of a problem solving domain during elicitation. Alexander et al. (1988) and Freiling et al. (1986) have discussed ontological analysis in some detail, and the following discussion is based on these two papers.

*Static ontology:* defines the physical objects, or primitive objects in a problem space, their properties and relationships. Alexander et al. (1988) have discussed this analysis as it begins with the enumeration of physical objects in the problem space and identifies their inherent properties and relationships. At this level the analysis performed is quite similar to the analysis in the entity-relationship technique due to Chen (1976).

*Dynamic ontology:* defines the state space of the problem-solving domain, and the actions that transform the problem from one state to another. The solution of a problem, in dynamic ontology as discussed by Alexander et al. (1988), consists of
selecting operators whose application transforms the current state into another terminal state. Dynamic ontology defines a problem space in terms of configurations of elements from static ontology, and then defines problem operators as transformations built on the domain of problem states. The dynamic ontology defines which knowledge is unchanged throughout the problem solving process (i.e. organizational charts) and which knowledge changes when the problem is solved (i.e. schedules and meeting plans). (1988:27)

**Epistemic ontology:** defines the constraints and methods that control the use of knowledge applied to the static and dynamic ontology. The epistemic ontology usually contains two different types of knowledge structures. Some are used to select which operations should be performed. Others control the actual performance of certain operations. The epistemic ontology appears to be needed to guide the classification of heuristic operations in the dynamic ontology. A notational language has been designed by Alexander et al. (1988) based on the concept of ontological analysis. The language consist of two types of statements. The first consists of domain equations and the second defines domain function declarations.

The approach in dynamic ontology is similar to that of identifying the state spaces of a problem domain and operators that change the state of a system in Petri-Nets and Finite State Machines description of a system. Tokenisation and its activation present a similar constraints to state-transition (in Petri-Nets) as available in Epistemic ontology.

**STRUCTURED ANALYSIS OF KNOWLEDGE**

Within the framework of structured analysis of knowledge, it is claimed that it is possible to bridge the conceptual gap between knowledge as expressed by an expert and the encoding of expertise in a software system. For the purpose of mapping verbal data onto knowledge, Hayward et al. (1988) have proposed a scheme comprising of five levels:

- knowledge identification;
- knowledge conceptualization;
- epistemological analysis;
- logical analysis;
- implementational analysis.

The following descriptions of each of these levels are based on Hayward et al.'s (1988) interpretations.

**Knowledge identification:** This level of analysis corresponds to simple recording what one or more experts report on their knowledge. Although the result may be in
a formalised form, the representational primitives on which this formalisation is based are linguistic (in the sense that Brachman uses this term). The same knowledge from different experts may have to be represented differently, because they use different terminology, or because their knowledge is structured in a different way.

Knowledge conceptualization: Aims at the formalisation of knowledge in terms of conceptual relations, primitive concepts and conceptual models. The knowledge of different experts, and possibly of different sub-domains, is unified within one conceptual framework.

Epistemological analysis: At the epistemological level the analysis uncovers structural properties of the conceptual knowledge, formalised in an epistemological framework. Such a framework is based on epistemological primitives representing types of concepts, types of knowledge sources, structuring relations (such as hierarchical relations, inheritance) and types of strategies.

Logical Analysis: The level of analysis applied to the formalism in which the knowledge on higher levels is expressed and which is responsible for inference making.

Implementational Analysis: At this level of analysis, mechanisms are uncovered on which higher levels are based. The representational primitives are the ones which are normally used when an implementation of an AI programme is described (e.g. matching, testing slot-filling, etc.).

Under the umbrella of knowledge-level analysis, there are several tools available. A knowledge acquisition tool was described by Breuker & Wielinga (1985) and Hayward (1988) which provides a framework for KADS methodology. Analysis of a domain at the knowledge level as proposed by Newell (1982) has been simulated in a system called OPAL, which allows medical specialists to enter and review cancer treatment plans for use by an expert system called ONCOCIN (Musen et al. 1988).

INTERVIEWING TECHNIQUES

The aim of this section is to investigate interviewing techniques for knowledge acquisition. In expert systems, interviewing techniques are much more important[2] and practically advanced than the techniques normally used in requirements elicitation.

---

2 For an expert system, a prime aim of such interviews is to get the expert's heuristic knowledge, which is generally accepted to be a hard task.
Chapter 2 Literature Survey: KA and IRS

A number of techniques are available in order to make an expert think more carefully in knowledge engineering. Greenwell (1988) has outlined a number of knowledge elicitation techniques for interviewing experts.

Before discussing some of the specific techniques used during interviewing activities, it is useful to classify the interviews. Interviews can be classified according to the different stages of familiarisation of application domain and based on 'where', 'when' and 'what' type of the information is required. These are: the informal or overview interview (where); the focused interview (when); and the structured interview (what), (Greenwell 1988). The style and application of each type of the interview will help to obtain much relevant information at different stages of familiarisation. The following discussion is a brief description of each type of interview and its relevance.

The Informal or Overview Interview: This is aimed at familiarising the knowledge engineer with the application domain. It is therefore likely to be the first interview.

The Focused Interview: Focused interviews are similar to ordinary 'chat show' conversations or discussions where the interviewer is interested in a topic about which the interviewee is knowledgeable. It is normally conducted by following a pre-determined agenda.

The Structured Interview: Structured interviews normally occur well into the knowledge acquisition phase and are conducted when information is required in greater depth and detail. The knowledge engineer will have prepared a list of topic headings rather than questions with which to conduct the interview.

Techniques Used in Interviews: The first significant technique used during an interviewing process for knowledge elicitation is known as probing or feedback (Greenwell 1988). In varying psychological conditions and at different stages of interviews, different probing techniques are recommended. Addition probing is used by the knowledge engineer after demonstrating some understanding of the subject matter and encouraging the expert with the message 'Go on tell me more about it'.

The reflecting probe could be used either in focused or structured interviews. It is recommended for the situation where the expert's current response tails off and the topic still remains incomplete.

The directive probe deals with the situation when the expert is judged to be too
specific on a single topic.

The *change of mode probe* is useful when a multiple viewpoint is required for the subject matter. For example, an alternative change of modes could include moving from an abstract or general example to specific examples.

The *defining probe* requires the expert to explain the meaning of a particular term or concept.

*Self report and think aloud protocols* technique is the second popular technique applied to knowledge acquisition. It is a technique borrowed directly from cognitive psychology. In this technique the expert is requested to provide a running commentary while performing a task.

The third important technique is generally known as *report by commentary*. In report by commentary, the expert explains what is happening or has happened with reference to a documented case study.

During requirements acquisition, expression and analysis, the main sources of inputs to an analyst principally depends on analysing interview transcripts and organisation documents. Both of these sources are directly related to communication activity in natural language between an analyst and a client. The formalised forms of this communication resulted in natural language documents. The subject which analyses natural language documents for useful information is discussed now.

### 2.4.2 Information Retrieval Strategies

One of the important uses of computers is to retrieve information. The importance of this activity has been recognised from the very first day the first computer appeared on the market. The availability of modern information retrieval systems has greatly improved storage and access facilities of information in various domains which were previously stored and retrieved manually. We believe that the information retrieval techniques provide some practical solutions to some of the problems which an analyst faces during requirements definition.

During requirements definition, the STARTS Guide argues that an analyst "should define purchaser's requirements in terms of functions, performance, interfaces, and design constraints" of a proposed system (1987:181). In a working environment, an analyst can be involved in seeking definitions, explanations, or in finding the correct context of the system's objects relative to the attributes of the other system's objects.
The sources available to the analyst typically include interview transcripts, organisational letters, reports, or documents related to the organisation. In this section, we discuss the techniques used in information retrieval systems with an intention to utilize them for an information system specially built for an analyst's understanding and elicitation.

The information retrieval system deals with the representation, storage of and access to information from documents (Salton & McGill 1983). The stored information in an information retrieval system is likely to include the natural language text of documents or of document excerpts and abstracts. The output of an information retrieval system in response to a search request consists of sets of references or definitions. These definitions and references are intended to provide information on items of potential interest to a system user.

The items of potential interest to an analyst during requirements definition are generally understood to include constraints in the system, system performance, the processes involved and the system interfaces. We believe that a large portion of this information can be retrieved from text or 'text fragments' from domain documents of a client organisation. In order to understand what is involved in an information system one needs to consider topics such as: Automatic Indexing, Retrieval Strategies, Natural Language Processing. The following discussion covers the key concepts for Automatic Indexing.

**AUTOMATIC INDEXING**
The basic operations involved in an indexing task consists of first choosing relevant content terms and second assigning weight to the terms according to their presumed value for the content identification (Salton & McGill 1983). Refinements are done later by assigning each stored item with its concept (i.e. a label or an index term) capable of representing the document content.

Most automatic indexing techniques start with computing the frequency of occurrence of individual word types. In natural language texts the importance of word frequencies for the purposes of content representation is generally recognised as important (Salton 1975, Halstead 1977). In order to calculate the presumed importance of each word, various term-weighting functions are available such as 'inverse document-frequency function', 'signal-to-noise ratio', and 'term discrimination value function'. The detail of these functions can be bound in (Salton & McGill 1983).

The second consideration, after terms weighting functions in information retrieval
strategies, is to improve the usefulness of index terms. Here, information retrieval system uses certain discrimination properties of terms such as associations, partitive, classes or subclasses in the hope of refining or broadening the interpretation of these terms. Any kind of term association can profitably be used in automatic indexing. The most natural source for this activity is to use a term thesaurus or synonym dictionaries.

2.5 Concluding Remarks
The review in Section 2.1 showed that conventional techniques and methods do not (directly) address the problems of requirements elicitation and domain understanding. It appears that most of the above mentioned techniques are provided with those aspects of software development which are needed mostly in a design stage of a software system. The semantics imposed by the constructs of these techniques and methods can constrain an analyst in representing the application domain knowledge. There are three points which emerge from our discussion:

1. In order to elicit the user requirements for a software system it is necessary to understand the functionality of a proposed system.

2. In order to understand the aims of the proposed system it is essential to understand the domain in which the system is going to work.

3. The available techniques, methods or tools do not possess such capabilities as to guide an analyst's mind to analyse the user requirements objectively in the client's domain specific language.

A recent paper by Martin and Tsai (1990) reporting on the results of using "N-Fold Inspection Method" supports our observation. In this method the authors have used 10 different teams and 5 different analyzing techniques for specifying a centralized traffic software system and found 92 different faults from a User Requirements Document (URD). The different methods used for specification in this research include: Structured Analysis (DeMarco 1978), Structured English (Birrell & Ould 1985), MSG (Berzins & Gray 1985) and Ada PDL (Ada 1983). The results of this research indicate various interesting points. Firstly, none of the teams discovered all the faults. Secondly, "Of the total faults, the average team found only 37 percent [of faults] during URD inspection" (1990:229). Thirdly, as the authors point out, the faults detected by each team were generally not the same: "The sparsity of URD faults found by a single team deserves further investigation. In this study, not many teams found the same faults."(1990:229) One can argue here that if these techniques/methods (as mentioned above) used in the N-Fold inspection method were so effective as to guide the analyst
team objectively, the question of sparsity of faults found would not have been arisen.

We have noted in Section 2.2, that expert system approaches are not adequate to deal with the diversity in application domains. The problem-solving ability used in such system are methods knowledge dependent which may be criticized as above. The expert system approach appears less effective in areas where software requirements elicitation and understanding are needed. Intelligent systems in the domain of requirements definition are geared more towards system validation issues (i.e. completeness or provability.). Most importantly, the requirements definition activity is not addressed globally and is restricted to the constructs provided by either method or techniques.

We have discussed natural language semantics in Section 2.3.1. In view of the client-analyst communication in natural language and the emphasis on (domain) 'understanding' lead us to concentrate on the mutual knowledge that exists between an analyst and a client (i.e. natural language) as a tool for requirements definition. In order to improve this mutual knowledge and to avoid ambiguities in the use of this tool, it is perhaps possible to use a natural language semantic schema. We have discussed two semantic schemata to find out the philosophy on which they have been developed. Conceptual graphs have been found as an appropriate schema to represent requirements statements. This scheme uses various knowledge of a language such as episodic, syntactic, and semantic. This schema is designed to do discourse analysis and (machine) comprehension.

In Section 2.4, we showed that knowledge acquisition techniques use psychological theories to deal with the knowledge of an (human) expert. The aim of knowledge acquisition, in this case, is to build an expert system, however, the techniques used to structure the knowledge (for understanding purposes) are equally important for requirements definition. We believe that techniques like protocol analysis, ontological analysis, knowledge structuring are useful, and can be used in an analyst's toolkit to acquire unambiguous and non-misleading requirements from the client.

Section 2.4 also discusses the automatic indexing techniques used in a information retrieval system. Those concerned with establishing the importance of terms and discriminating between them can be used for an analyst who is seeking machine assistance during requirements definition.
CHAPTER 3: RA DOMAIN KNOWLEDGE

In the previous chapters we have referred to the role of the knowledge-based expert systems in problem-solving, and we have alluded, in general terms, to the role of knowledge in systems specially designed for software requirements definition/analysis (RA) in Section 2.2. In general, the discussion in Section 2.2 reflects back to (typical) usage of method knowledge (of Section 2.1), but we argued there that this type of knowledge may not be very effective for the early parts of software requirements definition.

In this chapter, we describe the knowledge acquired from two expert analysts (Section 3.2) and the verification of this knowledge by a group of analysts (Section 3.4) for software development in general and for the requirements definition activity in particular. We have acquired this knowledge in an attempt to build an expert system for requirements definition. For the knowledge acquisition from expert analysts, we have used the 'face-to-face' interview techniques. Later this knowledge was verified by a questionnaire based approach. We have cited a 'state-of-art' survey of CASE tools (Section 3.6) which supports our arguments that current technology in software requirements definition is focussed on managerial aspects of software development and not on requirements elicitation and understanding. However, before discussing what knowledge we have acquired about requirements definition, we would like to discuss some background.

3.1 Knowledge Acquisition: an interview based approach in RA
Initially, we believed that it might be possible to build an expert system to provide advice and guidance related to problems in requirements definition. In other words, we considered the requirements definition activity as a problem in the domain of software development, which (in theory) could be solved by applying the knowledge of expert analysts. With such knowledge being represented in the knowledge-base of a 'requirements definition expert system', it could then be 'reasoned upon' to provide 'inferred advice' from the knowledge base (by using suitable inference mechanisms or reasoning strategies). The pre-stored knowledge in the knowledge-base would have comprised 'facts', 'rules' and 'heuristics' used by the expert in solving problems typical of the domain (i.e. software development). Specifically the problems which experts solve while eliciting requirements from a client.

The first phase in building expert system is that of knowledge acquisition and the verification of the acquired knowledge. The verification of the acquired knowledge is important and in our case it was more important because the experiential knowledge we
acquired is not documented before and as such not open to examination by others.

Our face to face interview failed to find the substantial amount of heuristic knowledge that would have been required to build an expert system. Our interview based findings, although sometimes at odds with the conventional 'text book' wisdom (Birrell and Ould 1985, Ramamoorthy 1987), did however reveal a wealth of experiential knowledge that was in large part verified by our questionnaire survey.

Our assessment was that an expert system for requirements definition was not feasible and perhaps, impossible. As we have argued in previous chapter (Section 1.3) that 'software engineering' is a specialist domain and by its nature very different to medicine (cf. MYCIN) or to geology (cf. PROSPECTOR). Moreover, the scope of 'requirements definition' encompasses the 'application domains' (the domain in which a system is going to work) and the 'problems' of the domain itself (i.e. the requirements definition activity) are not the same as the scope dealt by MYCIN (i.e. microbial infection therapy). Furthermore, the requirements definition activity is generally regarded as eliciting, understanding and specifying the 'problems' rather than 'solving' the problems.

We believe our attempts at knowledge acquisition for requirements definition are among the first reported in the literature.

3.2 Experiential Knowledge in RA
In the following sections, we first outline (Section 3.2.1) the questionnaire used during the interview. Next (Section 3.2.2) we describe our experts' experiences and their backgrounds. Subsequently (Section 3.2.3), we discuss our experience of the interview. Lastly (Section 3.2.4), an annotated summary of the experts' answers is discussed.

3.2.1 The Questionnaire
Our questionnaire was aimed to elicit the expert's (experiential) knowledge in three broad areas which we believe important to know for requirements definition/analysis. These areas are given in the following for RA domain knowledge:

- Purpose of requirements analysis (3),
- Monitoring and control used in requirements analysis (4),
- Formal/Empirical methods used in requirements analysis (3).

(note that the number in parentheses represents the total numbers of questions asked in these areas)
Chapter 3 RA Domain Knowledge

The questions related to 'purpose of requirements analysis' were asked to establish a common terminology between ourselves and the experts; to quantify the relative importance of the requirements analysis phase; to seek the views of one of the experts who specialises in dependability analysis; and to establish key issues in the requirements analysis phase. We followed the standard knowledge acquisition practice of establishing domain terminology or terminological relationship of the domain objects by asking questions of direct specialist interest, and hence to identify key issues in the domain. Questions 1-3 relating to 'purpose of RA' are given below:

1. What importance does System Definition (Requirements Analysis) have in the software development life-cycle?
   1.1 Do you think that the Requirement Analysis phase is the same as System Definition?
   1.2 Please can you comment on the statistics that the Requirement Analysis stage takes up to 5-12% of the total time of any project, and the effort required to correct an error in this stage 10-100 times less than any other stage of the project?
   1.3 What do you like to achieve after this stage?
   1.4 How are the results of this stage to be used?
   1.5 What stage of the Software life-cycle could mostly be affected by this phase?

2. How important to the user are dependability requirements in Requirement Analysis?
   2.1 What are the implications of stringent dependability requirement in carrying out requirements Analysis?

3. What are the key issues in the requirements analysis stage of software development, and what factors help you in going from Original Requirement to Requirement Specification?

Considerable interest is shown in software engineering literature on monitoring and controlling all aspects of software life-cycle (see for detail, Walston and Felix, 1977, Basili et al., 1983, 1984, 1986, Boehm 1981). Our next four questions were targeted to elicit expert opinion regarding the use of indicators/metrics of software development, the work breakdown structure, and reusability aspects of software requirements analysis activities. The questions asked in this respect include:

4. What type of indicator/metrics do you use in requirement analysis for monitoring/forecasting the project?

5. How do you view the System Definition/Requirement Analysis stage?(in terms of work to be done)
   5.1 Do you specify any task/work breakdown structure?
   5.2 Please rank the WBS in order of relevance to dependability?

6. Following is a set of rules used in System Definition (Requirements Analysis) stages. Please can you quantify them particularly with reference to the effect they have on System Productivity, System Complexity, Dependability, or any other aspect of the software development life-cycle?

7. How do you treat re-usability in the requirements analysis phase?
Finally, we asked our experts to comment on methods and/or models they use to do requirements analysis, whether or not they used abstraction and how did they tackle the issues related to software complexity in requirements analysis:

8. What level of abstraction is appropriate in the requirements analysis/system specification stage?
   8.1 Do you think comprehensibility is a problem and how do you cope with it?
   8.2 How do you tackle interconnection complexity in large software system?

9. How do you cope with logical inconsistency of coding in the system definition phase?

10. What TECHNIQUES/METHODS or MODELS do you use in Requirement Analysis?
    10.1 Please rank the methods/techniques in order of preference and schedule?
         (10 for more preferable, 0 for irrelevant).
    10.2 What are your particular areas of application experience?
        1. Communication Systems
        2. Real time Systems
        3. Distributed Systems
        4. Any other

3.2.2 The Experts' background
The experts involved in this study were originally employed by one of the leading hardware and software suppliers in UK, CAP Group Plc. (now SEMA-Group Plc.), which then employed more than 1200 skilled men and women. The group turnover of CAP was around 26 million pounds in 1984. We have benefitted from the know-how of two experts from this company. One expert was a Technical Director in CAP Scientific Limited, whom we have referred to as E2 in the subsequent discussion (E2 was also Senior Visiting Fellow at the University of Surrey). Our subject expert, subsequently referred to as E1, was recommended by E2. E1 has more than fifteen years experience as project manager, most of it with CAP, and he had worked for various large-scale software projects; he is currently attached to the UK Department of Trade and Industry as an advisor on matters related to safety critical (IT) systems.

3.2.3 The conduct of the interview
The interview was video-taped so as to preserve not only the verbal interaction of the experts but also the visual cues (e.g. diagrams, explanation charts, etc.) provided by the experts during the interview. The interview lasted a total of three hours, and was divided into two sessions. The whole transcription of the interview took around 45 hours spread over a period of six months.

The revised transcript of our interview with the two expert E1 and E2 is included in Appendix B. We refer to this transcript as revised because after word for word
transcription of the interview, the transcript was sent back to one of the experts. Expert E1 indicated errors and omissions in the original transcript and also added and deleted a number of sentences to the original transcript. The main cause of errors and omissions in the transcript was due to the poor quality of sound on the video-tape. However, the additions were generally qualifications to various statements, and the deletions were not numerous and perhaps due to E1's afterthought!

3.2.4 Annotated Experts' Remarks during the Interview
In the following (Section 3.2.4.1-3.2.4.3), the interview is discussed in three broad areas:

- Purpose of requirements analysis;
- Monitoring and control used in requirements analysis;
- Formal/Empirical methods used in requirements analysis.

3.2.4.1 The Starting Question: The Purpose of Requirements Analysis
Ramamoorthy et al. (1987) modelled requirements analysis as a two phased activity: phase dependent activities in which people usually prefer to do rapid prototyping, functional analysis, costing etc., and phase independent activities consisting of those aspects of requirements which deal with performance, reliability, security etc. However, if requirements analysis is treated as a phase oriented activity, the life-cycle approach to software development naturally follows. Birrell and Ould regard this activity as very important in the context of the life-cycle approach to software development (Birrell & Ould 1985). Bearing all this information in mind, we started with our opening question about the importance of the requirements analysis phase. E1 agreed with the importance of requirements analysis as it involves 'defining the outline design of the required system'. However, E1 disagreed with the statistics attributed to Boehm (1981), and argued that the 5-12% figure mentioned does not relate to the elapsed time of the project. Furthermore, E1 showed his disapproval of the phase-wise life-cycle approach to system development, particularly since in his experience, that of bespoke system development, changes in requirements never stop during software development and installation and requirements analysis has to be performed throughout the life of a project.

E1's expertise is mainly in bespoke systems development, where he claimed that it was difficult to make a distinction between separate phases of development. Bespoke system usually have to be dealt within the framework of the evolutionary approach to system development, where the analysts have to do a lot of exploratory analysis. E1 defines the evolutionary approach as:
"Evolutionary development is sideways out development, not the top down or bottom up. You give the client a few of the facilities he has specified in terms of what the end-user of the proposed system sees and uses at the end of day. You subsequently have to add other facilities. Now, in fact, to be able to do this doesn't mean to say that you supply one out of the 100 eventual facilities and that you only have to do one hundredth of the work initially knowing that you have to produce the other 99. [For example] You may have to develop the whole database management system on a little bit of data and a few application programs to sit on the top of it to provide one facility. This approach is a form of prototyping."

One of the main purposes of requirements analysis, as E1 suggests is to achieve more confidence in the project plan. According to E1, the output of this stage is used in subsequent stages but it depends largely on what method is being used. E1 also argues that notational method can sometimes "condition the analysts' thinking." E1 emphasised that life-cycle oriented requirements analysis may lead to the greatest shock in the coding phase, when the system developers suddenly realise that coding time is going up because of the complexity of the system.

The second expert, E2, however, did not appear to agree with E1, and argued that

"...the Software life-cycle influenced methodology forces the system builders to think in terms of the functionality of the system. But it generally hides the concerns related to performance, particularly in the more (fashionable) distributed systems."

E2 also argued that the evolutionary approach leads to fragmentation in the system, and then the system is delivered in a succession of parts:

"It is essential to fully understand the interdependency buried within the system. If you try to deliver a system in succession, as parts, and if you have lots of dependencies, then invariably lots of problems will surface during the acceptance stage. I think this approach means lots of deliverables and lots of stages and is only applicable to a very restrictive class of system."

In reply to criticisms of the evolutionary approach, E1 stated:

"... 'evolutionary approach' does work with highly interdependent systems but you deliver it in very small amounts at a time. I believe in changing only one facility at a time: a very few interdependent systems at a time, and it does mean a lot of management control on multiple releases all the time."

E2 then pointed out some of the associated problems which are common in change control, such as monitoring the changes, the criteria for software segmentation and its control, controlling the justification of priorities of user interest etc. We then asked E1 to distinguish between 'change control' and normal software development:

"Yes the whole thing [systems development] is change. I do not believe in development. I believe only in change control. So that is the line, to treat the whole project as change"
control.... The reason I distinguish between the two is that most people think in terms of the life cycle paradigm and there the development is considered as a sequence of steps: requirements; design; meta design and coding, and change only comes at the end of the cycle. However, the reality is the system builders start changing [the requirements] at the very beginning. This change is not change to a small segment of code but it is change to the build of [or the construction of] the system. This change could be through the addition of the new component or by replacement of the one by another."

E1 also argue fairly strongly against rapid prototyping and explained why he is against the throw-away prototype:

"Because it involves simply a vast expense of resources. The classic example could be a project which has well defined prototyping phases, and this phase involves developing an initial, intermediate and final prototype: each prototype has to be developed during a fixed length of time. But typically the team runs out of time during initial prototyping phase and the intermediate phase starts. This means that there isn't much time to learn from the initial prototype hence the initial prototype has to be thrown away."

E1 raised another argument against throw-away prototyping, that the team which develops the prototype generally does not develop the final product. This means that even if the prototyping phase was executed successfully the lessons learnt during that phase can not be passed in full by the prototyping team to the product-development team.

Since the personal interest of E1 lies in developing dependable software systems, we asked him to relate his dependability concerns with requirements analysis issues. He pointed out three possible relationships:

"(1) Dependability requirements specify what the user wants and if the system does not perform according to specification then a wrong system has been developed.
(2) The Customer generally tends to accept less than what he ought to be demanding.
(3) In certain fields the client actually has documented reliability, availability, specifications and standards but even there monitoring against specified requirements is not the norm."

The expert was then provided with a summary of what is regarded by academic experts in RA as being the key-issues in software requirements definition (Ramamoorthy et al. 1986). Table 2 shows these issues and what E1 thinks of their applicability during requirements analysis phase.
### Chapter 3 RA Domain Knowledge

**Requirements analysis issues cited in Ramamoorthy et al. (1986)**

<table>
<thead>
<tr>
<th>Key issues</th>
<th>Expert El's Opinion</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Continual/evolutionary change in requirements.</em></td>
<td>√(normal in bespoke sys.)</td>
</tr>
<tr>
<td>- during subsequent phases of the life-cycle.</td>
<td>NR</td>
</tr>
<tr>
<td>- due to incorrect prediction of resource requirements</td>
<td></td>
</tr>
<tr>
<td>- feasibility evaluation.</td>
<td>√</td>
</tr>
<tr>
<td>- testing.</td>
<td>√</td>
</tr>
<tr>
<td>- security.</td>
<td>×</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nonfunctional goals.</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>- performance.</td>
<td>√</td>
</tr>
<tr>
<td>- reliability.</td>
<td>√</td>
</tr>
<tr>
<td>- manpower coordination.</td>
<td>NR(RA outputs)</td>
</tr>
<tr>
<td>- managerial policies.</td>
<td>NR(RA outputs)</td>
</tr>
<tr>
<td>- quality management.</td>
<td>NR(RA outputs)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Phase dependent issues.</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>- metrics.</td>
<td>Ltd</td>
</tr>
<tr>
<td>- technical indicators.</td>
<td>df</td>
</tr>
<tr>
<td>- rapid prototyping.</td>
<td>√</td>
</tr>
<tr>
<td>- distributed systems specification.</td>
<td>x Design issue</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Phase independent issues.</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>- reusability.</td>
<td>×</td>
</tr>
<tr>
<td>- traceability.</td>
<td>×</td>
</tr>
<tr>
<td>- metrics.</td>
<td>?</td>
</tr>
</tbody>
</table>

Key:
- Not Relevant = NR
- Yes = √
- Not sure = ?
- Limited = Ltd
- Different views = df
- Not related = x

Table 2: A list of key issues deemed very important in requirements specification. Our expert opinion is shown in the second column.

Out of the 17 items we selected from Ramamoorthy et al. (1987), our expert (E1) appears to:

- agree with only six points: continual changes in requirements definition, feasibility evaluation, testing, performance, reliability and rapid prototyping are key issues in requirements definition.
- finds different/limited or no relevance to three items (i.e. metrics--related to both phase independent and dependent-- and technical indicators.)
• *suggests* that four items were of no relevance to RA: incorrect prediction of resource requirements, manpower coordination, managerial policies, and quality managements.

• *disagrees* with four points, as key issues in RA: security, distributed systems specification, reusability and traceability.

### 3.2.4.2 Monitoring and control used in RA

As requirements analysis is essential to software development, it is important to find out how this activity is monitored and controlled. In the software engineering literature, it is discussed that the monitoring and control of software development can be achieved using a variety of metrics (e.g. size/complexity, effort, changes, software science metrics and so on). The metrics and other cost drivers, we note, are generally related to the number of lines of code anticipated for the software project. E1 suggested that these factors are specifically case-dependent so that it will be very difficult to produce generic answers which can relate to productivity or dependability of the software:

"... it will be very difficult to give a generic answer to this question, if walkthrough is used in various phases; does it have any effect on productivity or dependability? It would depend on so much, on so many factors, ..."

Cost effectiveness in software development is related to the concept of reusability both in the sense of code fragments and in the reuse of already prepared chunks of functional specifications. We intended to get the expert's view on reusability for which he argued:

"I am not aware of any great successes in reusing the software. I am not saying you cannot, in general we don't. And I have a hunch that this because we do not make a lot of use of the parametric languages ..."

A supplementary question about the cost drivers used in software development was then asked: E1 pointed out that he has used SLIM[1] but not very successfully because it involves numbers of lines of code which he does not like to predict early in software development:

"I prefer not to work in terms of lines of code and it is not easy to do so early on in the project. I like to see design down to the module level, pretty well. Because in general the amount of effort to produce a module is pretty much independent of the complexity of the code ... Maybe

---

1 Putnam's work on the Rayleigh curve model models of the software development lifecycle has led to a commercially available cost estimation model known as SLIM. The principle estimation equation in SLIM is \( S=CK^{1/3}(TD)^{4/3} \), where \( S \) and \( TD \) are respectively the number of deliverable source instructions and development time, and \( K \) is the total lifecycle manpower effort and \( C \) is a constant which must be determined from historical data (Birrell & Oulds 1985:31).
one approach to software is to try to structure system development in the same sort of way, where there are complexity independent components, by looking at the system in terms of modules which have complexity independent cost associated with them: for example, configuration management, man-time related to test, design review and walkthrough. These are examples of stabilising influences, or at least influences which you can average over modules (or chunks of software). The creative design elements, thereby, become less significant in terms of the overall cost. Therefore, attempts should be made to spread or apportion all known costs over all the chunks of code (modules) thus reducing the relative contribution of one chunk. This means that in order to produce an accurate cost of the project detailed design must be undertaken.

The task of a system manager is always crucial in controlling the different aspects of system development. This involves his/her past experiences of successful or unsuccessful projects. Question five was aimed at determining how our expert views the work break down structure required in requirements analysis phase. In response to this question, the expert suggests that:

"This looks more like project planning, rather than the task or breakdown structure for the requirement analysis phase. We are talking here about that planning of system definition/requirements analysis phase where acceptance test specification ought to be a part of requirements analysis. It makes it look like a clean phased work, whereas the reality I think is that, continued requirement analysis does pervade in all future work. I do not think I could come up with a generic plan. I could do but I don't think it will be meaningful....

If I have to give an elementary lecture to someone who has never heard about the subject before, I will put something like:

* meeting with customer: initial requirements meeting,
* work out why the customer would like the system,
* then high level solution options which will meet the Requirements (e.g. whether the customer really needs an aircraft or something else will do instead; do you really need an intruder missile system or would a set of barrage be cheaper),
* Selection phase: one solution from the solution options,
* Requirements details of the chosen solution,
* Design options of the chosen solution....
But the above steps relate to an abstract solution; reality unfortunately is very different."

3.2.4.3 Formal/Empirical methods used in RA
In the software engineering literature, formal methods are extensively discussed. To the question: is there any formal method which is appropriate in requirements analysis as an abstraction tool for user requirements, our expert was quite outspoken:

"There is no such thing as formal method and if there are some formal notations, there is little you can do in formal notations. It is perhaps English or rather user-jargon that is the right-level of abstraction to use ..."

It has been argued that the large and complex software usually requires special
methods for the 'validation of specifications' to avoid inconsistencies in coding (see for detail discussion on SERM in Alford, 1977). In reply to our query, E1 argues:

"There are a number of projects I was involved in, where we put in a tremendous effort to provide traceability. However, I fear, that in general they are ignored when the systems are put in the field for use. Once in the field the customer changes his requirements and you change the specification in the system and function tests again and then the intermediate testing documents etc are just thrown away. I am looking at traceability, but I don't think it is a key issue in R.A., but poses a problem here because we don't know properly the way in which to show to customer that the actual system meets the requirements. We can say it does, the whole point of system testing is to do just that, but this does not automatically allow a mapping between requirements on the system, unless you use a method like JACKSON where the whole thing come automatically, where you can actually see the connections."

3.3 Conclusion: Knowledge Acquisition in RA

Analysis of the full transcript and the salient points as discussed above shows that the knowledge acquisition activity, in the domain of software development, does not produce a substantial amount of the so-called problem-solving 'heuristic' knowledge (e.g. 'if-then' type of statements) for an expert system. Indeed, both our experts were sceptical of the expert systems enterprise and our in-depth analyses led us to believe that perhaps expert system development is not feasible in this context. We believe that the 'expert systems' development by Kramer et al. (1988), Loucopoulos and Champion (1988), and Hughes et al. (1988) (see Section 2.2.1-2.2.3) are 'front-ends' for established methods like CORE, JSD and SADT. The expertise in these 'expert systems' were more related to computer-assisted instruction packages, where the stress is on elaboration of key learning concepts ('drill-and-practice') type exercise, and graphical navigation aids for novices.

Nevertheless, our interview did highlight various facets of requirements definition which are described below.

In an evolutionary model, the requirements definition activity is performed continuously throughout the software development and the product is delivered to the client in parts. This type of software development is characterised as follows:

(i) there is a tight control on various software versions of the system,
(ii) the changes in software requirements are implemented successively in parts of a system,
(iii) the aim is to use the trial period with the client to improve the system, this is similar to the evolutionary prototyping model of software development, but not
the type of prototyping usually termed as 'throw-it-away'[2].

In the evolutionary approach, system understanding starts at the level of functionality or main operations. This understanding extends to both higher and lower levels of analysis. The experts have, therefore, termed this approach the 'middle-out' approach. The STARTS guide (1987:196) termed it a 'pragmatic approach' which is useful for large, complex systems such as military command and control systems, where the boundaries are not well defined and the overall functionalities are sometimes difficult to express.

We found that one of our experts did not like the notion of a separate requirements definition phase for software development. Methods and techniques are used increasingly in the industry; however, our expert suspected that these methods conditioned the analysts' thinking.

Our experts feel that there are only few effective monitors (or indicators) used in software development. According to the views of E1, the number of lines of software code could not be used to predict the complexity of the software, and therefore, should not be used to calculate or monitor the cost of the software. Instead, our expert prefers 'function point analysis' to measure the complexity of the software (see Chen 1978 and McCabe 1976 for details on function point analysis).

The gulf between theory and practice was revealed during this survey. It is not clear how to 'scale-up' the theories in real practice for large scale software development projects. It was not surprising that our experts expressed doubt that currently available requirements definition methods are useful for large scale software projects.

---

[2] In the evolutionary approach mentioned above, the prototyping is similar to the 'face reconstruction process' by a police artist with the help of different facial parts (e.g. having a variety of noses, eyes, lips, forehead etc.), a confirmation is always achieved before the next facial part is fitted in during a criminal face recognition by a witness, and as a result a mental picture gets a physical form, and it is never thrown away during the case investigation.
3.4 Knowledge Verification: A questionnaire based approach
In Section 3.1, we indicated that knowledge acquisition, particularly interview-based knowledge acquisition, can be used to document the experiential knowledge of an expert analyst. Once this knowledge is documented, it is essential to verify this knowledge. In this section, we verify this knowledge through a questionnaire based approach. Basically, in this approach, we were exploring the knowledge on the same grounds as acquired from the expert analysts (i.e requirements definition in particular and software development in general, see Section 3.2). The results of our analysis of the questionnaire is presented here to verify the knowledge we have acquired so far.

We conclude in this section that the requirements definition activity plays an important part in software development, particularly in cost estimation of the software, time schedules and maintenance. This survey confirms that no single technique is uniquely popular among the analysts. We also note that during requirements definition most of the analysts in this survey prefer to analyse only the essential requirements of a system. A majority of analysts reported that they use a 'mixture' of textual and notational techniques for the requirements definition activity.

BACKGROUND TO THE SURVEY
The questionnaire used in Section 3.4 can be considered as an experiment in a focused interviewing technique as outlined in Section 2.4, and is used here to acquire the domain knowledge of the requirements definition activity.

Two criteria were used to select forty three different organisations for this survey. Firstly, they must deal with 'turn-key systems'; and secondly, they have more than 150 employees in the organisation (i.e. should be large to medium sized organisation [1]) A questionnaire containing 13 questions was prepared by the author and sent to these software development organisations. The questionnaire was composed of multiple choices[2] and deliberately kept small in number, in the hope of receiving a significant number of replies. However, we received only a moderate response: 13 replies from the 43 organisations (about 30 percent). There could be three reasons for this poor response: (i) commercial confidentiality, where organisations were unwilling to disclose their methods of software development; (ii) the complexity of the questionnaire; and (iii) the analysts were too busy.

1 These organisations were selected from the 'Computer User' Yearbook 1988. The yearbook contains the details of over 1000 organisations who deal with a diversified computing interests from software development to hardware retailing.

2 See Appendix C for questionnaire details.
BACKGROUND OF THE RESPONDENTS
Generally, the organisations we have selected were involved in 'turn-key' software development. Those who replied can be further divided into three categories:

SOFTWARE SYSTEMS HOUSES
- BIS Software,
- Data Logic,
- Logica,
- Scicon (now SD Scicon),
- Software Sciences.

MANAGEMENT CONSULTANCIES
- Arthur Young Managements,
- PE Consulting Services,
- PA Consulting Services,
- Touche Ross Management.

HARDWARE/SYSTEM GROUPS
- Easams,
- ICL (UK) Ltd.,
- Inbucon Technology.

These three categories cover a range of software systems: from real-time military systems to management information systems, and from stock control and inventory systems to command and control systems. These organisations also include those individuals (and organisations) who characterise software development as either a 'computing problem' (Hardware/System Groups) or a manifestation of the corporate strategy of the client's organisation (Consulting Groups).

THE EXPERIENCES OF THE RESPONDENTS
Most of our respondents either had a Mathematics degree or were Science/Engineering graduates (9 out of 13); 3 respondents had a Business Studies degree and one was graduated in an 'Arts subjects' (one respondent did not mention his education). These data indicate that a science degree, particularly a Mathematics degree, is favoured in this domain.

In our sample, the average practical experience in various posts was as follows:

- 8 years experience as a Project management (10 out of 13).
- 9 years as a project leader/System Analyst (11 out of 13), and
- 4 years as a programmer (9 out of 13).

(Three respondents failed to report any experience as a project manager; 2 did not show any experience as project leader and 4 had no experience as a programmer.)

Our results show that, in general, 'programming' constitutes a smaller part of our respondents' experience. The fact that most of them are experienced as a project leader
or as a project manager indicates that the respondents may have been involved in requirements definition quite early in their career and perhaps have a greater exposure to the clients. The background experience of this sample survey was compatible to the experience of our expert E1 as a project manager.

This following section covers the responses to each of the 13 questions given to the analysts (Section 3.4.1 to 3.4.13).

3.4.1 Notational versus Mixed method of Requirements Definition

The main aim under this heading was to find out how many analysts prefer to use 'notational techniques' (e.g. SADT, Flowcharts, Data Flow diagrams etc.), how many rely only on 'textual analysis' i.e. rely on using 'mental' or 'informal' analysis and use company related texts to conduct the analysis, and how many analysts use a 'mixture' of notational techniques. An overwhelming majority (10 out 13) reported that they use a 'mixture' of textual and notational techniques, only 3 out of 13 rely on 'textual analysis' alone, and none reported using 'notational techniques' only. Those relying on 'textual analysis' alone qualified their statements by noting that they use this analysis only for a new system and to build a conceptual model of the required system.

3.4.2 Tools and Techniques used in Requirements Definition

Following on from the question relating to the general preferences of techniques (notational, textual, or mixture of the two), we asked the analysts to indicate their specific use of well documented methods and techniques (like JSD, SADT, CORE, Data Flow Diagrams etc. or any other) on a three point scale: Quite Often, Sometimes, Depends. The following characterises the answers we received.

- Data-Flow Diagrams appear to be a popular technique (6 respondents use it quite often, 3 sometimes, and two depending on the requirements).

- SSADM: the Central Computers and Telecommunication Agency (CCTA, a permanent 'watch dog' agency of the UK government), approved methodology is also a popular methodology, and only two of our respondents use it quite often, while six use it sometimes and three depending upon the circumstances.

- Information Engineering Methodology, a product from James Martin Associates, also emerged as a 'quite often' used technique.

- The notational techniques like SADT influenced only one of our respondent, however, its use depends on the circumstances.

Table 3 compiles the responses received for question 2.
The above responses, to a certain extent, confirm two of the observations of our expert E1, when he (i) disapproved of the structured design notations, like DeMarco or Yourdon etc., because their semantic complexity in actual use 'could be enormous'; and (ii) showed his reluctance in using methods like JSD (although beneficial in certain cases), because such an approach 'forces you to identify every single type of event' of the system, and forces you to much lower level of details earlier on during requirements definition stage. Perhaps, another reason for the lack of use of JSD (or SSADM) is as E1 argued that the penalty of using such 'disciplined' techniques like JSD was that "the totality of information available at the end of requirements definition is pretty incomprehensible: most design [notations] don't give an easy way of structuring information."

3.4.3 Bottlenecks in Requirements Definition
Question 3 elicits respondents' views on various constraints which hinder the requirements definition activity (i.e. the bottlenecks). A total of nine such constraints were drawn from the literature for this question, ranging from 'changes in user requirements' to the 'misunderstanding of user terminology' by the analyst. The first few, which are thought to be major problems, are according to Ramamoorthy et al. (1987): (i) Functional and evolutionary changes in user requirements, (ii) Non-functional requirements. The others concern the knowledge of the client's domain, and the way this knowledge is encoded in the 'domain terminology' and how it is to be decoded (or understood) by the analyst. We have elicited respondents responses again on a three point scale: bottleneck encountered 'Quite Often', 'Sometimes', and 'Never'. Most of our respondents (12 out of 13) regarded 'changes in requirements' as the problem they encountered 'Quite Often', only one found it 'Sometimes'. Evolutionary changes of requirements is, therefore, a real problem (this is in accordance with experts E1's views).
The changes in requirements may be due to (i) a misconception of resource requirements, 6 out of 13 respondents encountered this problem as 'quite often' and a similar number of respondents faced it 'sometimes', only one respondent said 'never' had that problem; (ii) a 'misunderstanding' of user requirements, 5 out of 13 confirmed it as quite often the case, 6 out 13 said 'sometimes', and two respondents said they never have that problem. Table 4 contains the details of the respondents answers to the question relating to the three major 'bottlenecks' in requirements definition:

<table>
<thead>
<tr>
<th>Bottlenecks in RA</th>
<th>Frequency of Encounter</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>'Quite often'</td>
</tr>
<tr>
<td>1. Changes in requirements</td>
<td>12</td>
</tr>
<tr>
<td>2. Software performance requirements</td>
<td>7</td>
</tr>
<tr>
<td>3. User sophistication in requirements</td>
<td>6</td>
</tr>
<tr>
<td>4. Misconception in resource requirements</td>
<td>6</td>
</tr>
<tr>
<td>5. Misunderstanding in user requirements</td>
<td>5</td>
</tr>
<tr>
<td>6. Test criteria for software</td>
<td>5</td>
</tr>
<tr>
<td>7. Software reliability</td>
<td>5</td>
</tr>
<tr>
<td>8. Software security</td>
<td>4</td>
</tr>
<tr>
<td>9. User terminology</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 4: Comparing Analysts answers for the common issues during requirements definition stage of software development

The question related to user terminology was an interesting one: 3 out of the 13 respondents never had any problems with the user terminology—these respondents included both the hardware oriented groups in our survey and one software/systems house. However, more than half of the respondents 'sometimes' found user terminology a problem and another 3 found it so 'quite often'. However, we consider that user terminology plays an important role in the client's domain, and is thus essential to an analyst's understanding during requirements definition.

3.4.4 Level of detail at which Requirements are Specified

Question 4 deals with the level of detail in software specification. Typically, most of the analysts (8 out of 13) were in favour of including essential requirements only during the feasibility stages. Only three analysts preferred detailed requirements analysis including 'predictable software modules' in the feasibility documents (the term 'predictable modules' is defined in the glossary attached to the questionnaire, see Appendix C). Four analysts replied that it is not necessary to include predictable modules of a system in a software specification. However, they were in favour of including the details of requirements at the feasibility stage. This may indicate that
during requirements definition of a proposed system, at least during the feasibility stage, the details of the previous requirement specifications or other details of past projects are not included in current specification.

3.4.5 Domain Terminology And Formalisation

The Question 5 was: "Do you think formalisation of application domain terminology could help you analyse the client requirements?" Answers were requested in the form of 'Yes', 'No', and 'Not sure'. Five out of thirteen answers were not sure that formalisation techniques could help during requirements definition stage. Four respondents said 'no' it is no help during requirements definition stages. Four respondents said 'yes' that domain terminology formalisation can help during the requirements definition stage. Two of the respondents in the 'yes' category qualified their answers with comments. One respondent (who belonged to a management/systems group) suggested that: "Always build a glossary of the user terms". Another (software/system house) respondent considered such formalisation can be helpful, however he comments that "it is likely to be impractical", and adds that the "client would have to spend effort to define his terminology and 97-100% of the effort might be wasted".

3.4.6 Requirements Acquisition Methods

Question 6 was designed to elicit the respondents' opinion on their preferred methods of obtaining the clients' requirements, particularly on how the respondent obtained information about the client's organisation and the environment in which the proposed system is going to operate.

All the respondents indicated that they interview the end-users, and 4 out 13 said that they use 'passive observation' of the clients' organisation and its operation. All but one respondent indicated that he/she analyses documents which emanate from the client organisation. Most of our respondents (9 out of 13) preferred to interview the relevant technical experts in the domain: domain experts, business managers, computer professionals in the client's organisation etc. The fact that most analysts analyse user domain documents (and a number of them stress the importance of comprehending the terminology of the user's domain) led us to consider this area as an area of prime importance in our subsequent research into requirements definition.
The respondents also indicated that they use a variety of techniques which include:
- interactive business modelling using CASE generally known as Information Engineering Workbench;
- Kelly Grids;
- Teach back;
- Focused Interviews;
- Interviews at 'board level' in order to consider the business strategies (this was the answer of a respondent who is working in a management consultant's system group); and
- Interviewing the client's own computer systems department, which is often the source of the client's initial requirements specification (this was the response from a software system house).

3.4.7 Structured Domain and Requirements Acquisition

Question 7 was designed to determine the importance of a structured as compared to an unstructured application domain for requirements definition at the feasibility stages, and what methods the respondents use for this purpose. 3 out of 13 respondents said 'no', whereas, two respondent could not understand the question, and one respondents did not reply. However, the seven who said yes, also provided the methods we did ask for:

Yes: during feasibility study: segmentation of the overall corporate information model by function/entity affinity analysis.
Yes: I use 'common sense'
Yes: CASE tools which allow us to identify formally the existing structure and which document this structure in such a way that we can test our understanding of it with the user (e.g. SSADM, DFD's).
Yes: EASAMS has its own technical standards which we use by default (when a method is not specified by the customer). These standards and common techniques from the method.
Yes: I would usually try to obtain a functional breakdown of the domain (e.g using a top level data flow diagrams)
No: but a business strategy must be in place.
Yes, in outline: Functional and organisation hierarchy charts, data flow and organisation key data entity structure diagrams.

Table 5

It is evident from the above table that most of the analysts prefer DFDs and Entity-Relationship diagrams to structure the top-level information (e.g. organisational management, functional, data etc.) to analyse or specify the requirements at the feasibility stage.

3.4.8 Requirements Definition Work-Breakdown

Question 8 was designed to determine how the requirements definition task is broken down by our respondents. Recall, Expert E1's comments regarding monitoring and controlling used in RA; "I would go for a less glossy feasibility study and more prototyping". On the other hand, software engineering literature shows that 'requirements definition' activities are needed essentially during two phases of software
development. First, when the project is at an 'inception stage', where the client and developer are not committed to the project (this stage is normally referred to as the feasibility stage of the project). Second, during system definition phase when the client is, at least partially, committed to the project, and the analyst is committed to produce a number of documents: 'system definition management'; 'functional specification'; 'project plan'; 'quality management plan'; and 'system modelling' (Birrell & Ould 1985). This led us to ask the analysts, whether they divided this stage into an initial feasibility report, and a final report.

Less than half, 5 out 13, of the analysts indicated that they divide this phase into two; 4 out of 13 said that it is not done in this fashion, however, one of the negative respondents qualified his remarks by saying:

"But the requirements definition is often divided into feasibility study and detailed requirements analysis-- approval would be required separately for both tasks and approval for the full project would come only after the second."

3.4.9 Management and Key Deliverables
Question 9 was an elaboration of question 8 in that our aim was to identify the key deliverables relevant to initial and final feasibility studies. Birrell and Ould (1985) have discussed the system definition stage's work-breakdown structure with key deliverables after this stage: Functional Specification; Project Plan; Quality Management Plan; Acceptance Specification. Since we believe that the Acceptance Specification of the project needs careful analysis, it is dealt with in a subsequent question.

The analysts replies, for this question, are summarised in the following table.

<table>
<thead>
<tr>
<th></th>
<th>Final Report</th>
<th>Initial Report</th>
<th>Both Reports</th>
<th>After Contracts</th>
<th>Not Applicable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Plan</td>
<td>(7)</td>
<td>(3)</td>
<td>(3)</td>
<td>(nil)</td>
<td>(nil)</td>
</tr>
<tr>
<td>(for the client)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Development Guideline</td>
<td>(10)</td>
<td>(nil)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
</tr>
<tr>
<td>(quality management plan)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Resource Requirements</td>
<td>(8)</td>
<td>(3)</td>
<td>(1)</td>
<td>(1)</td>
<td>(nil)</td>
</tr>
</tbody>
</table>

Table 6: Stage wise relevance of typical software documents

It appears that the emphasis is on a final report of the feasibility study (rather than the initial report) and perhaps this report discusses Project Plan, Quality Management Plan and Resource Requirements for software development.
3.4.10 Software Test Criteria And Requirements Definition
It was stressed to us by our expert E1 that "you shouldn't have a Requirements Specification, you must have a Acceptance Specification", and "I would like to see acceptance test written before you go into the development phase." Question 10 was asked to determine the relevance of test criteria to feasibility studies of the project. Most of the respondents (9 out 13) indicated that they do not prefer to prepare the test criteria during the feasibility study, and two respondents were in favour of having test criteria during the feasibility study, only one respondent did not answer this question. A respondent who used the option 'depends', also includes his reservation: it is 'not usually an option but it is probably premature.'

3.4.11 Report Generators and CASE tools
Question 11 was asked in order to determine current practices regarding 'report generators' in the industry. Most of the respondents support three report generators: Information Engineering Workbench (IEW), AutoMate, and Excelerator. However, others respondents use either their own report generators or perhaps use some ordinary text editing facilities.

3.4.12 Documentation Standards
Every organisation appears to have its own standards for the documentation used for requirements definition during feasibility stages. For example, BIS Software Ltd. uses the standards available in BIS/IPSE system, marketed by BIS Applied systems in the UK, both for report generation, and for document standards. Data Logic Ltd. has its own document standards. A few respondents prefer leave the choice of standards for software requirements documentation to the client.

3.4.13 Ross & Schoman definition of Requirements Feasibilities
Question 13 was based on the Ross and Schoman (1977) notions of requirements definition activity, where the authors have stressed various feasibilities (including technical feasibility, operational feasibility, economic feasibility) of the proposed software. Most of the respondents (10 out of 13) have given their views on what they think appropriate during system feasibility studies. In the same question, we requested the analysts to provide information regarding the method(s) they used for assessing the cost of a software system. This information is arranged in tabulated form to present an overview of the concepts attached to different feasibility studies among the analysts, along with the software cost estimation strategies.
## Chapter 3: RA Domain Knowledge

### Method for Assessing Software Cost

<table>
<thead>
<tr>
<th>Technical Feasibility</th>
<th>Operational Feasibility</th>
<th>Economic Feasibility</th>
<th>Method for Assessing Software Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware in use, or and Propose, Appropriate Language, Constraints</td>
<td>Response times, User Interface, System Sops</td>
<td>Lifecycle costs covering: Hardware, Software, Staffing, Installation/Accommodation</td>
<td>Discounted Cash flow, Cost/Benefit Analysis</td>
</tr>
<tr>
<td>Suitability of the Product Strategy, Hardware Resources Required to Support both Development and Operational Phases of System Procurement</td>
<td>Number of Batch Jobs, Data Volumes, System Availability, Number of Users, Response Times, Staff Quality and Experience</td>
<td>Cost Justification against Time/Efforts, Number of Functions to Cost Analysis.</td>
<td>DATASOLVE Standards for Project Estimates</td>
</tr>
<tr>
<td>Must investigate major components/design structure which constitute the product architecture</td>
<td>Must propose major aspects of man machine interface.</td>
<td>Man day resource estimates must be produced</td>
<td>Based on previous estimates of career projects</td>
</tr>
<tr>
<td>This should always be considered, though in practice, I suspect it is usually done by default.</td>
<td>This should be considered though it rarely if perhaps because it is so difficult to do in the early stages when a feasibility study is done.</td>
<td>Essential, and this is usually done though I believe it is often optimistic. Benefits are usually over stated, cost under-stated. I also believe we are in urgent need of better techniques here!</td>
<td>Cost are established by costing resources and discussing cost with suppliers. Cost benefit analysis is often supported by discounted cashflow projections.</td>
</tr>
<tr>
<td>Not too relevant at this stage</td>
<td>Essential</td>
<td>Dependent on type of application and client. Some studies for legislative purposes</td>
<td>----</td>
</tr>
<tr>
<td>Required</td>
<td>Required</td>
<td>Required</td>
<td>Fully amortized implementation costs plus operational costs as estimated amortisation periods vary depending on type of expenditure and company policy.</td>
</tr>
<tr>
<td>Functions, Data flow, Data sources/Availability, Data stores, Storage Capacity, Throughput requirements.</td>
<td>MMI Response times, User Skills Required Training Requirement</td>
<td>Investment Cost, Life-cycle cost</td>
<td>Prompt, using a database of previous company software productivity.</td>
</tr>
<tr>
<td>Although the fine details will not be available at this stage, some aspects of technical feasibility should be considered at this stage. The only danger is in assuming technical feasibility when information is not available.</td>
<td>This is closely linked to technical feasibility and should be considered in conjunction with that.</td>
<td>All project must be cost justified, and the benefits must be tangible. If it appears at feasibility stage that this will not be possible then, the project should not proceed.</td>
<td>No formal methods, although an estimate of costs of hardware/software and manpower resources are made.</td>
</tr>
</tbody>
</table>

Table 7: Comparative Statements from 10 Analysts on Ross and Schoman Feasibility Strategies, the last column indicates their methods of assessing the Software Costs

It appears in the above table that there is no consensus regarding the definition of the terms 'technical feasibility', 'optional feasibility', and 'economic feasibility'. In general, however, technical feasibility is taken to mean the feasibility of the hardware platform and/or software tools for a given project. 'Operational feasibility' is regarded
in terms of the adequacy of response times, performance that is, human-computer interface. 'Economic feasibility' is defined, generally, in terms of costs and benefits of a software system. Similarly, the last column (i.e. method for assessing software cost) does not indicate any specific method popular among the respondents. However, the estimation of lines of code (e.g. COCOMO, Boehm 1981) or complexity in terms of functions (i.e. 'function point analysis', see Chen 1978 and McCabe 1976) to be preferred by the analysts for software costing.

3.5 Conclusion: questionnaire based approach
The above discussion identifies a number of key points associated with 'real world' software requirements definition practices. For instance, a mixture of notational and textual analysis is commonly used by analysts to understand the required system conceptually. It seems that rigorous analysis techniques (i.e. formal methods) are not used during the requirements definition stage. The whole conceptual model of the system starts with first considering the essential requirements for the system. The predictable modules of the system are either left for the detailed design of the system or seem unimportant during the initial investigation of the system requirements.

There is usually a vast amount of knowledge available to an analyst for development of systems which are generally characterised as database management systems or office information systems (or where a number of software systems are already working as computer applications). We believe that analysts mostly apply techniques such as entity-relationship diagrams, data flow diagrams and structure charts in these domains whose computerisation is already well understood.

Change in requirements seems to be an important issue during software requirements definition. This reflect the needs to understand the client's intentions. This, in fact, demands an understanding of the application domain. Analysts are obliged to know the domain terminology to understand the domain. However, in our survey, they seem less concerned with user terminology during requirements definition activities. It is possible that the respondent analysts tackle domains where a similar system is already available in the market and it is not needed to know user terminology at conceptual level. We believe that an appropriate understanding of domain terminology plays a significant role in requirements definition, and that this is essential to cope with changes in requirements.
3.6 CASE Tools Features

This section highlights the emergence of computer-aided software engineering (CASE) technology from a survey conducted by Rock-Evans (1989). The tool features involved in this survey indicate the vendors' confidence or popularity of certain techniques available in software development literature (see Section 2.1). These tools are claimed to be helpful for what is regarded as 'structured analysis or design' techniques in software development.

After looking at the features provided by these tools (Rock-Evans 1989), we conclude in this section that these tools appear particularly helpful in 'software project management'. At present, the CASE tools do not view requirements definition activity as a 'cognitive' or 'knowledge intensive' activity. These tools appear less supportive in the early stage of software development, which start from some vague ideas of what needs to be done. However, these tools are claimed to support all aspects of software development.

We note in Rock-Evans survey data (reported in 'INFOMATICS', April 1989) that around ten CASE products offer 10 or more features (by feature we mean data-flow diagrams, entity-relationship diagrams, data dictionary, any methods etc.) and/or their combination (e.g. the data-flow diagramming technique combined with entity-relationship diagramming etc.). Probably, this indicates the vendor's lack of confidence in any one technique, or that the vendors have taken into account the fact that analysts use number of techniques to tackle the onerous task of software development.

In the field of software development, methods like CORE, SSADM, Yourdon and DeMarco, JSD, Information Engineering, are well documented. Some of these methods were known to the analysts we interviewed (see Sections 3.1 and 3.3). The present CASE technology (based on these methods) appears to support the facilities in the areas which are labour intensive for software development. We found in this survey that a larger number of CASE products offered management facilities as compared to support for requirements definition. Requirements elicitation, understanding and specification, which lead to effective control of the software design, are either not mentioned as features or are implicit and less emphasised by the developers of the CASE products.
Chapter 3

RA Domain Knowledge

3.7 Concluding Remarks

Software development has a number of facets and involves different groups of people. The academics divide the issues in software development into phase dependent issues and phase independent issues. This distinction is less important in the real world, where the emphasis is more on productivity or on keeping down the cost for the software. Provability and correctness are considered less important in the real world than functional completeness and reliability. However, one aspect of software development, where academics and software house specialists agree is the importance of a correct 'exposition' of the client requirements. By exposition we mean eliciting the requirements correctly and being able to reproduce them for the client's confirmation and subsequent usage. It is commonly understood that requirements definition activity plays an important part in the software costs, time schedules, maintenance, and most importantly, the acceptability of the software by the client.

We also note that during requirements definition activities, most of the analysts in our questionnaire survey indicate their preference for analysing only the essential requirements of a system, and they are not very concerned to use the experience of past projects. However, considering the fact that specifications are written and the requirements are understood for computer systems without special emphasis on tools and techniques, the requirements definition activity can only be defined by an analyst's cognitive skills: acquisition and comprehension of the problems; imagery, representation, reasoning and decision making; and language based analysis of the problems.

The requirements definition activity for a software system has its own particular problems and issues. The most important, and probably unavoidable one, is concerned with the all-pervasiveness of natural language usage during this activity. It is used from the very start of a project, used throughout requirements acquisition from the client, and used during analysis of those requirements to produce a specification. Considering its critical role, natural language should be used with great care during requirements definition, especially to avoid ambiguities in the specification. This is perhaps easier at the 'production end', when producing the final requirements specification. However, we are more concerned with the problems at the 'receiving end', where the analyst is receiving knowledge from the client, trying to understand it and subsequently use it to design a system according to the requirements. The use of natural language is unavoidable during this process of understanding software requirements.
At present, problem-solving techniques based on formal mathematical theories are not widely used in the real world, especially during early stages of the requirements definition activity. As we have noted in Chapter 1, the requirements definition activity comprises 'acquisition', 'expression', 'analysis', and 'specification' of requirements, where we have proposed that these processes basically comprise overlapping activities of 'elicitation', and 'understanding and specification'. Here, the end products (e.g. diagrams, graphical representations and so on) represent the analysts' understanding of the requirements of the software.
CHAPTER 4: A Toolkit for Requirements Definition

So far, we have examined the requirements definition activity from three different perspectives.

First, we reviewed the relative importance of this activity both in software development models and independently. We have considered this activity in terms of requirements acquisition, expression, analysis and specification. We discussed the importance of various paradigms for this activity including: knowledge engineering (the knowledge acquisition techniques for software specification); information retrieval strategies (searching for key concepts underlying the user's domain directly from text); and knowledge representation schemata (for formalising the user's domain knowledge).

Second, we undertook a comparative study of existing methods and techniques used for requirements definition and discussed the importance of 'intelligent' systems, inspired by knowledge engineering in this context.

Third, we acquired, documented and verified the experiential knowledge for this activity from leading software development organisations, which specialise in the specification and management of software systems. We also mentioned briefly why current use of AI in 'software engineering', with its emphasis on the notion of 'highly domain specific problem-solving', will not help in the development of expert systems for requirements definition.

The three different perspectives led us to the following major conclusions:

i. The requirements definition activity comprises two interactive overlapping tasks, that of requirements 'elicitation' and 'requirements understanding and specification'. The emphasis of our work is on the identification of the above tasks performed during requirements definition and on the provision of computer based toolkit to expedite this activity.

ii. (a) Currently available methods and techniques, whether formal or informal, do not take into account (or indeed take advantage of) the extensive interaction between the client and the analyst through the medium of speech and text – i.e. the medium of natural languages.

ii. (b) In CASE environments, the requirements definition activity takes no account of the fact that the analyst has access to a large text archive which includes transcripts of the client interviews, documents describing the client's organisation and business practices, documents describing current computer systems used by the client and so on. In fact, the analyst needs a (special) information retrieval system, where key words may be used to retrieve the relevant knowledge for requirements definition.
The emphasis of our work (in the light of ii. a and b above) is on exploring and demonstrating the efficacy of what is known about language, linguistic data used in knowledge encoding/decoding and natural language processing. The other, equally important, emphasis of our work is to explore and demonstrate the efficacy of information retrieval using terminological data and statistical techniques.

iii. The (cognitive) skills used by an analyst for the requirements definition activity may include natural language-based communication, representation, problem-solving, reasoning and so on. We believe that computer assistance in the execution of these skills will lead to better elicitation and comprehension of user requirements. In our work we emphasise the role of representation schema (e.g. conceptual graphs and associated reasoning strategies) which draw their inspiration from fields such as 'Linguistics', 'Psychology' and 'Philosophy'. The use of such schema will preserve what was originally described by the client and will be easily traceable.

We describe how the above (cognitive) tasks related to requirements definition can be executed by using a natural language processing framework and by using conceptual graphs.

This chapter introduces a linguistic toolkit for requirements definition, particularly for the elicitation and understanding of user requirements. The architecture for the toolkit is as follows:

![Figure 13 A linguistic toolkit approach in requirements definition](image)

Section 4.1 comprises the background, including motivation, and an overview of the proposed toolkit. Section 4.2 is devoted to 'linguistic cues probing' for domain objects, their relations, rules and constraints of the proposed system. Section 4.3 discusses a combined approach based on elicitation and understanding the functionality of a proposed system in a natural language processing framework. Section 4.4 demonstrates the linguistic toolkit for requirements definition activities. The concluding remarks are discussed in Section 4.5.
4.1 Requirements Elicitation & Understanding in NLP Framework

In this section, we describe our motivation and overview of the proposed linguistic toolkit for requirements definition given in Figure 13.

Motivation for a Linguistic toolkit

We have observed that the task of requirements definition for software specification is performed in a limited amount of time (approximately 5-10% of the total time given to a software project, see Chapter 1). At the stage of requirements definition, the analyst starts working with a very limited knowledge of the actual requirements of a proposed system. The scope of the proposed system is generally not well defined at this stage, nor it is defined in terms of the particular functions or processes that are required. The key to progress at this stage is to (quickly) understand the scope of the proposed system. The question of understanding the scope of the system is directly related to the question of understanding the application domain of the system. We believe that the understanding of application domain can be expedited in terms of system requirements elicitation and understanding by the use of a linguistic toolkit within a framework of NLP techniques.

In the following, we present a summary of our observations about the requirements definition activity which further motivated us to develop a linguistic toolkit.

* A small amount of time is available for an analyst to generate the requirements specification.
* An analyst does not need to be an 'applications domain expert' before (or after) the system requirements definition process.
* The requirements specification document produced by the analyst is not a design document nor does it correspond to a system or a prototype.
* The client is often not clear about his/her own system requirements.
* The client aims to solve some problems whose solutions are not clear, even as to whether a computer solution is feasible or not.
* The client has a limited appreciation of the scope or constraints of Information technology.
* A complete spectrum of the advantages or disadvantages of the perceived computer solution is not known to the client (or the analyst) at the requirements definition stage.

It seems difficult to help an analyst during requirements definition considering the above stated constraints for this task. However, we believe that this task is manageable provided one thinks carefully about what is required in the context of software specification (i.e. contents knowledge or system knowledge). We shall use the term 'system knowledge' to refer to the application domain knowledge which can include system 'objects', 'relationships' between objects, 'system constraints', 'performance', 'interfaces' and 'functions'.
Chapter 4 A Toolkit for Requirements Definition

objects: physical or abstract concepts of the client domain
relationships: the object relationships that manifest themselves in class hierarchies; structural hierarchies in part-whole relationships; causal associations (cause/effects); material associations (composition) and so on
constraints: software design constraints; hardware design constraints
performance: significant characteristics of the proposed system regarding capacity, response times, system management, availability etc.
interfaces: Human Machine Interface; Hardware Interface; Software Interface
functions: the transformation mechanisms that are necessary to achieve outputs from given inputs; dynamic requirements; exception handling etc.

(note: see STARTS Guide 1987 for more details on these terms)

We believe that the information related to these terms for the purpose of requirements specification is available in the client domain documents, including the transcripts of the interviews specially arranged for requirements definition. Subsequently, we shall refer to the lexical data (i.e. dictionary words or phrases) which can be used to elicit this information from such texts.

Overview of the proposed Linguistic Toolkit
The linguistic toolkit, we propose, requires textual inputs (i.e. the requirement interview transcripts, or other documents as mentioned in item ii b. The outputs of such a toolkit are expected to be a set of requirements statements structured such that these statements can be discussed with the client to further elaborate and refine the analysts' understanding of the requirements. Such client-analyst interaction will eventually result in the production of requirements specification documents.

A linguistic toolkit for an analyst must deal with retrieving 'system knowledge' from the input text using certain kinds of *lexical entries*. We believe that these lexical entries can be used to probe the system knowledge as stated above (i.e. systems objects, their relations, heuristic rules, constraints and explanations). There are two reasons for confining this probe to these aspects: firstly, it is easy to find lexical entries for retrieving information relating to them, and secondly, we believe that this type of information is necessary for an analyst to understand the application domain. Our approach of general purpose linguistic probing for system knowledge can be compared to the ontological analyses used in knowledge acquisition for expert systems (see Section 2.4 for details). The linguistic cues probing is discussed in more detail in Section 4.2.

The general purpose linguistic cues deal essentially with words and phrases. There is another source to retrieve system knowledge, in the form of sentences containing

---

1The general purpose lexical entries, in linguistic terminology, are generally defined in the category of 'function words' or 'closed words class'.
'action' words to indicate process information for the proposed system. We can accomplish this task with the help of a special class of lexical entries which denote action (i.e. verbs). Consider the sentence, "The system must *analyse* all the incoming data to calculate a standard deviation." The action indicating verb 'analyse' gives some process information in a sentence. For a linguistic toolkit one can use lexical entries (as a lexical database) for selecting such sentences showing process information. However, in order to avoid the selection of irrelevant information, the inclusion of the context (in which the word appears) can help in selecting the relevant sentences. Subsequently, the selected sentences can be converted to conceptual graphs for achieving more understanding of the clients' requirements. The details of this approach are discussed in Section 4.3.

In order to emphasise the role of natural language (processing) in requirements elicitation and domain understanding, the above mentioned approaches are implemented for demonstration of a 'linguistic toolkit'. This demonstration is given in Section 4.4, and will show how the notions of information retrieval, NLP, and knowledge acquisition can be used for

(i) identification of system knowledge with the help of 'general purpose' lexical entries and
(ii) identification of system processes with the help of process indicating verbs which are relevant to the early stages of requirements definition.

### 4.2 Linguistic Cues Probing (LCP)

Under this heading, we shall discuss two kinds of probing: single *word probing* and *phrase probing*.

**Words for probing:** Grammarians have analysed languages, particularly languages like English, in terms of parts of speech (e.g. noun, verbs, adjectives etc.). These are generally classified as so-called 'closed classes' or 'open classes'. Examples are:

<table>
<thead>
<tr>
<th>Closed Classes</th>
<th>Open Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>determiner: the, a, that, every, some</td>
<td>noun: radar, room, graph</td>
</tr>
<tr>
<td>preposition: of, at, in, without</td>
<td>adjective: fast, slow</td>
</tr>
<tr>
<td>modal verb: can, must, will, could</td>
<td>full-verb: search, grow, play</td>
</tr>
<tr>
<td>primary verb: be, have, do</td>
<td>adverb: steadily, completely</td>
</tr>
<tr>
<td>pronoun: he, they etc.</td>
<td></td>
</tr>
</tbody>
</table>

The 'closed classes' are closed in the sense that the lexical items in this set cannot normally be extended by the creation of additional members. Quirk et al. (1985) describe the usage of these items in English as 'reciprocally exclusive' (i.e. the
decision to use one item in a given structure excludes the possibility of using any other), and 'reciprocally defining' (i.e. it is less easy to state the meaning of any individual items than to define it in relation of the rest of the structure). Certain items in closed classes can be used to distinguish subjects or objects in a sentence. For example, a determiner 'the', 'every', or 'some' can refer to something important in a sentence.

The other contrastive category in English is the so-called 'open class'. The parts of speech generally included in this class refer to 'nouns', 'verbs', 'adjectives' and 'adverbs' in English. Grammarians characterise this set as indefinitely extendable as it can always accept new words. Also, members of this set (particularly noun and adjective) refer to stable physical and abstract entities, hence called 'stative', and the verbs and adverb are characterised as 'dynamic' in that they indicate action, activity and temporary conditions.

Phrases for Probing: While the closed-class words help to elicit information about objects there are a number of phrases, mainly determiners and primary verb combinations, which are used to encode types, relations and hierarchies. Note that prepositions and prepositional phrases can also be used for this purpose. Furthermore, there are phrases (and words) which indicate the presence of rules: 'If and then' constructs used in "If <condition> then <action>" structure; phrases beginning with 'given that' connectives like 'so that'. Similarly, the domain knowledge is sometimes elaborated in text by the use of words (and phrases), like 'due to', 'because' etc.

In this section, we describe how words categorised in closed classes and word phrases can be used to elicit system knowledge.

Table 9 shows a set of example words and phrases which comprise the lexical database used in our toolkit for eliciting system knowledge. From now on, we shall refer to the words and phrase mentioned in this table as lexical entries. Four columns in Table 9 are arranged to probe four types of information. For example, system objects can be determined by the lexical entries categorised as 'Determiners & Quantifiers', relations and hierarchies for a system can be found with the help of the category labelled as 'Types, Relations & Hierarchies'. Similarly, the heuristic rules, constraints and explanations can be elicited for a proposed system by using the lexical entries mentioned under the columns labelled as 'Rule Indicators' and 'Explanation and Caution' respectively.
Table 9: Lexical entries for probing system knowledge

Figure 14 outlines our approach to use these lexical entries in a systematic manner. This figure is specified in SADT to highlight the requirements of typical lexical data for various kinds of system information. This figure shows that the LCP for quantifiers and determiners can precede other categories in Table 9 for system knowledge (see box 1 in Figure 14). Later, the LCP for lexical entries showing 'type', 'relations' and 'hierarchies' will help in understanding relations between the (system) objects (see box 2 in Figure 14). Similarly, other aspects of system knowledge can be elicited from domain documents in order to analyse and specify rules and constraints for a proposed system. We believe that the system knowledge retrieved in this fashion can help an analyst to elicit and understand the requirements of a proposed system, and the presence of above mentioned lexical entries in a sentence can be exploited for certain kinds of system knowledge essential for software specification.
We discuss now each of the lexical categories mentioned in Table 9 in detail (Section 4.2.1 to 4.2.4). The discussions address their lexical groupings in grammar, and their semantic role in system knowledge.

4.2.1 LCP with Determiners and Quantifiers
This category covers the lexical entries principally categorised as determiners and quantifiers in English grammar. Some of the members of this category provide definite or indefinite reference to an object, equivalent to the 'articles' in English. They generally occur as the head of a noun phrase. For example, 'a' in a noun phrase, a photocopier, and 'the' in, the milling machine. Similar referencing can be achieved with the lexical entries categorised in grammar as 'predeterminers', such as, 'half', 'all', 'double' and so on. Another grammatical category, 'postdeterminers', including 'cardinal' and 'ordinal' numerals, such as 'many', 'few' and so on, may also occur. (Quirk et al. 1985)

We have devised a search program to identify domain objects with the help of the quantifiers and determiners shown in Table 9. For example, consider the interview transcript provided by Skidmore and Wroe (1988) for a typical system dealing with seminar activities, where it is easy to select a sentence containing system objects with the help of the determiner and quantifier in bold:

"... Jim is always popping in asking me how well a particular course is booked or how many people attended the last seminar on a particular topic and if anyone couldn't get booked on it. ..."
(Skidmore & Wroe 1988:96)

4.2.2 LCP for Relationships and Hierarchies in the System
The second group contains a mixture of prepositions and prepositional phrases. "In the most general terms, a preposition expresses a relation between two entities, ... Most of the common English prepositions, such as at, in, and for, are SIMPLE, i.e. they consist of one word. Other prepositions, consisting of more than one word, are COMPLEX" (Quirk et al. 1985:657, 665). This group of lexical entries contains phrases such as 'is a' and 'a kind of' and so on which can help in finding the hierarchies between objects. For example, the sentences 'a photocopier is a machine' and 'a colour photocopier is a kind of photocopier', involve a hierarchy between the different machines.

The task of finding the relationships, types or hierarchies of the objects is crucially important for understanding the application domain. The LCP data discussed in this section selects the sentences containing strings of characters such as, 'is a', 'a kind of',
'contain' and so on. The objects identified by the determiner and quantifier (Section 4.2.1) can be used at this stage to identify their types and hierarchies for specific objects. A definition of types and hierarchical relationships between objects is crucial in designing the data structures during the design stages of software development.

4.2.3 LCP for System Rules
The lexical entries in this group are generally classified as 'conjuncts' and 'subordinating conjunctions' (i.e. 'if', 'then', 'because', and so on) in English. The conjunction and subordinating conjunctions encode cause/effect knowledge: how one object causes changes in other object.

4.2.4 LCP for System Constraints
A number of English words and phrases are used for 'explanations' and 'constraints' in discourse, such as, 'due to' and so on. Our linguistic toolkit can extract the explanations or constraints for system performance from the available text. The lexical entries provided by LCP data for this purpose, although belonging to different grammatical categories, have similar (semantic) functions in a sentence. For example, the word 'because' is defined grammatically as a 'subordinate conjunction', however, its function in a sentence is similar to the phrase 'due to', or the words 'must' or 'should', which are categorised as modal.

The constraints knowledge together with the system rules and heuristics can provide a substantial amount of help to an analyst in system understanding.

The second facet of our linguistic toolkit, which is concerned with more general processing of organisational documents for software requirements elicitation and understanding, is presented below.

4.3 Text Selection and Text Processing
Software requirements at 'specification level' are generally based on information such as 'inputs', 'processes' and 'outputs'. A detailed description may include functions, performance, interfaces and constraints of the system. A still more detailed description comprises tactical and future planning, involving hardware and software needs, strategic client needs and maintenance requirements for the system. At any level of detail, the system description task needs a variety of information during the requirements definition activity.

It is generally believed that a so-called 'solution endorsement' is not the main aim of a requirements specification for a software system (see for details Ramamoorthy et al.
1987). Instead, it is generally recommended to illustrate the problems as clearly as possible (STARTS 1987).

The general consensus on the basic ingredients of a software requirements specification (e.g. to IEEE Standards, 1988, and STARTS Guide, 1987) include the illustration and the discussion of the system's 'function', 'performance', 'interfaces' and 'design constraints'. To a certain extent the knowledge of functional activities in a system is central to a software requirement specification. Different methods use different terminology to identify these functional activities of a system. For example, in DFD and CORE these activities are termed 'PROCESSES', in an SADT description these tasks are termed 'ACTIVITIES' and in JSD they are called 'FUNCTIONS' of the system.

The information related to 'functions', 'performance', 'interfaces' and 'design constraints', in our view, can be elicited with the help of a certain class of verbs. Text fragments, which contain these verbs, can be extracted from a given text and presented to the analyst. However, such a general approach can give considerable amount of redundant information. This approach can be made specific if we can identify the context of the text and correlate it with process indicating words. This will provide a selection criterion for sentences which have substantial information regarding the processes of a proposed system. Later, the selected sentence can be used for understanding through a conceptual schema. This is the approach adopted in our linguistic toolkit as follows:

* Text selection: where a mixture of linguistic knowledge and information retrieval strategies are used to extract the processes knowledge directly from a text related to the domain of a proposed system.

* Text understanding: where the selected texts are mapped to a conceptual graph for understanding the software requirements and application domain quickly.

The inputs for text selection may include documents relating to the clients organisation; client's interview transcripts and so on. The output is intended to be conceptual graphs describing the functionality of the proposed system. A top level view of these activities is presented in Figure 15 (in SADT notation).
Chapter 4

A Toolkit for Requirements Definition

Figure 15: Top level specification for Text Processing in a linguistic toolkit

The details related to each activity (i.e. text selection and understanding) are discussed separately in the following subsections, which cover our design approach to this problem. Section 4.3.1 deals with text selection criteria, and Section 4.3.2 deals with text processing to generate conceptual graphs for the selected text.

4.3.1 The Text Selection

Text selection criteria are automatic and principally depend on two factors: (1) The presence of a content word whose frequency count is maximum. (2) The presence of a process word.

(1) Content words

In linguistics, an approach to word "classification recognises a class of 'content words', defined as words which have stateable meaning,... [as against] a few FUNCTION words, whose role is primarily to express GRAMMATICAL relationships." (Crystal 1989:70-71). Note, in order to avoid any confusion with these 'function words' and functions of a proposed system we will subsequently use the phrase 'process information' for functions of a system.

Function words comprise 40 to 50 percent of the text words (Salton & McGill 1982). In English, around 363 common words are generally characterised as function words (e.g. a, the, should, would and so on). It is easy to include them in a dictionary or in a so-called 'stop-list' (ibid 1982) for identification of content words. We have used these words in our program to identify the domain vocabulary. The program uses those function words which have been identified by Miller et al. (1958) in their work on the statistical properties of text. (The same list of function words was used by Halstead (1977) in his work on the quantitative
analysis of English prose). We have enhanced this list by including words like 'figure', 'mainly' and 'number', which are in addition to the original list of Miller et al. (1958). These function words are stored as PROLOG predicates (see Appendix D).

An efficient searching mechanism is required to compare, select and count an individual word against pre-stored lexical data-base of 'function words'. This process will determine a domain vocabulary of the content words which can be used to select a high frequency content word for further activities in the linguistic toolkit.

(ii) Process Words

To select a sentence containing process information, we have identified a list of process indicating words. In English, the lexical group classified as verbs can be used for this purpose, e.g. 'allow', 'provide', 'calculate' etc. An English thesaurus can be used as a source. The following table contains a list of words which can be used to indicate process information in a sentence. These words are used (in conjunction with a high frequency content word) in our linguistic toolkit as a lexical database for eliciting process knowledge from domain documents of a proposed system.

Note that this strategy could be extended to select not only sentences containing process information but also sentences indicating 'constraints', 'interface' or 'performance' information for specification purposes. However, this would need lexical databases similar to Table 10 for their respective categories.

<table>
<thead>
<tr>
<th>form</th>
<th>make</th>
<th>give</th>
<th>build</th>
<th>define</th>
<th>forms</th>
<th>given</th>
<th>makes</th>
<th>shows</th>
<th>size</th>
<th>builds</th>
<th>create</th>
<th>formed</th>
<th>defined</th>
<th>giving</th>
<th>making</th>
<th>stored</th>
<th>verify</th>
</tr>
</thead>
<tbody>
<tr>
<td>compute</td>
<td>collect</td>
<td>created</td>
<td>creating</td>
<td>forming</td>
<td>display</td>
<td>exhibit</td>
<td>extract</td>
<td>perform</td>
<td>provide</td>
<td>produce</td>
<td>showing</td>
<td>storing</td>
<td>building</td>
<td>computed</td>
<td>computes</td>
<td>creating</td>
<td>collects</td>
</tr>
<tr>
<td>provides</td>
<td>provided</td>
<td>produces</td>
<td>produced</td>
<td>retrieve</td>
<td>verified</td>
<td>calculated</td>
<td>collected</td>
<td>computing</td>
<td>displayed</td>
<td>generated</td>
<td>determined</td>
<td>demonstrated</td>
<td>determine</td>
<td>performed</td>
<td>producing</td>
<td>retrieved</td>
<td>verifying</td>
</tr>
<tr>
<td>storing</td>
<td>storing</td>
<td>stores</td>
<td>stored</td>
<td>verifying</td>
<td>calculates</td>
<td>calculating</td>
<td>collecting</td>
<td>determining</td>
<td>demonstrating</td>
<td>illustrating</td>
<td>illustrating</td>
<td>illustrating</td>
<td>illustrating</td>
<td>illustrating</td>
<td>illustrating</td>
<td>illustrating</td>
<td>88</td>
</tr>
</tbody>
</table>

Table 10: Process indicating words
The following figure summarises our proposed strategies pictorially, in two steps. The first step checks the frequency of a content word and the second correlates a process word and a high-frequency content word to select a sentence in the document. Some working demonstrations for this text selection are provided in Section 4.4.

4.3.2 Text Processing
We now discuss how a semantic interpretation may be obtained for a selected sentence. In AI, various knowledge representation schema are available (e.g. frames, scripts, semantic nets, conceptual dependency graphs and so on, see Rosenberg, 1980, for details). In Chapter 2, we have discussed a schema known as conceptual graphs. This schema is chosen for text processing because its origin is in the field of linguistics, and by implication we can use it for natural language text processing. In particular this schema can be used here for understanding or analysing the selected sentences. The reasons for selecting this schema against others such schemata are based on the following arguments.
According to Sowa and Way (1986) the knowledge in a sentence can be organised and understood into four basic categories:

- **Lexical**: information about word forms
- **Syntactic**: information about word and phrase categories and their ordering in sentences.
- **Semantic**: word definitions, constraints on the use of words in well-formed sentences, and background information about defaults and expectations.
- **Episodic**: assertions about particular things and events.

For an appropriate text understanding schema, all the above mentioned types of (linguistic) knowledge are necessary. Sowa's conceptual graphs can be used to represent knowledge (of a sentence) in all the four categories.

**The Basic ideas of Conceptual Graphs**

Sowa (1984) has claimed that the foundations of conceptual graphs are based on linguistics, psychology and philosophy. These disciplines provide different inputs to the conceptual graph theory, for example, Linguistic (knowledge) provides syntactic information for the discourse, 'concepts formation' is based on research in Psychology, whereas, 'category restrictions' are provided from Philosophy. We believe that during requirements analysis a representation schema, such as conceptual graphs, can be effectively used for requirements understanding.

Conceptual graphs contain only two types of constructs: concepts and relations. A conceptual graph consists of nodes connected by arcs. Each node represents a concept, and each arc represents some relationship between the concepts. The concepts-nodes represent entities, attributes, states or events and relation-nodes show how the concepts are interconnected.

In the following we now discuss Sowa's conceptual graphs applied to some of the requirements of a system, BUTEC (British Underwater Test and Evaluation) computer system. Since this system was designed for strategic military use, we are unable to discuss its requirements at great length. We will call this system a 'tracking system' in the subsequent discussion.

**A Conceptual Graph Generation Mechanism (CGGM): A Working Example**

We have implemented a program to generate conceptual graphs. In the following, we elaborate the mechanism for this program. The theory of conceptual graphs includes a set of standard operations on graphs. The basic operations include the four formation rules of copy, restrict, join, and simplify, [see Sowa (1984) and Sowa and Way (1986) for details]. These operations can be grouped in any order for a typical CGGM. We have grouped 'copy and restrict' and 'join and simplify' operations for our CGGM.
However, before using a CGGM for converting a natural language sentence into a conceptual graph, the sentence is parsed for 'subjects' and 'objects' information. Now consider a sentence selected after the text selection stage:

"The system shall allow the tracking in three dimensions for ten objects"

A parser can provide the following syntactic information:

(S:DECL (NP (DET the) (NOUN system)) (VP (AUXV shall) (TV allow) (NP (DET the) (NOUN tracking) (PP (pp in) (NUMERAL three) (NP (NOUN dimensions)) (PP (pp for) (NUMERAL ten) (NOUN objects)))))

There are a number of parsing programs available which can perform this task. For example, Sowa and Way (1986) have used in their semantic interpreter, an English parser called PLNLP (Programming Language for Natural Language Processing) due to Jensen and Heidorn (1983). Guenthner et al. (1986) have used ULG (User Language Generator) developed by IBM (1981) with a User Specialty Language (USL) System in their discourse representation theory (Appendix A). For our proposed linguistic toolkit, we have used a Recursive Transitive Network (RTN) parser, due to Gazdar & Mellish (1989), for syntactic analysis of sentences.

The next step after parsing a given sentence in a CGGM is to identify the semantic role of words (e.g. subject and object). These concepts can be determined from noun and verb phrases of a declarative (active or passive) sentence. In the above example sentence, the nouns, 'system', and 'objects', represent subject and object information, and a verb or verb phrase, represents process information and also a concept associated to some actions. The concept verb, 'allow', will form the process information for a conceptual graph.

**COPY AND RESTRICT**

Canonical graphs are predefined selectional constraints on permissible combinations of concepts with specific relations (see Section 2.3 for more details). A typical use of canonical graphs in a system is analogous to the factual database provided for a person who is looking for meaningful relations between important concepts of a domain. The canonical form of knowledge can be provided in a system independent of any domain.

The operations of copy and restrict are defined over a 'canonical knowledge base' for the linguistic toolkit. From the above mentioned parsed information of a sample sentence, when the CGGM encounters the concept type SYSTEM, as an agent of a main verb allow, the mechanism searches the dictionary definition for 'SYSTEM' already stored in a canonical form, which may be attached to a concept 'ALLOW' and...
can be made available to the CGGM for a final form of a bigger conceptual graph as:

\[\text{SYSTEM} \xrightarrow{\text{AGNT}} \text{ALLOW}\]

This graph presents a concept ALLOW which is attached to concept SYSTEM with a relation represented as AGNT and can be used for conceptual graph generation. However, the conceptual graphs generation mechanism will first confirm that the concept ALLOW is also present in the example sentence. If it is, this canonical graph will be selected for inclusion in a bigger conceptual graph.

Similarly, consider the next concept in our example sentence, the concept type 'TRACKING'. The graph generation mechanism will try to find information about the concept TRACKING, having found a canonical graph attached to this concept with a relation represented as MANR to the concept type ALLOW. The conceptual graphs generation mechanism will confirm that the concept TRACKING is also present in the example sentence. If it is, this canonical graph will also be selected.

\[\text{ALLOW} \xrightarrow{\text{MANR}} \text{TRACKING}\]

At this stage, the conceptual graphs generation mechanism will try to confirm that all the concepts attached to concept ALLOW are being selected. If that is not the case, the CGGM will first find all those concepts which are attached to the concept ALLOW with the restriction that they must exist in the selected sentence. In our example sentence, the concept ALLOW is also used with the concept 'objects'. The canonical graph for this concept will be selected for the concept OBJECT for a final form of a bigger conceptual graph as:

\[\text{ALLOW} \xrightarrow{\text{OBJ}} \text{OBJECT}\]

JOIN AND SIMPLIFY

A simple join creates a single graph by merging two graphs on a single matching concept. A join operation on the above selected graphs for matching concept ALLOW will generate more graphs:
The next step after the join operation, is simplification. The routines for simplification will eliminate the concepts that are duplicated in the context of a new graph. If duplicated relation nodes are found, they are also simplified, and the attached concept node is adjusted. In the above two graphs, the concept type SYSTEM is duplicated, it is necessary to delete one while maintaining the relation AGNT with the concept ALLOW. This will result in the formation of a new graph given in the following:

The resultant graph reflects the understanding of one of the main functions of the tracking system (i.e. ALLOW in this case). Other attributes of this function such as performance, constraints or interfaces could be attached to the concepts involved in the graph using similar steps as defined above. For example, the tracking dimensions in the example sentence are three, and the number of objects allowed by the tracking system are ten. The completed conceptual graph would be as shown in Figure 19.
Figure 19: A conceptual graph representing a complete requirements statement

The following figure presents a scheme for the above mentioned CGGM. Note that the figure is processing the same example sentence as discussed above.
Figure 20: A conceptual graph generation approach
Weakness of language based approach
There may be two weaknesses of a language based approach to requirements definition:

(1) Natural Languages are not rule governed compared to subjects like Physics and Chemistry in natural sciences.

(2) One weakness in Sowa's conceptual graphs is the possible non-availability of canonical graphs for the subject matter. In such cases, the issues involved in finding canonical graphs for a domain are similar to knowledge engineering issues.

4.4 A linguistic toolkit
This section demonstrates our implementation of the linguistic toolkit. The aim of this section is to illustrate the essential features of the toolkit, Linguistic Cues Probing, Text Selection and Text Understanding, integrated in one program (see Figure 13).

The interface to our linguistic toolkit is menu based and the program was written in QUINTUS PROLOG (environment) running under an UNIX operating system on a SUN SPARC station. (The codes are provided in Appendix D).

The input text required for processing must be an ASCII file and the toolkit is initiated by specifying the file name. The main menu appears on the screen after a short while. The program displays a screen containing a menu as shown below:

```
?- go(williams).
* Please Wait *
```

```
Document Name:williams
Domain Vocabulary (content words): 777 words
Words occurred frequently: [emissions:67,vehicles:57,nox:55]
1. Quantifiers  2. Relations and Hierarchies
3. Rule Indicators  4. Explanation
5. Sentences with Process Information & Conceptual graph
Enter your choice:
```

It is to be noted from this screen (i.e. Figure 21 a) that from the very start the program provides various items of information about the text being processed. This
information includes the name of the file being processed (e.g. 'williams'[2]), the total number of domain vocabulary counts (e.g. 777 words), and a list of three content words whose frequencies are found to be the highest in the document (e.g. *Emissions* appeared 67 times, *Vehicles* 57 times, *NOX* 55 times). The domain vocabulary is identified and counted in this program by removing the function words (see Section 4.3.1) from the text being processed.

One can use this content words frequency data in various information retrieval strategies (e.g. for document selection, important text fragments selection after a query to IRS and so on). We have used this knowledge to recognise the context of a document (or the topic of discussion), the text (williams) is dealing with concepts like 'emission', 'vehicle' and 'NOX' (and this type of information is essential for option 5 in this linguistic toolkit). The following screen shows the toolkit activity when option 1 is selected in the demonstration.

```
***************
Document Name: williams
Domain Vocabulary (content words): 777 words
Words occurred frequently: [emissions:67,vehicles:57,nox:55]
***************
1. Quantifiers  2. Relations and Hierarchies
3. Rule Indicators  4. Explanation
5. Sentences with Process Information & Conceptual graph
***************
Enter your choice: 1
Line Number: 67 * quantifier_indicator: [some] *
It is clear from this study of as-received cars that
vehicles at any particular time can in general be operating
at some large margin from the regulations and it
is interesting to note that pollutants such as CO may
well be being emitted in larger quantities than the regulations
would suggest, while for NOx it is possible that
emissions could be substantially lower than the Regulation.

more ?n
```

Figure 21b: Choice selection option 1

When a LCP option is executed in the program, the chosen LCP lexical entry and a line number are shown during text display; indicating the place where it is found in the document (the LCP lexical entries are provided in Table 9). The program then waits for user interaction to show more occurrences in the document. After finding all the occurrences in the document (up to the end of the file) other instances of quantifiers are searched in the document. If the user wants to go back to the main menu, the option

---

2 William is a text file and contains a research report in automobile engineering written by M.W. William (1988) of Warren Spring Laboratory UK, on "Relating vehicle emission regulations to air quality".

97
'more ?' may be negated by entering 'n' when the program waits for user interaction. Similarly we can try other options at the main menu level under the LCP framework, they are provided here as option 2 for relations and hierarchies, option 3 for rule indicators, and option 4 for explanations in the documents under investigation. Option 5 is discussed below separately. The next option 6 can be used to search a phrase in the document, the option 7 is provided to check with the domain vocabulary, and the option 8 is used to quit from toolkit (to PROLOG). The following few figures show the execution of some of these options.

```
yes
!?- go(sample1).

* Please Wait *

******************************************************************************
Document Name: sample1
Domain Vocabulary (content words): 113 words
Words occurred frequently: [system: 12, tracking: 5, allow: 3]
******************************************************************************
1. Quantifiers    2. Relations and Hierarchies
3. Rule Indicators   4. Explanation
5. Sentences with Process Information & Conceptual graph
******************************************************************************
Enter your choice: 1
Line Number: 10 * quantifier_indicator: [several] *
The system shall be compatible with the existing BUTEC computer system to
the extent of being operable in parallel with that system for a period of
several months.

more ?n
```

Figure 21c: Choice selection option 1

```
******************************************************************************
Document Name: sample1
Domain Vocabulary (content words): 113 words
Words occurred frequently: [system: 12, tracking: 5, allow: 3]
******************************************************************************
1. Quantifiers    2. Relations and Hierarchies
3. Rule Indicators   4. Explanation
5. Sentences with Process Information & Conceptual graph
******************************************************************************
Enter your choice: 2
Line Number: 9 * relation_indicator: [have] *
The system shall have a very high level of operating availability and
reliability and to assist in achieving this, extensive hardware back-up
facilities shall be provided.

more ?n
```

Figure 21d: Choice selection option 2
Option 5: this option is provided in the main menu for scanning the text for sentences containing process information in the document. The sentence selection criterion (as discussed in Section 4.3.1) requires both the presence of a high frequency content word and a process word (a list is provided in Table 10 above).

The following figure shows the execution of option 5 where a sample text being processed:

```
yes
? - go(sample1).

* Please Wait *

**************
Document Name: sample1
Domain Vocabulary (content words): 113 words
Words occurred frequently: [system:12, tracking:5, allow:3]
**************
1. Quantifiers  2. Relations and Hierarchies
3. Rule Indicators  4. Explanation
5. Sentences with Process Information & Conceptual graph
**************
Enter your choice: 5
* Line No.: 1, Content word: "system", Process word: allow *
The system shall allow tracking in three dimensions of up to 10 independent, mobile or static objects; each object using a choice of one of 5 different acoustic transmitter frequencies (in the range 11-23 kHz) and one of 3 different pulse repetition intervals (0.8, 1.6, 3.2 seconds).

* SYNTACTIC ANALYSIS *
* difficult *
more ?
```
The system shall allow the depth of each transmitting object to be decoded, from the acoustic telemetry pulse timing, with an overall accuracy within 2.5 meters of the telemetered depth.

The next activity in this option (5) is to generate a conceptual graph. In order to generate a conceptual graph a parser is required to analyse a sentence and to find agent and object information in a sentence (note that in the above figure it was difficult for the program to parse the sentence). A simple parser and around eight PROLOG routines are used in this option to represent a linear form of a conceptual graph. The suggested approach has been discussed in Section 4.3.2; a possible implementation with a limited parsing facility is demonstrated in the following:

```
yes
I  ?- go(sample).
```

Enter your choice: 5

```
1. Quantifiers 2. Relations and Hierarchies
3. Rule Indicators 4. Explanation
5. Sentences with Process Information & Conceptual graph

Enter your choice: 5

The system shall allow tracking in three dimensions for ten objects.

```
[allow]-
(agt)->[system];
(manr)->[tracking],
(space)->[three];
(obj)->[object],
(qnty)->[ten];
].
```

more ?y
The system generates its data fast.

The screen in Figure 21 (g) shows the execution of option 5, where a process indicating sentence is first selected then converted into a linear form of a conceptual graph. Here, a sample text file is selected for generation of conceptual graphs and the text available in this file contains simple declarative sentences for which the dictionary and grammar are available in the program. However, for a full facility various aspects of this toolkit need further research, such as a powerful parser with a comprehensive dictionary and canonical graphs extracting facilities, a conceptual graph generator which can cope with representing a highly linked graph, etc.

4.5 Concluding remarks

In this chapter we have presented our work on using notions of linguistics and information retrieval strategies for acquiring the specification knowledge from text related to the client's organisation. The linguistic cues probing notion is similar to the ontological analysis which is normally discussed for expert system during knowledge acquisition. General purpose lexical data has been used to collect the requirements specification knowledge in the form of system objects, relationships, rules and constraints from interview transcripts and other system documents.

The second consideration of this approach is based on eliciting the process knowledge available in the domain and the use of a meaning representation schemata to represent key sentences. An algorithm has been devised for selecting sentences which contain process or function knowledge. Special purpose lexical knowledge and content recognition strategies were used for this purpose to select meaningful sentences from the text. The semantic interpretation of these meaningful sentences is done by conceptual graphs, which starts after parsing a sentence for syntactic information. It determines the order of joining canonical graphs associated with each input word. The canonical dictionary definitions associated with each word can be used in this process to disambiguate an erroneous graph generation.
A linguistic toolkit has been developed in Prolog which is based on linguistic cue probing and retrieval of the process information from text related to client's organisation. We believe such a tool can help an analyst understand the application domain quickly. The activities of requirements elicitation and understanding are also supported by a conceptual schema that can be used to remove various ambiguities in a client's description of requirements. After using such a proposed linguistic toolkit for early stages of requirements definition, an analyst would then proceed to the specification process (using more conventional techniques and methods).
Several disciplines complementary to software requirements definitions were examined in this thesis. The purpose was to explore methods and techniques that could help an analyst in requirements elicitation and understanding. In this chapter we present our conclusions. First, we present a discussion of requirements definition activities in general terms (Section 5.1). Requirements capturing activities in software development appear to be similar to knowledge acquisition activities for an expert system: the relevance of this similarity is discussed next (Section 5.2). We note that natural language is used extensively during the requirements definition activity: the role of natural language processing in requirements elicitation and understanding is discussed next (Section 5.3). Finally, we discuss future work for software requirements elicitation and understanding in the natural language paradigm (Section 5.4).

5.1 Requirements Definition: A Cognitive Activity

The task of software requirements definition starts immediately after the inception of a software project, and it is principally aimed at producing a software specification. There are many tasks associated with specifying a software system which will satisfy a client's intentions. It is these tasks which distinguish the requirements definition activity from software design, coding and testing activities. We can elaborate on this distinction by noting three points. First, it can be argued that requirements definition activities are cognitive in nature and have close parallels with natural language acquisition, comprehension and production: an analyst produces a specification document (language production), after having performed certain knowledge acquisition tasks (language acquisition) and analysis and understanding tasks (language comprehension). The second distinguishing feature of the requirements definition activity is that it is knowledge-intensive, but in a different sense to that used in AI or used in other aspects of software development (i.e. software implementation). Here, the analyst is supposed to work with a set of software engineering concepts (e.g. inputs, outputs, functions, strategic needs, hardware requirements) and synthesises this knowledge with that of the application domain. The third distinguishing feature of requirements definition is that the analyst has to specify the application domain knowledge in such a way that it can be communicated to the client: this skill clearly distinguishes the experienced analyst from a novice. These distinctions are now discussed briefly in the following.

The Process of Producing a Specification

Due to the importance of client's requirements, it is essential to build a coherent picture of a proposed system prior to its specification. In order to produce a coherent picture,
it is essential to achieve a high level of understanding through the iterative processes of requirements acquisition, expression, analysis and specification. We have suggested in Chapter 1 that these processes can be viewed as the overlapping activities of elicitation, understanding and specification of user software requirements. These activities resemble human cognitive activities used for natural language based communication: 'acquisition', 'comprehension', and 'production' of natural language. These topics are discussed extensively in the psycholinguistics literature, where the researchers are more concerned with how a human acquires language, how humans comprehend language, and then produce it. This can be compared with an analyst who performs the requirements acquisition task during software requirements definition, then comprehends these requirements and finally produces a software specification.

When an understanding is achieved of any given situation, according to psycholinguistic theories, what results is called a 'global representation' (Clark & Clark 1977:161). The global representation is generally based on all the information taken together with world knowledge according to the reality principle[^1]. A global representation of a given situation is also essential to an analyst as a comprehender, when he or she communicates with a client. It can be argued that, in terms of the contents or knowledge of a software specification, the analyst's understanding of the proposed system will be based on his or her global representation of a given situation and his or her 'world knowledge'.

**Specification Knowledge for a Proposed System**

At the requirements definition stage, functional understanding of a proposed system is crucial to the success of the project. There are various levels at which we can understand the functionality of a system. At a minimum level, the functionality of a proposed system can be understood in terms of its inputs, processes and outputs. At a slightly higher level the functionality of a proposed system can be defined in terms of functions or processes, constraints, interfaces and performance. At a still higher level, the system may be described by its tactical and future planning needs, involving hardware and software needs, strategic client needs, maintenance requirements for the system etc. At any level of functional understanding, the system description task needs a variety of information: this information is either available in application domain documents (natural language text) or can be acquired from direct communication (speech or text) with the clients.

[^1]: According to the reality principle, listeners interpret sentences in the belief that the speaker is referring to a situation or set of ideas they can make sense of (see Clark and Clark, 1977).
Communication with the Client

We have noted (Chapter 3) that in the software industry analyst-client communication is mostly based on a notational representation mixed with natural languages for describing the behaviour of the proposed system. There are some advantages and some disadvantages attached to a notational representation depicting the behaviour of the software during the requirements definition activity. First, an analyst can present his or her understanding in notational form concisely, showing whatever he or she has understood about the proposed system behaviour. However, in producing this understanding he or she is influenced and constrained by the constructs available in the techniques or methods, and this can prejudice and prematurely suppress the available information for requirements expression. This disadvantage is not crucial if the application domain is already understood: here methods like SSADM, JSD, SADT or CORE are perhaps easy to use in dealing with the proposed software development. However, when the domain is unfamiliar these methods leave much to be desired at the initial stages of software development.

5.2 Capturing Requirements

Under this heading, let us first consider the types of knowledge acquired for an expert system from a human expert: knowledge acquisition systems are generally designed for eliciting specific types of knowledge such as problem-solving capabilities (procedural knowledge), heuristic knowledge (experiential knowledge or rules of thumb: 'if and then' rules) and taxonomical knowledge (the hierarchical knowledge regarding relationships among domain objects).

The above mentioned types of knowledge are also relevant for eliciting and understanding the requirements of a conventional software system, and therefore, it appears that the techniques developed for knowledge acquisition (i.e. for building expert systems) can be applied to requirements acquisition for a conventional system. This can lead us to knowledge acquisition/elicitation techniques available in AI namely: Protocol Analysis; Ontological Analysis; Structured Analysis of Knowledge and Interviewing Techniques (Section 2.4). Also there are a number of computer based automatic knowledge acquisition systems currently available, which are generally based on the above mentioned techniques, for example the ETSYSTEM reported by Boose (1984) and the KITTEN system by Shaw & Gaines (1988). Gaines and Boose have indicated that these systems have used most of the above mentioned techniques.

As far as requirements acquisition for a conventional system is concerned, there are
a number of constraints on the use of automatic knowledge acquisition systems. First, the domains for conventional systems are not as well defined as in expert system applications. Second, problem-solving activities are not centred around one person (an expert) in conventional systems. Thirdly, conventional computer systems are not generally designed for a well-motivated user, and are meant to solve a range of problems. In general, for a conventional system, an analyst refines and analyses a number of problems in many areas related to an application domain, in order to find the best possible computer solution, rather than merely involving him/herself in a so-called 'knowledge engineering' task (Jackson 1989) using specialised domain knowledge.

In the early period of our research we had intended to build a 'requirements definition/capture' expert system (Alam 1988). We were inspired by the successes, or more accurately the success stories, of expert systems in numerous application domains and we were encouraged to find that there were a number of other expert system projects in the requirements definition field. Some preceded us, like Stephens and Whitehead's 'Analyst' (1985) and some were reported later, like Hughes et al. 'ASPiS' (1988). There were a number of contemporary projects: Loucopoulos and Champion's RST project and Kramer et al.’s TARA.

The fact that the above mentioned systems, by and large, remain laboratory prototypes suggests that the 'knowledge' of requirements definition is more than problem-solving, domain specific knowledge: the type of knowledge characterising subjects like Microbial Infection Theory, Geochemical Prospecting or Chemical Plants. The knowledge which is used in requirements definition is the knowledge of how to decode knowledge, how to analyse the results of the decoding process and then how to encode the results of the analysis for communication with the people whose domain knowledge has been decoded. In terms of cognitive science, requirements definition depends on a range of cognitive faculties comprising language, representation, reasoning, comprehension, memory etc.

Our impression that the knowledge of requirements definition/capture was really not just the problem-solving knowledge of a specialised domain was further reinforced by our knowledge acquisition sessions with the expert analysts.

5.3 Natural Language Processing in Requirements Definition
Natural language is generally regarded as vague and ambiguous as compared to some other formal notations. Odell (1981) has outlined ten different points which preclude
the possibility of processing unrestricted natural language by computers[2]. Odell's arguments imply that in order to avoid ambiguity and vagueness one should use a restricted and formal artificial language. However, as Sowa (1984) has argued, these limitations are equally applicable to any artificial language. He supported his arguments by saying that "... most artificial languages may be viewed as extensions or abbreviated forms of natural languages", and "if English (a natural language) did not have a capability for being precise, the languages defined in terms of it could never be precise", (ibid 1984:341).

The point we wish to make here is that whilst we are aware of the inherent ambiguity and vagueness of natural language one cannot ignore the (obvious) fact that natural language is the principal medium for encoding/decoding knowledge. We believe an objective and non-intrusive way of analysing information in (client domain) documents would be to adopt schema based Natural Language Processing (NLP) techniques. Natural language representation schemata were discussed in Chapter 2, where we focussed on 'semantic nets' and 'conceptual graphs'. Particularly, we considered how these schemata help in various psycholinguistic topics including language acquisition, comprehension, and production (Clark & Clark 1977, Greene 1988). In computational linguistics (on the application side), a number of programs have been successfully developed to demonstrate the efficacy of schema based NLP (see Simmons 1984 for details). This has caused a renewed interest in such NLP techniques by various research communities like AI, Information Science, Linguistics and others.

In software engineering, natural language issues are mainly considered during the writing of a software document. Here, it is generally recommended that natural language should be used concisely and in an unambiguous manner. We have noticed some awareness of terminological issues at the requirements definition stages of software development during our surveys in Chapter 3. The importance of natural language is not seriously discussed in methods, tools and techniques for various activities of software requirements definition, particularly for understanding the client's intentions. Here, we believe that the notion of changes in requirements for a software system are directly related to the question of predicting the client's intentions. One way of solving this problem, that of changes in requirements, is to understand the application domain as quickly as possible. This has a direct bearing on understanding

---

2 The ten characteristics pointed out by Odell (1981) as limitations in natural language include: Context, Emphasis, Multiple speech acts, Intentionality, Non-functionality, Family resemblance, Overlapping and criss-crossing definitions, Open texture, Continuity and Sincerity.
the functionalities of a proposed system. Here, natural language processing techniques can play an important role.

We were motivated to use the concepts of natural language processing in the requirements definition stages for a variety of reasons:

1. specifications are written and understood in natural language;
2. changes to system requirements are the biggest issue in software development, and have a direct bearing on how requirements are understood and elicited for subsequent use;
3. clients used a domain specific language (user jargons) to disseminate their knowledge;
4. reaching agreement on what a term denotes is important during requirements definition;
5. it is easy to detect logical ambiguity in natural language, albeit its chances of occurring are more in natural language than in (formal) logic;
6. it is difficult to incorporate various forms of world knowledge in the specification documents other than by expressing such knowledge in natural language;
7. natural language does not constrain an analyst's thinking during the requirements definition stage, if it is used as a tool and provides no special constructs of its own.

We have described our work in Chapter 4 in the context of NLP framework, where we had some specific goals regarding requirements definition: to elicit and understand software requirements in terms of its system knowledge (i.e. objects, object relationships, object hierarchies, properties and attributes of objects, functions, performance, constraints and interfaces). We believe that system knowledge can be explored, elaborated, and quantified by examining certain classes of words and phrases (i.e. lexical entries) from text related to a client's organisation. Indeed, we notice that the instances of these lexical entries are (grammatically) categorised and have some specific semantic properties in a natural language (i.e. English). The instances of lexical entries used in everyday communication e.g. various forms of determiners, type and relation indicators, rule indicators and so on, illustrate our point. We believe that an analyst uses such lexical knowledge subjectively, without any emphasis on its grammatical categories or semantic properties, particularly for comprehending various information for the proposed system during analyst-client communications. We can supplement this activity in two ways. First, knowledge in the form of specific
sentences can be collected with the help of lexical entries (e.g. quantifies, determiners, so on), and then specific objects in the form of words and phrases can be extracted from the sentences for an analysis which is similar to ontological analysis. Second, a preprocessor can be used to highlight various attributes of a sentence (e.g. objects, agent information and their properties) in the documents as a 'visual aid'.

We have synthesised concepts and techniques from three fields for developing the linguistic toolkit that is, linguistics, information retrieval and knowledge representation. We consider this synthesis important because we believe that our approach expedites the understanding of the client's domain. It is essential to understand the domain by understanding the underlying knowledge of the domain, particularly the knowledge related to processes of the domain. In pursuance of eliciting this special knowledge, sentence selection criteria have been specified. The sentence selection criteria are based on information retrieval strategies, particularly frequency matrices, for automatically selecting the 'appropriate' or 'meaningful' sentences from documents of the application domain. Later, the 'conceptual graph formalism' was used for domain understanding by utilising of the knowledge embedded in the selected sentences, through a systematic use of 'concepts', 'relations' and 'concept-based canonical relations' of a domain. The following figure shows our scheme of text processing pictorially for the requirements definition stage.

![Figure 22: Text Processing for Requirements Definition Stage](image)

The work of Sowa and Way is the earliest cited literature on "implementing a
Chapter 5 Summary and Conclusion

semantic interpreter using conceptual graphs" (1986:57). There, the main motivation was to represent the meaning of the sentence automatically, using an interpreter which, in turn, joins canonical graphs associated with each input word to form a larger graph which represents the entire sentence. Here we are interested in the basic issues of elicitation and understanding during requirements definition for software development. In related but different work, the RST project (Section 2.2) carried out under an Alvey sponsored program, the authors have claimed that conceptual graphs are used for requirements eliciting strategies. Both the RST project and our linguistic toolkit emphasise the use of conceptual graphs. However, our goal was different from those of the RST project in that we use conceptual graphs for explicating the client's requirements, whereas, RST project used conceptual graphs for representing the so-called method knowledge only. Our work in this respect can also be distinguished by the notions of pre-processing the text for conceptual graphs generation. That is, eliciting the specialised knowledge directly from the text using a conceptual graphs formalism to represent its meaning.

5.4 Future Work
Researchers in the field of software requirements definition are generally interested in having a unified approach to requirements definition. The reality, of course, is different. Therefore, in order to answer the question as to why we find so many methods and techniques as opposed to one unified approach for software requirements definition activity, the following points are worth considering. First, advances in software engineering, particularly new programming paradigms, the use of formal methods, adaptation of mathematical theories and so on, inspires the software engineering community to develop new tools: this has led to technological proliferation of tools. Secondly, the requirements of one application are so different from the requirements of others that the requirements definition practices can not be generalised; hence, tool developers see the need for specialised methods. This has led to the proliferation of applications-oriented tools.

We have advocated the use of natural language processing tools for requirements elicitation and understanding and believe that the use of such tools will be a precursor to a unified requirements definition environment. A proper linguistic tool will provide an unified and non-intrusive approach to analysts-client communication activity. Immediate next steps toward this goal would be as follows:
1. Full implementation of an integrated system as suggested in Section 4.2, and 4.3;
2. Better representation schema for natural language knowledge;
3. Integrated use of this requirements definition linguistic toolkit with more conventional system modelling facilities (e.g. SADT).

Our work may be seen as a useful pre-requisite to Greenspan's work (1984) for producing a RML, which is based on SADT modelling of a proposed system. An ASPIs type of project (Section 2.2) requires a similar pre-requisite, as the questions of elicitation and understanding for software requirements definition were again not addressed. The following picture suggests how our work might contribute to an integrated requirements modelling system:

![Diagram](image)

Figure 23: Pictorial view of an integrated approach to an analyst workbench

In conclusion, requirements definition is a (task-oriented) cognitive activity. It needs methods, tools and techniques which must be developed with the knowledge of human cognitive faculties in mind.
Appendix A: Natural Language Syntax and Semantics

A.1 Natural Language basics: natural language is basically an instrument of communication (Clark and Clark 1977). Greene (1988) has noted two important functions of language: "external communications with other people and internal representation of our own thought." The external communications include both written or spoken forms of the language, where a variety of linguistic knowledge is used to understand the speaker's intentions. Traditionally the types of knowledge required for language understanding have been divided into the following categories:

language basics:
(a) Prosody: the rhythm and intonation patterns of language,
(b) Phonology: the sounds or phonemes of language,
(c) Morphology: the meaningful elements or morphemes that make up the word,
(d) Lexical meanings: meanings of words listed in the dictionary,

Language structure:
(e) In the form of Syntax: the rules for combining words into phrases and sentences.

Language Function:
(f) In the form of semantics: meaning and its expression,
(g) Pragmatics: the use of language and its effects on the listener,
[Sowa, 1984, Greene 1988].

The study of language in linguistics generally includes the categories of knowledge mentioned above from (a) to (g), whereas, in language psychology or psycholinguistics domain, researchers are more concerned about representation, comprehension, production, and learning of natural language by humans. Since our aim is to deal with this knowledge in the context of a machine's understanding it is perhaps better to discuss the key concepts of natural language processing in this appendix.

A.2 Natural Language Syntax
Phrase Structure Grammar: There are many models used in syntactical analysis of natural language, the fore-runner of most of them is the 'phrase structure grammar' (or context-free phrase structure grammar). For example, in the English language a sentence (S) is generally defined at a syntactic level in terms of the structures of its constituents: Noun Phrase (NP) and Verb Phrase (VP). A context-free grammar form of this structure would look like this:

Rule [simple sentence formation]
\[ S \rightarrow NP + VP \]
Rule [transitive verb]
\[ VP \rightarrow TV + NP \]
TV := drink, eat
Rule [intransitive verb]
\[ VP \rightarrow IV \]
IV := flies, melt
Rule [minimal requirement: a noun]
\[ NP \rightarrow N \]
N := birds, snows
Rule [pre-modification]
\[ NP \rightarrow PreM + N \]
PreM := IDENTIFIER, NUMERAL/QUANTIFIER,
Appendix A

ADJECTIVE, NOUN_MODIFIER

Rule (post modification of relative clauses)
NP := NP + PostM
PostM := S1 {Relative Clauses}
...
...
...
Rule (Relative Clauses)
S1 := S

This type of context-free grammar constitutes a general parsing system for defining the expressions of a language in terms of rules. The rules are recursive equations over expression types (e.g. Verb Phrase, Noun Phrase) called non-terminals; and primitive expressions (e.g. Nouns, Verbs) called terminals in the literature. However, it is now generally believed that a context-free phrase structure grammar, in its simplest form, is not sufficient to analyse the whole range of syntactic structures which occur in natural language, either in speech or in text. This was realised after the attempt to parse sentences like time flies like an arrow. The first three words in this sentence belong to more then one part of speech: time and flies could be either nouns or verbs. Sowa (1984) suggested that a possible resolution of these ambiguities would be to use context. For more details on this subject the reader is referred to Welin, (1979), Winograd, (1984), Gazdar and Mellish, (1989).

The second mostly widely discussed approach to sentence parsing is the Augmented Transition Network.

Augmented Transition Networks (ATN): Augmented transition network grammars (ATNs) and their descendants are currently one of the most common methods of parsing natural language in computer systems [Winograd (1983), Gazdar and Mellish, (1989)]. ATNs were first developed for use in natural language understanding systems such as LUNAR (Woods 1973), a system that answers English questions about the Apollo II moon rocks.

Basically, ATNs consist of 'recursive transitive networks' (RTN) with labels and arcs which characterise syntactic categories instead of the rules used in context-free grammars. The augmentation centres around the addition of so-called conditions and actions associated with the arcs of a RTN. Conditions restrict the circumstances under which an arc can be selected, while the 'actions' perform feature-making and structure building operations. Conditions and actions make use of registers for roles and features, associated with the nodes of the parse tree being constructed. Registers are similar to the variables of a programming language, each having a name and storing some information. This formalism deals directly with concepts such as alternative sequences, optional constituents, and arbitrarily repeatable constituents in natural language. A BNF specification of the ATN from Shapiro (1989) is given below:

```
<ATN> -> (state> <state>*
<state> -> (state-name> <arc> <arc>*
<arc> -> (CAT <category-name> <augmentation>* (TO state-name>) I
(WRD <English-word> <augmentation>* (TO <state-name>) I
(PUSH <state-name> <augmentation>* (TO <state-name>) I
(POP <expression> <augmentation>*
(JUMP <state-name> <augmentation>*

<augmentation> -> (VERIFY <condition>) I <action>
?action> -> <register-name> <- <expression>
(SENDR <register-name> <expression>) I
(<defined-operator> <expression>*
```

113

Woods himself has elaborated this BNF as:

"It is similar to most ATN formalisms, except that conditions on arcs are expressed in terms of an action (VERIFY<condition>), and an infix assignment operator (<-) is used in place of the more customary SETR function, and functions (NE and PC) are used to refer to the next input element and the parsed constituent of a push arc, respectively (in place of asterisk, which served both purposes in, [Woods, 1970]).

In this notation an ATN specification consists of a list of state specifications each of which consists of a state name and set of arc specifications. Arcs can be one of the five indicated types. A CAT arc accepts a word that is recorded in a dictionary as belonging to the specified syntactic (or semantic) category; a WRD arc accepts the specific word named on the arc; a PUSH arc invokes a subordinate level to the ATN to recognize a phrase beginning with the specified state; a POP arc signals the completion of a phrase and specifies an expression for the value that is to be returned as the structure for that phrase. A JUMP arc specifies a transfer of control from one state to another without consuming any input." (1989:328)

The diagrammatic representation of an ATN grammar is shown in Figure 24. This figure effectively represents the recursive nature of phrases which occur in natural language, where states are represented by small circles and arcs are represented by arrows connecting states. Each arc is labelled with the name of the kind of constituent that will enable that transition if it is found at that point in the input string. This sample grammar has three levels: S for sentence, NP for noun phrase, and PP for prepositional phrase.

Each level begins with a state whose name indicates the kind of constituent being searched for. In the naming convention used here, a state name consists of the name of...
the constituent being sought, followed by a slash (/), followed by a brief mnemonic indication of what has been found so far. This naming convention is not an essential part of a transition network grammar but is a useful device for making grammars readable. Each level ends with one or more final states (indicated by a short arrow labelled as POP), which marks the successful completion of a phrase. A sequence of arcs from a start state to a final state defines a sequence of constituents that can make up a phrase of the kind sought by the start state.

A variety of extended ATNs formalism are reported in the literature (e.g. Boguraev, 1983, Winograd 1983, Woods in Shapiro 1989). "Although ATNs still get used, there are number of trends in NLP that are now leading to their decline. The original theory of transformational grammar whose operations they sought to embody has changed out of all recognition" (Gazdar & Mellish 1989:96).

A.3 Natural Language Semantics: Conceptual Dependency (CD)
Schank's conceptual dependency structure is another formalism which attempts to model 'meaning structure in a discourse'. Schank (1969) has based his CD theory on the claims that there is a language-free predetermined set of possible relationships that make an interlingual translation possible during (human) language translation. Schank first divides the core of an event into four categories of words. The distinction is made on words as 'Picture Producers' (PP), and those which are 'Picture Aiders' (PA) in producing a picture. The terms which represent 'actions' and 'action helpers' have been separately classified as ACTS and AA respectively. Schank's classification of words, especially words which are used to represent knowledge in effect are words which are regarded as 'open class' of words in English (i.e. noun, adjective, verb and adverb).

The dependencies between the PP's and PA's words for a given situation are specified in a 'conceptualisation'. Basically a conceptualisation represents an actor (ACTOR) performing an action (ACT) under a variety of constraints and supports. Note that the actual number of ACTS primitives, as defined by Schank, has varied over the years. Rosenberg, however, has arranged eleven ACT primitives of the CD theory into four groups:

**Physical ACTS**
- PROPEL: apply a force to (e.g. push, throw) a body
- MOVE: move a body part (e.g. scratch, kiss)
- INGEST: take something inside (e.g. eat, drink)
- EXPEL: take something outside or force out (e.g. sweat, exhale)

**Global ACTS**
- PTRANS: expresses a change in the physical location of an object (e.g. go, put)
- ATRANS: operates on abstract relationships (e.g. ownership and possession, give, buy, take)

**Instrumental ACTS**
- ATTEND: takes sense organs (eyes, ears) as objects (see, listen)
- SPEAK: produce sounds; its objects are sound (e.g. say)

**Mental ACTS**
- MTRANS: to handle the flow of information to and from the conscious mind (underlies verbs like recall, memorise, perceive)
- MBUILD: account for though combination (e.g. conclude, resolve, solve)

According to Schank this set of action primitives generalises all the most important
primitive action words. Schank has elaborated his ideas by arguing that:

"In CD, TRANSFER POSSESSION is called ATRANS, which is a primitive action. Whatever we call it, that name has no real meaning in an understanding system apart from what happens in that system as a result of its occurrence. In other words, the actual meaning of ATRANS in the system is the set of inference rules that fire off because of its presence, no more and no less." (1981:16)

In the above we have defined four categories of words and about eleven primitives actions (ACTs) in CD theory. To complete the language of CD theory, Schank includes conceptual cases of events analogous to the case structure of Fillmore, (1968). Hence, with the help of these 'conceptual cases' Schank represents events in the form of a network of graphs. Some of these cases include: 'objective' case which relates an objective PP to an ACT; the 'recipient' case, which relates a donor PP and recipient PP to an ACT; the 'directive' case, which relates a direction (to and from) to an ACT, and the 'instrumental' case, which link conceptualisations instrumental to an ACT to a conceptualisation containing the ACT. This case relation between Actors and Actions may be elaborated as follows:

Actors perform actions.
Actions have objects.
Actions have instruments.
Actions may have recipients.
Actions may have directions.

The conceptualisation in a CD graph also includes tense markers to indicate past, future, conditional, or continuation of an event. A graphical representation of conceptual dependencies, as given originally by Rosenberg 1980, will help to illustrate some of the notation form of CD theory:

(1) Mary sold her car to Beth

Mary transferred ownership of the car from herself to Beth which caused Beth to transfer her money from herself to Mary.

The interesting aspect of the CD representation scheme, from the point of view of requirements elicitation and understanding, is its inferential power. The inferences drawn by Schank et al.'s MARGIE (1973) program in natural language understanding demonstrate how such a system can be used for requirements definition activities: the input sentence given to MARGIE for inferences was "John gave Mary an aspirin." (Rosenberg 1980)

1. John believes that Mary wants an aspirin
2. Mary is sick.
3. Mary wants to feel better.
4. Mary will ingest the aspirin.

Conceptual dependency theory and its computational realisations, indicate the relevance of such an approach to systems which may be developed for requirements elicitation. In particular we believe the inferential power of a CD-based system will help to elicit user needs and requirements by analysing sentences taken from documents produced by the user. The representation of these sentences together with the possible inferences drawn from them will be of help to the analyst. Similarly, sentences from an analyst can be analysed for verifying 'conceptual' dependencies which the analyst thinks characterise a system built to satisfy the user's needs.

There are, however, two constraints in the use of CD theory for requirements definition. First, this representation does not produce a 'clutter-free' graph for the purpose of discussion with the client, which are frequently needed during the requirements definition stage. Secondly, because of its strategy of keeping the primitive processes to a minimum, CD theory can introduce constraints during requirements analysis. In the next section we discuss another meaning-representation schema which might be more relevant to the needs of a requirements elicitation system.

Case Features: The notion of case traditionally refers to the classification of nouns according to their syntactic role in a sentence, as it is inflected by various forms of nouns at a syntactic level (Bruce & Moser 1989). In English, pronouns have these case inflections according to their use as subject, object, or possessive article (e.g. the first person singular pronoun is 'I', nominative case, 'me', accusative/objective case, or 'my' genitive/possessive case). In languages such as Greek all nouns are given affixes that indicate their case. "The idea of a direct relationship between inflections and cases is one kind of case, also called 'surface' or 'syntactic level' case" (ibid 1989:333). These surface cases do not invoke any semantic relations with the action verb in a sentence, hence cases of this type are not very helpful in natural language processing.

Another sense of 'case' is a categorisation of noun phrases according to their 'conceptual roles' in accordance with the action verb in a sentence ([Jespersen 1965), (Fillmore 1968)]. These cases are distinguished from the above by calling them 'deep cases', 'semantic case' or 'theta role' of noun phrases. For example, in the action described by a sentence "John kicked the football with his foot", John is playing the conceptual role of a kicker of a physical object playing the role of 'football'. These roles could be formalised in a 'case frame' for the verb 'kick', where 'John' is an animate object generalised as an agent in the slot of case frame of 'kick' as:

{agent}: animate object,
(object): physical object,
(instrument): physical object,
(source): location,
(goal): location).

Here, the curly brackets are used to indicate those slots which are optional in the case structure. This case frame will help in rejecting a sentence like "John kicked the new idea." This sentence is unreasonable because the sense of 'kick' used here seems to require a concrete object, and in this case 'new idea' is not a concrete object.

The theory of case features suggests an approach to the representation of sentence meaning and is important in relating the structure of a sentence to that of its meanings. Application of case theory to intelligent systems have ranged from a medical model of glaucoma to speech understanding (references can be found in Bruce & Moser 1989). Most natural-language systems make use of these ideas in some form (For a survey of implemented systems using case grammars, see Bruce, 1975).
Discourse Representation Theory (DRT): The analysis of a discourse proceeds in DRT, first by constructing an 'intermediate structure' in USL. The 'intermediate structure' is a tree structure which is constructed through syntactic analysis of the sentences and consists of the following different type of nodes:

* RELATION nodes (R-nodes) consisting of a predicator and a list of ARGUMENT nodes.
* ARGUMENT nodes (A-nodes) consisting of a role name and a node of type NOMSTR or VERBSTR.
* NOMSTR nodes (NOM-nodes), which list features of nouns (including quantification and negation) and an R-node or a constant.
* VERBSTR nodes (V-nodes), which list features of verbs (including verb negation) and an R-node.

The 'intermediate structure' of a sentence "every accident is an event" thus looks like:

```
V(
  R(is,
    (A(NOM, N(every, R(accident, nil))),
     A(NOM, N(a, R(event, nil)))))
```

Note that the processing of the 'intermediate structure' always starts with the verb at the top node. Conditions and arguments expressed by the verb are then recursively processed to write down the whole structure with the help of typed nodes as defined above, where NOM is use in this case to indicate nominative role name of the sentence.

In the second stage of DRT a Discourse Representation Structure (DRS) of the sentence is constructed, which basically comprises a pair consisting of a set of 'reference-markers' and a set of a conditions, in the form <U:C>. These schemes are termed as DRS in DRT. A Backus-Naur Form to characterise the syntax of DRSs is presented in part (B) of this appendix. To write down the condition expressed by the verb, all its arguments are processed recursively. The predicator and the arguments in the 'intermediate representation', in the above example sentence, thus provide a DRS reference-markers and conditions as:

```
[e1:accident(e1)] ->[e2:event(e2).e1 = e2]].
```

The representation in DRT is completed after the resolution of contextual references as well as elimination of redundant conditions. For this, the DRT requires application of meaning rules. In DTR theory meaning rules can be presented in one of the different schemata of knowledge representation (e.g. semantic nets, conceptual graphs or frames).

**Backus-Naur Form to characterise the syntax of DRSs**

```
<drs> ::=<reference-marker-list>:<conditions>l
    [<conditions>]l
    [<reference-marker-list>]

<reference-marker-list> ::=<reference-marker>l
    <reference-marker>, <reference-marker-list>

<reference-marker> ::=<object-marker>l
    <et-marker>
```

118
<object-marker> ::= u<number>
<et-marker> ::=<event-marker> | <time-interval-marker>
<event-marker> ::= e<number>
<time-interval-marker> ::= t<number> | n
<conditions> ::=<condition> | <condition>, <conditions>
<condition> ::=<atomic-condition> |<conditional-condition> |<disjunctive-condition> |<negative-condition> |<event-condition>
<atomic-condition> ::=<predicator> (<argument-list>) |<predicator>|<term> = <term>|<et-marker> ≦<et-marker>|<et-marker> < <et-marker>|<et-marker> o <et-marker>
<predicator> ::=<identifier>
<argument_list> ::=<term> |<argument_list>
<term> ::=<reference-marker> |<number>|<functor> (<argument-list>)
<functor> ::=<identifier>
<conditional-condition> ::=<drs> -> <drs>
<negative-condition> ::=¬<drs>
<even-condition> ::=<even-marker> : <drs>
A.4 Natural Language Processing System

The main theme on which most of the Natural Language Processing (NLP) systems work, in the context of natural language understanding, is the so-called 'schema theory': "The basic idea, originally suggested by Bartlett (1932), is that human memory consists of high-level structures known as schemas, each of which encapsulates our knowledge about everything connected with a particular object or event" (Greene 1988:34). In recent times Schank (1972) and Minsky (1977) have discussed schemas of knowledge "that represent the general knowledge which aids the understanding of conversations and texts as well as real-life events" (Greene 1988:35). However, when these schemas are required for machine's understanding, a variety of (language) knowledge is required (see above).

The problem of transforming a text into a well-structured data base has received much attention in NLP research. Practical methods of representing the content of the text as network of linguistic or philosophical units of words or concepts for natural language processing were first introduced by Quillian (1968). These network of linguistic units are commonly known as semantic networks, and have found some successful applications. At MIT Winograd's SHRDLU program could command a robot hand and answer questions in English about a microworld of blocks. The work of the Yale group on the subject of NLP, particularly that related to 'conceptual graphs', has been documented in books by Schank and Abelson (1977), Lehnert (1978) and Schank and Riesbeck (1980).

Many of these early systems have been implemented on computers of the 1970's which were constrained either in terms of speed or memory or both. The present interest in NLP has arisen from a changing view of the nature of computers, a shift in emphasis from number-crunching to symbolic processing. The symbols that computers can now manipulate represent more complex objects like words, sentences, trees or networks (Gazdar & Mellish 1989).

We discuss three recently reported language processing systems which have relevance to the theme of this thesis: requirements elicitation and understanding. The first system deals with traffic law and is meant to advise a lawyer in typical consultation mode. The second system is devised to perform a variety of tasks performed by an experienced indexer. The third system provides the semantic interpretation of technical text and can also acts as an intelligent tutor.

A system development in NLP starts with the aim that such programs will make a computer behave 'as if' it is intelligent. "Interestingly, the aim is usually stated as getting computer programs to understand natural language, i.e. ordinary human language with all its messy ambiguities, in contrast to the precision of the programming languages used in computer programs" (Greene 1988:101). The early attempts in this direction are documented for the programs such as ELIZA Weizenbaum (1966), BASEBALL (Green et al. 1963), SHRDLU (Winograd 1972), and MARGIE (Schank 1980). ELIZA was designed to simulate the responses of a non-directive therapist, and BASEBALL was designed to provide information about baseball games played in one season. "Winograd's computer program SHRDLU was a great step forward in modelling the human language understander's ability to produce meaningful representations using knowledge-based inference" (Greene 1988:115). "The basis of the MARGIE program was to generate inferences from CD representations of sentences, particularly inferences about all the possible causes and consequences of the events described. This led to an inferential explosion" (ibid 1988:134).

The research which started in the late fifties and early sixties in natural language processing resulted in what can essentially be recognised as demonstration programs. The research aims during the 70's and 80's, however, included application of natural language processing to specific. For example, Franz Guenthner and others (1986)
have reported a system which is intended to be used by lawyers or judges for consultation about a particular case description. Note that Guenthner et al. have entitled their paper as "A theory for the representation of knowledge". Similarly, a system due to Vleduts-Stokolov (1987), for automatic indexing, entitled as "Concept Recognition in an Automatic Text-Processing ...", is designed to recognise biological concepts in the broad domain of biological sciences. A specific application is described in a paper entitled "Semantic Interpretation of Technical Texts" by Sebastiani et al. (1986). This system deals with natural language processing and understanding of those sentences describing the mode of operation of a physical device, or the structure of complex objects. The aim of this section is, therefore, to discuss the principles of these programs in order to understand current trends in natural language processing.

A.4.1 Traffic Law processor
This prototype was intended to deal with German traffic laws in natural language (Guenthner et al. 1986). The project's objective was to use a machine for natural language discourse analysis and to have expert-like inferences and understanding in the domain of (German) traffic law. This consolidated approach utilises both semantic and pragmatic aspects of a given language, which distinguishes it from most previous approaches to the study of natural language semantics. An overall view of the system is presented in Figure 25, where three distinct processes are shown as essential for such a system:

First of all, a parsing system, the so-called 'User Specialty Language' (USL), is first activated to transform the natural language text. This USL system was originally developed for natural language-based interaction with databases [(Lehmann 1980) (Ott & Zoeppritz 1979) (Lehmann 1980) (Zoeppritz 1984)] to process English, French, Italian and Spanish as well as German query sentences. Secondly, the parsed information are translated into logical form in 'discourse representing structures' to make it available for a question-answering system. This logical transformation is achieved through a formalism termed by the authors as 'Discourse Representation Theory': DTR (see above). The discourse representation in DRT completes after the resolution of contextual references as well as elimination of redundant conditions. For this, the DRT requires pre-processed fragments of natural language text in USL. At the third stage, where question-answering is achieved in this system, various meaning structures can be used such as 'frames', 'semantic nets' or 'conceptual graphs'.

Guenthner et al. (1986) have discussed a number of extensions as research areas for this project in the future:

Figure 25 System overview of Traffic Law Processor
1. Problems of adequate semantic representation of phenomena, such as causality, ability, etc.
2. Problems of discourse pragmatics, on which depend the proper treatment of contextual references, appropriate system reactions in a dialogue with a user, and other phenomena which in conventional computer applications are addressed under the label of 'user friendliness'.
3. Problems of deductive strategies, which have often been addressed in the Artificial Intelligence literature, but are widely accepted as still needing much further investigation." (ibid 1986:54)

Next we discuss a system with pre-stored indexing knowledge of an expert indexer on a subject of biological sciences in significance of natural language processing.

A.4.2 Automatic Indexing with Concept Recognition
Of all the procedures normally used in a document processing environment, the most important and also the most difficult one, is to recognise the bibliographic items capable of representing the document content (Salton & McGill 1983). These activities require the capabilities of a highly skilled indexer. A system reported in Vleduts-Stokolov (1987) is claimed to be capable of recognising biological concepts from text in biological sciences.

Initially, the system translates the subject titles (which require classification) into a semantic representation. The second process conceptualises these semantic representations into subject headings (indexed). The 'part-whole' relationship is used here as an intellectual process for the adequate recognition of many 'conceptual heading'. There are around 600 recognisable biological concepts (i.e. concept headings) in the system. The following figure summarises Vleduts-Stokolov's work into a SADT type of representation, (Figure 26):

Figure 26 Automatic Indexing System description at block level

The semantic vocabulary, used during the translation process (see Box 1 in the above figure) of the formal representation, defines 'significant' versus 'insignificant' meanings within the system with the help of concept primitives. The theory of Componential Analysis[1] has been used to define around 440 primitives in the domain of biological sciences. A primitive is basically defined in Componential Analysis by the meaning of more complex concept in the form of its sense components. For example, the word 'capillaries' could have a primitive 'Blood Vessels' assigned to it, and a title 'Ultrastructural changes in capillaries' containing this word may indicate the meaning 'Blood Vessels Pathology'.

From the above inferred primitive 'Blood Vessels', other primitives such as

---

1 See Lyons (1968, 1977) for more details on Componential Analysis.
'Cardiovascular System' could be inferred. This is the second process in Vledut-Stokolov's system having inferencing rules in the form of whole-part relationship such as:

"If 'Physiology of rumen' then 'Physiology of digestive system'"
"If 'Anatomy of heart' then 'Anatomy of cardiovascular system'"

Thus the above title 'Ultrastructural changes in capillaries' with the word 'capillaries' will be conceptualised first as 'Cardiovascular system' and indexed as 'Cardiovascular system Anatomy', etc. by these inferencing rules.

This natural language processing system is designed as an automatic aid to indexing with concept headings. It is based on a vast body of knowledge corresponding to approximately 15,000 concepts frequently used in biological literature. This body of knowledge associated with 440 primitive concepts represents a formalised language in the system's knowledge base, which also contains the disambiguation rules. Indexing policy rules are described by concept heading definitions in the same formalised language of primitives.

A semantic interpretation of technical texts is a difficult area for NLP. We now discuss a NLP system in the following which deals with technical text.

A.4.3 Semantic Interpretation of Technical Texts
The interpretation of 'technical text', or sentences containing specific details of scientific nature has been not only concerned the indexers but also that of knowledge engineers. A (sentence in) technical text describes the mode of operation of a physical device, or the structure of complex objects. According to the authors of this system, Sebastiani et al., "this restricted linguistic domain has a straightforward representation in terms of [knowledge representation language] KL-MAGMA" (1986:15). The KL-MAGMA (Cappelli & Moretti 1983) representation schema is generally categorised as a variant of 'structured-inheritance nets' and a version of KL-ONE [(Brachman et al. 1978), (Brachman & Schmolze 1985)]. The following figure, Figure 27, presents an overview of the system, comprising at least three distinct stages:

The system uses KL-MAGMA knowledge language to represent individual technical sentences in a structured-inheritance network scheme organised around a 'pivot' concept. The 'meaningful' elements, the noun phrase(s) of a sentence, are represented by means of instance roles of this 'pivot' concept. Instance rules are defined by stating the relationship between any lexical item to its 'superconcept' or other concepts. These representations form a 'prototypical-knowledge network', and when interconnected with other similar forms, represent the "semantic knowledge" of Sebastiani et al.'s system.
This semantic representation of the input sentence in KL-MAGMA is achieved in three stages. First, the individual sentence is parsed into its constituents parts (the system use Functional Grammar as discussed by Key, 1979). At this stage, the constituent verb of the sentence is matched with a predefined set of verbs along with their case-frames. Such a selection is necessary for subsequent stages of the system.

Second, the selected case frame is translated into a modified version of case frames. Here, Sebastiani et al. have referred to the problems with traditional case frames. This version of case frame takes care of the 'degree of binding' between a case and a verb (first suggested by Bruce 1975), which provides semantic constraint due to the inflexible binding between the case and the verb. This drawback is minimised by associating a set of case frames with each verb, and pairing them with a semantic condition. This process is intended to allow a particular case-frame to be chosen from a set depending upon the semantic properties of the linguistic items of the input sentence. Therefore, the case-frame is chosen according to the context in which the verb appears. It is not representative of the verb itself, but of a contextually bound instance of the verb.

In the third stage, a prototypical-knowledge network is instantiated with the help of other lexical items occurring in the input sentence. Sebastiani et al.'s system treats these lexical items as generic KL-MAGMA concepts, and the properties are represented either by means of roles of the concept itself or of 'superconcept' relationships with other concepts. The preceding stage could be processed in a top-down fashion, with the case-frame enforcing constraints on the search for fillers through their functional behaviour in the sentence. In this stage, however, Sebastiani et al. use a bottom-up procedure, where individual lexical items of the sentences are analysed as a possible role filler candidate in the selected case frame. All the frames associated with the verb are scanned until the corresponding cases are fully identified and added to the representation.
A Summary of Lexical semantic relations found in (Evens 1980: 118) note: reference to individual authors can be found in Evens (1980).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>class inclusion comparison</td>
<td>Gener</td>
<td>SUP/SUB</td>
<td>superordinate subordinate</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>modification attributive exemplification</td>
<td>A (generic attribute), Magn, Bon, Son S ?</td>
<td>MOD</td>
<td>infralogical relations</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>synonymy synonymy</td>
<td>Syn Figur</td>
<td>EQ</td>
<td>synonymy</td>
<td>complex (limiting of T)</td>
<td></td>
</tr>
<tr>
<td>antonymy antonymy</td>
<td>Conv Anti</td>
<td>NEG</td>
<td>antonymy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cause contingency</td>
<td>Caus</td>
<td>BECAUSE IMPLY BY</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>part/whole consituent</td>
<td>Cap, Centr Sign, Mult Equip</td>
<td>PART OF HAS PART</td>
<td>parts/whole complex</td>
<td></td>
<td></td>
</tr>
<tr>
<td>provenience provenience</td>
<td></td>
<td></td>
<td>substance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>case relations spatial function operational</td>
<td>S_{loc}</td>
<td>LOC, Theme location GOAL, Source Causal Actant</td>
<td>propositions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>text relations</td>
<td>AND, OR, NOT SINCE</td>
<td></td>
<td>plans with Q, etc.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 11 SUMMARY OF RELATIONS
Appendix B : Experts' Interview Transcript.

Q.1: What importance does system definition or Requirements Analysis (RA) has in the software development lifecycle? This is the opening question.

E1: I'm a bit surprised at the way this is expressed, because RA seems to me to be put in as an alternative definition of system definition. I would distinguish between the two. If we are really talking about requirement analysis then that is the most important part of all. I had assumed that system definition means defining an outline design of the system. If it means here, defining what the system is supposed to do, then I would understand the equation of system definition with requirements analysis.

SQ: Can you comment on the statistics that the requirement analysis stage takes up to 5-12% of the total time of any project, and the effort required to correct an error in this stage is 10-100 times less than any other stage of the project.

E1: This refers to the classical academic text book type of software cycle, you do all the RA, then design, detailed design, and so on. First of all I take issue with that assumption because I do not think that is like that in real life. I do not have much faith in the life-cycle model (RA>Design>Coding>etc.), because requirements change all the time and RA covers the whole span of most projects, especially for the bespoke systems. So the 5-12% figure mentioned above can not be related to the elapsed time of the project. Perhaps it refers to the total project effort. Because of the progressive nature of actual requirements analysis, as opposed to any 'RA phase', it is difficult to pull out the separate contribution of RA. (The original source of this quotation does not think that he was right in quoting this figure, and also the pay-back for the effort in RA does not necessarily lead to a 100 times saving!).

SQ: Can I put this in the context that your background is basically on bespoke projects rather than on software products.

E1: That is right.

SQ: It is important to distinguish between a product development environment and bespoke system development: In the former it is easy to make a distinction between the various life cycle phases, whereas with bespoke system development this is not the case.

E1: Yes, I think you are probably right, that is a useful distinction to make. Going back to my argument for an evolutionary approach (as I may not have said it, not in so many words so far), which I favour for bespoke development, it is again rather difficult to say when you find out the errors, you are finding them out all the time. I think again this relates to the assumption of a distinct set of phases where you don't actually see that you have got a requirement wrong until 10 years later when you give the system to the customer. Whereas, with the evolutionary approach you should be actually seeing something earlier in the design. But even with an evolutionary approach you have to do lots of up front analysis.

SQ: What do you like to achieve after this stage?

E1: I like to achieve more confidence in my project plan, because the RA have a very dramatic effect on the total complexity of the system. During RA, system design of some kind is usually carried out. So you will get some feeling as to whether your estimates, and your intended approach to design are correct.

SQ: How do you use the results of this stage?

E1: Same way as I would use any other information (I don't understand this question)
**E2**: What they are asking is how you define interfaces between the input and the output of consecutive phases.

**E1**: That depends to some extent on what method is used: if one of the design methods is used for requirements expression, then RA is used very formally as the first stage of design. (As in our use of JACKSON's method in one of our projects, to do just that). Subsequently, this first cut design is elaborated further to add more details to it, and might well be transformed in to some suitable form for design. However, in many cases it merely conditions the thinking. The analyst absorbs what it is required and then by some process of inspiration comes up with some kind of design. Then you think through whether you are able to provide the required facilities with the design. There is only a tangible link between the end of RA and the design process in this scenario.

**SQ**: Does the use of a particular methodology put a different slant on the Requirement Analysis phase?

**E1**: It can do, yes.

**SQ**: So what you are saying is that an analyst first selects a methodology and then thinks of requirements.

**E1**: No. There are two points to note here: (1) The use of a design language will obviously condition the way in which you extract requirements. (2) The use of a design language (method?) creates the impression of an obvious link between RA and design: Design merely becomes a process of elaborating the Requirements. This might be useful in terms of traceability. But in many cases you use the RA phase to characterise the design of a software such as saying "what it is in a data-based project." Then you add a requirement "it should be highly secure and be available non-stop." This evidence may lead a builder to look for systems which can fulfil these criteria, for example, in tandem. Then you analyse how these requirements may be fulfilled with the available tools. It is often not possible to discern a logical flow from requirements to design. Though, for bespoke systems the logical flow must be discernible.

**SQ**: What stage of the software lifecycle could mostly be affected by this phase?

**E1**: Acceptance.

**SQ**: So effectively the earlier you do something the later it gets tested?.

**E1**: Oh, no I think acceptance is affected by everything. Interestingly enough, the way you can get most shock, certainly with conventional 3GL like approach, if these languages are used in the development of bespoke systems, it is probably in the coding phase, when you suddenly see the coding time is going up because the complexity of the system has been under-estimated. This is the problem with the structured process-oriented design notations, like the early form of Yourdon or DeMarco, where bubbles could be used to denote complex processes. Because you had defined the bubble with a name, you thought that you had done the job. The reality was that the semantic complexity of the bubble could be enormous and that was generally found much later.

That is why I quite like the Jackson approach, because it forces you to identify every single type of event of the system, and forces you to a much lower level of detail earlier on during the RA phase. Any method which does that means that you could have less of a shock later on. Unfortunate the corollary of such an approach, that of Jackson, is that the totality of the information available at the end of RA is pretty incomprehensible.
Most design methods (or notation methods) don't give an easy way of structuring the information. The hierarchical process oriented methods facilitate structuring, but they lead to the other problems that you don't know how much of the hierarchy is yet to be discovered.

SQ: You just mentioned that it is the acceptance phase where you feel the most effect of the requirement analysis phase. Aren't you just making a tautological remark?

E1: Yes. In terms of shocks, I think, it is in the coding phase, you suddenly realise the problem is more complex than you thought, and if you get the requirements wrong that is the stage in which you get most effect. But again it depends on the interface with the customer, the nature of the project, (how evolutionary the requirement) and how many interim deliveries you are expected to make. If there is not much contact with the customer, it is actually possible to have some crude implementations of the vaguely specified requirements and then only at the delivery of the system or even in trials, it can be found out that the system does not do what the customer wants. So there is not a clear answer. To some extent it depends upon the interaction with the customer. This is again an argument for an evolutionary approach in which there would be lots of stages, and lots of deliverables and the customer would have many more opportunities of commenting — i.e. earlier elicitation of the requirements.

E2: The software lifecycle influenced methodologies force the system builders to think in terms of the functionality of the system. But they generally hide the concerns related to performance, particularly in the more (fashionable) distributed systems. This means that the project manager has to try to decide how these performances are to be provided across several set of hardware connected together by some distributed system. And, indeed how well you can use certain particular hardware. You must have to imagine all these things together with how to provide that level of performance.

E1: This is not quite relevant, but valid, most methods concentrate on functionalities and ignore other attributes of a system.

SQ: Very briefly what do you mean by evolutionary development as compared to the traditional software lifecycle oriented methodologies?

E1: Evolutionary development is sideways out development not the top down or bottom up. You give the client a few of the facilities he has specified in terms of what the end-user of the proposed system sees and uses at the end of day. You just add to these facilities. Now, in fact, to be able to do this doesn't mean to say that if you supply one out of the 100 eventual facilities then you only have to do one hundredth of the work. Initially, you may have to develop the whole database management system on a little bit of data with a few application programs to sit on the top of it to provide one facility. This sort of approach (e.g. prototyping), but not prototyping to throw them away. Throw-away prototypes are developed to learn lessons from them, then are thrown away and move onto development of the next prototype.

But there are serious problems with throw-away prototyping. The classic example is a project which has well defined prototyping phases, such as an initial, intermediate and final prototype. But typically one team overruns during the initial prototyping phase and the intermediate phase starts. This means that the initial prototype team continues putting effort into work which no one looks at anyway. A very careful management is required with prototyping, because later teams will not pick up the work of earlier teams unless it is the same product. So the prototyping should be some stages in software evolution rather than throw-away modules.

E2: It is essential to fully understand the interdependency buried within the system. If you try to deliver a system in parts and if you have lots of dependencies between them then it is possible that a number of problems will surface during the acceptance stage.
think this approach, of lots of deliverables and lots of stages, is only applicable to a very restricted class of systems.

E1: Perhaps I can take an issue with that statement. I believe that the evolutionary approach works with highly interdependent systems. However, you introduce a small amount of change at a time. Perhaps, I am talking about adding only one facility at a time, whether or not they are delivered in much small stages.

E2: Yes.

E1: I am taking only a few interdependencies at a time, and it does mean a lot of management control on multiple releases all the time.

E2: Then you got management problems which are common in change control.

E1: Yes, the whole thing is change control, I do not believe in development. I believe only in change control. So that is the line to be adopted in software, I mean to treat the whole project as change control.

SQ: Please distinguish between change control and development.

E1: The reason why I distinguish between the two is that too often people think in terms of the simple lifecycle paradigm, where the development is considered as a sequence of steps, requirements, design, detail design and coding, and change only comes at the end of the cycle. However, in my view change control management starts right from the very beginning. This change is not change to a small segment of code but it is change to the built (construction) of the system. This change control management starts right after addition of the a new component or by replacement of one by another.

Q.2: Can you comment on dependability requirements of the user. Especially in the view of the fact that some dependability requirements may be unrealistic.

E1: Most users don't demand dependability requirements. But then, may be they shouldn't, after all, once the customer has specified his functional and performance requirements that's what he wants and what he doesn't want. What he doesn't want and if it happen some of the time, and in case the system works all the time and it does not perform according to specification, then a wrong system has been developed.

Many customers accept less than what they ought to be demanding. However, in certain fields the clients actually have documented reliability, and availability specifications and standards, but even there monitoring against specified requirements are not the norm.

SQ: What are the implications of stringent dependability requirement while you are carrying out requirements analysis? Does it change the work you do, or the way you do it?

E1: Philosophically it probably shouldn't, in that you might argue that we are morally committed to provide the user what he wants, whether or not he asked for it explicitly in a dependability statement or whether or not he will complain if we don't supply it.

In the case of stringent requirements from the user it is true that you would structure the requirements much more carefully and concentrate on the safety critical and mission critical requirements. This affects the design as well as RA and lead to modularisation of the design. It affects the design techniques as to what you would use as a whole, not only some X and Y techniques to establish that you had got it right.
Stringent dependability requirements may affect the design dramatically. Structurally, requiring fault tolerance and perhaps more automation, as well as effective use of a design method. Then they can also affect the cost dramatically. Again, if you do not give these dependability requirements due consideration sufficiently early then you will be shocked when they are eventually addressed. For example, a project in which I was involved did have a stringent requirement that in case the human operator dies in the mission the vehicle was to be brought back to its base, because the vehicle was very expensive.

**E2:** On the other hand onerous dependability requirements may lead to a much more general approach which overcomes the problems of dealing with lots of special cases. The general approach can lead to a slightly cleaner system.

**E1:** I agree to that we should distinguish between dependability and availability. In the example you just quoted, a bank operating 24 hours-a-day requires high availability, as compared to a number of options which are available to the bank working 8 hours-a-day. So there is no option to go back to a transaction for general recovery. This mean that one ends up with a different design then if you did cater for the case of reverting to a transaction mode.

**Q.3:** What are the key issues in a RA stage of software development, and what factors help you in going from the Original Requirement to Requirement specification?

**E1:** Feasibility evaluation is interesting and it is connected with my thoughts on prototyping, and throw-away prototyping. Far too often I have seen beautiful feasibility studies saying things are feasible which are too complex to be realised in a real system. Such as accuracy of a target tracking system. But you really would not do it as it was done in the feasibility study. The study is thrown away, without any benefit to a eventual system.

**SQ:** Please elaborate the connection between continual evolutionary change and feasibility evaluation.

**E1:** The connection is quite direct: a feasibility study changes the perception of the clients, I would go for less glossy feasibility studies and more prototyping leading to evolution.

**SQ:** Sorry, perhaps we are talking about testing phase of software development.

**E1:** Do you mean testing to reveal problems during requirements definition?

**SQ:** What about the changes in requirements which create changes in testing?

**E2:** I thought they (interviewers) are trying to get to the fact that in requirement phase you (E1) also have to plan how you are going to test the system. If you don't, and people sometimes don't, then they find that they are way down the path before they realise that they have to change things. Because they don't have a plan of how to put it together.

**E1:** Yes, I have a standard line for this, "you shouldn't have a requirement specification, you must have an acceptance specification." This is not feasible, because often we have systems more complex than we can test adequately, but in principle it is a good approach to try. I would like to see acceptance tests written first, before you go into development. The tests will change, again it is back to evolutionary change, you could say evolutionary development, which allows you to develop the acceptance test as you go along. Also this will bring testing phase much closer to specification phase for any particular set of circumstances.
E2: I think there is one thing you just mentioned which has not been high-lighted: the cost effectiveness in designing, there is a trade-off between extra cost, extra functions or extra dependability. This is to offer people a balance between what they choose, sometimes they regret what they have chosen but at least you try to put a price on it.

E1: Yes, I think the idea of clean analysis for the requirements, including my own idea of monitoring the acceptance specification first, is idealistic because the customer is actually relying upon you. You try to design a system to elaborate for him own requirements: what the customer actually wants.

E2: And equally the user does not know what does he actually want as a system, because the system is going to change his business. So if you ask him what he wants, he is so busy working with his present system, which may be a manual system, that he does not really know. The system designer or analyst then has to try to evaluate what effect the proposed system will have on the client's business.

SQ: There are at least two meanings of security, one is the security in the sense of unauthorised person in to the system and the other security in the sense of a physical crash.

E1: If we are talking about security requirements, my experience is that they contribute to evolutionary change only if you ignore them initially. It is not changes in requirements, it is a change of understanding and hence change of attitude towards requirements. Security requirements do grow sometimes because the user suddenly realises he has some security requirements that he did not appreciate previously, not only did he not mention them, but he did not really fully understand his own requirements.

E2: Some changes in requirements are not predictable, for instance hacking, which is not particularly considered seriously for a while. Now they have realised that quite a lot of systems which apparently didn't need to be secure in terms of data, now need to be secure. So you now get a change which was not required in the early stage of the project, because the requirement has changed.

E1: I think that performance and reliability, particularly performance (which is not usually specified) can be considered for banking system that might have to run 24 hours a day and you don't have the facility to go back to a transaction in general, to recover it, at least, not in the same way.

Coming back to availability, it can have dramatic change on the style of design you adopt. A year or two years ago, I worked on a system which had to have 99.99% availability. Given that, you are allowed to work on it continuously. However, if you are asked to run a system at any time with optional resources, then you have to have a very high availability and that have far-reaching effects on the design approach of the system. But I think that could be an extreme case.

As for real time requirements, consider the same project, we could not in fact use the computer system that we had assumed during feasibility study, simply because we were trying to achieve a cheap solution using standard Vax operating system. But the process switching time took up most of the response time available to us and it was also indeterminate, so things like that can affect the design quite dramatically during RA, I suppose these are key issues in RA which affect the design approach you use later on and of course affect the cost.

SQ: How do you see manpower management in requirement analysis phase?

E1: It comes out of it, I do not see it as part of RA. Certainly in term of establishing man power requirements, they are important. Whereas, managerial polices and quality
management, since I have been a quality manager, I don't think it is significant to RA

SQ: They are quite important!

E1: They are quite important.

SQ: Can we go back, manpower co-ordination, managerial polices are two facets of the same question, do you structure team at the stage of RA, that is the first question, if you do that, what are the possible polices to manage them?

E1: Certainly I think that there are lot of advantages in for instance having a separate integration team. But one thing you must not do, that is to give them any right to alter the code, changes must always go back to the development, otherwise the integration team will evolve the system away from the main design approach. The design team will be busy working with the system, and their new versions won't integrate. Is that the sort of issue you are trying to find out?

SQ: Is that the sort of issue you have when you do requirement analysis?

E1: Well no, that is more to do with project planning really, it is not the part of requirement analysis.

E2: I believe it is, because you need to set up a development team, or identify specific extra software, that would be needed in the later stages and make sure that the delivery of that is OK, or plan any specific deliveries of different hardware that are coming later, if you are working for different a machine. They will affect the way in which you set up a project team, and that comes out fairly early in software development.

E1: Yes it does, but I don't see it as part of RA though it will come out during the same phase. Perhaps these things are managed when we are creating the outline design or the outline project plan, which we shall be doing during this phase, but I don't see, how manpower co-ordination actually affects software requirements.

SQ: How does requirements affect the ultimate co-ordination in manpower?

E2: Well I think, the instances I mentioned will do, because you set up structure to allow for these items, like special tools, like making early delivery of part of the system, and so on.

E1: The whole manpower co-ordination stems from RA, but I don't see that it is tightly coupled, since the technical nature of the requirements is not usually going to have a dramatic affect on the manpower co-ordination. The manager will have his check list of things to bear in mind, but that is not determined by the particular requirements.

SQ: Can you say anything for software metrics?

E1: I think one can't properly answer that, certainly I cannot, because they have not been used, apart from metrics of progress, I don't see that as phase dependent, is that dependent on the RA phase or a later phase?

SQ: What are the metrics which are used in RA phase? Have you used any metrics in RA phase?

E1: Requirement Analysis Phase, well yes; Progress. I have seen complexity metrics used: we do that anyway as part of our estimation procedure. Recently we used function point counting complexity metric, supposedly it is a measured functionality, independent of cost. Unfortunately, in my view, we couple it with cost: for the calculation of cost, not its functionality. But I could say that, that progress metrics I
have seen. What do you say E2?

E2: I think people use more traditional metrics, such as break down program size, no of lines of code, at an early stages even before requirement analysis because one has to do this, at the call for tender. But when you are refining your ideas of software requirements, so I think it is more refinements than an introduction of metrics during RA.

E1: There will be some metrics like performance metric for instance!

E2: You size the project, that will mean breaking it down into the type of resources you want, in fact you are refining that during RA. In fact you are refining it all the way, throughout the project.

E1: I can think it is actually performance measurements. I suppose in performance analysis of a system involves notions of design.

E2: I think what you (E1) are saying is that there are far few instances of objective measurement or metrics than the people realise, still a very much an individual practice. People say I believe this looks like about twice as big as the old project I did, so it could be twice as big. It is very much a craft.

E1: It's true.

SQ: What sort of technical indicator do you used in terms of quality in the requirement analysis phase?

E1: Would you count performance loading!

SQ: Do you count performance?

E1: Do you count it as technical indicator?

SQ: No.

E1: I used it as a phrase, I still don't understand what you mean by this.

SQ: Well, I mean as you said, measure of progress, lines of code, which really just to get the feel for how the project is going?

E1: Well, you wouldn't measure lines of code during RA phase. You predict the lines of code, that you actually measure in terms of the effort to continue to elucidate the requirements. I think also, back to complexity measures, I have seen cohesion and coupling metrics used....

E2: I think there is a real problem here. It seems to me, that is the classical software lifecycle available in software literatures do not deals with the projects E1 have been doing. There are no sequential series of events, where it is no problem with performance or reliability. You just choose the machine and you know you got plenty of resources and you have a very simple team structure of people just taking you through from analysis to design.

So I think the problem we are getting here is that E1 is used to systems where requirements are often not understood completely, because of the very level of complexity and interdependency. So it is never easy, you got very much more feedback loops right the way through out the development. You try to control as you can, and if the system seems to go wrong like Nimrod or BATES, it seems it is only because that the level of confidence between the project team and the customer, and so
on, seems to fail.

I have seen a case in CAP, where the developing team have lost confidence with each
other, but some how they did come through at the end of the day. And I have also seen
other situations where they had lost the confidence at the wrong point, when all the
problems have been sorted out. So I think there is a problem here. I think the sort of
situation E1 is dealing with are not usually covered by classical lifecycle. Equally it is
misleading when you hear people talking about formal specifications they assume that
they can cover these complex situations by one technique, that does not work either.

E1: You are talking about the sort of projects which can follow a conventional
lifecycle, but I am not aware of any that have.

E2: Well some of your projects you had in Scientific, where in fact you worked on
two or three Fortran programs, your mathematician designed some analysis of a
problem and you write two or three Fortran programs so it was very small, single team
almost.

E1: Well, even then if you are talking about an OR project, doing mathematical
modelling, they change them all the time, perhaps they didn't understand what was
required, what the customer had in mind as a result of feedback.

E2: But it was a very much controlled change management project, E1, because you
delivered the system then you found it is not what the client wants, then you changed it.

E1: You should do all systems like that, that is my point, if it is possible and it is not
always possible.

E2: That is a good point.

E2: Sorry, I am interfering again it seems to me that this is the problem with this
approach, I don't think we are going to get around because if you want contribute you
are not supposed to go for very small system, because you are not going to prove any
thing.

E1: (Next sub question was on Rapid Prototyping: as a key issue, our expert continue
from the questionnaire) Rapid prototyping- yes, very important, very useful approach,
though I'm not that familiar with many examples of it being done. We seem to be
moving into an era in which we are able to put together some thing which looks fairly
representative to a customer. My line is that we should not then throw it away, but
deliver the prototype system or evolve the actual system from it (key issue: distributed
system specification).

SQ: People now use one sheet of data on one machine and next on other, they want
certain data in certain place, while the application is running in some other place: a fairly
complex system. We are asking you the question, is this the key issue in RA phase.

E1: In our world, (In Scientific), it is we who decide to what extent the system is to be
distributed, in order to provide right level of design, fault tolerance etc. But it is under
our control. It doesn't come from the customer. Where the customers asks us to
integrate with another system, then it is obviously a key factor but we don't have that
many like that.

E2: But it is a key factor at this stage. You have got to determine it at this stage. It is
dangerous if you ignore it.

E1: Certainly, if you get it wrong in the system design phase then the cost effects will
be significant.
E2: So it is a key issue you have to bear in mind.

E1: Is not a key issue in requirement analysis, as much as in design. Here we are talking about the customer requirements.

E2: So, when you specify the system then?

E1: Dependent upon the system again, and the approach you take. It depends upon what flexibility you put into your design. You might be able to throw away the existing distributed system, and put in totally different one. With the rate of change of technology that often is feasible, to throw away a distributed system and put in a monolithic system. That is feasible, but that's as much as I can say about it.

SQ: Reusability as a next category of phase independent issues.

E1: I am not aware of any great successes in reusing software in a system. I am not saying you cannot, but in general we don't. I have a hunch that this is because you do not make a lot of use of the parametric packages, like, 4GL, etc...

SQ: Sorry, maybe a programmer can reuse his codes.

E1: Rare, very rare.

SQ: Does this not depend very much on the field you are in?. My previous company had a library of routines and procedures, which were reused. And if you wanted to do something fairly low level like string handling, you would fine that there was a procedure to do that, and it does not happen to be exactly the same: you can take one there, change it, use it and then store it back in the library as well the original one.

E1: Were you using similar hardware each time?

SQ: It was exactly the same hardware. We were a product company, as well.

E1: So there is no problem.

E2: It depends what you mean by reusability. What you are saying that there is what everybody would expect to have a particular function report for your type of applications. I mean going back to the very least, trig function, and so on. But people build libraries in much more complex way for repeat business. The real problem is the environment which is changing all the time, new architectures you expected to use, new network techniques and so on, and so people who have invested in reusable components have often lost money because they find that it is very difficult to achieve reusability. It is easy to put in the library but very difficult get people to retrieve them.

SQ: I mean, you are not objecting to it.

E1: No, it is not something which comes up in the requirement analysis. You will try to bend the way in which you satisfy your requirements, so as to reuse code, but it does not come out as a key issue in RA in general.

E2: What about reuse within the project. There was one example within CAP, on a big commercial project, there was a determined attempt by the project manager to make sure that, for very large print programs for example, which will normally be written by each programmer individually, people were forced to design a super-set which they could reuse. He said it was a tremendous job to get it to happen. We did make savings, but the problems of trying to achieve it means he will never try again.
E1: I think that either modern technology or modern approaches to system design are making that unnecessary. We are going to a more virtual machine approach to design nowadays. So it never occurs to someone to write his own print program: You will have some output facility provided.

E2: Sorry but that is equivalent to what you have said about the library.

E1: That's right, but it is not something we have to think about constantly as a library. It is first the way you develop the system are virtual machines. (Traceability as a key issue)

E1: Traceability from requirements through to final system?

SQ: How much emphasis is required in RA phase, about the traceability, I mean the specification should meet the final coding.

E1: Yes, you should be able to trace back to requirements.

E2: You test the functions of requirements, rather than after you have built the system, testing particular programs.

SQ: It could be an issue?

E1: Yes, certainly tremendous amount of effort put in to provide traceability, I fear a lot of it nugatory in the end. In fact, that when we eventually put it in the field, the customer changed their mind anyway, we then changed it, you did function tests, system test, and throw away all the intermediate test. But I don't think it is a key issue in RA. But I think there is a problem here, in that we don't know properly how to show the way in which the actual system meets the requirements. We can say it does and point to system tests which demonstrate it, but the actual mapping from the requirements is a very hard thing to do. Again unless you go for something like Jackson, where you can actually see the connections.

E2: Traceability is perhaps more important to product development because you are very concerned with regression test, to make sure that the changes you introduced does not cause any side effect, things used to work, now don't; that is one form of traceability.

E1: That is true of bespoke systems as well.

E2: Yes, true.

E2: So, with incremental change it is important to be able to ... not just traceability, it is actually checkability, where you actually approve what works, or find out what works or doesn't work.

E1: I think it is very hard.

E2: I think, now people have started using structured methods like Jackson, you mentioned, do you feel that it's actually changing the relationship between analyst and the user, to be able to provide the better feedback from the user to allow them to understand, or to show that you understood their requirements. Does it help you? Do you feel any difference?

E1: I think it should do, but I think it doesn't, whether the notation I have seen used, the users have been overwhelmed. They have nodded their heads and they sign the documents, but at the end of the day, what they say that they did not know what was that? Even on a recent Jackson review, where I was actually supposed to be putting
myself in the customer's point of view and asking questions from their side, just in case they had missed any thing. At the end of day the customer said, thanks for so much help. They are not shy of asking questions but actually they don't know how to ask. When I asked them the questions, "have you understood what you should asked". But until then they had been nodding their heads around the table.

E2: It does not surprise me. I think it is important. We still have not got a good way for having a dialogue between the users and analyst. The users must be able to say what they want and able to understand the implication of what the analyst has understood from their requirements.

E1: It is no good showing the user all the pieces of paper and only to have their eye glazed over something.

Q4: What type of indicator/metrics do you use in RA for monitoring/forecasting the project.

E1: I have used SLIM some years ago, not very successfully and it suffered from the problem which most of these do -- they want an estimation of lines of code in the first place. That is the hard part. I have a hunch that we actually estimated the size of the project with very little rationale for the estimate of the number of lines of delivered source and it was mostly thinking. Oh it is a big project so it is bound to be a 50 million pound project, so that means delivery of that much of code, because that is the sort of code to deliver on this size of project, and what is the cost of that? "Oh—about 50 million pounds."

SQ: So what SLIM does for you?

E1: Not a lot, It allow you to "what if" and then rationalise the answer you first thought of say it is like the story of Zanzi bar: At North end of Island there was an old man who each day fires a noon day gun. One day a tourist asks him "how do you know when to fire the gun?" The man replied, "Every week, when I go shopping in the town at the other end of the Island, I set my watch by the clocks in the clockmaker's shop." One day when the tourist finds the clockmaker in the town, he then asks him: "How do you keep all yours clocks in time?" The clockmaker replied: "Every day, there is a chap at the other end of the Island who fires a noon day's gun. I have a feeling that number of estimation is like that.

E1: There is something which is quite interesting that has come out in the area of estimation, considering function point counting which we have been using for estimation in recent time. On one project, in particular, the lines of source code went up by a factor of 4 during a particular period of the project. That is not bad for a software project...

SQ: Sorry, you mean from an estimate to the actually delivery?

E1: From an estimate to delivery, the number went up by the factor of 4; which of course makes nonsense of all these estimate, which are based upon lines of source code. But on this particular project the estimate of lines of code went up by a factor of 4 from estimation to actual delivery, and the function point went up only by a factor of 1.2 and the requirement had not changed dramatically and I think that it was therefore a very good measure of complexity, a measure of function we were delivering. That is different from cost drivers. The cost did grow by a factor of more than 4, because of the growth in over all complexity and the over all job which has been done.

So I think, we must distinguish between measures of complexity of the functions which we have to deliver, from the size of the job. Now I prefer not to work in term of lines of code but, although it is difficult early on in the project, actually I like to see
design down to module level, pretty well. Because, in general, the amount of effort to produce a module is pretty much independent of the complexity of the code.

I suppose it has some sort of analogy to hardware. Where it is very easy to estimate hardware cost, because the cost of a board used to be 250 pounds, because it is the cost of components and workshop production cost. Actually the actual designed element was lost in the production cost. Maybe one approach to software is to try to structure system development in the same sort of way, with complexity-independent components.

By looking at the system in terms of modules, the cost associated with these become stabilising factors which average over all modules, and the creative design elements become less significant. So what you want to do is to try spread or try to apportion all your known cost over all the chunks of code, so you reduce the relative contribution of the individual elements. In the past I have done that—produced a very detailed design. If I were take over as a project manager, a project for which the estimate is based on the number of lines of code, I will feel very unhappy unless I thought that the estimate of lines of source code was based upon the fact that some body thought "it is twice as big as a previous project", which is something I can feel is much more tangible and I can actually believe. If he is actually looking at the requirements as specified and saying "that is going to be 70 thousand lines of code" then I would feel very uneasy. If he says "I built one before and I think that this one is twice as hard then that" is much better—more realistic.

E2: But also, where you can develop a system where you able to reduce the interdependency between modules then it becomes more like a set of self contain projects. It is easier then.

E1: One of my problems with incremental or evolutionary development is of course that you don't get a full understanding of the complexity at the very beginning.

Q5: How do you view the system definition or requirement analysis stage in terms of work to be done? Moreover, do you specify task or work break down structure?

E1: Yes Ok, I can now beginning to understand what was meant by system definition by looking at story board 5.2. This looks more like project planning rather than the task or work-break down, structure for the requirement analysis phase. Are we?

<table>
<thead>
<tr>
<th>Story Board 5.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>System definition Mangement.</td>
</tr>
<tr>
<td>* Produce Functional Specification.</td>
</tr>
<tr>
<td>* Produce Quality Management Plan.</td>
</tr>
<tr>
<td>* Produce Acceptance Test Specification.</td>
</tr>
<tr>
<td>* Perform System Modelling.</td>
</tr>
</tbody>
</table>

It looks too optimistic to me, where in reality continued requirement analysis does pervade all future work. I do not think I could come up with a generic plan. However, if I have to give an elementary lecture to someone who never heard about the subject before. I am sure, I can put something like, meeting with customer, initial requirements meetings, really discussions, then you have to work out really why he wants the system. Then you have options:options of different ways of meeting these ultimate requirements, so to say high level of solution options. Then after a selection phase, you go into requirements details of chosen solution, and then design option perhaps. Then I might move on to details of acceptance, specification, design reviews
and so on. But that's how I do it as an abstract idea. It does sound very plausible but I am not sure that we do it on any individual project, E2?

E2: If you are talking about ideal system definition or RA, it includes all of that, you have on this story board. In reality often they are not. We can count for just the last bit (See Story Board 5.2). The features, they have here, then you (E1) come in and you are expected to, in fact refine their set of large volume of features.

E1: I suppose, we are talking about feasibility.

E2: Yes, but it does effect the system definition doesn't it?

E1: Oh yes, very much so.

E2: So feasibility is just a part of RA.

E1: Certainly it affects your understanding of what is required.

E2: Sometimes it is missing altogether or not done very well.

E1: I have actually seen published papers or conference papers, arguing that, you only have to specify 'what' rather than 'how'. It is totally wrong, you must know why, you are working on it: We are talking about options availability.

SQ: So you like this idea of exploring the question of why?

E1: Yes definitely, it is very important, to keep in mind throughout the project.

E2: That is why the story board 5.2 is not really talking about work break down, because work break down will includes all those things in which academia are agreed. You (E1: the expert) and the user (client) like to explore the question of 'why' a software is needed. Is not then you start to say how do I go about it? Then you mark the key area.

E1: Certainly, you will do that at some stage. Here, assuming you are actually done these activities and then go to design, selection design and so on. You might be producing work breakdown structure, goes on to selection and outline design. You will be producing a quality management plan for the future at this stage. It is difficult to say which I find most important. All you are doing things which are necessary and extent to put effort into it which is absolutely essential, so every thing (in the story board 5.2) is absolutely essential. However, I find it difficult to prioritise the items which is in this story board 5.2 regarding task breakdown structure.

Obviously yes I do, although it may look like beautiful generic, philosophically ideal approach to software. You asked me the relevance of dependability: I said they are essential for doing the job. I am not quite sure that they are all equally important to dependability. For example, in true design option dependability is not a relevant issue. For dependability the first thing to do is to get the requirements right. Except for system, where you actually need fault tolerance of high availability hardware or some thing like that. So, the dependability is not much contributed in this phase. Again I have to say unfortunately, that quality management plan is not much contributed to dependability either.

Q.6: Following is a set of rules used in system definition (RA) stages. Please can you quantify them particularly with reference to the effect they have on system productivity, system complexity, dependability, or any other related aspect of software development lifecycle.
E1: I will condition my answer by saying that productivity, I suppose we are talking about producing a dependable system as oppose to code cut, which is I think marginally different.

SQ: Is it useful thing to do, I mean this kind of complicated table filling up, do you think it will lead us any where?

E1: Unfortunately, the answer will be in this situation is yes, and in other situations is something else. It will very difficult to provide a generic answer to the question, however, if walkthrough is use in various phases does it have any effect on productivity? Or dependability?

In fact it would depend on so many factors. Take a case history where we have modules of walkthrough in a project. But for some good reasons, the walkthrough had become part of the quality management system. To label a quality management on systems where you have a sets of forms with questions like 'timing', 'constraints' and so on. In some sense, this activity has stopped creativity of the development team. And when walkthrough have been shared by some quality assurance organisation then the development team washed their hands of it. So walkthrough like that was not really worth anybody's time. It costs you lots of effort but did not actually get you anywhere: you dredged off every time the quality assurance inspectors came from the ministry. It's was felt absolutely useless at time. Coming back to what I talked about keeping integration team and requirements team separate, I find this a better idea.

Q.7: How do you treat reusability in the RA phase?

(This question has been answered while discussing sub-question 3, as a phase independent issue, so it was decided not to talk about it)

Q.8: What level of abstraction is appropriate in the RA/system specification stage? Do you think that formal method will do here, abstraction in that sense also?

E1: There is no such thing as formal methods: there is formal notation, there is little you can do in formal notations. It is perhaps English or rather user-jargon is the right-level of abstraction to use. The way the user want to use the system. Most analysts will take the user jargon to be English, as the jargon is not formal enough : The user-jargon appears to lack a formal semantic definition.

It is essential that work in formal methods is directed towards formalising the jargon of some of the applications area. That is the level I would like to start at and I think there is no choice. Certainly it is entirely wrong to have just the design-methods type of abstraction: this is where we become dreadfully unstuck where the systems analyst/designer have understood or they had understood what they were presenting to the customer and the customer did not understand it. Although it looks so obvious, give your customer a half-day course in a method like DeMarco, and present him or her with an analysis; nine times out of ten nobody (customer) can understand the explanation. Therefore, the presentation or abstraction must be in the user's jargon and it is not a problem. (Here, we can note the importance of user terminology for requirements definition activities. Moreover, if we take note of the sentence, "the presentation or abstraction must be in the user's jargon," semantic schemas for requirements statements are useful for understanding).

If you can formalise the way in which the user thinks about his or her system. So much the better. This is already happening:SSADM methods are being re-thought of as Event-response chains. The way people think of real-time embedded system. Such an approach is much more useful (and tangible) then say, an approach based on notionally formal notations which is less comprehensible to a user.
SQ: This all about user comprehensibility, what about comprehensibility of the design.

E1: This is an argument for formal methods not for notational techniques. So the disadvantage of those formal methods which are making software requirements totally incomprehensible and the next chap (client) knows nothing about it. If you use something which looks incomprehensible there is a great scope for misunderstanding or alternative interpretation which you didn't quite appreciate: comprehensibility is a problem but apparent comprehensibility is the worst problem.

Certainly one can cope with these problems with formalising (and dealing) in user jargon terms. (In the software world we describe the deliverables for customer) in terms of software, e.g. in terms of screens, reports generators and so on. In some environment the customer does understand this software-specification approach. But often there will be a mismatch between description of a system in term of the concept understood by the design-ers and the descriptions of the requirements, which the customer can not understand. One approach which I have not mentioned so far, we called it facility metric, where each paragraph was taken as an elemental part of the requirement. There were system versions which were ticked off against the paragraphs. We identified facilities with the paragraph number. It was a way of getting across the linguistic barrier.

SQ: Would you say methods like MASCOT were really designed to do communication work because they use communication jargon like pools and channels, and methods like JACKSON were developed for doing data base work because they use the same jargon?

E1: Well initially I can say that Jackson was developed for data base or data oriented, (or actually file handling rather than data base, data base didn't exist when the Jackson started as a common jargon). MASCOT was not really for communication. It's around early 70's that we perceived a number of things about system design, such as message passing, treating that every thing differently for communications. We saw in a number of software systems if you could communicate by discrete time independent channels of information, which is actually a new concept, discrete time independent packages being messages, then it made thing lots easier. It mean that you could test and integrate more easily because things are totally asynchronous: because you did not spend half of your time in synchronisation, it made life a lot easier.

It means you could plug/unplug modules more easily and test them independently. Of course, in many systems we could not do that because we did not have data bases there, so we have pools. There is a nice example, given by Ken Jackson earlier on where the revision, he put out at his lecture (or presentation he gave at conferences) showed a very nice system, where all components communicated by channels. If you actually looked to the real thing, it was a simplification of a pool in the middle of every thing, and every thing talked to this pool, it is nice to have descriptions which look clean and healthy. Considering that example, I would not say it is developed for the communications world. I can say it was developed with some of those concepts at the back of their mind. But it was driven by the desire to take out time dependency from the design. I think this deals with communication work reasonably well. The problem these methods give you is of really hanging over to deal with system design where people prioritise processes, so MASCOT, and similar methods tend to prioritise the individual activity, task. These are always felt difficult in system design.

SQ. 8.2: How do you tackle interconnection complexity in the large software system? Remember the Ken Jackson example, every thing fell in the pool, can you see that in requirement phase, every thing fell in the pool?
E1: In those terms, yes, there is a chance of that. I am looking at the relevance to requirement analysis. Very often the user will describe his data requirements, and the requirements generally to you in a way which will imply a pool. (I am using the term pool in the general sense and not in the MASCOT sense) The team may not differentiate between different class of data and one of the activities of the requirements analysis may be to see if you can break things down specially to see whether they are detachable. It is not always possible, requirements analysis activity might be to see that if you can get more decoupling to what apparent from the user first description of the system.

Take the tracking system which command and control in terms of data-base of the target. I think it may not be possible to structure them in modern design techniques. I have not seen it done yet. I am not saying that it is not possible, it may be possible to automatically extract the necessary coupling which is required for the processes of the systems then you only get the minimal coupling. You may get the minimum coupling required for the system. But later when current understanding needs more details you may find it complex. It would be difficult to change the design to accommodate an extra coupling and it might be very painful.

The classic example is that of a data base management system, where the dictum is that you should "tune tomorrow and not today", because the chances are you will not even (steady) the system tomorrow. Data base management system is an area, where you do need to think fairly early how you going to handle them, because if you structure your data on the basis of first obvious access path you can think of and then later on you require more sophisticated access path than it may be painfully slow or it may cost you great deal to provide right amount of facility. I am saying that the design method may lead you to the minimal cut-time solutions, but it may not be very helpful because later on you may be required different minimal cut to implement it and it may be quite hard.

Q. 9: How do you cope with logical inconsistency of coding in the system definition phase? In fact, I am talking about a programmer misunderstanding of what you wanted him to do. A single mistake which does not appear during implementation, than later after integration of the system it reveals.

E1: What you do about it that possibility in advance, you mean trying to pre-empt error or to build some thing to detect them or to anticipate them.

SQ: Is there any method that can be used to eliminate this possibility.

E1: That is something in which you apply what you got something down stream to do the proof of equivalence or the partial equivalence between eventual code and original design, process of verification!

SQ: It is not actually being in practice, that sort of proving but you do apply some sort of technique for that situations, what is it you apply?

E1: We had one project for software specified in "Z". It was intended to check its own consistency. The intention was to introduce some kind of proof on the code. Although, we did not quite get that far for checking its own consistency. But again back to earlier comments it was quite incomprehensible and no body realised that the incomprehensibility was the actual problem. I am not convinced when people suggest to add comments to formal specifications in order to show the relationship between user jargon and to explain the mapping, particularly, in specifications dealing mathematical notation. What you got in return is lots of the semantic content of the intention are left unspecified. What was the original intent left in the mapping rather than in the specification itself. Lots of mapping on the other hand is left totally informal. Then you ask to accept it as acceptable mapping. I am not convinced that could be done. I will prefer to go for a approach which reveal the user jargon with some formality by
using logic and temporal logic. After everybody in that field become familiar with that and realised the abstraction and the concepts, then you can specify it in that language and you go automatically to the next stage of the mathematical notations. You can not have transformation rules which cause error of interpretations, which you might have in any transformations. How do you cope with it in general? I have seen an example of using Z, and it run into the classic problems: the specification was proven self-consistent but yet not consistent with their code. It turned out to be wrong system. However, there are some counter examples where I have seen proves, for nice and easy problems, where the concepts are generally deterministic and it is not too hard then for the designer to think in terms of user jargon. There you can not find a mismatch between user jargon and the language of specification. This perhaps set the issue aside of proving specification to code level. I have not seen any example of formally proving code against specifications but I am not saying that there are not any examples, I mean it is not in my experience.

E2: What about Viper as an example?

E1: Viper is interesting: It is a microprocessor to built by RSRE(Malvern). It was initially specified in ADA, then in LCF/LSM and other languages. One of the implementation was to be in UK5000.

E2: Its main aim was provability.

E1: Yes, but I am not convinced of that. Although, I have not seen the final analysis. It was an example where functions at the highest level were well understood concepts. However, the potential for mismatch between specification and implementation was there, regarding what could be built and what is specified. Therefore, as long as you could define it in pretty well understood mathematical notation which is known by every one. Perhaps in that case it is possible, but if I am asked to build a snap down missile target tracking system, there is not an universal understanding of what all that mean, you can not than go for provability, or for automation that earlier.

SQ: In fact I think this is a deep question in the sense that, can you do any thing to stop people coding inconsistently which is not there in specification within the process of software development cycle?

E1: If you mean automation and self consistent coding? That is what we are doing, precisely giving much emphasis on high level languages.

E2: That is what they are arguing. You (academia) could do it if you (E1: expert) could actually identify the problems. I think it is useful to automate the process. However, there are various issues attached to it relating cluster of things that are too complex and that can make the job difficult.

E1: It is possible that you may not wish to automate the process. Looking at these limitation, it is possible that you end up with a particular implementation. It is also possible that you may have lot of choices, each of which could be logically proved to be equivalent to original requirements. But you may not wish to implement them. But I think that is an argument against automation in general. However, if we consider hardware requirements, checking its requirements through provability is not worthwhile. I think it's highly irregular concept. But in general the questions of issue of design of code and provability, I think is interesting one. Although as I have said earlier, you can achieve results from structured programming even using assembly language or any thing similar Structured Fortran being used but it was not actually structured Fortran code gives structures to a software but what is needed is a very carefully control to achieve this.

Now I am not saying that you can not do that. The advantage of automation, assuming
the automation is correct, is actually executing the specification. We have a code generation tool, which prompt us to put in the algorithmic components of coding, code modules, similar to ADA like format and then we have automatic ADA code generator. If all these tools work properly with their hardware then this question of provability doesn't arrive, because it bound to be self consistent, you are actually executing the specification.

SQ: The need was there, I mean the need to check the consistency.

E1: Yes you have to adopt techniques, design methods and approaches which minimise that problem, what we could do is to automate the programming, I guess it is more reliable.
Appendix C: Analysts survey
Consultants Addresses with proprietors names:

1. Mr. J. W. Chapman, Admiral Computing Group, Camberley Surrey GU15 3JR.
3. Dr. B. O. R. Williams, Arthur D Little Ltd., Berkeley Square House Berkeley Square, London W1B.
4. Mr. S. Smith, BIS Software Ltd., York House 199 Westminster Bridge Road, London SE1 7UT.
5. Mr. Nick Parkin, Bluebird Software Ltd., Sutherland House 34 Argyll Street London W1V JAD.
6. Mr. David Booth, Boeing Computer Services (Europe) Ltd., PO Box 747 364 Euston Road London NW1 3BQ.
7. Mr. Mike Buckmaster, British Medical Data Systems Ltd., Paddington House Town Centre, Basingstoke Hants RG1 1GJ.
8. Mr. Brian Reynolds, Burroughs Machines Ltd., Heathrow House Bath Road, Cranford Hounslow Middlesex TW5 9QL.
9. Mr. Roger Evans, Computer People Group, 58-69 St Martin's Lane, London W1X 4DS.
10. Mr. K. Nicol, Computac Group Co Ltd., Heathcote House 20 Savile Row London W1E 1AE.
11. Mr. P. M. G. Collinson, Consultants (Computer and Financial), Eldon House 2-3 Eldon Street, London EC2M 7LS.
12. Mr. P. M. Burnham, Coopers & Lybrand Associates Ltd., Plum Tree Court London EC4A 4HT.
13. Mr. D. D. Clare, Data Logic Inc., Heathcote House 20 Savile Row London W1E 1AE.
14. Mr. Chris Wood, Datacom Ltd., 99 Stains Road, West Surbiton-on-Thames Middx TW16 7AN.
15. Mr. D. J. Harris, Easana Ltd., Lyon Way Finley, Camberley, Surrey GU16 6DX.
16. Mr. David England, D M England & Partners Ltd., Lytham Court Lytham Road, Woodley Reading RG5 3PG.
17. Mr. B. E. Avis, Ferranti Computer System Ltd. (Cambridge Department), Tylor Way Cambridge Great H N44 7XX.
18. Mrs. N. J. Coulter, GEC Avionics, Airborne Soft., Division Elistree Way Borehamwood Herts WD6 1RX.
19. Mr. G. Johnson, Grand Information Service Ltd., Wyvern Way Rockingham Road Uckfield Middlesex U88 2UD.
20. Mr. R. Gregory, Honeywell & Partners Ltd., Kennedy House Stratford Street, Altrincham Cheshire WA14 1ET.
21. Mr. A. Infopoint, ICL UK Ltd Bridge House (North), Putney Bridge Fulham London SW6 3XZ.
22. Mr. Repoter, Johnson Technology P-K Computer Service, Park House Wick Rd, Egham, Surrey TW20 0DN.
23. Mr. A. R. Sparrow, Infosystems Consultants Ltd., Plesant House 29 Mount Pleasant London WC1X OAR.
24. Mr. J. B. Hartland-Swann, Ingersoll Engineers, Bourton Hall Bourton on Dunmore Rugby, Warwickshire CV3 9DQ.
25. Mr. Peter Morris, Intal Ltd., PO Box 5 Grosvenor House Prospect Hill, Reddish Worsley M50 4QY.
26. Mr. Fred Black, James Martin Associates, Spa House 11-15 Worple Road Wimbledon, London SW19 7UT.
27. Mr. Ken Rasmussen, LOGIC (UK) Ltd., 64, Newman Street, London W1E 4SE.
28. Mr. D. A. Sharmer, P E Consulting Services Ltd., Park House Wick Road Egham Surrey TW20 0DN.
29. Mr. P. E. Thomas, PA Consulting Group Amplitude House, Gates Way, Stevenage Herts SG13HL.
30. Mr. Alan Austin, Pafez Ltd., Strelley Hall Strelley Nottingham NG8 6PE.
31. Mr. G. Castro, Peat Mackenzie & Co., 1 Puddledock Blackfriars London EC4V 3PD.
32. Mr. David Bevan, Praxis Systems plc., 20 Manvers Street Bath BA1 1PX.
33. Mr. P. Newton, Prolog System Ltd., Data House & Alexander Road Wimbledon, London SW19 7TB.
34. Mr. E. H. Marzilione, Rand Information Systems Ltd., 29 Thames Street Kingston upon Thames, Surrey KT1 1PS.
35. Mr. R. H. Marson Smith, Scoics Ltd., Wavendon Tower Wavendon Milton Keynes MK17 8LX.
36. Mr. J. Green, Sherwood Computer Ltd., Renolds House Whitfield Street Gloucester GL1 1PG.
37. Mr. P. Bennett, Small Systems Software Support Ltd., 9 Beechcroft Road Cippenham Kent BR6 1AN.
38. Mr. Brian Hill, Software Ireland Ltd., 26 Limerick Street Belfast BT2 8PJ.
39. Mr. S. D. Ingham, Software Sciences Limited, Farnborough Hampshire England GU14 7KS.
40. Mr. Elaine Pendreich, Thorn EMI Computer Software, Boundary House Boston Road London W7 2QG.
41. Mr. M. Braithwaite, Touche Ross Management Consultants, HILL HOUSE 1, Little New Street London EC4A 3TR.
42. Mr. Mark Clifford, Turnkey Software Ltd., 12 High Street Chalfont St Giles Bucks HP8 4QA.
43. Mr. J. Warburton, WM Data Ltd., Data House 72 Station Road, Southwater West Sussex RH13 7DT.
We first discuss the preamble used to introduce ourselves to the analysts in Section C.1. The glossary of terms, which was used to present our understanding of the terminology used in the questionnaire, is presented in the same section. Section C.2 includes the questions regarding the analysts' background, qualification and experience. Analysts replies are given in Section C.3.

C.1 Preamble to the Questionnaire
In the context of this survey, we introduced the task, as a research activity related to a PhD project, which aims to study the use of the knowledge involved in the requirements definition stage of software development. The following indented paragraph gives the preamble used with every questionnaire for this purpose.

This study is part of a PhD project and we are investigating the knowledge mainly involved in requirements definition stage of software development. The questionnaire prepared, therefore, does not intentionally cover other aspects such as project management and planning of system development. We have concentrated on the problems associated with feasibility study and/or requirements definition of software system. The information you will provide, will help us to rationalize the real world knowledge against what is available in the text book of software engineering. Whereas, the analysis of this survey will help us to give the judgment on the popularity and practices for various tools, methods and techniques.

The Glossary used in the Questionnaire
The following glossary of terms was enclosed with the questionnaire sent to the analyst, with the intention of removing ambiguities in the use of the terminology and to have a common base-line for communication.

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Textual analysis</td>
<td>By analysing user initial requirements and domain documents only.</td>
</tr>
<tr>
<td>Notational techniques</td>
<td>Using some kind of notational representation techniques for analysing the domain.</td>
</tr>
<tr>
<td>Mixture</td>
<td>Mixture of the above two.</td>
</tr>
<tr>
<td>Requirement definition</td>
<td>Feasibility study, System requirements report project objectives (for which an approval is required).</td>
</tr>
<tr>
<td>Test criteria</td>
<td>System assurance, (which provides an objective evaluation of project reviews on a routine basis. These objective and impartial reviews determine if the project is proceeding according to plan and whether the development guidelines are being followed.</td>
</tr>
<tr>
<td>Customer</td>
<td>Client (for which the is system required)</td>
</tr>
<tr>
<td>Analyst</td>
<td>A system definer for designer and implementors and a main contact between software house and the client.</td>
</tr>
<tr>
<td>A model</td>
<td>An analyst terminological and diagrammatic findings with relations to the application domain concepts and agreed among client, analyst and implementor.</td>
</tr>
<tr>
<td>Predictable Modules</td>
<td>Modules essential for the system operation but not mentioned in user initial requirements, such as, number of files must be open during system operation, or a specific file required to maintain information/data during operation; solutions.</td>
</tr>
<tr>
<td>Essential Requirements</td>
<td>Functional requirements of the system only.</td>
</tr>
<tr>
<td>Formalisation</td>
<td>Using &quot;Z&quot; or &quot;VDM&quot; approach for requirements analysis.</td>
</tr>
<tr>
<td>CASE</td>
<td>Computer Aided Software Engineering.</td>
</tr>
</tbody>
</table>

C.2 The Questionnaire: An Outline
The questionnaire was designed to elicit information about four major aspects of requirements definition. These aspects were either identified by our review of the existing literature (Chapter 2) or stressed by our experts, whom we interviewed at length (Section 3.1).
These four aspects were:
- What tools and techniques were used or deemed potentially useful, by the working analyst? Question 1 and 2 of the questionnaire dealt with this aspect.

Q.1 How do you build your conceptual model for the required system during feasibility study/requirements definitions? (please tick)

(a) Textual analysis
(b) Notational techniques
(c) Mixture

Q.2 Do you get any help from the following method/techniques during feasibility study or does it help you in requirements definition stage of Software Development? (please tick)

<table>
<thead>
<tr>
<th>Method/Technique</th>
<th>Quite often</th>
<th>Sometimes</th>
<th>Depends</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) JSD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(b) SADT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(c) CORE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(d) SSADM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(e) Data flow</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(f) Any other(Please specify)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 28

- What were the opinions of the analyst on changes in user requirements and on the elicitation of non-functional requirements like performance, reliability and security of a yet-to-be developed software system? (the relevant questions [3-5] in Figure 29).

Q.3 Following are the key issues generally recognised in the literature for software development, do you find them relevant during requirements definitions or feasibility stage?

<table>
<thead>
<tr>
<th>Issue</th>
<th>Never</th>
<th>Sometimes</th>
<th>Quite often</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) Continual/evolutionary changes in requirements</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(b) Inability to understand user terminology</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(c) User sophistication</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(d) Misconception of resource requirements</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(e) Misconception of user requirements</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(f) Non-functional requirements, such as:</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Appendix C

1. Performance requirements of the system

2. Reliability requirements of the system

3. Security of the system

4. Test criteria for the system

Q.4. At what level do you specify the required system during feasibility stage?

(a) Essential requirements only ........................................... □
(b) Detail requirements with predictable modules of the system.......... □
(c) Detail requirements but without predictable modules of the system... □

Q.5 Do you think, a domain terminology formalisation can help you to analyse the client requirements?

(a) Yes.................. □
(b) No.................. □
(c) Not sure.......... □

Figure 29

- What methods are used to acquire knowledge of the clients' (or user's) domain? (the relevant question 6 and 7 in Figure 30)

Q.6 What methods do you use for requirements extraction?
(please tick as many as you like)

(a) Interview the end user.......................... □
(b) Passive observation.......................... □
(c) Analysis of organisation documents and its structure .......... □
(d) Interview with technical expert.......................... □
(e) Others..........................................
(Please specify)

Q.7 If the application domain is not already structured, is it necessary to structure the application domain first before feasibility study?

(a) Yes,........................................□
what method would you prefer to use for it?
Could you briefly tell us about it:...................................................

(b) No........................................□

Figure 30

- How do the analysts break-down the feasibility studies and how do they plan the
preparation and relevant documents during requirement definition stage. (Question 8-13 in Figure 31).

Q.8 Do you divide your feasibility study into two parts as initial (for which a partial approval is required) and final (for which a firmed approval is required)?

(a) Yes......................... □
(b) No......................... □

Q.9 When do you prepare following documents?

<table>
<thead>
<tr>
<th>Initial studies</th>
<th>Final Report</th>
<th>Not applicable</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) Project plan</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>(b) Development guidelines</td>
<td>□  □  □</td>
<td></td>
</tr>
<tr>
<td>(c) Resource requirements</td>
<td>□  □  □</td>
<td></td>
</tr>
</tbody>
</table>

Q.10 Do you prefer to prepare Test criteria during feasibility stage?

(a) Yes........ □
(b) No.......... □
(c) Depends
   (please specify) .................................................................
   ..........................................................................

Q.11 Do you use any report generators or CASE tools set during feasibility study report?

(a) Yes,........ □
   (could you name them for us, please).
   ..........................................................................
   .................................................................
   ..........................................................................
   ..........................................................................
(b) No........ □

Q.12 Do you use any documentation standards during the preparation of feasibility reports?

(a) Yes........ □
   (please specify)
   .................................................................
   ..........................................................................
   .................................................................
(b) No........ □

Q.13 Could you provide us yours views on the following (what one must include or not required during the system' feasibility studies).

(a) Technical feasibility:..........................................................
(b) Operational feasibility:.....................................................
(c) Economic feasibility:......................................................
(d) Do you use any method for assessing the cost of solutions?
   (please specify)
   ......................................................................................

Figure 31

There was a separate section where the analyst was supposed to indicate his/her
organisational experience as Manager, Project Leader/Analyst or Programmer. The questionnaire was enclosed with a brief note requesting the addressee to enclose some information in the form of a 'table of contents' of a requirements specification document for any successful projects he or she has done in the past.

The following section is used to discuss the answers received from the respondents, here, the subsections are arranged according to type of each question.
Appendix D: PROLOG Source Code.

:- reconsult('functwords.pl').
:- reconsult('processwords.pl').
:- reconsult('dictionary.pl').
:- reconsult('canonical.pl').
:- reconsult('lcp.pl').
:- reconsult('cg.pl').

:- op(400, xfy, :).

re2:- reconsult('analyst1.pl').

cls:- unix(shell(clear)).

go(File):-
    prepare(File, No_Do_Vocab, H_Words),
    repeat,
    present_lcp_menu(File, No_Do_Vocab, H_Words, Choice, Choice),
    see_finish.

quantifiers:-
    from_start,
    perform_choice(quantifier_indicator), !.

rule_indicators:-
    from_start,
    perform_choice(rule_indicator), !.

relation:-
    from_start,
    perform_choice(relation_indicator), !.

explanation:-
    from_start,
    perform_choice(explanation_indicator), !.

process:-
    nfile(Stream),
    process_id(LineNo, ConWord, Word, Posi),
    stream_position(Stream, _, Posi),
    write("*
    "),
    write('Line No.: '), write(LineNo),
    write('), write(ConWord),
    write('), Content word: "'), write(ConWord),
    write('), Process word: '), write(Word),
    write(' *'), nl,
    senasitis,
    stream_position(Stream, Posi),
    readsent(WordsList),
    graph(WordsList),
    write('more ?'),
    fget_line(user, [Ansl], 10), nl,
    ( Ans \= 110 -> finish_no
    ; from_start, finish_yes
    ),
Appendix D

see_finish, finish_no, !.

phrase :-
  from_start,
  write('Enter a phrase: ')
  getsent2(user, L2),
  assert(lcp(phrase, L2)),
  perform_choice(phrase),
  retract(lcp(phrase, _)),
  !.

domain:- con(Word, Frequency),
  write('Content word: '),
  write(Word),
  write(', Frequency: '),
  write(Frequency), nl,
  write('more?'),
  fget_line(user, [Ans[, 10], nl,
  ( Ans \= 110 -> finish_no
   ; finish_yes
   ),
  see_finish, finish_no, !.

quit: close_all_streams, finish_yes.

prepare(File, No_Do_Vocab, H_Words):-
  nl, write('* Please Wait *'), nl,
  abolish(con/2),
  abolish(chw/2),
  open(File, read, Stream),
  abolish(nfile/1),
  asserta(nfile(Stream)),
  stream_position(Stream, Position),
  abolish(top_posi/1),
  asserta(top_posi(Position)),
  finish_no,
  abolish(numb/1),
  assert(numb(0)),
  concord,
  dom_info(No_Do_Vocab, H_Words),
  from_start,
  abolish(process_id/4),
  correlation,
  from_start,
  finish_no.

concord:-
  repeat,
  action1,
  see_finish,
  finish_no.

action1: -
  readsent(Words_List), !,
  ( Words_List \== end-of-file \rightarrow getcontent(Words_List) ;
finish_yes
).

correlation:-
    repeat,
    action2,
    see_finish.

action2:- !,
    readsent(L),
    (L \= end-of-file -> select_sentence(L)
     ; finish_yes)
).

select_sentence(L):-
    chw(Word_),
    member(Word,L), !, % Is high frequency content word present
    getproc(L).

getproc([]).
getproc((First,Last)) :-
    atom_length(First,Length),
    prv(Length,LProcess), !,
    (member(First, LProcess) -> getposition(First)
     ; getproc(Last))
).

getposition(Word):-
    retract(newposi_1(NewPosi)),
    numb(LineNo),
    chw(ConWord_),
    assert(process_id(LineNo, ConWord, Word, NewPosi)).

cHECKPROC(Word,Length):-
    prv(Length,List), !,
    member(Word,List).

getcontent([]).
getcontent((First,Last)):-
    number(First), !,
    getcontent(Last).

getcontent((First,Last)) :-
    atom_length(First,LenFirst),
    checkcontent(First,LenFirst),
    getcontent(Last).

checkcontent(Word,Length):-
    fw(Length,List),!,
    (+member(Word,List) -> ch_count(Word)
     ; true).

% Checking words frequency and selection
ch_count(Word):-
  retract(con(Word,N)) -> incres(N,M), assert(con(Word,M))
; assert(con(Word,1)).

incres(N,M):-
  M is N + 1.

dom_info(No_Do_Vocab, H_Words):-
  findall(X,con(_,X),List),
  length(List,No_Do_Vocab),
  mysort(List, List2),
  mmber(1,F1,List2),
  retract(con(Word1,F1)),
  assert(chw(Word1,F1)),
  mmber(2,F2,List2),
  retract(con(Word2,F2)),
  mmber(3,F3,List2),
  retract(con(Word3,F3)),

mysort([],[]).
mysort([H|T],S):-
  split(H,T,U1,U2),
  mysort(U1,V1), mysort(U2,V2),
  append(V1,[H|V2],S).

split(X, [Y|T], [Y|U1], U2):- X =< Y,!, split(X,T,U1,U2).
split(X, [Y|T], U1, [Y|U2]):- split(X,T,U1,U2).
split(_,[],[]).

perform_choice(Choice):-
  lcp(Choice, ValueChoice),
  all_choices(Choice, ValueChoice),
  see_finish, finish_no.

perform_choice(_).

all_choices(Choice, ValueChoice):-
  readsent(Sentence),
  check_sent(Choice, ValueChoice, Sentence).

check_sent(Choice, ValueChoice, Sentence):-
  (Sentence == end-of-file ->
   do_action(Choice, ValueChoice, Sentence)
  ;
   from_start).

do_action(Choice, ValueChoice, Sentence):-
  list_matching(ValueChoice, Sentence),
  perform_read(Choice, ValueChoice).

do_action(Choice, ValueChoice, _):-
  finish(yes)
; all_choices(Choice, ValueChoice).

perform_read(Choice, ValueChoice):-
  nfile(Stream),

Appendix D
Appendix D

define newPosi_l(Position),
stream_position(Stream,_,Position),
numb(LineNumber),
write("Line Number: ", write(LineNumber),
write( "+ "), write(Choice), write( "+"),
write(ValueChoice), write( "+"), nl,
seenasitis,
write(" more ? " ),
fGet_line(user, [Ans_], 10), nl,
( Ans \(== 110 \rightarrow \) finish_no
 ;
from_start, finish_yes
 ).

list_matching([], _).
list_matching([X,Y,Z], L):-
member(N,X,L),
M is N + 1,
member(M,Y,L),
list_matching(Z,L).

list_matching([X],L):-
member(X,L), !. % the cut will avoid multiple instances of X in L

member(1,X,[X|_]),
member(N,X,[_|Y]),
\ N \(<\ M +1.

finish_yes:- abolish(finish/1), assert(finish(yes)).
finish_no:- abolish(finish/1), assert(finish(no)).

from_start:-
nfile(Stream),top_posi(X), stream_position(Stream,_,X),
abolish(numb/1), assert(numb(0)).

see_finish :- finish(yes) \(\rightarrow \) true; fail.

seenasitis:-
nfile(Stream),
get0(Stream, Char),
\(\rightarrow \) test_next(Stream, Char) \(\rightarrow \) (ttyput(Char), seenasitis)
 ; fullline(46).

fullline(Char):- ttyput(Char), ttyln, ttyln.

test_next(Stream, Char):-
member(Char, [46,63,33]),
stream_position(Stream, P),
( get0(Stream, C2),
\(\rightarrow \) member(C2, [10,32,13]) \(\rightarrow \) stream_position(Stream,_,P), fail ; true).

readsent(Sentence):-
nfile(Stream),
capital(Stream, Sentence),
( Sentence \(\equiv \) end-of-file \(\rightarrow \)
(abolish(newposi_1/1),

stream_position(Stream, Posi_1),
asserta(newposi_1(Posi_1)),
getsentence(Sentence))
;
true  % the file is finished and Sentence == end-of-file
).

readsent(Sentence):-
  Sentence == end-of-file.

capital(Stream, Sentence):-
  peek_char(Stream, C),
  ( +member(C,
    [65,66,67,68,69,70,71,72,73,74,  
     75,76,77,78,79,80,81,82,83,84,85,86,87,89,90]) ->
    increase_read_pointer(Stream, Sentence); incr_a_number).

increase_read_pointer(Stream, Sentence):-
  getO(Stream, Char),
  Char == -1 -> capital(Stream, Sentence); Sentence = end-of-file.

incr_a_number:-
  retract(numb(X)),
  Y is X +1,
  assert(numb(Y)).

atom_length(X,L):- !, % new
  name(X, S),
  length(S, L).

% Reading a sentence in a lower case from an open stream.

getsentence(Wordlist) :-
  nfile(Stream),
  getO(Stream, Char),
  Char == -1 ->
    to_lower(Char,LowerC),
    getrest(LowerC, Wordlist);
    getrest(Char, Wordlist).

getrest(46, []) :- !.                     % 
getrest(63, []) :- !.                     % !
getrest(-1, []) :- !. [end_of_file marker]

getrest(Char, Wordlist) :-
  member(Char,[10,13,32,34,38,39,40,41,44,45,58,59,95]), !,
  getsentence(Wordlist).

getrest(Letter, [Word|Wordlist]) :-
  getletters(Letter, Letters, Nextchar),
  name(Word, Letters),
  getrest(Nextchar, Wordlist).

getletters(10, [], 10) :- !.                 % carriage return
getletters(13, [], 13) :- !.                 % line feed
Appendix D

getletters(32, [], 32) :- !.
getletters(33, [], 33) :- !.
getletters(34, [], 34) :- !.
getletters(38, [], 38) :- !.
getletters(39, [], 39) :- !.
getletters(40, [], 40) :- !.
getletters(41, [], 41) :- !.
getletters(44, [], 44) :- !.
getletters(45, [], 45) :- !.
getletters(46, [], 46) :- !.
getletters(58, [], 58) :- !.
getletters(59, [], 59) :- !.
getletters(63, [], 63) :- !.
getletters(95, [], 95) :- !.
getletters(-1, [], -1) :- !.

getletters(Let,[LetLetters],Nextchar):-
  nfile(Stream),
  get0(Stream, Char),
  Char \= -1 ->
    to_lower(Char,LowerC),
    getletters(LowerC, Letters, Nextchar);
  getletters(Char, Letters, Nextchar).

% Interaction with a user who end his entry with a linefeed

getsent2(Stream, Wordlist) :-
  get0(Stream, Char),
  to_lower(Char,LowerC),
  getrest2(Stream,LowerC, Wordlist).

getrest2(S, 10, []) :- !. % 'linefeed

getrest2(S, Char, Wordlist) :-
  member(Char,[32,34,38,39,40,41,44,45,58,59,95]), !,
  getsent2(S, Wordlist).

getrest2(S, Letter, [Word|Wordlist]) :-
  getlet2(S, Letter, Letters, Nextchar),
  name(Word, Letters),
  getrest2(S, Nextchar, Wordlist).

getlet2(S, 10, [], 10) :- !. % carriage return
getlet2(S, 13, [], 13) :- !. % line feed
getlet2(S, 32, [], 32) :- !. % Space
getlet2(S, 33, [], 33) :- !. % !
getlet2(S, 34, [], 34) :- !. % "
getlet2(S, 38, [], 38) :- !. % &
getlet2(S, 39, [], 39) :- !. % '
getlet2(S, 40, [], 40) :- !. % ( 
getlet2(S, 41, [], 41) :- !. % )
getlet2(S, 44, [], 44) :- !. % ,
getlet2(S, 45, [], 45) :- !. % -
getlet2(S, 46, [], 46) :- !. % .
getlet2(S, 58, [], 58) :- !. % :
getlet2(S, 59, [], 59) :- !.
getlet2(S, 63, [], 63) :- !.
getlet2(S, 95, [], 95) :- !.

% Main Menu

present_lcp_menu(File, No_Do_Vocab, H_Words, Choice):-

write('Document Name:'), write(File),
write('Domain Vocabulary (content words): '),
write(No_Do_Vocab),write(' words'),
write('Words occurred frequently: '), write(H_Words),
write('1. Quantifiers  2. Relations and Hierarchies'),
write('3. Rule Indicators  4. Explanation '),
write('5. Sentences with Process Information & Conceptual graph'),
write('Enter your choice: '),
fget_line(user,[Char|_], 10),
unify_choice(Char,Choice).

unify_choice(49,quantifiers).
unify_choice(50,relation).
unify_choice(51,rule_indicators).
unify_choice(52,explanation).
unify_choice(53,process).
unify_choice(54,phrase).
unify_choice(55,domain).
unify_choice(56,quit).

% file: cg2.pl

:- reconsult('dictionary.pl').
re4:- reconsult('cg2.pl').
noc:- nodebug.
start:-
prompt(_,'>.. '),
repeat,
write('>
read_in(L),
delete_last(L, Words),
(L=[q..] -> true ;
graph(Words) -> fail).

graph(Words_List):-
abolish(senten/1),
parsed_data_abolish,
write("* SYNTACTIC ANALYSIS *"),
(\+synt_info(Words_List) -> nl, write("* difficult *");
assert(senten(Words_List)),
write("* CONCEPTUAL GRAPH *"), nl,
chw(ConAgent_1),
(\+agnt(_,ConAgent) -> nl, write("* difficult *");
concept_graph(ConAgent)
)
), !.

delete_last(L,M):-
append(M, L).

synt_info(Words_List):-
traverse(s,Words_List,[],Parse), nl,
write(Parse), nl, nl,
append([s],NewParse, Parse),
append(NP,VP,NewPase),
NP=[],
seperat_mvb(MainVerb,Object, VP),
seperat_concept(Agent, NP),
retract(s_c(X)),
(X == passive ->
(assert(agnt(MainVerb, Object)),
assert(obj(MainVerb, Agent)))
; (assert(agnt(MainVerb, Agent)),
assert(obj(MainVerb, Object)))
).

seperat_concept(Agent, NP):-
simplify_list(NP, NP1),
member(X, [n,np]),
seperate([X,A gent], NP1),
nonvar(Agent).

seperat_mvb(Mvb, Object, VP):-
simplify_list(VP, VP1),
(member([], VP1) -> fail; true),
seperate([mnb,Mvb], VP1),
seperat_concept(Object, VP1),
wid(mnv,Mvb:-V),
get_status(Mvb, X, V, VP1),
(  
  X == active -> assert(s_c(active))
  ; assert(s_c(passive))
).

get_status(Mvb, X, V, VP1):-
  member([past], V),
  (  
    separate([Mvb, PP], VP1),
    PP == pp, nl,
    separate([pp, PP1], VP1),
    PP1 == by
  ) -> X = passive
  ; X = active.

get_status(_, active, _, _).

separate(_, []).

separate([X, Y], [X, Y\_]):- Y \= [], X \= Y.

separate([X, Y], [\_L]):-
  separate([X, Y], L).

parsed_data_abolish:-
  abolish(agnt/2),
  abolish(obj/2).

parse(Net, Node, X, X, []; ParseXYIParsesYZ):-
  final(Node, Net).

parse(Net, Node_1, X, Z, [ParseXYIParsesYZ]):-
  arc(Node_1, Node_2, Label, Net),
  traverse(Label, X, Y, ParseXY),
  parse(Net, Node_2, Y, Z, ParsesYZ).

traverse(Word, [Word\_X], X, [Word]):-
  not(special(Word)).

traverse(Category, [Word\_X], X, [Category, Word]):-
  wid(Category, Word; _; _).

traverse(Net, String, StringLeft, [Net\_Parses]):-
  initial(Node, Net),
  parse(Net, Node, String, StringLeft, Parses).

traverse('£', X, X, []).

% simplify_list([], []).

simplify_list([X|Y], [X, B]):- atomic(X), simplify_list(Y, B).

simplify_list(F, L):-
  arg(1, F, X), simplify_list(X, V1),
  arg(2, F, Y), simplify_list(Y, V2),
  append(V1, V2, L).
/* Conceptual Graphs implementation

concept_graph(ConAgent):-
    agnt(MainVerb,ConAgent),
    generic(GenAgent,ConAgent),
    obj(MainVerb,Object),
    generic(GenObj, Object),
    builtcg(GenAgent, GenObj).

builtcg(C1,C2):-
    abolish(condition/1),
    abolish(h_state/1),
    cgO(_,X1:R2:C2),
    cgO(_,X2:R1:C1),
    X1 == X2, R2 != R1,
    write(''), nl,
    c_write(X1),
    h_write,
    assert(h_state(X1)),
    p_generate(X1).

p_generate(C1):-
    cgO(_,C1:R:C2),
    simplify(R,C2),
    fail.

p_generate(_):- end_write.

simplify(R,C2):-
    senten(L),
    valid(R,C2),
    generic(C2,C3),
    member(C3,L),
    rc_write(R,C2),
    a_generate(C2), !.

a_generate(C1):-
    cgO(_,C1:R:C2),
    simplify2(R,C2),
    fail.

a_generate(_).

simplify2(R,C2):-
    senten(L),
    valid(R,C2),
    generic(C2,C3),
    member(C3,L),
    rc_write(R,C2), !.

valid(R1,Y1):-
    senten(L),
    cgO(_,C:R1:Y1),
    \+h_state(C),
    cgO(_,C__:Y2),
    generic(Y2,Y3),
member(Y3,L),
assert(cndtion([],)).

valid(_,Y1):-
  senten(L),
  cg0(_,Y1::Y2),
  \+h_state(Y1),
  generic(Y1,Y3),
  member(Y3,L),
  assert(cndtion([]).)).!

valid(R1,Y1):-
  senten(L),
  cg0(_,R1:Y1),
  \+h_state(R1),
  cg0(_,R2:Y2),
  generic(Y1,Y3),
  member(Y3,L),
  generic(Y2,Y4),
  member(Y4,L),
  R1 \= R2,
  assert(cndtion([\=])).).

valid(R1,Y1):-
  senten(L),
  cg0(_,R1:Y1),
  generic(Y1,Y2),
  \+member(Y2,L),
  assert(cndtion([\+])).).

valid(_,Y):-
  assert(cndtion([\+])).

newcg0(_,C1:C2):-
  cg0(_,C1:C2), !.

rc_write(R,C1):-
  retract(cndtion(Cndtion)),
  r_write(R),
  c_write(C1),
  write(Cndtion),
  nl, !.

h_write:- !,
  write('!'), nl.

r_write(X):- !,
  write('!'),
  write(X),
  write('!->').

c_write(X):- !,
  write('!'),
  write(X).

cn_end:- !,
  write('!'), nl.

Appendix D

generic(C2,C1):-
    wid(_,C1:C2:_),!.

% % English 1
% % the S network
% initial(0,s).
final(2,s).
% arc(0,1,np,s).
arc(1,2,vp,s).
% % the NP network
% initial(0,np).
final(2,np).
% arc(0,1, det,np).
arc(0,1,'£',np).
arc(1,1,adj,np).
arc(1,1,n,np).
arc(1,2,n,np).
arc(2,2,pp,np).
arc(2,3,wh,np).
arc(3,2,vp,np).
% % PP phrase
% initial(0,pp).
final(1,pp).
arc(0,1, np, pp).

% % the VP network
% initial(0,vp).
final(1,vp).
final(2,vp).
% arc(0,1,aux,vp).
arc(0,1,'£',vp).
arc(1,1,mnv,vp).
arc(1,2,np,vp).
arc(1,2,'£',vp).
arc(2,2,pp,vp).
arc(2,2,adv,vp).
arc(1,3,that,vp).
ar(3,2,s,vp).

% file:dictionary.pl
%
/*****************************/
% common dictionary
/*****************************/
Appendix D

wid(adj, mobile:mobile:*).
% wid(mod, very:very:*).
% wid(adv, often:often:*).
wid(adv, always:always:*).
wid(adv, sometimes:sometimes:*).
wid(adv, fast:fast:*).
wid(adv, fastly:fast:*).
% wid(adv, in:in:*).
wid(adv, quickly:fast:*).
wid(adv, rapidly:fast:*).
%
wid(pp, in:in:*)
wid(pp, under:under:*)
wid(pp, to:to:*)
wid(pp, by:by:*)
wid(pp, for:for:*)

/***************************************************/
% Canonical conceptual graphs
% cg(0, identity, 'canonical form of a concept':relation:
% 'canonical form of a concept'),
/***************************************************/
cg0(1, chase:agnt:cat).
cg0(2, chase:obj:mouse).
cg0(3, generate:agnt:system).
cg0(4, generate:obj:data).
cg0(5, data:manr:fast).
cg0(6, data:manr:slow).
cg0(7, generate:obj:electricity).
cg0(8, electricity:manr:fast).
cg0(9, act:agnt:animate).
cg0(10, age:chrc:entity).
cg0(11, age:ptim:time).
cg0(12, arrive:agnt:mobile_entity).
cg0(13, arrive:loc:place).
cg0(14, entity:attr:attribute).
cg0(15, believe:expr:animate).
cg0(16, believe:obj:proposition).
cg0(17, phyobj:size:big).
cg0(18, big:comp:phyobj).
cg0(19, person:chld:child).
cg0(20, order:obj:command).
cg0(21, order:rcpt:person).
cg0(22, phyobj:attr:color).
cg0(23, allow:agnt:system).
cg0(33, object:type:mobile).
cg0(25, allow:manr:tracking).
cg0(26, produce:agnt:system).
cg0(27, produce:obj:data).
cg0(28, object:qnty:ten).
cg0(29, allow:obj:object).
cg0(30, tracking:space:three).
cg0(31, allow:agnt:program).
%
172
% file lcp.pl
:- dynamic lcp/2.

/********************************************/
% LCP Data
/********************************************/

lcp(quantifier_indicator, [some]).
lcp(quantifier_indicator, [several]).
lcp(quantifier_indicator, [all]).
lcp(quantifier_indicator, [one]).
lcp(quantifier_indicator, [most]).
lcp(quantifier_indicator, [many]).
lcp(quantifier_indicator, [more]).
lcp(quantifier_indicator, [seldom]).
lcp(quantifier_indicator, [often]).
lcp(relation_indicator, [contain]).
lcp(relation_indicator, [contains]).
lcp(relation_indicator, [represents]).
lcp(relation_indicator, [represent]).
lcp(relation_indicator, [represented,by]).
lcp(relation_indicator, [properties]).
lcp(relation_indicator, [structures]).
lcp(relation_indicator, [consisting]).
lcp(relation_indicator, [has]).
lcp(relation_indicator, [have]).
lcp(relation_indicator, [with]).
lcp(relation_indicator, [is,a]).
lcp(relation_indicator, [type]).
lcp(relation_indicator, [a,kind,of]).
lcp(rule_indicator,[if]).
lcp(rule_indicator,[then]).
lcp(rule_indicator,[unless]).
lcp(rule_indicator,[given,that]).
lcp(rule_indicator,[there,are]).
lcp(rule_indicator,[so,that]).
lcp(rule_indicator,[in,addition,to]).
lcp(explanation_indicator,[because]).
lcp(explanation_indicator,[in,case,of]).
lcp(explanation_indicator,[due,to]).
lcp(explanation_indicator,[shall,be]).
lcp(explanation_indicator,[there,are]).
lcp(explanation_indicator,[there,is]).

% Function words in file 'functionwords.pl'

fw(1,[a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0]).
fw(2,[am, an, as, at, be, by, do, he, if, in, is, it, me, my, no, of, oh, on, or, so, to, up, us, we, ye,10,11,12]).
fw(3,[ado, all, and, any, are, but, can, did, etc, few, for, get, got, had, has, her, hes, him, his, how, its, may, may, nor, not, now, off, one, out, own, per, she, six, ten, the, too, two, was, way, who, why, yea, yes, yet, you]).
fw(4, [also, anon, away, been, both, does, done, dont, down, each, else, even, ever, five, four, from, gets, have, here, into, just, keep, kept, less, lest, many, mine, more, most, much, must, next, nine, noes, none, once, ones, only, onto, ours, over, past, plus, real, same, self, some, such, than, that, thee, them, then, they, this, thou, thus, unto, upon, very, well, were, what, when, with, whom, will]).

fw(5, [about, along, being, first, later, other, shall, their, there, these, thine, thing, third, those, three, truly, twice, under, until, wasnt, where, which, while, whose, would, yeses, yours]).

fw(6, [across, almost, always, amount, anyone, around, awhile, befor, behind, behind, beyond, cannot, during, eighth, eighty, either, eleven, enough, except, fairly, figure, fourth, hardly, having, height, herein, hither, indeed, inward, itself, mainly, middle, mighty, myself, nobody, number, others, please, pretty, rather, really, second, selves, should, theirs, thence, things, thirds, thirty, through, thurice, toward, twelve, twenty, unless, upward, weight, whence, whilst, withal, whence, within]).

fw(7, [against, already, another, anybody, awfully, because, between, farther, forever, forward, further, herself, himself, howbeit, howbeit, however, hundred, insofar, instead, million, neither, nothing, nowhere, outside, outward, perhaps, seventy, several, sixteen, someday, thereby, therein, thereof, thereon, thither, through, thyself, undoing, whereas, wherein, wherewith, thereby]).

fw(8, [although, anything, anywhere, backward, eighteen, evermore, everyone, fourteen, inasmuch, insomuch, likewise, millenia, millions, nowadays, overmuch, somebody, somewhat, thousand, together, whatever, whenever, wherefor, yourself]).

fw(9, [aforesaid, elsewise, forasmuch, foregoing, halfdozen, otherwise, ourselves, something, sometimes, shouldest, therefore, therewith, twothirds, wherefore, wherewith]).

fw(10, [beforehand, everything, everywhere, fourteen, henceforth, heretofore, oftentimes, themselves, thereafter, throughout, underneath, yourselves]).

fw(11, [furthermore, midthirties, theretofore, twentyseven]).

fw(12, [backwardness, nevertheless, whereinsoever]).

fw(_, []).

% General purpose processes and actions indicating words in file:‘processwords.pl’

prv(4, [form, make, made, give, show]).
prv(5, [allow, build, built, define, forms, given, gives, makes, shows, store]).
prv(6, [allows, builds, create, formed, defined, defines, giving, making, stored, verify]).
prv(7, [allowed, compute, collect, created, creates, forming, display, exhibit, extract, perform, provide, produce, showing, storing]).
prv(8, [allowing, building, computed, computes, creating, collects, defining, displays, exhibits, generate, performs, provides, provided, produces, produced, retrieve, verified]).
prv(9, [calculate, collected, computing, displayed, generates, generated, extracted, determine, performed, producing, retrieved, verifying]).
prv(10, [calculated, calculates, collecting, determined, determines, 

174
displaying, exhibiting, generating, illustrate, performing)).
prv(11, [calculating, determining, demonstrate, illustrated,
retrieving ]).
prv(12, [demonstrated, demonstrates]).
prv(13, [demonstrating, illustrating]).
prv(_, []).

175
Appendix E Software Development Glossary


**acceptance**
The act by which the user indicates to the supplier that the agreed goods and services have been provided satisfactorily. It is quite common for acceptance to be agreed with minor defects and omissions identified, but with an understanding that these will be corrected by an agreed date.

**acceptance testing**
Formal testing conducted to determine whether or not a system satisfies its acceptance criteria and hence to enable the user to determine whether or not to accept the system.

**acceptance testing documentation**
The supplier's documentation concerned with testing the system to demonstrate its capability of meeting the requirements of the Functional Specification.

**access security**
Hardware or software features, operating procedures or management procedures designed to permit authorised access and prevent unauthorised access to a computer system.

**accuracy**
The ratio of error to total value. It is usually expressed as percentage.

**adaptability**
The ease with which software allows differing system constraints and user needs to be satisfied.

**algorithm**
A finite set of well-defined rules for the solution of a problem in a finite number of steps; e.g. a full statement of an arithmetic procedure for evaluating \( \sin x \) to a stated precision.

**application language**
A problem-oriented language whose statements contain or resemble the terminology of the occupation or profession of the user.

**application software**
Software specifically produced for the functional use of a computer system (contrast with environment software).

**archiving**
The transfer of current operational data and software to a permanent storage medium to allow later regeneration in case of corruption or loss (or may be required for audit purposes).

**as-built system**
The tested and approved system with all supporting documentation, supplies and spares.

**asynchronous**
A timing system in which each event or the performance of each operation starts as result of a signal generated by the completion of the previous event or operation, or on the availability of these parts of the system required for the next event or operation.

**automatic control (mode)**
Control or process by automatic means.

**availability (system)**
The ratio of system up-time to total operating time.
backing store  
Storage that is accessible by a compute only through input-output channels.

back-up  
Provisions made for the recovery of data files or software, for restart of processing, or for use of alternative equipment or procedures after a system failure.

baseline  
A specification or build of a computer system that has been formally reviewed and agreed upon, that thereafter serves as basis for further development, and that can be changed only through formal change control procedure.

benchmark  
A standard or point of reference against which a particular feature of a system is measured. The usual benchmark test are those which compare different computers with each other using the criteria of speed of operation, throughput, responses, etc.

CCITT  
Consultative Committee International Telegraph and Telephone. A committee within the International Telecommunication Union. It concerns itself with the conventions which enable transfer of data between electronics systems. Its most well known series of recommendations are its 'V' series and X25 specifications. These conventions are commonly referred to in user requirements specifications and supplier tenders.

change control  
The process by which a change is proposed, evaluated, approved or rejected, scheduled and tracked.

configuration  
The totality of the hardware, software, firmware, services and supplies required for the successful operation of a computer-based system or associated group of systems at a given reference point in time.

data acquisition  
A general term for the capture of data from various sensors and the processing of data for presentation to the operator in the form of VDU displays, printed logs, charts, etc.

database  
A collection of data fundamental to a system.

design review  
The formal review of an existing or proposed design for the purpose of detection and remedy of design deficiencies that could affect fitness for use and environmental aspects of the product or process, and/or for identification of potential improvements of performance, safety and economic aspects.

ergonomic design  
Design for efficiency and ease of use, e.g. most frequently used keys within easy reach of the operator, or VDU screens readily discernible in ambient light conditions and formats arranged for ready recognition of critical information.

environment software  
The total software which provides an operating environment for application software. It includes the operating system and other usually standard software components such as language interpreters, database management packages and other run-time utility software.

fall-back control  
Provision made for the use of alternative hardware or procedures in the event of a system failure.
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>feasibility study</td>
<td>A study to identify and analyse the problems associated with an outline proposal for a system development project in order to demonstrate its viability, costs and benefits.</td>
</tr>
<tr>
<td>formal language</td>
<td>A language whose rules (syntax) are explicitly established prior to its use, e.g. programming language. Contrast with natural language.</td>
</tr>
<tr>
<td>freeze date</td>
<td>The date beyond which no change to a specification or formal report will be accepted.</td>
</tr>
<tr>
<td>functional specification</td>
<td>A document agreed between supplier and user, which defines what will be provided under the contract.</td>
</tr>
<tr>
<td>issue level</td>
<td>The specification level of a software product at the time of release. See software release.</td>
</tr>
<tr>
<td>kernel (software)</td>
<td>The nucleus or core of an operation system.</td>
</tr>
<tr>
<td>life cycle</td>
<td>A succession of discrete activities or phases covering the total life of a project from initial conception to disposal.</td>
</tr>
<tr>
<td>maintainability</td>
<td>The ease with which maintenance of a functional unit can be performed in accordance with prescribed requirements. It may also be defined as the probability that a failed functional unit will be returned to operational effectiveness within a given period of time.</td>
</tr>
<tr>
<td>natural language</td>
<td>A language whose rules are based on current usage without being explicitly prescribed, e.g. English, French, etc.</td>
</tr>
<tr>
<td>operational data (file)</td>
<td>Data essential to the normal operation of a system.</td>
</tr>
<tr>
<td>PERT</td>
<td>Project Evaluation and Review Technique; a project management system.</td>
</tr>
<tr>
<td>post-installation documentation</td>
<td>The totality of documents supplied to the user by the supplier of operational support and, where applicable, future development of the system.</td>
</tr>
<tr>
<td>redundancy</td>
<td>The inclusion of duplicate or alternate system elements to improve operational reliability by ensuring continued operation in the event that a primary element fails.</td>
</tr>
<tr>
<td>security</td>
<td>The protection of computer hardware and software from accidental or malicious access, use modification, destruction or disclosure.</td>
</tr>
<tr>
<td>simulation</td>
<td>The representation of physical phenomena by means of operations performed by a computer system.</td>
</tr>
<tr>
<td>soak test</td>
<td>The process of submitting a system to continual working under a prespecified environment and workload for a predetermined duration. It may include operation at an abnormally high temperature.</td>
</tr>
<tr>
<td>software system specification</td>
<td>The supplier's documentation detailing how the software is designed to meet the requirements of the Functional Specification,</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>takeover</td>
<td>Usually the stage in the fulfilment of a contract at which the customer takes possession of equipment in terms of ownership and custody. It does not necessarily imply agreement that the equipment is satisfactory or complete. Contrast with acceptance.</td>
</tr>
<tr>
<td>tender</td>
<td>A suppliers's response indicating his proposal for meeting all the user's specified requirements.</td>
</tr>
<tr>
<td>turnkey</td>
<td>A contract in which an agent undertakes to furnish for a fixed price all materials and labour, and to do all the work needed to complete a system.</td>
</tr>
<tr>
<td>user requirements specification</td>
<td>A statement by the user of his total requirements</td>
</tr>
<tr>
<td>validation</td>
<td>A test of module, subsystem or system, within its environment and against its requirements, to determine whether it fulfils it functions.</td>
</tr>
<tr>
<td>walkthrough</td>
<td>A review process in which a designer leads one or more other members of the development team through a segment of design or code, while the other members comment on technique, style, possible errors, violation of development standards and other problems.</td>
</tr>
</tbody>
</table>
Bibliography

(Acly 1988)

(Ada 1983)

(Ahmad et al. 1987)

(Alam 1988)

(Alexander et al. 1988)

(ANSI/IEEE 1984)
ANSI/IEEE: An American National Standard, The Institute of Electrical and Electronics Engineers, Inc. 345 East 47th Street, New York, NY10017, USA.

(Bainbridge 1986)

(Balkovich & Engelberg 1976)

(Baranofsky 1974)

(Barr & Feigenbaum 1981)

(Bartlett 1932)

(Basili et al.1983)

(Basili & Weiss 1984)
Victor R. Basili and David M. Weiss, "A Methodology for Collecting Valid Software

(Basili et. al 1986)

(Bates 1978)

(BCS 1977)

(Berzins & Gray 1985)

(Birrell & Ould 1985)

(Boehm 1981)

(Boehm 1988)

(Boehm & Papaccio 1988)

(Boguraev 1983)

(Boose 1984)

(Boose 1986)

(Boose & Bradshaw 1988)

(Boreham & Niblett 1976)
Bibliography

(Borgida et al. 1985)

(Brachman 1979)

(Brachman et al. 1978)

(Brachman & Schmolze 1985)

(Braun & Givone 1981)

(Bresnan 1982)

(Breuker & Wielinga 1985)

(Brown 1989)

(Bruce 1975)

(Bruce & Moser 1989)

(Buchanan & Feigenbaum 1978)

(Buchanan 1981)

(Buchanan et al. 1983)

(Cameron 1986)
(Celko 1982)

(Cercone 1980)

(Chen 1978)

(Chen 1976)

(Chikofsky & Rubenstein 1988)

(Chomsky 1957)

(Chomsky 1965)

(Chomsky 1982)

(Chow 1978)

(Chvalovsky' 1983)

(CLark & Clark 1977)

(Clocksin & Mellish 1981)

(COLLINS 1987)

(Connor 1977)
(Connor 1985)

(Cottrell & Small's 1983)
G. W. Cottrell and S. L. Small, "A connectionist scheme for modelling word sense disambiguation", Cognition and Brain Theory, 6, 89-120.

(Dawson 1989)

(Demarco 1978)

(Diederich et al 1986)

(Diederich et al. 1988)

(Duda 1981)

(Evens et al. 1980)

(Feigenbaum & McCorduck 1983)

(Finkelstein & Potts 1985)

(Finkelstein 1989)

(Freiling et al. 1986)

(Gaines & Boose 1988)
Bibliography


(Gevarter 1982)

(Greene 1988)

(Greenspan 1984)

(Greenwell 1988)

(Grover 1983)

(Guenthner et al. 1986)

(Halstead 1977)

(Hart 1986)

(Hayward et al. 1988)

(Hekmatpour & Ince 1986)

(Hughes et al. 1988)

(Hurley 1983)

(IBM 1981)

(Infomatics 1989)
A survey on CASE tools, "For Systems and Marking Management: infomatics,"
INFOMATICS, APRIL 1989.

(Jackson 1980)

(Jackson 1982)

(Jackson 1986)

(Jackson 1990)

(Janardan & Salvendy 1988)

(Jensen & Heidorn 1983)

(Kelly 1955)

(Key 1979)

(Koch & Heicking 1980)

(Kramer et al. 1988)

(Kuipers & Kassirer 1983)

(Kuipers & Kassirer 1984)

(Lehmann 1978)

(Lehmann 1980)
Bibliography


(Lehman 1984)

(Loucopoulos & Champion 1988)

(Lowery & Duran 1989)

(Lyons 1968)

(Lyons 1977)

(Martin & Tsai 1990)

(McCabe 1976)

(Metzner & Barnes 1977)

(Meyer & Skuce 1989)

(Miller et al. 1958)

(Minsky 1975)

(Montgomery 1972)

(Mullery 1979)
G., Mullery, "CORE—a method for controlled requirements specification,"

(Musen et al. 1988)

(Myers 1978)

(Newell & Simon 1972)

(Newell 1982)
A. Newell, "The knowledge level," Artificial Intelligence, 18, pp. 87-127.

(Nisbett & Wilson 1977)
R. E. Nisbett and T. D. Wilson, "Telling more than we can know: verbal reports on mental processes," Psychological Review, 84, pp. 231-259.

(Odell 1981)
S. Jack Odell, "Are Natural Language Interfaces Possible?" Report TR 73-024, IBM Systems Research Institute, New York.

(Ohnishi et al. 1985)
Atsushi Ohnishi, Kiyoshi Agusa, Yutak Ohno, "Requirements Model and Method of Requirements Definition," Department of Information Science, Kyoto University, Kyoto 606, Japan.

(Osgood & Sebeok 1965)

(Ott & Zoeppritz 1979)
N. Ott and M. Zoeppritz, "USL—An Experimental Information System Based on Natural Language," In L. Bolc. (Ed.) Natural Language Based Computer Systems, Hanser, Munich, 1979, p. 4.

(Ould et al. 1982)

(OXFORD 1986)

(Parkinson et al. 1977)

(Pereira & Shieber 1987)
(Phillips & Pugh 1987)

(Porter 1980)

(Quillian 1968)

(Quirk et al. 1986)

(Ramamoorthy et al 1987)

(Rich & Waters 1988)

(Rosenberg 1980)

(Ross 1977)

(Ross & Schoman 1977)

(Ross 1977)

(Royce 1970)

(Rusk & Krone 1984)

(Sager et al. 1980)

(Salter 1976)
(Salton 1975)

(Salton & McGill 1983)

(Schank 1973)

(Schank et al. 1973)

(Schank 1975a)

(Schank 1975b)

(Schank & Abelson 1977)

(Schank & Riesbeck 1981)

(Sebastiani et al. 1989)

(Shapiro 1989)

(Shaw & Gaines 1988)

(Shorter 1987)

(Silva & Montgomery 1977)
(Simmons 1984)  

(Skidmore & Wroe 1988)  

(Sloman 1980)  

(Smith & Baker 1983)  

(Sommerville 1987)  

(Sowa & Way 1986)  

(Sowa 1983)  

(Sowa 1984)  

(Sowa 1989)  

(STARTS 1987)  
The STARTS Guide, Second Editions, Volum1, and 2, DTI and NCC.

(Stephens & Whitehead 1985)  

(Stefik 1982)  

(Stratton & Hayes 1988)  
Peter Stratton and Nicky Hayes, "A Student's Dictionary of Psychology," Publisher Edward Arnold.

(Vleduts-Stokolov 1987)  

(Walker 1986)  
(Waltz&Pollack 1985)

(Waterman 1985)

(Weizenbaum 1966)

(Welin 1979)

(Wells 1990)

(Wilks 1977)

(Winograd 1983)

(Wolff 1989)

(Woods 1970)

(Woods 1973)

(Yourdon & Constantine 1979)

(Zoeppritz 1984)