City Research Online

City, University of London Institutional Repository

Citation: Cook, S.C. (1990). A knowledge-based system for computer-aided generation of measuring instrument specifications. (Unpublished Doctoral thesis, City, University of London)

This is the accepted version of the paper.

This version of the publication may differ from the final published version.

Permanent repository link: https://openaccess.city.ac.uk/id/eprint/29073/

Link to published version:

Copyright: City Research Online aims to make research outputs of City, University of London available to a wider audience. Copyright and Moral Rights remain with the author(s) and/or copyright holders. URLs from City Research Online may be freely distributed and linked to.

Reuse: Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge. Provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way.

City Research Online: http://openaccess.city.ac.uk/ publications@city.ac.uk

A Knowledge-Based System for Computer-Aided Generation of Measuring Instrument Specifications

by

Stephen Clive Cook

A thesis submitted to
City University
for the degree of

DOCTOR OF PHILOSOPHY

Measurement and Instrumentation Centre,
Department of Electrical, Electronic and Information
Engineering, City University, London EC1V 0HB, U.K.

November 1990

Contents

List of Figures

Acknowledgements

I would like to acknowledge the support and encouragement given to me by my joint supervisors Professors L. Finkelstein and P.H. Sydenham and thank them for accommodating my need to start the research programme part-time. I would like to express my gratitude to Dr. A Finkelstein for expanding my awareness of knowledge representation and the application of formal methods.

A post-graduate research programme, especially one largely conducted part-time, has a significant impact on one's family. I would particularly like thank my wife, Hilary, for her support and tolerance over what has been a very long haul. My sons, Aaron and Robert, small as they are, have also done their best to give me the time I needed to complete this work. Of the many friends and family who have given us help in our relocation to the U.K. this year, my wife's parents Mr & Mrs Burke, deserve special mention.

All three employers I have had during the period of the research programme have provided essential support. I am grateful to British Aerospace Australia Limited who granted me paid study leave each week and enabled me to combine visits to the City University with business trips. I am also grateful to Vision Systems who allowed me study leave. I am particularly grateful to my current employer, the Electronics Research Laboratory of the Defence Science and Technology Organisation, who not only granted me study leave for a year's part-time study but also awarded me a Postgraduate Fellowship for 1990 to allow me to complete this work full time. I would also like to thank members of the Terrestrial Transmissions Systems of ERL for their informative communications throughout the year and help with many of the drawings.

Declaration

I grant powers of discretion to the University Librarian to allow this thesis to be copied in whole or in part without further reference to me. This permission covers only single copies made for study purposes, subject to normal conditions of acknowledgement.

Abstract

The task of writing specifications for measuring instruments requires knowledge from many fields, including specification writing practices, measurement science and current instrumentation practice. The subject of this thesis is the automation of this difficult specialist task. This topic forms part of larger systems concerned with instrumentation system design automation at the Measurement and Instrumentation Centre at City University and the Measurement and Instrumentation Systems Centre at the South Australian Institute of Technology. The thesis commences with a thorough review of the specification process as applied to measuring instruments. This examination of the specification process, was performed with cognisance of the engineering management procedures in common use in industry. The outcome of this study was the extraction of salient methods and techniques needed to systemise the production of formatted instrument specifications. These were ensconced in a procedural program suite, *Specriter 1*, which demonstrated the potential of computerising the process. The valuable knowledge gained in the production and use of this system is discussed. The body of the thesis covers the more ambitious limited formal system, *Specriter 3*, which incorporates integral domain knowledge to provide substantial user assistance, and checking for consistency and reasonableness. The frame-based generalised documentation generation shell constructed for this task, is discussed together with the features which make it particularly useful for *Specriter 3*, such as, the ability to represent knowledge of the human interface, the output text, and the problem domain in a modularised fashion, the ability to reason with the contents of the knowledge base directly, and the complete independence of the knowledge base from the shell. The knowledge-base editor *Framedt*, is described, which enables the knowledge base to be readily altered or a new one constructed for a different document generation task. *Specriter* has been used in two practical applications, and these are referenced in the text.

Chapter 1
Introduction

1.1 Preface

This thesis describes the work performed as a Research Student at the City University Measurement and Instrumentation Centre (MIC) and the South Australian Institute of Technology Measurement and Instrumentation Systems Centre (MISC). The majority of the research was conducted part time in Australia in conjunction with MISC whilst employed as an Engineering Manager, with the final year being undertaken full time at MIC at City University.

The subject of the research is the creation of a computer tool to assist in the production of measuring instrument specifications. This work forms an integral part of the Computer-Aided Engineering of Instruments (CAEINST) research program at MISC which seeks to provide the necessary knowledge for a user to specify, create, install and apply capable measurement and control systems (Sydenham 1987). It also forms part of the research programme of the Design Theory and Methods Group, at MIC, aimed at automating the measuring instrument specification process (Finkelstein et. al., 1990). The design process can be thought of as a sequence of stages each of which consists of information gathering and organisation, formulation, generation of candidate designs, analysis of candidate designs, and finally decision (Finkelstein and Finkelstein, 1983). Each of these can be automated separately and this research is concerned with the first stage of the design process, that is, the preparation of the requirements statement.

1.2 Aim of the Research

The aim was to produce an automated, or at least a computer-assisted method of generating a measuring instrument requirements specifications from a requirements

analysis. A further aim was to produce a specification in a form that can be used for both technical, management, and contractual purposes. Another objective was to provide direct interfacing to the other instrument design packages under development at the two research centres, for example, measurement system requirements elicitation (Sydenham et. al., 1990), and design concept generation (Mirza et. al., 1990).

1.3 Background

The current CAEINST concept was derived from the observation that there is an underlying structure to the process of designing measuring instruments, and that there exists commonly applicable principles and methodologies for this design field (Bosman, 1978; Finkelstein & Finkelstein, 1983 & 1985; Sydenham, 1984). From this, it was believed that it would be possible to construct a computer-aided design package using this structure together with stored knowledge. From this background, the context for the research program was framed:

(a) The users can be expected to be educated, but not expert in measurement science, instrumentation, or specification writing.
(b) In the first instance, concentrate on the incorporation and structuring of existing knowledge and techniques.
(c) Project management aspects are to be considered, not just technical ones.
(d) At a later stage, the resulting software would be integrated into larger systems.

1.4 Structure of the Thesis

The thesis is divided into eight chapters. The introduction and conclusion have been kept small to encourage complete reading. The remaining chapters are outlined below. Firstly, Chapter 2 discusses the various types of measuring instrument specifications used in industry and how they relate to the literature of design methodology. A brief review of the instrument design cycle highlights development requirements specifications as the principal focus for this research effort.

Chapter 3 opens by examining the specification practices currently in use for

measuring instruments. In this treatment, emphasis is given to the more formal specification practices generally found when the customer and supplier are from different organisations and cost and timescales are important. The outcome from this review is a choice of specification format. The chapter continues with a review of specification language and style. Against this background, the case for automating specification generation is argued. There are no precedents for this type of task, so the human specification generation method was scrutinized and from this a process which could be implemented on a computer is developed. The remainder of the chapter is devoted to describing methods which can be used to specify each of the paragraphs identified in the chosen specification format.

Chapter 4 opens by enumerating the design aims for a software tool to generate measuring instrument specifications. *Specriter 1,* the first known computer-aided tool of its type, is then described. A great deal was learned from this activity and this is recorded later in the chapter.

Chapter 5 examines theoretical concepts that are of interest for further development of the *Specriter* idea. It commences by examining the possibility of applying formal specification methods to measuring instruments. This is followed by a review of knowledge representation techniques suitable for the domains of instrument engineering and specification writing. The chapter concludes by choosing a formal specification paradigm and a cooperating knowledge representation technique suitable for implementation on available computer systems.

Chapter 6 describes *Specriter 3*, the software system which embraces the ideas of Chapter 5. Whereas, Chapter 7 describes how *Specriter 3* can be used to generate, edit and print measuring instrument requirements specifications. As the mechanics of the software have been discussed in the previous chapter, this description is designed to illustrate the contents of the measuring instrument knowledge base used by *Specriter 3.*

Chapter 8 concludes the thesis by summing up the achievements of all phases of the project and the contribution it has made to knowledge.

Chapter 2
Review of Specifications as Engineering Management Tools

2.1 Introduction

The relevant Oxford English Dictionary (1989) definition of the word *specification* states:

> "defn. 4d. *techn.* A detailed description of the particulars of some projected work in building, engineering, or the like, giving the dimensions, materials, quantities etc., of the work, together with directions to be followed by the builder or contractor; the document containing this."

The dictionary traces the usage of the word *specification* in this context to 1833 when it was introduced by architects to describe the materials and workmanship required for building contracts. From the inception of engineering specifications as we know them, these documents have formed part of the agreement, or contract, between a customer and a supplier (Mead et. al. 1956; Dunham et. el. 1979). Thus the engineering specification takes on the dual role of describing the product or service in technical terms and becoming a management tool. In their latter role, specifications are used for estimating labour and material costs, controlling the conduct of the work, and finally for determining acceptability of the final deliverable. When a specification becomes part of a contract, the level of detail is necessarily increased and consideration has to be given to the legal and financial consequences of ambiguity and lack of completeness (Mead et. al. 1956). It is this form of specification that this thesis is primarily concerned with. This chapter discusses the various types of measuring instrument specifications used in industry and how they relate to the literature of design methodology.

Instruments are often part of larger systems. System engineering employs numerous

specification types and it is advantageous to examine these to obtain the broadest perspective on specification methods and applications. The first section of this chapter is devoted to this topic.

The instrument design cycle is reviewed next. The exact type of specification this project is concerned with is then identified within the system engineering framework and the instrument design cycle.

The chapter concludes by consolidating this industrial background against more theoretical viewpoints on instrument design methodology.

2.2 Specifications in Systems Engineering

Systems engineering has gained increasing attention since its recognition as a discipline following the second World War. This has been stimulated by the increasing cost and technical complexity of development and acquisition programs. Some of this attention is no doubt due to large program failures which could possibly have been avoided, or at least mitigated through the use of system engineering (M'Pherson, 1980; DSMC, 1983).

The purpose of systems engineering is to prevent these failures through a unified approach that completely defines all requirements on the system and establishes a system configuration which can be shown to meet those requirement before detailed development commences. The definition of the terms *system* and *system engineering* depend on the application. The US Department of Defense definition given in DSMC (1983), derived from MIL-STD-499A (1974), is perhaps the broadest and the most common interpretation:

> "System Engineering is the application of scientific and engineering efforts to (a) transform an operational need into a description of system performance parameters and a system configuration through the use of an iterative process of definition, synthesis, analysis, design, test, and evaluation; (b) integrate related technical parameters and ensure compatibility of all physical, functional, and program

interfaces in a manner that optimizes the total system definition and design; (c) integrate reliability, maintainability, safety, survivability, human, and other such factors into the total engineering effort to meet cost, schedule, and technical performance objectives."

In its simplest terms, systems engineering is both a technical and a management process and is often referred to as a "front end" process. That is, the majority of system engineering tasks are completed in the initial phase of the program, termed *realisation* by M'Pherson, when about 5% of the program's funding is expended (DSMC 1983).

Only those aspects of systems engineering which relate to measuring instruments as system components will be pursued here. Of particular interest to this research project is the first part of the system engineering definition, namely, the generation of performance parameters and system configuration from an operational needs statement. The outcome of this process is a complete set of system requirements documented in a system specification and augmented by supporting specifications and by a number of subsystem and lower level development specifications. Figure 2-1 illustrates a typical system specification tree for a space-flight instrument.

The system support specifications of Figure 2-1 are used to hold system specific requirements which are common to many levels of the specification hierarchy. Typical example include quality assurance requirements, safety standards, workmanship standards, electromagnetic compatibility design standards and practices, software standards and practices. The advantage of placing these requirements in separate documents is to allow citation from each level of the specification tree thereby avoiding needless repetition.

The system design process partitions the system requirements by function or discipline until a level is reached where specific hardware items or software routines are identified. The partitioning process usually involves the use of analysis and simulation to translate and allocate all the system-level requirements to equipment-level requirements (DSMC 1983). Examples include error budgets, mass, power

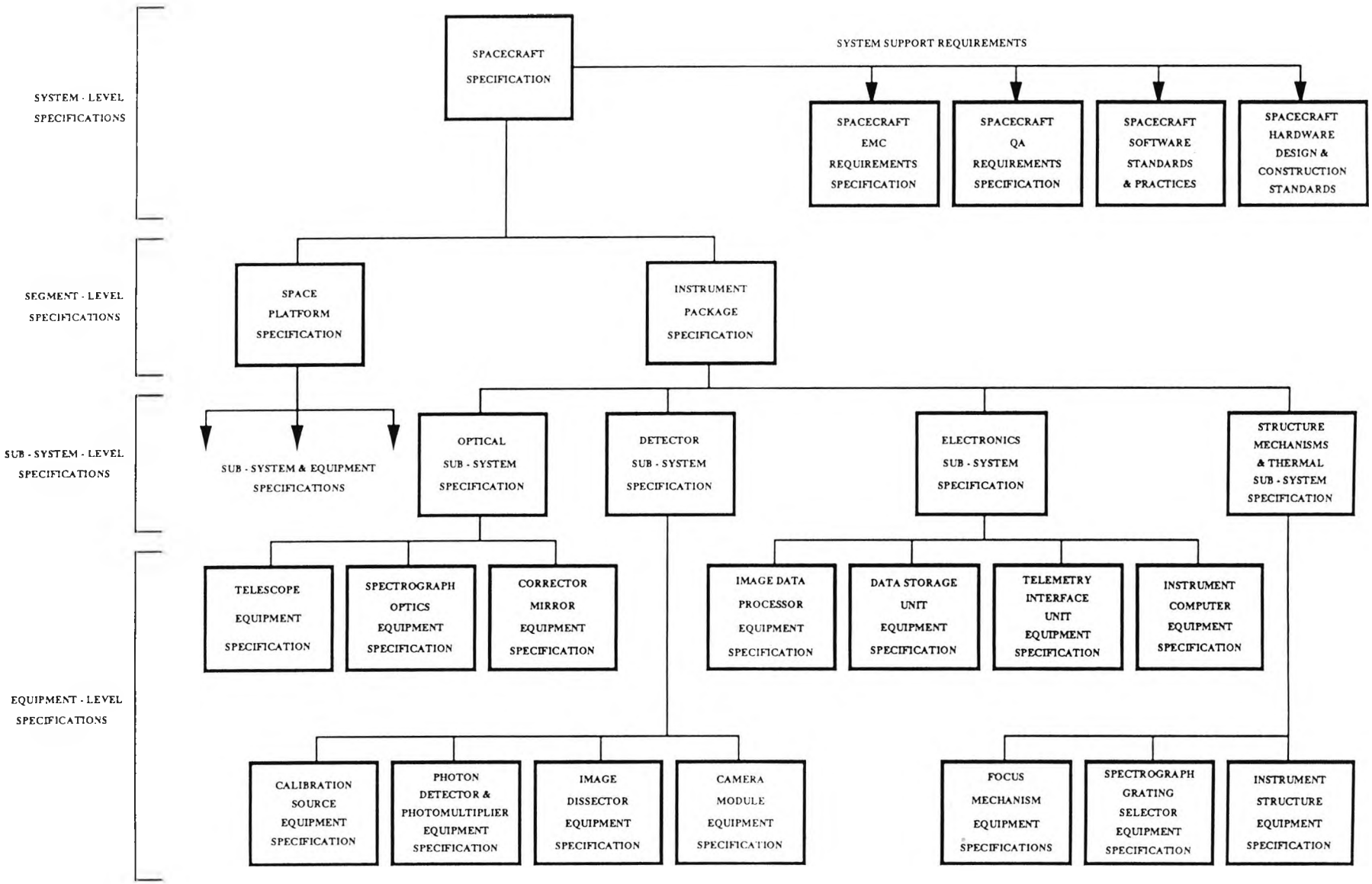


Figure 2-1 Typical System Specification Tree

consumption, reliability, structural alignment, communication bus loading. Note that non-functional requirements are very definitely included in this process. Thus from a system perspective, an individual instrument appears as one component. The common conception of an instrument specification - a single page of predominantly functional requirements - is dangerously inadequate. The complete system may fail if inadequate attention is paid to such factors as availability, reliability, maintainability, supportability, survivability, ergonomic factors, safety, and internal and external compatibility (M'Pherson, 1980). Hence it is not only desirable but essential for the specification to encompass all of these subjects.

The designer of an instrument destined to become part of a larger system, is isolated from the ultimate customer's desires and preferences by the system designers and by a lack of a system-wide perspective (M'Pherson 1981). This places greater emphasis on the completeness of the specification.

The acquisition of the various equipments which comprise a system may be achieved through purchase of existing products or alternatively through commissioning the development of purpose-built entities. This choice is usually determined by circumstances rather than by choice. It is far less expensive to purchase existing equipment, where possible, and this would be the preferred route for most industrial systems. Military and space systems also make use of existing equipment, in particular general purpose items such as computers, however a higher proportion of new developments is common. The type of specification needed in either case is different. This issue is the topic of discussion for the next section and is elaborated further in Chapter 3.

2.3 The Use of Specifications in the Instrument Development Process

2.3.1 The Industrial Instrument Development Process

The instrument development process is preceded by the conversion of the measurement problem description, inherent in which is an understanding of knowledge the instrument is to obtain, into a development specification. If the

instrument is part of a larger system, then this task will be performed as part of the system design process (M'Pherson, 1981). A similar process must also occur if the instrument is the highest level entity under consideration. Stage 1 of the Measurement Process Algorithm (Sydenham, 1985b) is apt in either case.

The measuring instrument life cycle is generally considered to commence from the design specification through to production and later post production support. This process has been established for many years and differs little now from Figure 2-2 extracted from Draper, McKay & Lees (1952). The process commences from what they refer to as the *design specification* (which is also commonly referred to as a *requirement specification, design aim,* or *development specification)* and proceeds into the research and development phase. The outcome of this phase, often referred to as the *A Phase* or *Alpha Phase*, is all the information needed to develop a production model. The physical principle of operation will have been determined, theoretical performance calculated, and design realisation pursued to a functional prototype. This prototype, often called the *Engineering Model* will be functionally representative of the finished product but will be fabricated by the most expedient means.

The first phase embraces the majority of the development risk. At its conclusion, the performance and physical characteristics of the finished product will have been determined. At this stage, the development specification is converted into a product specification which contains more detail and more constrictions on the final article. Such things as the physical principle of operation, enclosure material and manufacturing method may be included now.

The production design phase, often referred to as the *B Phase* or *Beta Phase*, then commences. The mechanical components are now designed to suit available production processes. Electronic design activities concentrate on developing the existing circuits into a form appropriate for the desired scale of production followed by selecting production components, and finally laying out the circuit boards. The resulting pre-production prototypes will then be exposed to both rigorous laboratory testing and extensive field trials in the intended operating

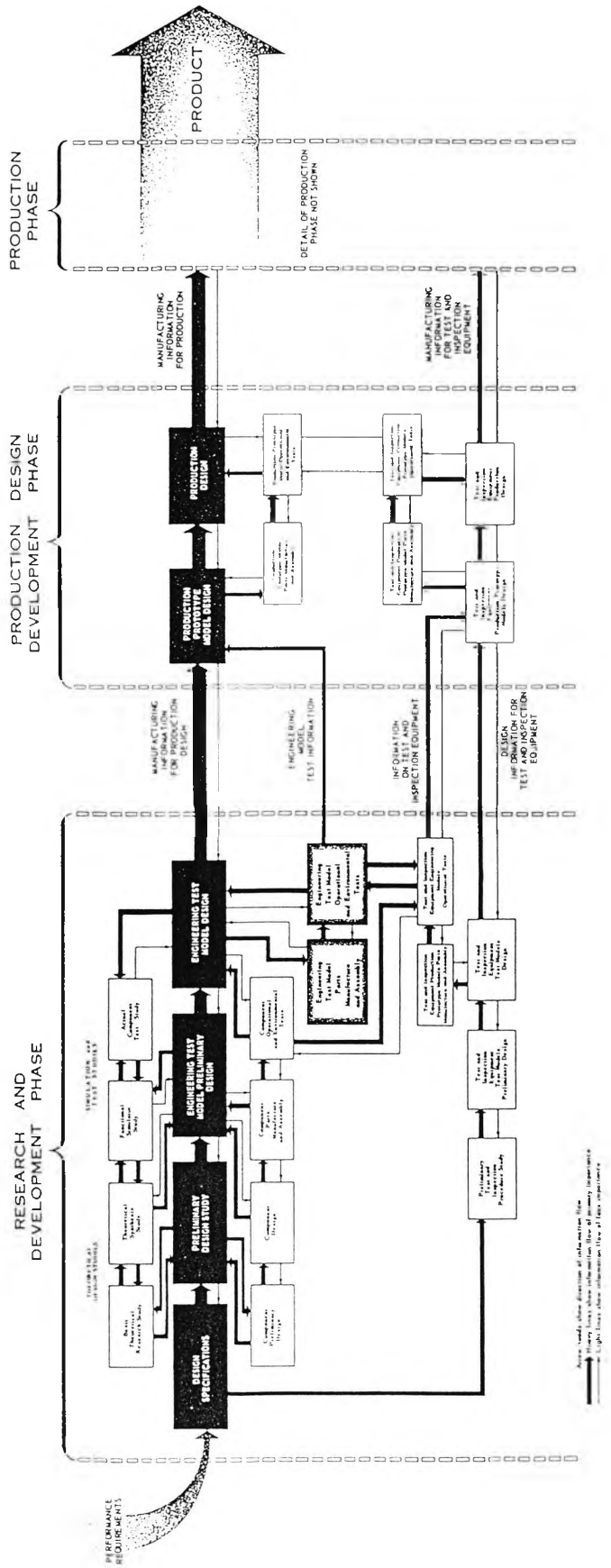


Figure 2-2 Information Flow Diagram for the Development of an Instrument (from Draper, McKay & Lees, 1952)

environment. The output of this phase is the manufacturing or production data pack which will include a complete set of drawings, a product specification which reflects the characteristics to be achieved by the production instruments, all the usual production planning documentation, together with specifications for functional testing, physical inspection, and environmental testing.

2.3.2 Distinction Between the Specifications used in the Development Process

Confusion often exists when one talks about a specification for an instrument. This is not surprising as there are so many different specifications which are used within industry. Each functional group within an organisation usually only sees one type, and there is a natural tendency to refer to it as *the* specification.

The specifications used by production staff are the easiest to distinguish. These documents describe product inspections, functional tests, environmental tests, acceptance tests and the like. Often a single document is issued to cover these topics, although in certain circumstances, in particular in high risk projects, numerous documents can be manifest. Typical names for these documents include Inspection Requirements Specification, Test Requirements Specification, Environmental Test Requirements Specification, EMC Verification Specification. The common feature of all of these documents is that they refer to the verification of the performance or some other characteristic of the instrument which should be acceptable whenever the instrument has been correctly manufactured.

Perhaps the most difficult distinction is the one between development specifications and product specifications. A development specification should contain a minimum of constraints on the creativity of the designer. These constraints should be limited to either system-level requirements which flow down, such as the number and mass of enclosures or a few key features identified by the marketing staff as being crucial to the product's success. The latter category would usually include size, colour, appearance and sometimes work's cost price. Development specifications describe *what* has to be done and should put no restrictions on how it is to be achieved providing the specifications are met. In

contrast, a product function or fabrication specification has a different role. These specifications ensure form, fit and function interchangeability and lower level component interchangeability respectively. There is little flexibility left, and that is the intention.

Another contributing factor to the confusion is the often messy interface between the system engineers or marketing staff and the equipment designers. All too often the requirement generators place unnecessary restrictions on the design process because they wish to decide some aspect of the equipment design. In addition, the system design activity may often include an A Phase for each unit to provide verification of system budget partitions and costs etc. This is often necessary to prepare the submission for funding of a major capital project. Thus it is often the case that when the equipment designer first sees the instrument specification, the design effort remaining is little more than that associated with the production development phase.

Unfortunately the term *Requirements Specification* doesn't have a consistent meaning, although it usually is taken to mean a development specification. All specifications which place requirements on the vendor can be called requirements specifications. The MIL-STD-490A no longer uses the term for hardware items preferring instead development, product function or product fabrication specifications. However, it does retain the term for software and interfaces.

2.4 The Specifications Considered for Computer-Assisted Generation

The ensuing sections illustrate why it is important to make it clear exactly what type of instrument specification is being discussed. This research project is not concerned with all of the possible specifications which may be used in the development of an instrument.

The primary interest of this research is to assist in the preparation of development specifications. These specifications can then be used as input to the instrument design process. Nonetheless, it will be shown in Chapter 3 that the format of

design process. Nonetheless, it will be shown in Chapter 3 that the format of product function specifications differs little from a development specification and that it is possible to generate both types using the same tools.

2.5 Consolidating Industrial Practice with Design Methodology Research

Finkelstein and Finkelstein (1983), showed that the process usually drawn as a linear flow such as in Figure 2-2, can be thought of as a number of iterations through the design process shown in Figure 2-3 reproduced from that paper. Each design process is fuelled by either a needs statement or a requirements specification.

This more general description of the design process is quite consistent with industrial practice and takes into account the flexibility that can occur in the number of stages employed. The first stage is the system design or alternatively, the conversion of the customer needs statement into a development specification. Each level of system partition will usually involve another stage of the design process. Next comes the equipment, in this case the instrument, research and development phase. The production design phase is the final design phase, although this is often divided into two to reflect the two-stage product refinement strategy commonly used in industry.

Within the context of this model, this research project and the resulting software aim to undertake the information gathering and organisation for the cycle which precedes the research and development phase. Capability to produce the more constrained specifications needed before the production design phase will also be considered.

2.6 Conclusion

This chapter has reviewed the varied specifications used in industry and those discussed in the systems engineering and design methodology literature. Broad equivalence between the specification types referred to in these fields has been shown. The type of specification which forms the subject of this research has been identified as a requirements specification suitable for input to the research and development phase of the instrument design process. An industrial perspective has

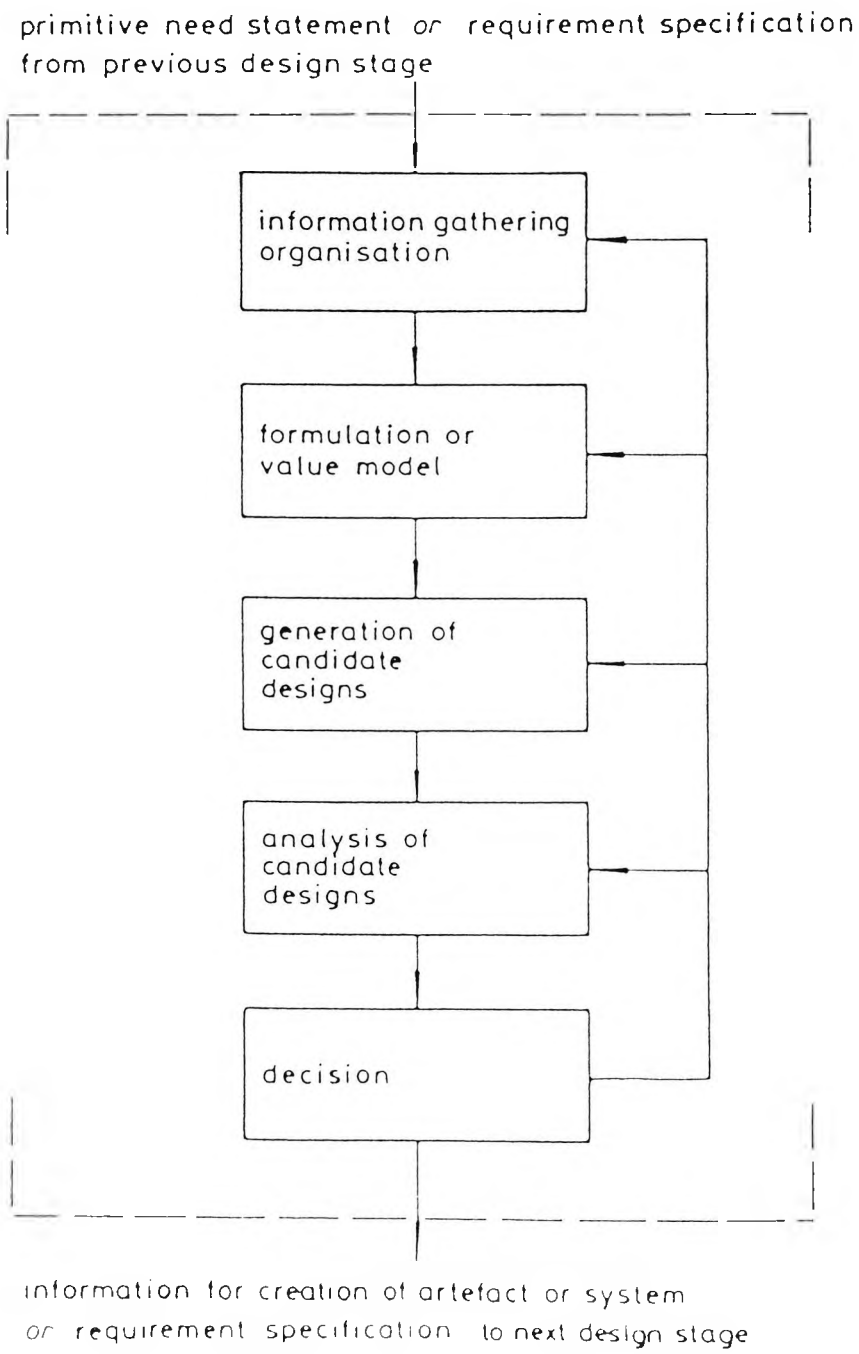


Figure 2-3 Stages of the Design Process (from Finkelstein & Finkelstein, 1983)

been adopted in this chapter because the practices developed to produce measuring instrument specifications have come from that domain. The next chapter examines specification practices and specification generation and proposes a general specification method for measuring instruments which can be automated.

Chapter 3
A Methodology for Automating the Specification Generation Process

3.1 Introduction

In Chapter 2 the various types of specifications used in industry were discussed. It was shown that the type of specification needed for development of an instrument is what is variously known as a requirements specification, development specification, design aim or design specification. These documents are characterised by describing *what* the entity to be developed is to do and the design, implementation, and project management aspects are specifically excluded. This chapter examines the practices used in the production of requirements specifications for measuring instruments, suggests what is to be gained by automating it, and proceeds to describe techniques which could be used for that purpose.

All instruments are part of a larger universe and there are more constraints than just the performance of the instrument characterised, by say, its input to output transfer function. These additional constraints are so numerous that standard requirements specification formats have been developed to aid completeness amongst other things. This chapter commences by examining specification practices and selecting the most appropriate practice, format and type for the general case of measuring instruments. The language and style adopted to write specifications is covered here.

The next section presents the case for automating the specification process against the background material previously presented. It concludes with a list of design aims for a specification generation tool.

This is followed by a section which commences by examining the process used by specification writers to create specifications for measuring instruments. An underlying algorithm is identified which is exploited to form the basis for

automation. The remainder of the section describes a method that can be adopted to generate each portion of the specification. The outcome of this section is a method of producing English text which avoids the complexity of true natural language generation.

The chapter closes by discussing what has been shown and how it can be used to fulfil the goals of the research project.

3.2 An Examination of Specification Practices for Measuring Instruments

3.2.1 Specification Practices

Many items are still designed and produced in informal environments with little or no written description. This practice is only workable if the supplier has an intimate knowledge of the customer's requirements and the application environment of the finished product. It can, say, be both feasible and efficient in a research environment for colleagues to task each other to develop instruments. However, the success of this approach does rely on a vast amount of shared knowledge between the parties involved in the task. It is also limited to situations where money does not change hands or the amounts are insubstantial. This represents one extreme of the continuum of specification practices.

In the more general case, a supplier will be expected to produce an instrument for a customer at fixed price to be delivered on an agreed date. As the stakes increase, the quality of the requirements specification, which is the cornerstone of the contractual documentation, also needs to increase. Hence specifications for scientific instruments to fly on spacecraft, are very detailed as the program budget will usually be in the tens of millions of pounds and there is no second chance to perfect the product. This represents the other extreme.

In recent years, Australia, in line with other countries has been extending the use of formal quality requirements and encouraging their more widespread use in industry to rectify the poor quality control of products. This has culminated in

the phasing in of stringent requirements regarding supplier's quality assurance provisions for all Australian Government purchases (SAA, 1985). For example, organisations responsible for the design and manufacture of the products must possess a quality assurance system conforming to AS 1821 (1985) which is broadly equivalent to ISO 9001 and BS 5750 (1979). An important part of such quality programs is design control which covers contract and specification review, document preparation, design reviews, change control and adherence to company standard procedures (Stebbing, 1979). Thus, the use of comprehensive formatted specifications has expanded from military and avionics contractors to a much wider cross-section of industry.

It would seem there is little value in generating a back-of-the-envelope-type specification as used in an informal environment. Thus the selection of a suitable practice really becomes a question of how detailed and rigorous one wishes to become. In view of the changes occurring in industry and the desire to tackle all the issues, at least briefly, it was decided to examine practices originating from tightly controlled industries.

There are a number of specification formats in use. Government agencies, large companies and industry associations all have their preferred format. The one that is probably the most commonly used is MIL-STD-490 (1968). This format has been described in the literature by Burgess (1969) shortly after the publication of the standard. In comparison, other formats, for example Leech (1972), suffer from a lack of completeness. Since its introduction, adherence to MIL-STD-490 has been mandatory for all United States Military procurements and is also extensively used by U.S. government agencies such as NASA. It is also the basis for the standard used for Australian Defence procurements. After 17 years the standard was revised to become MIL-STD-490A (1985). The newer document is largely unchanged from its predecessor but does embody a few refinements such as where descriptive text should be placed in a specification.

MIL-STD-490A (1985) was selected for use in this research project, because the format described is not only time proven but has recently been revised to reflect

the experience of long and widespread use. It can be argued that the actual format chosen is less important than its completeness because it is not a difficult task to present the information in an alterative form should this be necessary. In regard to completeness, MIL-STD-490A is second to none.

3.2.2 Selecting the Specification Type

Measuring instruments are a sub-set of all the things MIL-STD-490A can be used to specify. Hence the first decision to be made was which of the range of specification types available, shown in Figure 3-1, is the most applicable to requirement specifications for measuring instruments. Guidance for this is available in the military specification MIL-S-83490 (1968).

```
Type A - System Specification
Type B - Development Specification
     Type B1 - Prime Item
     Type B2 - Critical Item
     Type B3 - Non-Complex Item
     Type B4 - Facility or Ship
     Type B5 - Computer Program
Type C - Product Specifications
     Type C1a - Prime Item Function
     Type C1b - Prime Item Fabrication
     Type C2a - Critical Item Function
     Type C2b - Critical Item Fabrication
     Type C3  - Non-Complex Item Fabrication
     Type C4  - Inventory Item
     Type C5  - Computer Program
Type D - Process Specification
Type E - Material Specification
```

Figure 3-1 Military Specification Types (from MIL-STD-490A & MIL-S-83490)

It is important to define the terminology used in Figure 3-1 as the meanings are not uniform across the world and some are peculiar to the U.S. Military. MIL-S-83490 defines a Type A System Specification as one which shall:

> "... state all necessary requirements in terms of performance, including test provisions to assure that all requirements are achieved. Type A specifications shall state the technical and mission requirements of the system as an entity. Specifications shall include requirements for functional areas, interfaces between functional areas, interfaces with other systems, and application of any known specific existing equipment."

Whereas Type B Development Specifications are defined as ones which:

> "... shall state all necessary requirements in terms of performance. Essential physical constraints shall be included. Type B specifications shall state requirements for the development of items, other than systems. Specifications shall specify all of the required item functional characteristics and the tests required to demonstrate achievement of those characteristics."

A Type C Product Specification is used when the item already exists and contains all the necessary detail to ensure that anything supplied to the specification will have form, fit, and function interchangeability. In practice these specifications are modified versions of the development specification containing increased level of physical detail. They are prepared at the end of development phase B2 as described in Chapter 2.

Types C and D refer to considerations in the production phase of the product life cycle and will not be considered further.

Types A or B are the most suitable for the purpose of describing items which may be either developed or purchased. Choosing which one to use is based on what defines a system. An instrument would normally be considered at best to be a sub-system and generally no more than an item, in particular when it is one of

many on a ship, for example. Even in the case where the purpose for the platform is to carry an instrument, for example a spacecraft instrument or a sonar buoy the instrument is considered to be a sub-system to be described by a Type B description. When the instrument is a small desk-top unit, such as, say, a laboratory thermometer, it could then be considered a system. However this problem is resolved by the Intended Use paragraph of MIL-S-83490 which states that:

> "... Type A specifications are generally intended for use only for systems of significant size and complexity."

Thus the use of Type B development specification is indicated for general application to measuring instruments. The next decision is to determine which subgroup is the most appropriate. The last two are clearly not applicable as instruments are not computer programs or facilities. MIL-STD-490A defines non-complex items as simple parts of prime or critical items which can be shown to be suitable for the intended application by inspection or demonstration. Examples given of non-complex items include special tools, work stands, fixtures, dollies and brackets. Often the specification requires little more than a drawing. Thus Type B3 Non-complex item specifications are unsuitable for even the simplest measuring instrument.

The decision between choosing Type B1 and Type B2 is more difficult and can be dependant on the nature of the instrument being specified and how it is to be dealt with by the design authority. A definition of the two terms is the key to understanding the difficulty in making the distinction. A prime item is considered to be a complex item such as an aircraft, missile, radar set, or training equipment. MIL-STD-490A states that a Type B1 specification is indicated if:

> " ... A prime item development specification may be used as the functional baseline for a single configuration item development program or as a part of the allocated baseline where the configuration item covered is part of a larger system development program."

It goes on to support this by stating that such an item will normally be formally accepted, that spare parts will need to be provided for it, manuals produced, and quality conformance inspection will be required for each item as opposed to sampling. In contrast, a critical item is declared to be below the complexity of a prime item but is nonetheless engineering or logistics critical because of its complexity, its effect on system reliability, the need for spare parts or because it has been designated an item for multiple source procurement.

The distinction between the two is largely a matter of the nature of the whole task and the way the configuration control program has been organised. If the overall task is to equip a new facility with instrumentation then each instrument would be a critical item. Configuration control, the discipline of knowing exactly the revision status of each item in a system, would be applied to the system as a whole, and each instrument would appear as a minor element in the overall configuration management plan. On the other hand, if the task is to design, build and commission a major scientific instrument such as a wide aperture radiometer then it is a prime item if not a system. Examination of Appendix II and Appendix III of MIL-STD-490A which detail the contents of prime item development specifications and critical item development specifications respectively, shows that the major difference is in the amount of detail that needs to be included in the requirements section, in particular interface definition and characteristics. Thus a prime item specification can be considered to be a superset of the contents of a critical item specification.

When considering the automation of the specification generation process, using more than one format is an unnecessary complexity. A single format must be chosen that is sufficiently general to cover a large range of instrument development situations. There is little value in attempting to automate the production of specifications for really large systems because they tend to have unusual requirements and because by very nature and expense, these projects tend to challenge established techniques. Hence generation of system specifications will not be considered further. Prime item development specifications cover all the items presented in critical item specifications, so this type was seen as the

logical choice, and the Type B1 format was selected. Excessive detail can be handled, as necessary to suit the project, using the techniques presented later in the chapter.

3.2.3 A Review of Specification Language and Style

It has already been stated that specifications have the dual role of conveying essential technical requirements from a customer to a supplier and acting as a management tool for the conduct of the work. In their latter capacity a specification will often form the basis for a legally binding contract. In the same way that the legal profession has found it necessary to adopt a definite style of language, specifications tend to be written in a particular style and certain words and phrases have taken on specific meanings.

3.2.3.1 General Points

There is a large number of references that offer guidance on what should appear in a specification, the pitfalls to avoid, and some advice on which practices should not be used in specification generation (Mead et. al., 1956; Hill, 1970; Leech 1972; Dunham et al., 1979; Sydenham, 1986). These points can be summed up by the realisation that specifications are documents which fill a particular role and must be written with that thought firmly entrenched.

It is commonly pointed out that a specification is not a novel but a series of requirement statements which must be written in clear and simple language, and, as such, everything should be sacrificed for clarity. It is also advised that specifications are not a treatise and should be as brief as possible, consistent with completeness and exactness. A point which receives considerable attention is that the requirements should be definite and unambiguous. Indeterminate specifications especially those which seek to exercise arbitrary control are warned about in Mead et. al. (1956). This is more a problem in civil engineering where terms such as "as the engineer or architect shall direct" are used which leave the entire matter open to interpretation. Further problems to

be avoided include overspecification, which simply increases the cost to no advantage, and unnecessary restrictions which stifle creativity (Voelcker 1988).

Specifications should not include management information such as statements of work, schedules, warranty provisions, payment details etc. These should be covered in the general provisions of the contract.

All this advice is correct and valuable but does little to help the fledgling specification writer learn *how* to write a specification or describe the style to be used. This information is not readily available as it is considered proprietary by commercial organisations. Once again examination of military methods provides some assistance.

3.2.3.2 The Military Specification Style

The United States of America Department of Defense mandates not only the format of specifications in MIL-STD-490A (1985), but also constrains the use of language and writing style. The standard directs that short sentences with a minimum of punctuation, consistent with correct interpretation, are to be used and long sentences containing compound clauses should be rewritten in the required format. Consistency in terminology is also required hence the common practice of using more or less synonymous words for the sake of euphony is not only discouraged as it is by Mead et. al. (1956), but prohibited. These points are valuable when considering the automation of specification generation. The language style adopted by the US military is much easier to generate than free form prose.

The next most important point is the meaning attributed to certain words by MIL-STD-490A (1985). The word *shall* is required whenever a specification expresses a provision that is binding. The infinitive form of the verb or the word *must* as used to convey the same meaning in commercial specifications, (Leech, 1972) are not acceptable. The words *should* and *may* are to be used whenever it is necessary to express non-mandatory provisions. DSMC (1983)

expands on this by stating that the use of *should* or *preferred* means that the use of an alternative must be justified whereas *may* indicates that the contractor's selection will be acceptable. The use of *will* is limited to cases where simple futurity is to be indicated. For example, the statement "The instrument shall survive its operating environment." means that it is to be designed and manufactured in such a way that it will survive its operating environment. Whereas if "shall" were replaced by "will", the statement would be interpreted as a fact that would be of no concern to the designers, i.e., the very nature of the instrument would be such as to permit it to survive the environment. Hence, use of the word "will" is constrained to statements of the type: "The Government will provide the following equipment to support the development ... ". Use of the present tense is taken as expressing a fact and does not form part of the specification. The standard requires that the emphatic form of the verb shall be used throughout the specification, for example: "The indicator shall be designed to measure ...". The exception is in the test provisions section where the imperative form is permissable when preceded by the text: "the following tests shall be performed". The example given for the emphatic form is:

> "The indicator shall be turned to zero and 230 volts alternating current applied."

Which may expressed in imperative form as:

> "Turn the indicator to zero and apply 230 volts alternating current."

MIL-STD-490A (1985) also denotes certain phrases be used in certain situations. For example, when citing a reference document one of the following is mandated "conforming to ..." , "as specified in ...", or "in accordance with ...". When referencing a requirement in the specification which is rather obvious or not difficult to locate "as specified herein" is to be used. "Unless otherwise specified" placed at the start of a sentence or paragraph is used to indicate an alternative course of action but only where it

is possible to clarify the meaning by providing a reference. This will usually be the contract or a higher level specification. When it necessary to use a proprietary name because it is the only adequate method of describing an item, the words "or equal" are used to permit competition.

Positive limitations are described so that the number stated is included in what is acceptable. This effectively amounts to specifying measurable quantities as greater than or equal to or less than or equal to the stated limit. The example given is:

"The diameter shall be no greater than ... ".

"Not applicable" indicates that the specifier has thought about the subject of the paragraph and has deliberately chosen not to state a requirement. (Irrelevant or inapplicable paragraphs are not removed from the specification but labelled "not applicable".) This treatment conveys the intention of the writer, retains the structure of the specification, and obviates correspondence on the topic from concerned contractors.

3.2.3.3 Other Standard Techniques

Specification writers employ a number of techniques to enable them to release draft documents before all the information they need is to hand. Perhaps the most useful one is *TBS*. This is an abbreviation for "to be specified" and is used when it is known that the parameter will be specified before the design is complete. Initial issues of specifications often have TBS against parameters which are derived from higher level system specifications, for example, quality provisions or operating and storage environment. Another related term is "to be determined" abbreviated to TBD. This is used when values are not yet available usually because the system design is not complete. While releasing a specification to tender containing TBD's and TBS's is undesirable, it is a common practice as the system designers often need the feedback contained in the responses to tender to know what can be achieved by which organisation

and make the final system-level trades-off accordingly. From a contractor's viewpoint, more information is conveyed using these forms than if the uncompleted paragraph was simply omitted to be made a contract amendment at a later date.

The most common technique for specifying requirements, approaches, procedures, or testing to be used in the development and production process, is to reference an appropriate specification, standard, or handbook. This practice provides new programs with the benefit of previous experience, promotes common testing techniques, minimises logistics costs and simplifies the specification process (DSMC, 1983). Providing the potential contractor is familiar with the reference document and the consequences of the paragraph referenced, it can be quite efficient. In addition, well established industrial standards are used, were applicable, by the courts in the United Kingdom as yardsticks with which to interpret acts of Parliament (Lucas, 1981). This gives them a firm legal standing. On the negative side, there is a tendency to become entrapped in a chain of referenced standards. Each one that is referenced can reference further ones until it sometimes seems that the entire standards library is required before what should be a simple paragraph, can be interpreted. As there are many thousands of military and civil standards this is a major problem. In a large system this can be compounded by the fact that standards are referenced at all levels of the specification tree and hence at the uppermost level, vast numbers of documents can be incorporated by cross referencing at successively lower levels. Some consolation is offered by DSMC (1983) which observes that while the number of references could theoretically reach hundreds of thousands, in actuality, multiple referencing tends to make the number level out rather than increase exponentially as the number of specification levels increases. Nonetheless, the resultant number can still be very large.

Another serious problem is that either or both of the parties may not truly understand what has been called up, especially when there is a large number of references cited. Meeting certain quality, environmental testing, and in

particular electromagnetic compatibility (EMC) requirements, can be extremely costly. An example of this, from personal experience, is EMC testing. The supplier thought that conforming to the EMC requirement in the cited standard, MIL-STD-461, would be straightforward and easy to verify. Unfamiliarity with the implications of the standard caused massive cost overruns to meet the stringent specification. It became necessary to use nested enclosures around the circuitry and very expensive filtered connectors and high frequency absorbing external cables. However, these design and production costs were swamped by the verification costs which necessitated the hiring of an instrumented EMC test chamber which could measure emissions between 0 and 40 GHz and subject the item to fields over the same range up to 400 V/m. Several months facility hire at £5000 a day certainly imparted a lesson never forgotten.

The general rule is to write out simple statements rather than reference a standard when at all possible. The standard can then be designated a "guideline" document. This means that the standard may contain useful information and is probably worth having, but does not have to be explicitly conformed to. Statements such as "in general conformance with" a particular document, are often used here. The uninitiated might interpret this as a requirement to meet the standard with only minor exceptions. In practice, it means that the customer cannot afford to invoke that standard and wants a mechanism to try and achieve the best for the money available. Knowledgeable contractors will always state their agreement to comply to requirements stated in this way, because it shows willing, and because they realise there is no firm requirement to meet. Such techniques are frequently used to separate out contractors who are familiar with the industry from those who are not.

3.3 The Case for Automating the Specification Generation Process

The arguments which support the automation of the production of measuring instrument specification can be subdivided into five largely self-contained sections:

(a) Importance.
(b) Technology Transfer.
(c) Efficiency.
(d) Adequacy.
(e) Overspecification.

3.3.1 The Importance of Specification Generation

Specifications form the basis for contracts between customers and suppliers. Each cannot exist without the other so it is mutually beneficial for work to be conducted according to an agreed specification and be completed to time and budget. At tendering time, bidders learn from the specifications not only information about the nature of the work but also they form some idea as to competence and fairness of the parties who prepared the documents and the treatment they can expect to receive during the progress of the work (Mead et. al., 1956). If the specification is not what would be expected for the type of work offered then suppliers will be reluctant to tender. Signs suppliers look for are indiscriminate referencing of standards, poor format and use of language, ambiguity, and lack of clarity, consistency, and reasonableness. If a bid is made at all, the potential supplier will add contingency costs to cover the uncertainties in the specification.

Just as suppliers employ senior staff to prepare responses for requests for tender, customers need to employ knowledgeable, experienced staff to compile the specifications used. Specification writing, therefore, must be seen as an important, labour-intensive, and expensive process which is crucial to the long-term success of an organisation.

3.3.2 Technology Transfer

Organisations which design and manufacture expensive and often complex instruments for discerning customers, are very familiar with the production, interpretation and configuration control of requirements specifications. They are also familiar with the commonly cited standards used in their industry and other knowledge needed to form good working relationships with their customers.

Young expanding companies, in contrast, frequently lack this knowledge and are often even bereft of standard procedures. There is a clear case for technology transfer to assist such companies and this is usually provided by consultants and selective recruitment. The increasing importance of formal quality systems, as mentioned earlier, has resulted in a shortage of staff skilled in this area. It is not easy to expand the number of specification writers quickly because a wide basis of industrial experience is needed to write a specification which is adequate, complete and consistent and at the same time incorporates cost effective verification procedures and takes advantage of cost versus utility functions for the many parameters to be specified. Mead et.al. (1956) devotes the first part of Chapter 16 to the knowledge needed to be able to write a good specification and concludes that:

> "Such knowledge is acquired only by extended study, observation, and experience, and is not usually possessed by the young engineer."

For example, the underlying physical principal that can be employed to measure temperature say, is dependant on the measuring range, accuracy required and so on. Each method will have a typical cost associated with its use, and while it is not the specification writer's job to select which to use, it is important not to inadvertently exclude one, particularly if it the least expensive, because a single parameter was arbitrarily, too tightly specified.

Another example is quality assurance. The more rigorous the quality system demanded by the specification, the more expensive the instrument will be. Thus

to specify quality assurance provisions appropriately, it is necessary to derive an expected cost and have some knowledge of the potential suppliers' standard operating practices and approach to quality management.

There is a definite need to provide some of this expertise in a form readily available to assist inexperienced specification writers.

3.3.3 Efficiency Limitations of the Manual Method

A fundamental limitation of the manual method is that it takes considerable effort to prepare a comprehensive specification, in the region of days to months. Specification writers are highly skilled and experienced people who acquire their skill over many years. Novice writers will take much longer to prepare specifications, even if they have a solid industrial background, because of unfamiliarity with the formats and lack of specification preparation knowledge, including usage of language, standards to cite, industry practice outside their experience and associated bounds of reasonableness.

3.3.4 Adequacy Limitations of the Manual Method

As has been already stated, it is very difficult to write a good specification, that is, one which is complete, consistent, unambiguous and not overly restrictive. Specifications are written in natural language, and it is an unfortunate fact that any such document cannot avoid being open to interpretation. This means they never stand in isolation without human interaction between the customer and the supplier. It is possible, however, to reduce dissention, in particular near the end of a development, if all the requirements are recorded. This is not an easy thing to achieve as not all requirements may be known at the time of writing. This is where the specification writer and systems engineer use their experience and knowledge to provide sensible defaults. It can be said that detecting a lack of completeness is a specialist task in itself, see Section 3.4.

3.3.5 Overspecification

There is a tendency to overspecify equipment. This is particularly true when supply is to go to open tender as a somewhat tighter specification is seen as a method to improve the probability of receiving what is actually required (Wheeldon, 1974).

Overspecification is hard to avoid especially in larger instruments which are extensively partitioned. At each specification level, tolerances are tightened in an effort to ensure the overall system performance can be met (Sydenham, 1986). Overspecification very often leads to unnecessary expense but is commonly overlooked unless it threatens the viability of the project. In the extreme, a reasonable system requirement can be translated into such a stringent equipment requirement three or four levels down the specification tree, that the system becomes infeasible.

3.3.6 Design Aims for a Specification Generation Tool

The arguments presented to support the need for automating specification generation can be converted into initial design aims for a computer-aided engineering tool.

Experienced writers and novices have been identified as potential users. This requirement places demands on the human interface and the logic behind the assistance tendered. Frequent users require little help and often wish to avoid tedious human interface procedures which can be of great benefit to novice or infrequent users. In addition, there is no doubt that an experienced writer would possess knowledge in some areas beyond that available in any conceivable program. Thus facilities must exist for the program to be overridden when desired.

Given that producing a specification usually takes many days, there is an obvious desire to speed the process.

Consistency and lack of ambiguity would appear to be relatively straightforward to achieve; this is not always so as specifications for large systems can run into many hundreds of pages. There is no effective way for humans to ensure consistency in such large documents; the best that can be done is to have a number of cognisant people review the document. Thus some form of computerised completeness, consistency and reasonableness checking would be highly desirable.

There is a need for a mechanism to check that numerical values generated in the final specification can be traced back to the original user requirement to control overspecification. This would be the ultimate aim of a system-level reasonableness checking facility. At the equipment level, where this research project is focused, assistance would probably be limited to providing sensible defaults and flagging unreasonably expensive requirements.

3.4 A Basis for the Automation of Specification Generation

3.4.1 The Human Specification Generation Process

When planning to automate an intellectual task, it is often fruitful to study the human method of performing the task to determine if there is a systematic basis which can be formulated as an algorithm. From personal experience, there is indeed a specification generation algorithm which is often followed. This is illustrated in Figure 3-2, specifically for measuring instruments.

The algorithm starts when the writer selects the desired document format. This choice will depend on the a range of factors such as whether the instrument is to be developed, the level of detail indicated by complexity and cost of the instrument, the expected relationship between customer and supplier. In practice this is usually pre-determined by company standard procedures or customer requirements.

The next two tasks can be conducted in parallel. The most straightforward, albeit

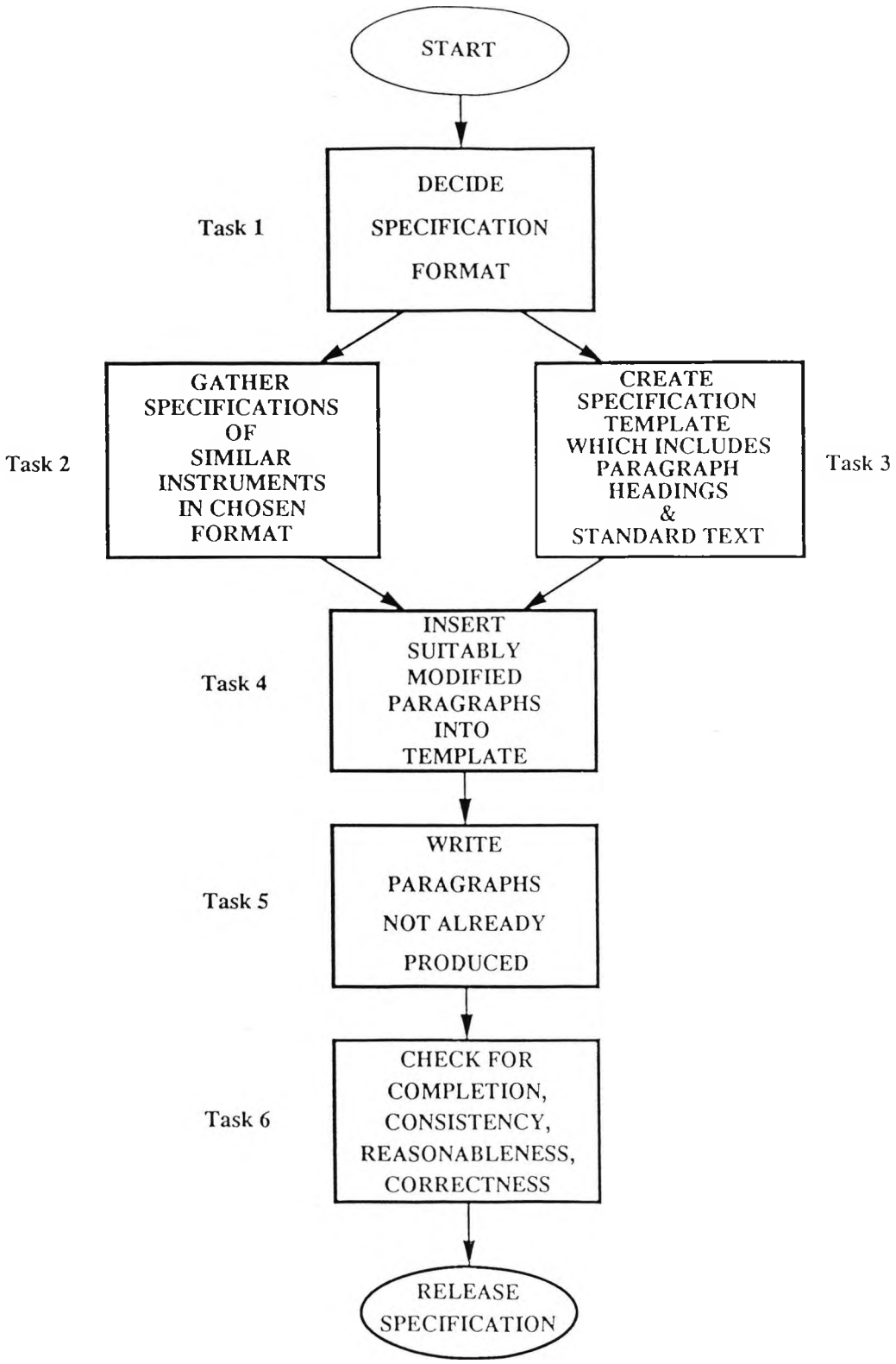


Figure 3-2 The Human Specification Generation Algorithm

time consuming, is to collect a representative sample of recent specifications for similar instruments. Greatest benefit can be extracted from specifications for equipment destined for the same industry and operating environment and written to conform to the selected format. Larger companies are at an advantage here because sufficient material can often be uncovered by a search through the files.

The parallel task is to create a skeleton of the document from the format description including paragraph headings, and the standard paragraphs which invariably appear, such as disclaimers, description of citation practice for applicable documents, etc. In the case of an often used format, this skeleton, or template, is likely to be incorporated into company standard procedures and may well be available as a word processing file.

The next task requires a level of intellectual input. In essence it is a "cut and paste" operation. For each paragraph of the template, the most appropriate paragraph from the reference material is selected, modified to suit the instrument under consideration, and inserted into the new document. Whenever a particular paragraph cannot be completed this way, it is simply left blank.

The next process involves greater intellectual input. Each of the paragraphs not yet dealt with has to be written. This needs to be done in a style consistent with that of the paragraphs already grafted into the document. Matching the style may not be all that difficult because specifications tend to be written in stereotyped language as described earlier.

The last process is the most demanding. Here the draft document is reviewed for completeness, consistency, reasonableness and conformance against the contractual requirements for the project and the company standard procedures. This is usually a multi-level task conducted by the Project Engineer, Engineering Manager, Quality Manager and finally the individual with the authority to release the document.

In organisations which regularly produce specifications, this algorithm is efficiently

streamlined and detailed specifications of complex items can be produced in around two weeks.

3.4.2 A Computer Specification Generation Process

3.4.2.1 The Process

The process described above can be used as the basis for a computerised realisation. The first task, deciding on the specification format to use, is not required as it has already been decided that a Type B1 development specification, defined by MIL-STD-490A (1985), is well suited to the task.

The task of creating a specification template can be replaced by the provision of a single fixed template applicable to a broad class of measuring instruments. Thus the concept of making the template available as a word processing file has been extended into incorporating it into a purpose built software package. Note no processing is needed.

The purpose of acquiring specifications, Task 2 of Figure 3-2, is to provide knowledge on the methods that can be used to specify each paragraph. The ultimate realisation of this task would be to access a database containing a large number of specifications, extract the most suitable documents for the specification task in hand, and then from these, compile a list of specification methods. This is not feasible at present because the reference specifications are not readily available to the public in printed form let alone electronic form; their owners regard them as proprietary material. In addition, building a natural language processor capable of performing this task using techniques available at the start of this project (Winston, 1984), would have been a vast task.

An obvious alternative is to perform a general survey of methods for specifying measuring instruments and store this information within the computer. Thus this task can also be replaced by stored knowledge.

The next two tasks can be combined and become alternatives. Completion of the paragraphs can be achieved from either the stored methods where adequate, or by the user entering a paragraph.

The checking function is performed last after the draft specification is completed in the same way that a human manager will generally only review a complete document. Interactive checking could be accommodated in the entry process.

Figure 3-3 illustrates the complete process. An additional task has to be added to convert the internal representation used by the computer into a printed document.

3.4.2.2 Restrictions on Generality

A tool to specify all classes of measuring instruments was thought to be too bold a task and beyond what could be achieved. Hence it is decided that some limits to generality would be necessary to make the project tractable. These are:

(a) Instruments are to be limited to one measurand;
(b) This measurand is to be one dimensional.

Thus a proportion of instruments, for example optical imagers which measure intensity in two dimensions, had to be excluded. These could be added later but there would need to be a template created for each number of measuring dimensions.

3.4.2.3 A Specification Template for Measuring Instruments

The generation of a list of paragraph headings is the first step in the process of creating a template. The headings selected are shown in Figure 3-4 and follow the numbering scheme described in MIL-STD-490A (1985). The list of

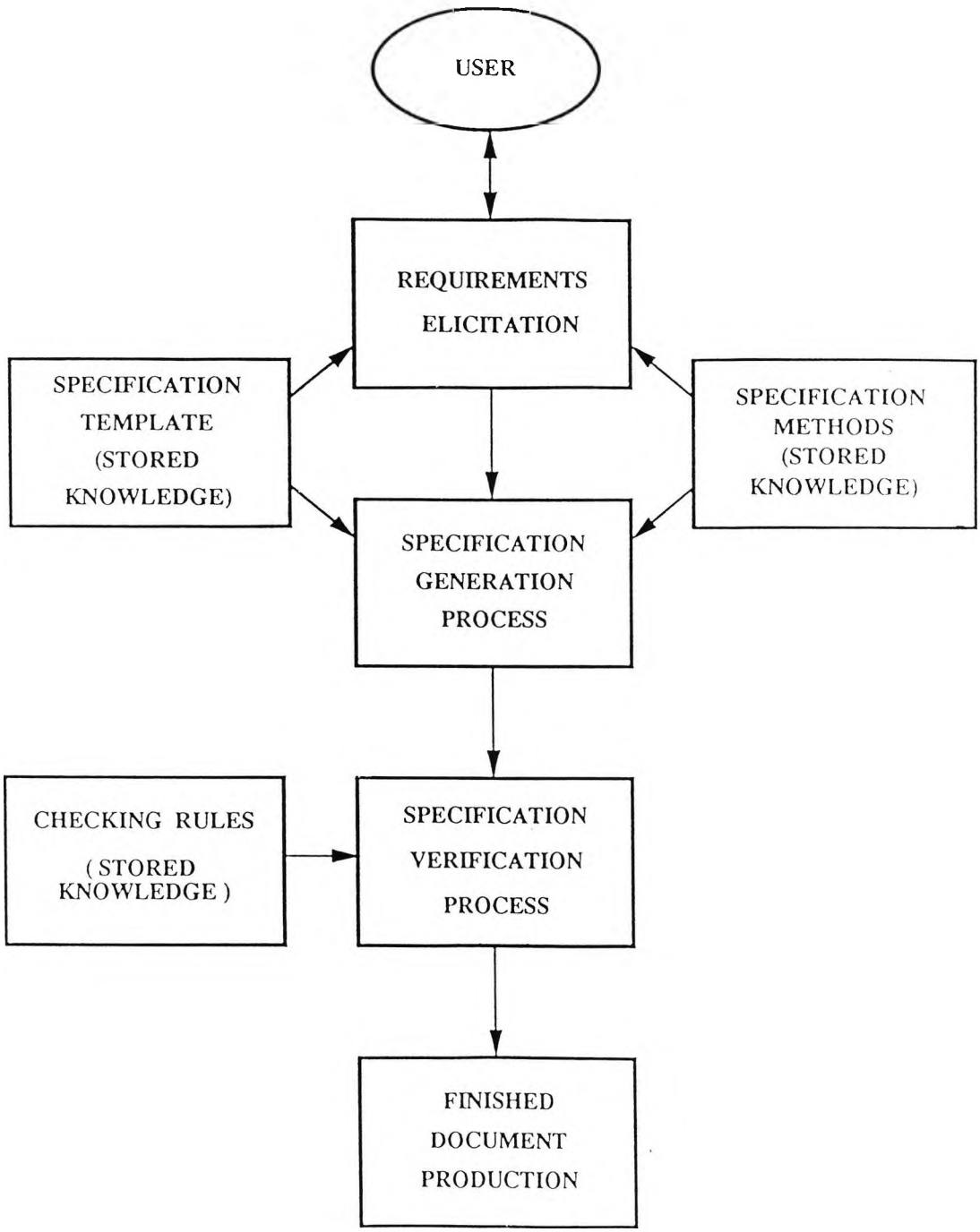


Figure 3-3 A Computer-Aided Specification Generation Process

performance characteristics and their definitions was taken from AS 1514 (1980). Further detail on the derivation of this template is given in Section 3.5. This list can be widely applied to instruments which have one single-dimensional measurand.

For simple instruments, it may be thought that the detail is excessive but this can be countered by examining the amount of useful information presented that might otherwise have been omitted. It is often lack of attention to this sort of detail which leads to confrontation between suppliers and customers. The most valid complaint about the preparation of comprehensive specifications is the time and effort required. Computer assistance could well render that argument invalid.

For major expense instruments, the specification structure dictated by the paragraph headings is still valid and in fact has been commonly used for such tasks for many years (Auspace, 1984; ESA, 1983; AEL, 1987) Each of the paragraphs tend to be much more comprehensive, in particular requirements not related directly to performance such as reliability, maintainability, workmanship standards etc.

Completing the template is the task of the specification generation process. This is covered in the next section.

3.5 Specification Methods

Experience of writing and reviewing instrument specifications reveals that there is an underlying finite set of methods used to specify each paragraph of a natural language specification. In many instances there may be a limited number of options available and simply identifying a specific option is sufficient specification. Alternatively, descriptive text can often be generated by inserting numeric values or phrases into standard sentences. Appropriate use of this technique can cover a wide range of eventualities and exceptions can be handled by having the user write the whole paragraph.

By employing these techniques to advantage, it is possible to prepare a limited set of general paragraphs for most of the template headings and instantiate the appropriate paragraphs with suitable parameters for the instrument under consideration. These general paragraphs, often referred to in the computer industry as "potted text", (possibly a metaphor alluding to potted electronic assemblies which are very hard to alter) can incorporate the special language and techniques of the specification writer described earlier.

The remainder of this section is devoted to outlining the methods selected for specifying the paragraphs identified in Figure 3-4. In the example output text, italics is used to refer to text stored in the computer while the text enclosed in "<>" denotes non-terminal symbols. These symbols represent strings which would be entered by the user or a string which would be generated by the program in response to the selection of an option. The text generation process needs to insert these strings into the stored text and make grammatical adjustments such as selecting between *a* and *an* and decisions on capitalisation.

3.5.1 Scope

The example from MIL-STD-490A can be employed directly to form the paragraph. All that needs to be inserted is the name of the instrument.

> *This specification establishes the performance, design, development, and test requirements for a* <instrument name>.

At first sight it might appear that this paragraph is only applicable to development specifications. This is not the case as the word *design* appears in the product function specification example in Appendix VII of MIL-STD-490A.

3.5.2 Applicable Documents

All the documents referenced in the text of the specification must appear here. It is preferable to state the exact version number which for contractual reasons may

1 SCOPE
2 APPLICABLE DOCUMENTS
3 REQUIREMENTS
  3.1 Instrument Definition
    3.1.1 General Description
    3.1.2 Interface Definition
      3.1.2.1 Electrical Interface
        3.1.2.1.1 Power Interface
        3.1.2.1.2 Communications Interface
        3.1.2.1.3 Electromagnetic Compatibility
      3.1.2.2 Mechanical Interface
        3.1.2.2.1 Sensor
        3.1.2.2.2 Data Processor
        3.1.2.2.3 Display
      3.1.2.3 Thermal Interface
        3.1.2.3.1 Sensor
        3.1.2.3.2 Data Processor
        3.1.2.3.3 Display
  3.2 Instrument Characteristics
    3.2.1 Performance Characteristics
      3.2.1.1 Measuring Range
      3.2.1.2 Discrimination
      3.2.1.3 Repeatability
      3.2.1.4 Hysteresis
      3.2.1.5 Drift
      3.2.1.6 Dynamic Response
      3.2.1.7 Measuring Error
      3.2.1.8 Power Consumption
    3.2.2 Physical Characteristics
      3.2.2.1 Enclosures
        3.2.2.1.1 Sensor
        3.2.2.1.2 Data Processor
        3.2.2.1.3 Display
      3.2.2.2 Mass
        3.2.2.2.1 Sensor
        3.2.2.2.2 Data Processor
        3.2.2.2.3 Display
    3.2.3 Reliability
    3.2.4 Maintainability
    3.2.5 Environmental Conditions
  3.3 Design and Construction
4. QUALITY ASSURANCE
5. PREPARATION FOR DELIVERY
6. NOTES

Figure 3-4 Paragraph Headings for Requirement Specifications for Measuring Instruments

not be the current one. If no version is given then the current version is assumed but this is poor practice as it makes the requirements subject to changes in the reference documents. MIL-STD-490A states that only documents specifically referenced should be listed but in practice it can be useful to include guideline documents. The following statement can be used to cover this instance and the extent of reference. This would be immediately followed by the applicable document list.

The documents listed hereunder form a part of this specification to the extent invoked by specific reference in other paragraphs of this specification. If a specification is listed but not referenced in any specific paragraph, then the specification is applicable as a design guideline.

<applicable document list>

3.5.3 Requirements

The requirements section must include the essential requirements and descriptions that apply to performance, design, reliability, interfaces etc. In fact this section comprises all the information most people think of as a specification. The requirements section should be so written that compliance with all the requirements will assure the suitability of the instrument for its intended purpose. The corollary is that non-compliance with any requirement will indicate unsuitability for the intended purpose. Development specifications can include design goals in addition to minimum requirements, but each must be identified clearly to avoid confusion. Only essential design constraints should be included as requirements, for example, restrictions on the use of certain materials due to toxicity or dimensional or functional restrictions to assure compatibility with associated equipments. A short sentence is all that is necessary under the section heading before each of the requirements is spelt out in detail.

The measuring instrument described by all the requirements of this section shall pass the examinations, analyses and tests specified in Section 4.

3.5.4 General Description

This paragraph is used to define the item in terms of major physical parts and functions. Block diagrams, schematic diagrams and pertinent operational, organisational and logistic considerations and concepts can be included here. In the interests of generality, it was decided to restrict the definition to the physical partition of the instrument followed by a statement of what the instrument is intended to measure and its place in any system hierarchy. The provision of a block diagram was not considered justified. The description here is determined by the physical characteristics option selected in paragraph 3.2.2 of the specification and covered in paragraph 3.5.11 of this thesis. Hence one of five paragraphs is selected as appropriate. The one for separate housings is shown.

> *The* <instrument name> *comprises three modules, namely, the* <Measurand> *Sensor, the Data Processor and a Display. Division of functions between the three enclosures including compensation for temperature and influence effects is unrestrained. The purpose of the instrument is to measure* <measurand> *in a(n)* <environment type> *environment. The <instrument name> forms a part of* <larger system>.

The general description paragraph does not contain any requirements, rather it is information for the reader, hence the word *shall* is not used.

3.5.5 Interface Definition

This paragraph covers the functional and physical interfaces between the item being specified, in this case a measuring instrument, and other external items. The internal interface between the units which make up the instrument is specifically excluded as these should be determined by the supplier or designer. The sub-section heading simply carries an introduction while the detail is contained in later sub-paragraphs which cover electrical, mechanical and thermal interfacing aspects.

The <instrument name> *will interface to the system being measured, power supplies, equipment mountings and the operator.*

3.5.6 Electrical Interface

The electrical interface was sub-divided into three areas: power, communications and electromagnetic compatibility. This paragraph can get quite complicated for some items which require multiple power supplies with possibly different frequencies. The trend over the last decade or so has been to include a suitable power supplies in the instrument rather than rely on the availability of external supplies. This simplifies the power interface and its specification. In particularly sensitive instruments, the AC to DC convertor may well be housed in a separate enclosure but this does not prevent it from being considered part of the instrument. The chosen specification method is to opt for a single supply and require that all internal supplies be generated from this. After which a specification for the power plug is given. If the power supply is AC, for example the mains or aircraft power, the frequency can be included as part of the power supply specification i.e., "50Hz British power mains conforming to BS ....", or "400Hz aircraft power conforming to MIL-STD-748A".

The performance of the <instrument name> *shall be met when connected to a(n)* <power supply specification>. *Connection to the power source shall be achieved by a flexible power lead. Plugs conforming to* <power plug specification> *shall be fitted to these leads. The peak current drawn by the instrument shall not exceed* <peak current>.

If the supply is not well specified, for example a battery that is subject to intermittent charge, then voltage limits can be included in the power supply specification, for example "12 V lead acid battery with terminal voltage in the range 11 to 14.5 V".

There is some difficulty deciding where the power consumption and hence current

drawn from the power supply should be specified. The interface is what the instrument has to be designed or purchased to work with and that includes electrical and mechanical constraints. Thus supply voltage, noise and fluctuations correctly belong to the interface. The current flow is determined by the instrument and is constrained by a performance characteristic. The view taken is that it is the responsibility of the specifier to ensure that the power source has adequate average current output to provide for the specified power consumption, but it is worth specifying a limit for the peak current.

Many modern instruments can be configured to form part of an information gathering network by connecting them to a computer or other information processing equipment. Common interface standards employed are the serial RS 232-C and the parallel instrumentation bus standard IEEE-488, (the Hewlett Packard Interface Bus HP-IB). The only effective way to specify the communication interface is to reference the required standard. It is useful to distinguish between the raw sensor data and the processed data which is that normally displayed.

> *Processed and corrected measurement data shall be available to external equipment via a communication port conforming to* <communications interface standard>.

ElectroMagnetic Compatibility (EMC) is a discipline of growing importance. As the electromagnetic spectrum becomes increasing heavily used, shielding requirements become greater. For this reason, commercial electrical equipment now has to meet relevant national limits for conducted and radiated emissions. The second aspect to EMC is susceptibility. Many instruments are susceptible to electromagnetic radiation emanating from automotive ignitions, airport radars and nearby broadcast transitions. This is a particularly difficult area to specify well and is a specialist field in its own right. The two prominent standards are MIL-STD-461 "Electromagnetic Compatibility Requirements" and MIL-STD-462 "EMC Verification Methods". The first deals with the amplitude levels versus frequency, while the second deals with how this can be verified. The frequency responses

and amplitude levels are often tailored to meet project requirements. The general rule is that there should be at least 6dB margin between expected field strengths at a particular frequency and the field strength which causes the equipment to malfunction. Thus if a geophysical instrument say, is intended to be located near a broadcast transmitter, there would be a requirement for the instrument to operate in the expected field without malfunction. EMC requirements are especially important for integrated systems where individual units from a variety of vendors have to work in close proximity, for example on an aircraft or even in a computer room.

When wishing to specify EMC, two problems arise: the first is what the emission and susceptibility levels should be used for each piece of equipment; and secondly how these are to be verified. Complete verification requires the use of extensive and expensive facilities. EMC is another area which is strongly related to the amount of money at risk. It is not feasible to place anything but the most general requirement on a low price commercial instrument. Statements such as "The instrument shall not be affected by electrical plant noise." are of minimal value if an instrument is to be developed but may enable the customer to return an off-the-shelf unit. At the other end of the financial spectrum it would have major safety and economic consequences if a purpose-designed avionics instrument was found not to be compatible with the aircraft into which it was intended to be fitted. Consequently all avionic and space flight equipment undergoes rigorous EMC testing.

The general approach adopted was to cite susceptibility and emission standards which might be national standards, a system EMC standard, or the previously mentioned United States military EMC standards.

> *The* <instrument name> *shall meet the emission requirements of* <emission standard & limits> *and the susceptibility requirements of* <susceptibility standard & limits>.

3.5.8 Mechanical Interface

The mechanical interface specification is dependant on the chosen physical partitioning of the instrument. Obviously the number of enclosures affects the number of interfaces which need to be specified. The example shown here is for three separate enclosures.

The Sensor shall be <sensor mounting method>.

And similarly for the other two housings.

The Data Processor shall be <data processor mounting method>.

The Display shall be <display mounting method>.

3.5.9 Thermal Interface

The thermal interface deals with heat removal and consequently, temperature control. In most applications, temperature control of the instrument will be achieved by limiting the temperature rise of critical components by suitable heat sinking to the environment. Specification of thermal interface can be achieved by selection of one the following from a list:

(a) Conduction to the mounting face
(b) Natural convection
(c) Forced convection
(d) Radiation
(e) Cooling not specified

Convection is the usual cooling method for all classes of applications except for military and space instruments which tend to use conduction to temperature controlled cold plates. Natural convection is the preferred method whenever feasible because of the freedom from fan noise and the avoidance of concerns

about fan failure. Forced convection would be specified when the power consumption is high, when the maximum operating temperature is high, or when the operating temperature of the unit is to be kept low for reliability or other reasons.

Radiation is only a significant heat loss mechanism in vacuum and even then is generally limited to lower power units with a large radiating area. The option not to specify a cooling mechanism for an enclosure is intended to be used whenever the power consumption is very low, or zero.

There can be more to specifying the thermal interface than the heat transfer method. The heat transfer surfaces may need to be specified with a drawing or perhaps reference to a manufacturer's part number for a heat sink. These instances can be handled by permitting a text string to be entered as for other attributes.

Standard option selections can be interpreted to give:

The Display is to be conductively cooled via the mounting faces described in paragraph 3.1.2.2.2.

The Data Processor is to be convectively cooled.

The Sensor has negligible power dissipation and no thermal interface is specified.

3.5.10 Performance Characteristics

The structure and contents of this sub-section are highly dependant on the item being specified. The only guidance given in MIL-STD-490A Appendix II is that this sub-section is to state what the item, in this case the measuring instrument, shall do, including both upper and lower performance limits.

Section 5 of AS 1514 (1980) lists the meterological properties of measuring instruments and defines each term. This list which applies only to one-dimensional, single measurand instruments, was used as the basis for a performance requirements list. The introduction to this section of the specification is:

> *The* <instrument name> *shall meet the performance requirements specified herein when operated in the environment specified in Section 3.2.5.*

Strictly speaking this introduction is redundant as it is inherent in the fabric of the specification, however, the reason it has been included is to alert the supplier to the need for the instrument to work over the complete range of environmental conditions encountered later in the document. This is particularly important when it is intended for the instrument to operate in harsh or unusual environments, for example, exposed outdoor locations, low Earth orbit, or underwater.

3.5.10.1 Measuring Range

The detailed paragraphs commence with measuring range.

> *The instrument shall be able to measure* <measurand> *from* <lower range limit> *to* <upper range limit> *with at least the performance described by the remaining paragraphs of Section 3.2.1.*

Once again the final clause is redundant but serves to remind the supplier that the remaining performance specifications need to be met across the entire measuring range.

3.5.10.2 Static Performance

Static performance is characterised by parameters such as discrimination, repeatability, hysteresis, drift, and total error. These terms are well defined (AS 1514, 1980) and well known and hence were selected for inclusion into

the general method of specifying static performance. These parameters do not exhaustively deal with the topic, however. Complex performance limitations such as non-linearity (Sydenham, 1983a), and the effects of digital signal processing such as quantisation error, rounding errors and filter coefficient quantisation (Proakis & Manolakis, 1988) could also need to be specified. There is no general way of dealing with these and other specialised performance characteristics, so the approach selected was to offer a user entered special requirements paragraph.

There are two ways given in AS 1514 to apply bounds to the properties of measuring instruments:

(a) As absolute values in the measuring units, and;
(b) Percentage of indicated value.

Another method occasionally encountered is to specify uncertainty in terms of percentage of measuring range.

A further consideration is that measurement is a stochastic process (Doebelin, 1983; Sydenham, 1983a). In a measuring situation where noise of known statistics is encountered, it could be useful to specify the measurement mean, variance and perhaps other statistical parameters.

All of these specification methods can be found in specifications and it was decided to allow the specifier to use whichever one was felt to be appropriate. This desire is achieved by including the necessary text into the specification string, for example, "3% of measuring range" or "13 Pascals". The static performance parameters are dealt with thus:

The <instrument name> *shall have a discrimination of* <discrimination> *or better.*

Any two measurements performed on any given identical value of

<measurand> *within the range specified in para 3.2.1.1 shall indicate values within* <repeatability>.

Hysteresis effects owing to alteration of the direction of the <measurand> *change, including any dead band, shall be less than* <hysteresis>.

Drift resulting from aging effects shall be less than <drift> *over* <drift period>.

The error of measurement including systematic error, random error, reading error, repeatability error and hysteresis error shall be less than <total error>.

<special static performance characteristics.>

The method of specifying drift needs some explanation. The term *drift* is used in a general sense to describe changes in performance of a measuring instrument due to changes in influences quantities such as temperature, humidity etc. These are dealt with by specifying that the performance shall be within the stated bounds over the range of environmental conditions specified in 3.2.5. AS 1514 (1980) specifies drift as:

"That property of a measuring instrument as a result of which its metrological properties change with time, under defined conditions of use."

The definition of drift used here refers to the change in mean indicated value with time. This most closely matches the term "stability" perhaps, but the term "drift" was retained because its use is more widespread.

3.5.10.3 Dynamic Performance

Dynamic response can be specified in a number of ways, the following were pursued:

(a) Mathematical

(b) Empirical.

(c) User entered paragraph.

The mathematical specification technique was based on the technique of Sydenham (1983b). This treatment concentrates on systems of zeroth, first, and second order based on the observation that higher order systems tend to display characteristics which can be described by one of these in the frequency area of interest. It is stated that this assumption has been found adequate in practice. A zeroth order instrument indicates that the indicated value follows the input signal time features faithfully. A perfect zero order system is a mathematical abstraction as infinite frequency response is implied, however, this frequently desirable response can be approximated if the maximum rate of change of the input signal is very slow compared to the dynamic performance of the instrument. In practice it is not feasible to specify a zeroth order system because it is impossible to implement one. If instead a specification such as "zeroth order response up to 100 Hz" were used, the conundrum of determining what constitutes a response which is no longer zeroth order is presented. This could take the form of maximum phase shift or amplitude attenuation but it is probably better to describe the response in terms of a first order system of sufficiently short time constant to give the desired characteristics. In addition, from a practical point of view, as noise power is proportional to bandwidth, it is always useful to limit the bandwidth of a measuring system. For these reasons, the option of specifying zeroth order response was later withdrawn from the mathematical description option list. The option of specifying zeroth order response with suitable qualification is still available via a user entered paragraph. First order response is characterised by a time constant while second order response requires both natural frequency and damping ratio.

The <instrument name> *shall exhibit first order response. The time constant shall be* <time constant>.

The <instrument name> *shall exhibit second order response. The natural frequency shall be* <natural frequency> *and the damping ratio shall be* <damping>.

These specification methods rely on the specification strings containing the limits. It was decided not to restrict the user to say just a maximum time constant because the time constant may have been introduced to curtail noise or deliberately restrict dynamic performance. The values for second order systems would normally be bounded but to cater for all possibilities, limits for these too were left for the user to explicitly specify.

The "mathematical" method is really a frequency domain specification paradigm. Anything that has to be specified must be verifiable (MIL-STD-490A, 1985). In many cases it is impractical to verify the performance of an instrument by sinusoidally varying the measurand over a range of frequencies. The alternative method, is to specify time domain dynamic performance to a step input. The "empirical" specification technique is so named because it based on the parameters that are conveniently measured and effect the practical application of the instrument.

When subjected to a step input of <step size>*% of measuring range, the* <instrument name> *shall have a 10 to 90% rise time of* <rise time> *and shall settle to* <settling limit> *within* <settling time>. *The -3dB frequency response cutoff shall be* <frequency response> *and the ripple in the pass band shall be less than* <ripple>.

Once again the limits have to be included in the specification string. This specification technique has the advantage of easier verification and is often preferred because it uses terms more familiar to testing personnel. Another benefit is that it can be used successfully regardless of the response order. However, care is needed to ensure that the time and frequency domain specifications are consistent. If desired, one or other of the methods could be left unspecified by setting the appropriate specification string to "unspecified"

or "not applicable".

Systems that cannot be conveniently specified with the previous two methods, such as those which cannot be subjected to step or sinusoidal inputs, can be dealt with by a user controlled paragraph. This option allows the user total freedom of expression, but as with all such paragraphs the possibility of automated consistency checking is lost.

3.5.10.4 Power Consumption

The approach taken with power consumption was just to specify an upper limit as this is adequate in most circumstances. An exception that could arise would be the need for a minimum heat output to be maintained for thermal control. Another could be a more comprehensive description of the load on an AC supply. In the latter such items as power factor, crest factor and peak inrush current might need to be specified. This paragraph, however, deals with performance while most of the items mentioned relate more to an interface requirement and as such could be dealt with there by reference to a power subsystem specification or a standard.

The power consumption of the <instrument name> *shall not be greater than* <maximum power consumption>.

3.5.11 Physical Characteristics

The area of physical characteristics is where most diversity of specification occurs. For a development specification, the physical characteristics of a measuring instrument should include the following as applicable (MIL-STD-490A, 1985):

(a) Physical configuration, including dimensional and volume limitations.
(b) Mass limit of the instrument.

(c) Requirements for transport and storage, such as tiedowns, pallets, packaging and containers.
(d) Durability factors to indicate degree of ruggedness.
(e) Health and safety criteria including adverse effects on operators.

The prime physical characteristics of an instrument are associated with its number and type of housings. It is not necessary to specify these if their choice is unimportant, but usually there will be some constraint on the size, mass and configuration derived from the intended installation. For example, many companies prefer to rack mount all displays and processing equipment and will always write this into a development specification as a requirement.

An instrument must have three basic functions: a sensor, a data processor or signal conditioner, and a human or machine readable display. These often form a natural physical partition as well, although one, or indeed all, functions may be combined. The sensible combinations of the functions, shown below, were used as the starting point to automate the process of describing physical characteristics:

(a) Single enclosure.
(b) Separate sensor, data processor and display units.
(c) Separate sensor, combined data processor and display.
(d) Combined sensor and data processor, separate display.
(e) User specified physical arrangement.

The option chosen affects the interface definition and the mass distribution. When an instrument is partitioned into more than three physical units, as indicated by Sydenham (1983) for the process control industry, then the user specified paragraph must be used. The user will need to indicate the number of enclosures, their names, and their physical characteristics. The latter can be dealt with as below.

The mass of each enclosure is specified rather than a total mass as this was thought to be the most useful. For each enclosure, one of three options can be

used to specify the housing:

(a) Rack mount.
(b) Standard off-the-shelf enclosure.
(c) User paragraph.

Most rack mount equipment conforms to IEC 914, or the national adoption of this ubiquitous 19 inch rack mounting standard. Rack width and mounting hole positions for front panels are specified in the standard. Rack height is always specified in terms of rack units of 1.875 inches. Thus a 1U rack occupies 1.875 inches of rack height while, say 6U occupies 11.25 inches. Rack depth does not appear to be well modularised and rack boxes are made in a range of depths to suit the equipment to be housed. Thus to guarantee physical interchangeability it is necessary to specify the number of rack units and the maximum rack depth.

The specification of three separate enclosures is shown; the other physical partitioning variants are derived from this. A different specification method is used for each to illustrate the variety available even without having to resort to the user paragraph.

The <instrument name> *Sensor shall be housed in an enclosure conforming to* <sensor housing specification>.

The mass of the <instrument name> *Sensor shall be less than* <sensor mass>.

The <instrument name> *Data Processor shall be housed in a* <number of rack units for data processor>*U rack-mounted enclosure conforming to IEC 914. The maximum depth of the enclosure shall be* <maximum data processor rack depth>.

The mass of the <instrument name> *Data Processor shall be less than* <data

processor mass>.

The <instrument name> *Display shall be housed in an enclosure whose maximum dimensions are not to exceed* <display dimensions>.

The mass of the <instrument name> *Display shall be less than* <display mass>.

3.5.12 Reliability

Reliability is the probability that an item will perform its required function, under stated conditions, for a stated period of time (ISO 8402, 1986; BS 4778, 1987; MIL-STD-721B). This can be paraphrased as "the probability of non-failure in a given period" (Smith, 1985). High risk, high reliability items, in particular space-based instruments and military systems are indeed specified this way. This option takes the form:

The probability of the <instrument name> *surviving for* <survival period> *shall be greater than* <probability of survival>. *The probability of survival (Ps) shall be determined using* <basis for reliability calculations>.

A major advantage of this specification method is that it is straightforward to calculate system reliability for the survival period (O'Conner, 1985) without needing to make assumptions about the failure rate statistics. The reliability calculation basis refers to both the calculation technique and failure rates used. The foremost handbook used is MIL-HDBK-217E (1987) although others are given in Smith (1985). This handbook describes two methods; parts count and stress analysis. The former is intended for quick appraisals of potential reliability and sees most use in system design trade-offs. The stress analysis method uses complex algebraic formulas to calculate the probability of survival for each component, mechanical or electrical, right down to solder joints. Such factors as operating temperature, device procurement specification, stress factors, and device maturity, feature in every calculation. Calculations of systems reliability are

usually performed using computer models which allow ready alteration of any of the thousands of components which comprise the system. Unless otherwise stated, the worst case environmental and electrical conditions are used for the reliability analysis which will always give rise to a pessimistic result. Hence it is important when specifying the basis for reliability calculations that the method and the relevant average environmental conditions are stated. For example: "MIL-STD-217E Parts Stress Analysis assuming an ambient temperature of 20 degrees C and maximum power supply voltage".

For other types of instruments, reliability requirements are seldom stated in probability terms (Smith, 1985). Parameters such as Mean Time Between Failure (MTBF) or failure rate are usually specified instead.

> *The mean time between failure (MTBF) of the* <instrument name> *shall be greater than* <MTBF> *when operated in the environment specified in paragraph 3.2.5.1. The MTBF shall be calculated using* <basis for reliability calculations>.

This method of specification is often used for purchased components such as power supplies and computers. In this case, the basis for reliability calculations could well be observed reliability data from the manufacturer. If the instrument is to be incorporated into a larger system, it is however, problematical to convert MTBF into a probability for inclusion into system reliability models. Probability of survival can be determined from any of the established failure rate probability density functions (Amstadter, 1971). However, in converting MTBF to Ps, it is necessary to make some assumption about the failure rate statistics. The constant failure rate negative exponential distribution is usually selected which represents the Useful Life region of the well-known "bathtub curve". If the observed MTBF data includes infant mortality and wearout failures then the calculated Ps will be incorrect and significantly pessimistic. If redundancy is employed in the system, small errors of this type can be magnified to the extent where years are removed from the projected system life expectancy. Conversely, if non-constant failure rate conditions are to be modelled, in particular wear-out, then once again deriving Ps

from MTBF is misleading at best.

The last reliability specification option is simply to state that reliability has not been specified. In instances where an instrument is expected to comprise few components, the nominal MTBF will be so long, typically, hundreds of years, that any models used to derive the value will be invalid.

3.5.13 Maintainability

Maintainability can be defined in similar terms to reliability: the probability that a failed item will be restored to operational effectiveness within a given period of time when the repair action is performed in accordance with prescribed procedures (Goldman & Slattery, 1964; Smith, 1985). MIL-STD-490A (1985) takes a somewhat broader view as shown by the extracts below:

"a. Time (e.g., mean and maximum downtime, reaction time, turnaround time, mean and maximum time to repair, mean time between maintenance actions).

b. Rate (e.g., maintenance manhours per flying hour, maintenance manhours per specific maintenance action, operational ready rate, maintenance hours per operating hours, frequency of preventative maintenance).

c. Maintenance complexity (e.g., number of people and skill levels, variety of support equipment)."

It goes on to state that maintainability is to be specified quantitatively and that the requirements shall apply to maintenance activities conducted in the planned maintenance and support environment. Maintenance philosophy, per se, does not belong in a requirements specification but it does influence where the maintenance actions will be conducted, and the level of skill needed in the maintenance staff. Once the maintenance philosophy is determined, it is then possible to determine the level of detail required. A list of options was prepared which covers a wide

range of eventualities. These are listed in rough order of maintainability priority:

(1) Full maintenance support - on-site repair personnel
(2) Full maintenance support - off-site repair personnel
(3) Assembly replacement by owner.
(4) Despatch to service organisation for repair.
(5) Throw-away instrument - need not be designed to be repaired
(6) Application environment prohibits maintenance.
(7) Unusual - enter a paragraph.

The first two are traditionally favoured by owners of large, expensive, but failure prone systems whose availability is essential to the well being of the organisation. Examples include computer systems, industrial plant instrumentation, and many military systems. The term full maintenance support means having qualified and experienced repair personnel on standby, in case of failure, in addition to resources dedicated to routine maintenance. Option one implies that down time must be minimal and that a comprehensive inventory of spares be held on-site.

The military concept of maintainability given above, is very strongly biased towards complex systems, such as a combat aircraft, which have a system failure rate in the order of a few hours and as such are continually under repair, calibration or other maintenance activity. Maintenance in an industrial or laboratory setting, in contrast, usually takes the form of routine preventative maintenance, re-calibration at lengthy intervals, and hopefully infrequent repairs. The rate of maintenance method is commonly applied to complex systems, such as combat aircraft which have low availability and very high maintenance requirements. In general, the routine servicing of instruments would be required no more than a few times per year. Hence the most suitable specification method is to specify the routine maintenance interval, maximum down time for routine maintenance, and mean time to repair.

To ensure performance is within specification, all instruments need to be calibrated at intervals specified either by the manufacturer or by statutes. Formal

calibration against traceable standards comprises a complete functional verification of the performance of the instrument (Sydenham, 1982) and as such is convenient to combine with any other routine maintenance. Hence the calibration interval can become the minimum routine maintenance interval. These points are combined to create the text for option one:

> *The* <instrument name> *shall be designed to permit on-site routine maintenance, calibration and repair by trained servicing personnel. The instrument should be designed so that routine maintenance should not need to be conducted at more frequent intervals than the calibration period. The routine maintenance interval (calibration period) shall not be less than* <calibration period> and *down time shall be limited to* <routine maintenance down time>. *Mean time to repair shall not be greater than* <mean time to repair>.

Comprehensive specification of maintainability is a complex task and Smith (1985) devotes a chapter of his book to this topic. The general method above covers most situations applicable to measuring instruments and uses well known terms. Extensions to this will be specific to the instrument being specified and are best handled with a user entered paragraph.

Option two implies that the delay incurred in repair personnel arriving can be tolerated which may flow down to reduced spares holding as short delays obtaining infrequently used parts from a central depot may also be acceptable. What can be inferred is the skill level of the maintenance staff. The breakdown maintenance staff, in particular, would usually be experienced technicians backed up by an engineer. The text is virtually unchanged from option one:

> *The* <instrument name> *shall be designed to permit on-site routine maintenance, calibration and repair by trained servicing personnel. The instrument should be designed so that routine maintenance should not need to be conducted at more frequent intervals than the calibration period. The routine maintenance interval (calibration period) shall not be less than*

<calibration period> and *down time shall be limited to* <routine maintenance down time>. *Mean time to repair shall not be greater than* <mean time to repair> *from the arrival of servicing personnel.*

Note that the response time of the maintenance contractors is a contractual issue and forms no part of the specification. Once again a user entered paragraph is needed for special requirements.

The third option imposes definite design requirements. The "user" here is assumed to be the instrument operator who would normally be expected to possess only modest technical skills but would nonetheless be capable of following fault isolation and rectification procedures after suitable training. The use of modular design techniques combined with elaborate self test incorporating automatic fault location is indicated. User replaceable modules would need to be readily available.

The <instrument name> *shall be designed to be repaired by the user replacing one or more modules on-site. The instrument should incorporate self test and fault indication to facilitate this. The User's Manual shall reflect this philosophy by including fault rectification procedures. The instrument should be designed so that routine maintenance should not need to be conducted at more frequent intervals than the calibration period. The routine maintenance interval (calibration period) shall not be less than* <calibration period> *and the mean time to repair shall not be greater than* <mean time to repair>.

It was decided not to specify MTTR for this option because it is highly dependant on the ability of the user and their familiarity with the fault finding procedures in the user's guide. Specific cases can be dealt with in a user entered paragraph.

The fourth option, is to send the instrument to the manufacturer, or their representative, for repair in the event of failure. It is the least expense but repairs can take months; not an attractive proposition for owners who cannot afford to lose the utility of the instrument. It is expected that repair staff would be experienced technicians with the back up of the engineering staff.

In the event that the <instrument name> *requires maintenance, it is expected that the instrument is to be despatched to the manufacturer's service organisation for servicing. The instrument should be designed so that routine maintenance should not need to be conducted at more frequent intervals than the calibration period. The routine maintenance interval (calibration period) shall not be less than* <calibration period>.

The next alternative is often forgotten but probably the commonest for inexpensive items. Calibration is still required and hence a calibration interval must still be specified. Any other maintenance is limited to replacement of the instrument with a new one. Hence a complete instrument would need to be stocked as a maintenance item, should short down time be a system objective.

The <instrument name> *does not need to be designed to be repaired. The calibration period shall not be less than* <calibration period>. *In the event of malfunction, it is intended that the instrument be replaced with a new one.*

The next choice is another method of specifying "not applicable" but has the advantage of reinforcing that repair is not possible after deployment. This would be the case for most spacecraft and performance monitors in undersea cables.

Once installed in its intended operating environment, the <instrument name> *is not accessible for maintenance. There is no maintainability requirement.*

The last option is the catch all user entered paragraph.

<Maintenance philosophy paragraph.>

3.5.14 Environmental Conditions

The environments that the instrument can expected to experience in shipment, storage and service are specified in this paragraph. A distinction needs to be made between environments which the instrument has to survive without impairment and the operating environment where the performance characteristics must be met. This was achieved by separating the environmental characteristics into two categories; operating and non-operating which covers transport and storage. This division is introduced by the following statement and followed by details of the operating environment:

> *The environment the instrument is to withstand is divided into operating and non-operating environments. The* <instrument name> *shall meet the performance characteristics specified in section 3.2.1 when operated within any of the combinations of environmental conditions defined in the operating environment. It is not necessary for the* <instrument name> *to be able to meet its specified performance whilst being subjected to non-operating environment extremes that exceed those for the operating environment.*

The last sentence covers the eventuality of an instrument which is being operated whilst being subjected to environmental stress beyond its operational limit due to say, transportation.

Temperature is generally the most important influence parameter which affects an instrument's performance and this appears first. The temperature range refers to ambient air temperature so if the instrument is to operate in sunlight where direct heating of the enclosure occurs, this should be specified in the special environment paragraph.

> *The ambient air operating temperature range over which the* <instrument name> *shall be able to operate shall be* <minimum operating temperature> *to* <maximum operating temperature>.

Humidity is dealt with next. If the upper limit approaches 100% it should be stated in the upper limit specification string whether it is to operate in the presence of condensation or not because this will make a significant impact on the mechanical design of the housings and perhaps necessitate a sealed enclosure.

The humidity range over which the <instrument name> *is to operate shall be from* <minimum operating humidity> *to* <maximum operating humidity>.

The vibration specification follows. The rather simplistic method given is to specify frequency limits and a maximum vibration amplitude, however, this would be sufficient for many non-demanding applications. The most obvious alternative method would be to give an amplitude versus frequency graph which allows the vibration specification to be tailored for particular measured environments such as space craft launch, rail transport, factory plant operating vibration etc. This could be cited using the user entered paragraph.

The <instrument name> *shall meet the performance requirements of section 3.2.1 when subjected to vibration over the range from* <lower operating vibration frequency> *to* <upper operating vibration frequency limit> *of amplitudes up to* <maximum operating vibration amplitude>.

The ambient pressure range is important for two reasons. Firstly, ambient pressure can be an influence variable for many instruments. Secondly, natural convection is the commonest cooling method for instruments. If the instrument is to operate in an unpressurised aircraft, in space or even on a mountain, overheating may occur if this factor has been ignored in the design.

The atmospheric pressure range over which the <instrument name> *is to operate over shall be from* <lower ambient pressure range limit> *to* <upper ambient pressure range limit>.

In addition, a user entered paragraph is available for special environmental characteristics not covered by the standard paragraphs.

The storage time statement then follows:

> *The instrument shall survive the storage environment for* <storage time> *without deterioration. If the length of time in storage exceeds the calibration interval the instrument shall be calibrated before use.*

The storage and transportation environment specification method is a repeat of that for the operating environment and need not be shown. Although an instrument need not operate in storage and transportation environments which are often more severe than the operating environment, it is a requirement that they will function within specification once the environment returns within the operating bounds. This requires that not only is the instrument undamaged by the harsher environment but it remains within calibration and that the calibration interval is not affected. This can be more difficult to achieve than it first seems, especially if the transportation environment is particularly harsh, for example space craft launch, or that encountered during tactical manoeuvres. It is not recommended that the storage and transportation environment be marked as not applicable even for low cost instruments. Most electronic equipment is air freighted. Inadequately designed or packaged equipment frequently suffers damage from large or heavy components breaking off circuit cards and insufficiently restrained assemblies becoming loose often as a result of screws falling out during transit. For this reason, the default transportation environment is airfreight.

3.5.15 Design and Construction

This section covers minimum or essential requirements not covered by performance characteristics, interface requirements, or reference documents. These can include design standards which have general applicability and are pertinent to the class of equipment into which the instrument falls, for example, process control, avionics etc. Requirements governing the use or selection of materials, parts and processes, interchangeability requirements, safety requirements and the like are also included in this section. It is intended that this section should

reference established standards to the maximum extent possible.

This section is very product and industry specific. The best way this can be dealt with is to have a standard section for each type generic application. Automation would then comprise of assisting the choice of the most appropriate section. This is a major task and so the selected method is to present a general default paragraph which can be modified by the user as required.

<potted design and construction paragraph>

This paragraph was deliberately poorly specified to ensure generality. Help information could be made available on suitable standards for a range of situations. This topic, which is primarily relevant to product specifications and not development specifications, is an area for further research.

3.5.16 Quality Assurance Provisions

This section covers the formal tests and verifications which the instrument must pass before it will be accepted by the customer. MIL-STD-490A requires that there be subparagraphs covering reliability testing, engineering evaluation to define extent of test program, qualification testing, installation testing and finally a formal test verification of performance characteristics to demonstrate that prime item requirements in Section 3 of the specification have been satisfied.

Quality assurance programs cost money (Smith, 1985; Stebbing, 1987) and if a rigorous quality assurance program is specified the instrument will cost many times more than one delivered with minimal performance verification. It is essential to match the quality assurance provisions to the risk involved with the delivery of a non-conforming instrument and to the type of supplier. For example, if the instrument is low cost and likely to be supplied by an organisation that caters for the domestic market, the lowest level of quality assurance is indicated. In that case the instrument would most probably be supplied off the shelf. If the instrument did not function correctly or did not meet its advertised

performance it could be returned for replacement, repair or refund. This minimum standard is usually referred to as "good commercial practice". Acceptance testing is usually limited to batch testing, with perhaps individual visual inspection or a perfunctory functional test.

Large organisations, such as government agencies and major companies, generally specify more rigorous quality requirements consistent with their procurement and quality control procedures. Furthermore, if the instrument is to form part of a larger project, then serious financial and timescale consequences can arise if the instrument is not fit for its intended purpose for whatever reason. It is too great a risk to permit the chance of the supplier unilaterally making a small change from the agreed specification which might render the instrument unusable. In these cases 100% functional testing will be called for and the customer will usually insist on a certificate of compliance stating that the specification has been met. It is usual for the customer to require that their representatives have right of access during testing.

The ultimate level of quality control is that employed in the space industry with military following only a little behind. For the purposes of producing general statements these can be grouped together. These industries usually require that each item is 100% functionally tested in a range of operating environments and that each stage of the process is vetted by the customer's representatives.

Quality assurance specification is by selection of one of the three options described above or a blank user paragraph:

(1) Good commercial practice.
(2) 100% testing of performance & certificate of compliance.
(3) Rigorous compliance against each paragraph of specification.
(4) User paragraph.

These stored paragraphs for the first two are quite short:

The <instrument name> *shall be built to a good commercial standard and be able to meet the performance characteristics specified in Section 3.2.1 in the environmental conditions specified herein.*

The <instrument name> *shall be built to a good commercial standard and be able to meet its performance characteristics in the environmental conditions specified herein. The performance criteria specified in Section 3.2.1 shall be 100% tested. Certificates of performance and calibration, signed by the supplier's Quality representative shall be despatched with the instrument. The calibration of the instruments used to test the* <instrument name> *shall be traceable to national standards.*

The text for option three is considerably longer and the paragraph headings are included for clarity.

4.1 General

4.1.1 Responsibility for Tests

The vendor shall be responsible for the performing the tests, analysis or inspections specified in Figure 4.1 the Verification Matrix. Any non-compliancies encountered during the examinations or tests shall cause rejection of the instrument.

4.1.2 Test Reports and Certificates

A test report shall be prepared following the testing of the instrument. A copy of this report shall be available on request. A calibration certificate shall be included with the instrument when despatched.

4.2 Quality Conformance Verification

The requirements for and the methods used to verify that the design and

performance requirements of section 3 will be satisfied, are tabulated in the Verification Matrix, Figure 4.1.

4.2.1 Test Sample

Each instrument delivered shall be inspected and acceptance tested.

4.2.2 Test Sequence

Acceptance inspection and tests shall be consist of the examinations and tests in the following sequence.

(1) Physical Examination
(2) Functional Test
(3) Environmental Tests
(4) Post Environmental Functional Test

4.2.2.1 Physical Examination

Examination of the instrument shall be performed prior to functional testing.

4.2.2.2 Functional Testing

Functional tests shall be performed prior to, during and where appropriate, following environmental testing.

4.3 Verification Methods

4.3.1 Inspection

Inspection shall consist of physical examination of the product, engineering drawings and/or other documentation to determine

conformance to the requirements. The <instrument name> *shall be inspected to determine conformance to the physical characteristics specified in paragraph 3.2.2 and the applicable drawings and specifications.*

4.3.2 Analysis

The analysis method shall consist of review of analytical data resulting from analyses performed by generally recognised techniques for requirements that cannot readily be demonstrated through conventional testing techniques. Computer simulation, is the preferred method, where appropriate.

4.3.3 Testing

4.3.3.1 Functional Tests

Verification that the <instrument name> *performs as specified herein shall be achieved by performing functional tests specified in Figure 4.1.*

4.3.3.2 Environmental Tests

Testing of performance shall be performed at both extremes of the range of each environmental characteristic. It shall not be necessary to combine tests of more than one environmental characteristic in a single test, i.e. high temperature and low pressure. Instrument performance in such situations shall be verified by analysis as indicated in Figure 4.1.

4.4 Test Procedures

All tests shall be performed in accordance with approved, released

procedures.

4.5 Rejection and Retest

If a failure occurs during a test, testing shall be discontinued until an analysis is performed to determine whether the condition warrants continuation of the tests or discontinuation of the tests for more detailed failure analysis. The test procedure shall be repeated until completed successfully. If corrective action substantially affects the significance of results of previously completed tests, such tests shall be repeated also.

Figure 3-5 is an example verification matrix typical of that referenced as Figure 4-1 in the text above. Verification methods are based on either inspection, analysis or testing. Selection of the method used has to be considered carefully. Functional requirements are usually verified by testing unless the cost of the necessary environmental or measurement facilities renders this impractical. Inspection is used for verifying physical parameters. Analysis is used on all other requirements which cannot reasonably be measured, for example, reliability, maintainability, and certain functional characteristics at more than one operating environmental extreme. An example of the latter might be physical dimensions at low temperature, vibration and vacuum. The matrix of compliance lists the selected verification method for each paragraph of Section 3. This matrix would be based on the default matrix of Figure 3-5 edited as required by the user.

Verification is a large topic and much more could be added on methods and standards used. Fortunately, method 3 tends to be applied mainly to instruments which are intended to become components of larger systems. In which case the verification methods will not be duplicated in each instrument specification but will reside in the system specification, or most likely, the system quality assurance requirements document.

PARAGRAPH	VERIFICATION METHOD
3.1.2.1.1 Power Interface	A
3.1.2.1.2 Communications Interface	I
3.1.2.1.3 Electromagnetic Compatibility	T
3.1.2.2.1 Sensor Mechanical Interface	I
3.1.2.2.2 Data Processor Mechanical Interface	I
3.1.2.2.3 Display Mechanical Interface	I
3.1.2.3.1 Sensor Thermal Interface	A
3.1.2.3.2 Data Processor Thermal Interface	A
3.1.2.3.3 Display Thermal Interface	A
3.2.1.1 Measuring Range	T
3.2.1.2 Discrimination	T
3.2.1.3 Repeatability	T
3.2.1.4 Hysteresis	T
3.2.1.5 Drift	A
3.2.1.6 Dynamic Response	T
3.2.1.7 Measuring Error	A
3.2.1.8 Power Consumption	T
3.2.2.1.1 Sensor Enclosure	I
3.2.2.1.2 Data Processor Enclosure	I
3.2.2.1.3 Display Enclosure	I
3.2.2.2.1 Sensor Mass	T
3.2.2.2.2 Data Processor Mass	T
3.2.2.2.3 Display Mass	T
3.2.3 Reliability	A
3.2.4 Maintainability	A
3.2.5 Environmental Conditions	A,T
3.3 Design and Construction	I
5. PREPARATION FOR DELIVERY	A,I

Legend: A = Analysis
I = Inspection
T = Testing

Figure 3-5 Default Matrix of Compliance

3.5.17 Preparation for Delivery

This section is generally applicable to product specifications only, but is included here to keep the specification compatible with both development and product specifications. This is worthwhile because development specifications can lead to limited manufacture and delivery of working models without the production of product specifications.

It is useful to make a distinction between what covers *packaging* as opposed to *packing*. MIL-STD-490A states that preservation and packaging covers:

> " ... requirements for cleaning, drying, and preservation methods adequate to prevent deterioration, protective wrapping, package cushioning, interior containers and package identification marking up to but not including the shipping container. ... ".

Whereas packing covers:

> " ... the exterior shipping container, the assembly of items therein or packages therein, necessary blocking, bracing, cushioning, and weatherproofing."

The method selected to handle this section was to provide a default text containing only the most general of requirements which the specifier could modify to suit special requirements.

> *The* <instrument.name> *shall be packaged in such a way that its performance shall not be impaired by storage for periods up to the storage time specified in Section 3.2.5.*

> *The packing shall be designed to support the instrument during transportation and storage and provide absorbent protection from mechanical shock. The packing shall not contaminate the instrument during storage and shall keep it free from scratches, dust and chemical attack.*

All containers or parcels which comprise the instrument shall be marked with a minimum of the manufacture's name and the product name. In addition, where appropriate, labels detailing special handling requirements shall be affixed, for example, "Fragile Device", "Do not expose to X-rays", "This Way Up", "Anchor during Transport", etc.

3.5.18 Notes

The notes section contains information of a general or explanatory nature, but no requirements and nothing which is contractually binding. MIL-STD-490A lists the possible contents of the notes section in the order they are to appear:

(a) Intended use.
(b) Ordering data.
(c) Preproduction sample, pilot model, or pilot lot, if any.
(d) Standard sample, if any.
(e) Definitions, if any.
(f) Qualification provisions.
(g) Cross reference of classifications.
(h) Miscellaneous notes.

The most common use of the notes section is to explain the intended use of the item as this important information can be obscured in the detailed requirements. Contractors can then be in a position to identify inconsistencies between detailed requirements and the intended use of the instrument. This feedback forms an important part of the human specification generation process, because although legally if an item is produced to the letter of the specification it has to be accepted and paid for, there are no long term business prospects in producing useless products.

3.6 Discussion and Conclusion

This chapter examined current practices for generating measuring instrument specifications. It was suggested that the human specification process as distilled into the human specification generation algorithm, could be investigated as a route to automation. A generalised specification approach applicable to a wide range of measuring instruments was then proposed using the MIL-STD-490 Prime Item format.

A discussion on specification methods followed which outlined standard practices developed over the last forty years. A general method of completing the paragraphs of the chosen format was then proposed. Each paragraph of the specification format was then discussed and suitable specification methods described.

The important outcome of the work described in this chapter is that by examining the human specification generation method an algorithm has been identified which indicates there is structure to the tasks which can be exploited to assist automation. The seemingly infinite number of specification options has in fact been shown to be tractable. This has been achieved by enumerating all the viable alternative specification methods for each paragraph and giving suitable text for each one. The chosen text for each paragraph has to be customised by inserting short strings which contain the essence of the specification. Provision for user entered custom paragraphs has been included as an alternative to cope with diverse situations.

Chapter 4 shows how the methods derived in this chapter can be successfully used to assist in the production of specifications for measuring instruments.

Chapter 4

Specriter 1: The First Automation of the Specification Generation Process

4.1 Introduction

Chapter 3 proposed a computer-aided specification generation process, derived from a human one, which could be used as the backbone of a computer-aided engineering (CAE) tool. The key concept in the proposed procedure was the use of a fixed, general-purpose instrument specification format, ensconced as a template. The latter half of Chapter 3 was devoted to describing alternative methods for specifying each paragraph of this template.

The last chapter also established some functional design aims for a tool to assist in the production of measuring instrument requirements specifications. These can be summarised as follows:

(1) Improve the efficiency of generating specifications compared with manual processes.
(2) Provide knowledge-based assistance in generating the specification, covering areas such as specification format and use of language.
(3) Provide a user interface suitable for both novice and experienced specification writers.
(4) Provide some mechanisms for completeness, consistency and reasonableness checking.
(5) Provide sensible defaults preferably generated from information already entered.

To these, can be added those requirements, not already covered, which flow from the Computer-Aided Engineering of Measuring Instruments (CAEINST) project philosophy, see Chapter 1:

(6) Concentrate on the incorporation and structuring of existing knowledge and techniques.
(7) Incorporate project management aspects in addition to technical ones.
(8) Consider that the resulting software may later form part of a larger entity.

Armed with this list of requirements, work began on the software suite which eventually became *Specriter 1*. The initial work looked into human computer interface methods and how to generate a document from a question and answer session. Development of the software then proceeded. This activity took place on a sequence of three machines over the course of two and a half years. The final software implementation, (Cook, 1988a), receives most attention as all earlier versions are subsumed into it.

The chapter concludes by summing up the attainments, and inadequacies of the software. This summary, together with the knowledge gained in producing and using *Specriter 1*, are distilled into a brief description of improvements needed for future realisations.

4.2 Initial VAX Implementation

At the commencement of the task, the only computing facility available was the central VAX/VMS cluster at the South Australian Institute of Technology. VAX Pascal was the only modern, general-purpose language available on this computer and so was selected for all tasks except the human interface. To permit future portability, it was decided to constrain programming to the ANSI Pascal subset of VAX Pascal and to prohibit the use of system calls from within the programs.

4.2.1 Human Interface

4.2.1.1 A Look at a Dialogue System

Considerable thought was given to the human interface. Character strings

containing the essence of the specification information had to be extracted from the user in a form which would require minimal processing before insertion into the stored paragraphs described in Chapter 3. Program control also had to be derived from user responses as most paragraphs could be specified using one of many alternative methods. The first tack was to examine an intelligent dialogue systems called SYNICS (Edmonds & Quest 1979) available on the SAIT VAX cluster. This system possesses two features of interest for this task: facilities to build a natural language parser and node-hopping control.

A parser can be used to extract the meaning of the user's response and this allows the user to express an answer to a query in his own words. The meaning, or semantics, can then be used by the node-hopping control to alter the question sequence to suit responses to previous questions.

In practice, natural language understanding was not well suited to *Specriter* and later was also found not to be useful for other related tasks (Sydenham et. al., 1990). The fundamental difficulty is that free form entry was found to give insufficient guidance to users. A second consideration was the realisation that building an adequate natural language parser for the joint domains of measuring instruments and specification preparation would be a major task. Instead, a more conventional form filling procedure was used which proved effective and did not suffer from the substantial overhead involved with SYNICS.

A useful output from the investigation was the discovery that certain parameters to be specified formed natural interrelated sets and that these are best displayed together. Another finding was the emergence of a question sequence based on responses. The concept of response-directed control was thought valuable and was carried through into the later versions.

4.2.1.2 The Screen-Based Human Interface

VAX Pascal was selected for all further work on the human interface. All terminals connected to the SAIT VAX featured screen addressing and as this

facility was becoming common, it was decided to investigate its use in the design of a human interface.

Cursor addressing permits defined placement of characters on the screen. This permits help text, titles, and previous questions and answers to be remain on the screen when new questions are asked rather than have the entire screen output scroll up from the bottom.

The question sequence was partitioned into eleven areas to be tackled in the order shown in Figure 4-1. Eleven screens were designed for the standard VT-100 terminal following the recommendations by Galitz (1985). Each screen had the major topic heading appearing in inverse video at the top immediately followed by help information to aid question answering. The questions are asked in sequential order separated by sub-topic headings also in inverse video. Previous questions stayed on the screen whenever space permitted. The context in which the current question was being asked could easily be determined by scanning the nested and indented topic headings. Figure 4-2 shows a typical *Specriter 1* screen including user responses.

To cater for the fixed format of the screen, and the inability of standard ANSI Pascal to cope with variable length strings, responses had to be limited to 75 characters.

4.2.2 Document Generation

Use was made of *Digital Standard Runoff* which is a VMS system utility identical in concept to Unix *nroff* in that it is a text formatter controlled by command strings imbedded in the source text. It can be thought of as a very primitive word processor but with the control codes explicitly generated and always visible. While it is a poor substitute for a word processor, Runoff is ideal for the task of automatic text generation because it does not require human interaction from a keyboard. A program was written to accept the user responses from the user human interface program, generate paragraphs of text according to these responses,

1. Measurand Definition
2. Performance characteristics
3. Interface definition
4. Physical characteristics
5. Environmental conditions
6. Quality assurance provisions
7. Preparation for delivery
8. Reliability
9. Maintainability
10. Design and construction
11. Notes

Figure 4-1 Order of Tackling Topics

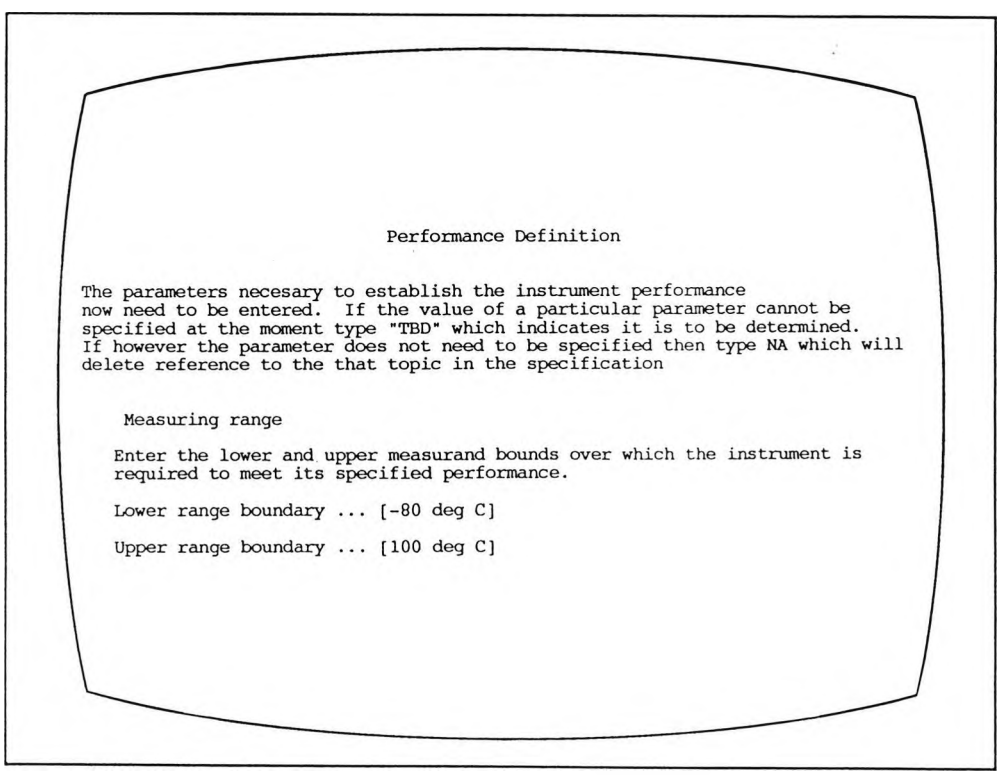


Figure 4-2 Typical *Specriter 1* Editing Screen

and finally produce an output file in a suitable form for the text formatter. The final document was then produced using the text formatter.

4.3 The Need to Change Development Environments

Well before *Specriter 1* was finished, it became obvious that the SAIT VAX Cluster was not an appropriate host for a system such as *Specriter*. The foremost reason is that VAX Pascal did not support modular compilation and this meant the entire program had to be re-compiled after every change. Even with just a few thousand lines of code, the edit, compile, link and run cycle extended into many minutes, even when sole user. When the system was heavily loaded, program development became effectively impossible. Another major factor which encouraged a change of host, was access to the computer; screen-based programs do not work effectively over low-rate modems. Additional concerns included the lack of any expert system software or artificial intelligence languages which could be attached to the developing system. The cost of such packages for VMS systems made their acquisition unlikely.

For these reasons, it was decided to expend the effort to port *Specriter* to a VAX 11/750 running Unix 4.2 BSD. This was successful but development was even slower as Unix is tailored for the C programming language and the Pascal development environment lacked necessary tools, for example, symbolic debuggers. Furthermore, despite the precautions taken, the Pascal source code proved hard to install on other Unix sites, such as City University, due to differences in Pascal implementations and variations in terminal control codes. A complete re-write in C was considered but rejected as yet another consumer of hours for very little benefit.

At about this time, it also become clear to other MISC researchers that the potential users of CAEINST, MINDS and other software under development, were unlikely to possess expensive VAX systems. Hence, they too began to consider alternatives.

4.4 The Personal Computer Implementation

4.4.1 The Selection of a More Suitable Host for *Specriter*

Microcomputers were shown to be useful in instrument system design automation by Gardener (1985) for the following reasons:

(1) The computer is dedicated to the user ensuring good availability and full allocation of the machine's resources.
(2) Microcomputers require no specialist support staff and no special facilities such as airconditioned rooms.
(3) Users have greater control and better interaction with the system.
(4) Hardware and in particular software are inexpensive.
(5) Widely available throughout the world.

These points are even more valid now because of the vastly increased computing power offered by this class of machine. In fact the term personal computer has largely replaced the diminutive term microcomputer which has become misrepresentative. Of particular value for this project, is the superior screen handling capability inherent in most personal computers which can be exploited to create sophisticated human interfaces. Another consideration for this application, is the ability to connect a dedicated printer directly to the computer. This facility certainly speeds the process of creating documents.

The target machine chosen for the final port from Unix was the ubiquitous IBM PC/AT style personal computer. The reasons for this choice were based on affordability, good software availability, good processing performance and the desire to achieve good hardware availability through the huge installed base of these machines. Time has vindicated this decision as development of backward-compatible high-performance machines continues unabated. However, it was known that IBM PC/ATs possess some major limitations, in particular the inability of the normal operating system, DOS, to access more than 640k of main memory and the lack of virtual memory capability. Nonetheless, if carefully designed,

large powerful programs can execute on IBM PCs.

Borland's Turbo Pascal Version 3.0 was selected because of its high development efficiency, low price and because it implements a super-set of ANSI Pascal thus easing the task of installation. The VT-100 style display was retained but a much more rapid screen update rate was achieved by mapping the *Specriter* screen functions onto the Turbo Pascal equivalents.

4.4.2 The Final Version - *Specriter 1.41*

Once established in the Borland development environment, progress on completing *Specriter 1* accelerated. The final version, *Specriter 1.41*, was produced in April 1988. The installation and use of the system are described in the User's Guide, Appendix 1, and will not be repeated here. The design and facilities of the software are described below.

4.4.3 *Specriter 1* Structure

Figure 4-3 illustrates the structure of *Specriter 1* which was implemented as six separate executable modules tied together with an operating system command language program. This approach was used for two reasons. The first is because the text formatter and document display utility were proprietary software which could only be invoked from the operating system. The second was because this early version of the Turbo Pascal compiler was constrained to a maximum executable module size of 64k bytes which equates to roughly 4,000 lines of code. This was circumvented by dividing the roughly 13,000 lines of source code by function, as below:

(a) Main menu production and task sequencing.
(b) Operating system command language generator.
(c) Attribute entry and editing.
(d) Text generation.

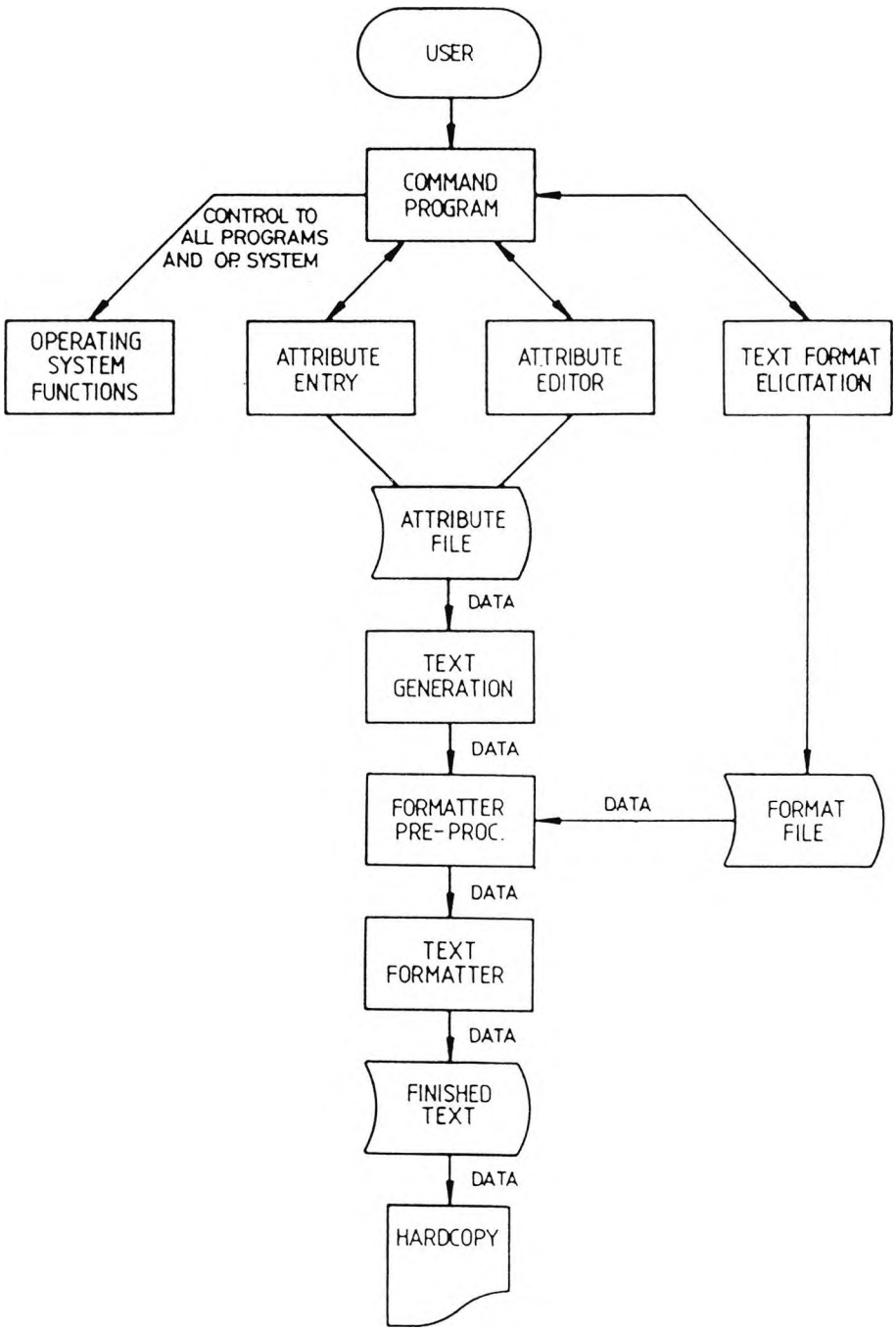


Figure 4-3 *Specriter 1* Structure

4.4.3.1 Command Program

The command program provides the main menu shown in Figure 4-4 and a sequencer which calls the various modules to be executed. The latter is quite cumbersome as Turbo Pascal Version 3 cannot call executable modules directly from the language. To overcome this, the original idea from the VMS and Unix implementations of using an operating system command language program was adopted. The paucity of control available in the DOS batch file language, necessitated extra complexity. The effective but messy solution was to have one Pascal module to produce the human interface and a second to generate a DOS batch file which could subsequently be executed. On completion of this batch file, the command program was reloaded and executed and thus the cycle continues until the user elects to terminate the program.

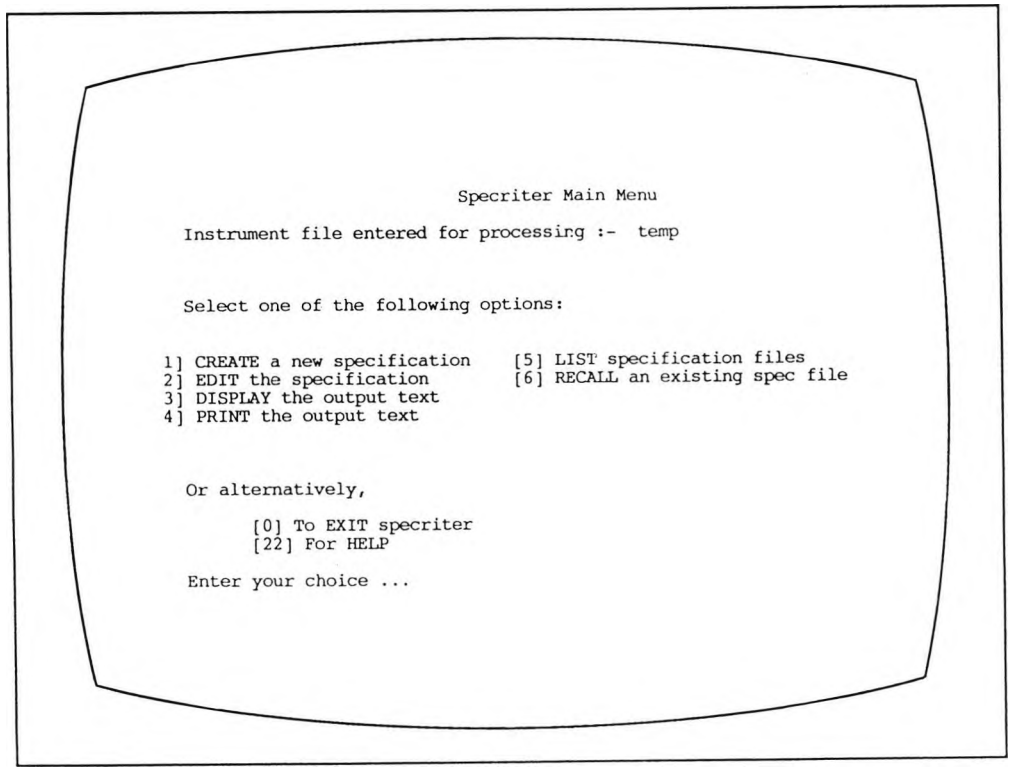


Figure 4-4 *Specriter 1* Main Menu

The main menu provides eight options. Numbers one to four call up purpose-written programs which are described in subsequent sub-sections. The HELP option calls up the Borland Readme program which reads the help file and displays it on the screen. Readme supports backward and forward scrolling and has string search facilities. Control is returned to the main menu by pressing the Escape key.

Rudimentary file handling facilities are provided with LIST and RECALL. These were included to support a workfile concept. Individual instrument specifications are held as instrument attribute files and the one currently under consideration is nominated the workfile and its name is shown in bold on the main menu. Being able to store the essence of the specification in this way means that it is possible to return to a previous instrument without having to repeat the creation process. It also opens the possibility of having a library of instrument specifications available. An alternative to the CREATE process could then be to select and edit the most appropriate library file. LIST displays the instrument attribute files held in the current directory and RECALL is used to designate the workfile. The workfile then becomes the instrument attribute file that the EDIT, DISPLAY and PRINT programs load and process.

4.4.3.2 The Entry and Editing Program

This program, known as *Edit*, is the heart of *Specriter 1*. The same program is employed for creation and editing. The mode of operation is set by status information written into a control file. When CREATE mode is selected from the main menu of the command program, the user is prompted for a file name for the instrument to be described and once this is entered *Edit* is invoked in entry mode. A new workfile is created using the name given. It is loaded with defaults which can be used if questions are skipped or the program terminates abnormally.

Information entry is accomplished by a question and answer format. The order

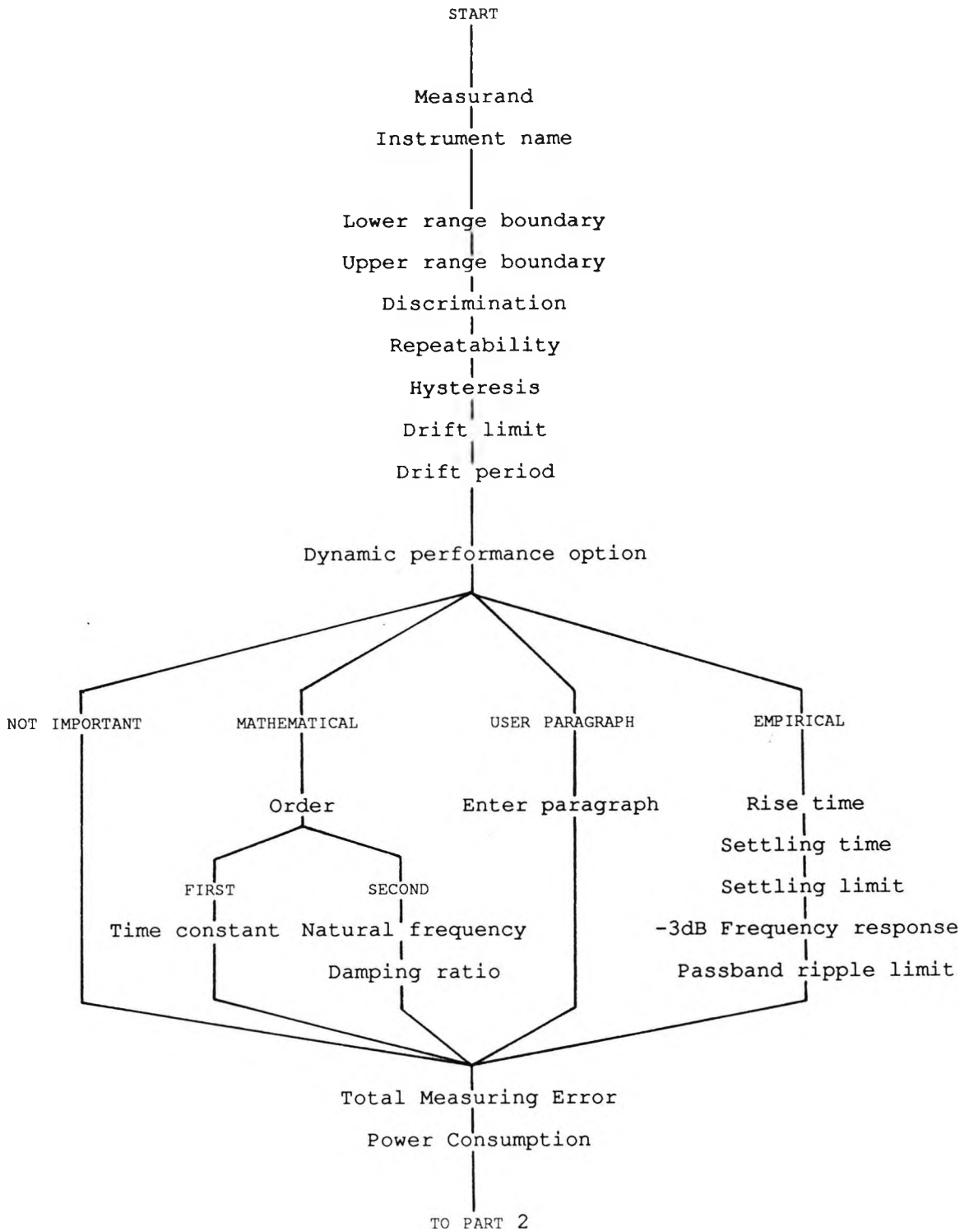
of the questions is based upon the concept of extracting a high proportion of the information in the early questions with the detail following. The idea was to try and build a mental image of the instrument in the mind of the user before the detail was tackled. The order the topics are tackled in was shown earlier in Figure 4-1. Requested responses fall into one of the following categories:

(a) A text string containing such items as numeric limits, qualifiers and measuring units.
(b) A selection of one of a given set of alternatives.
(c) A user entered descriptive paragraph.

The number of questions presented to the user is minimised. This is achieved by interpreting responses to key questions, in particular those configured as alternatives, and selecting the appropriate continuation sequence. Figure 4-5 illustrates the full question sequence.

After completion of information entry, the essence of the information gathered is written to the instrument attribute file named at the start. Control is then returned to the main menu.

The editor can now be used to alter any of the responses. It is called from the main menu by selecting the EDIT option. *Edit* is invoked in editing mode and the specification just entered, being the current workfile, is loaded. Editing takes the form of replacing the previous responses to the questions. To avoid sequencing through the entire question list, the questions are divided into eleven topics and displayed on the editor menu, Figure 4-6. The user selects the topic containing the question(s) to be altered and the question sequence starts from the beginning of that topic. For each question of the topic, the user is presented with the original question followed by the previous response enclosed in square brackets. The user can change the response simply by typing in a replacement or alternatively, can elect to leave the previous one unaltered by simply pressing the return key without making an entry. One or



Legend: SMALL CAPS = OPTION NAMES

Courier font = questions

Figure 4-5 *Specriter 1* Question Sequence (Part 1)

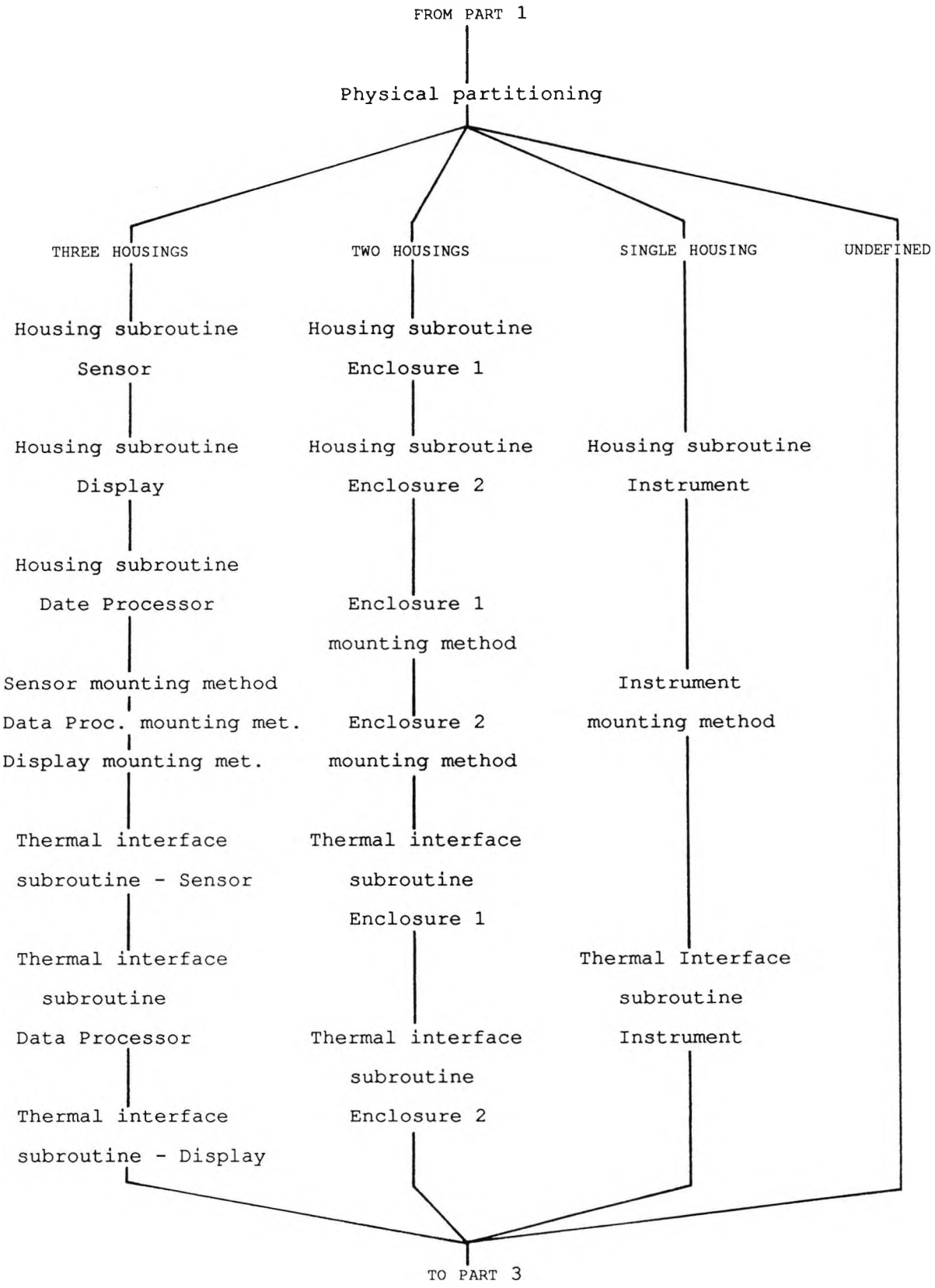


Figure 4-5 *Specriter 1* Question Sequence (Part 2)

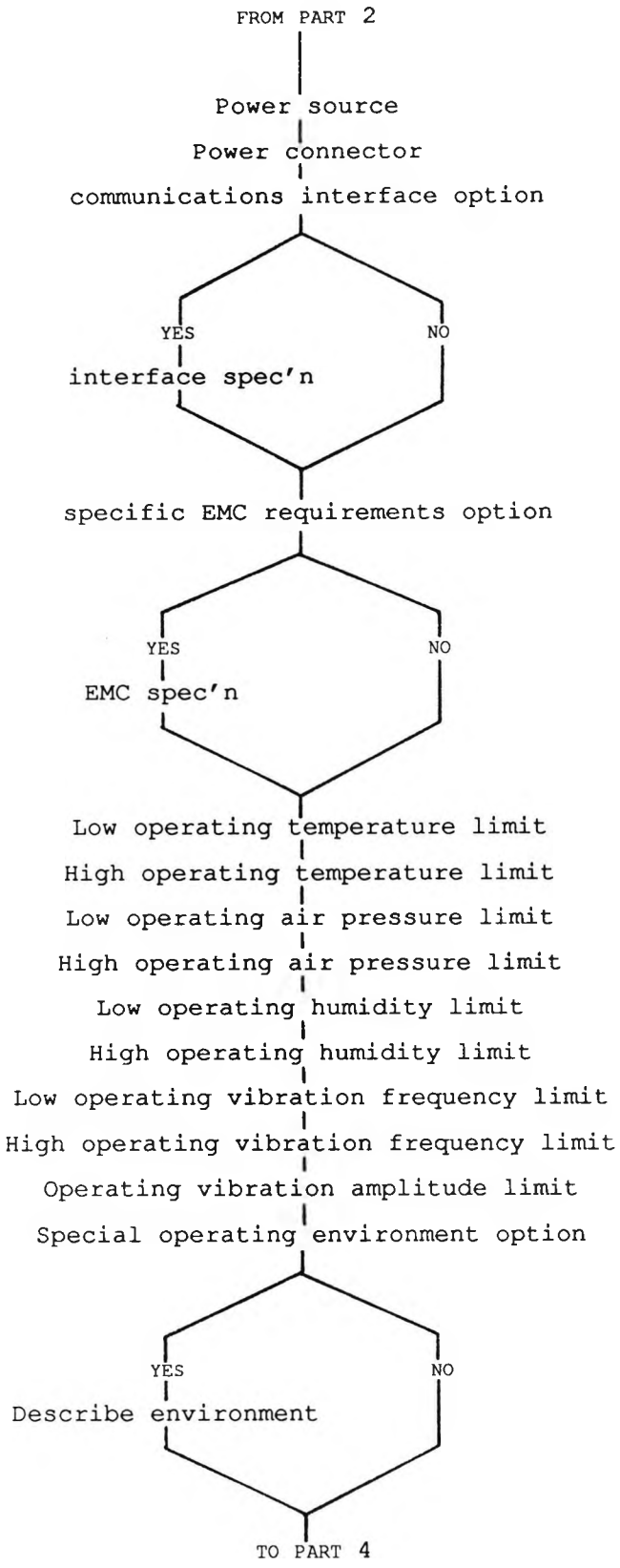


Figure 4-5 *Specriter 1* Question Sequence (Part 3)

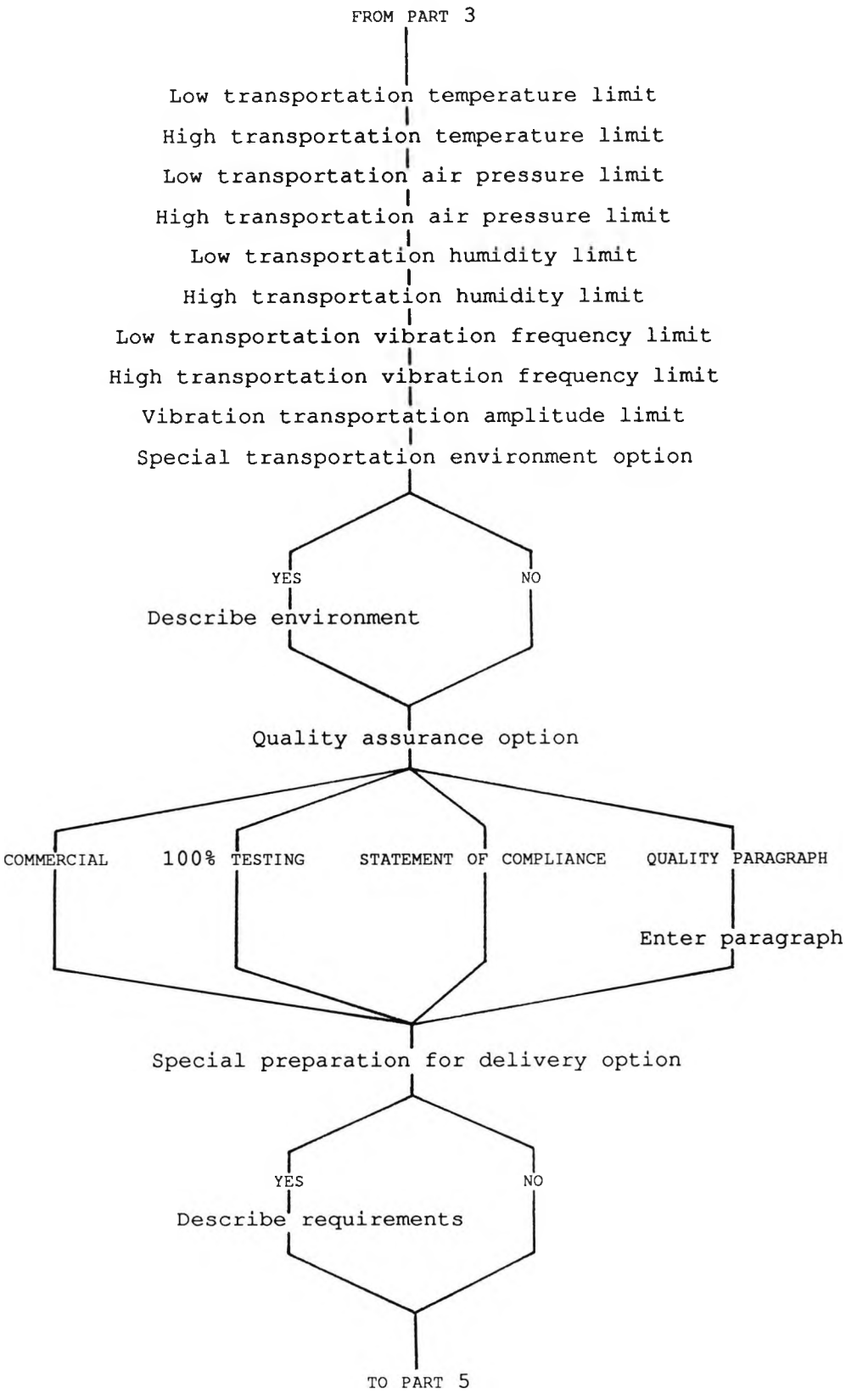


Figure 4-5 *Specriter 1* Question Sequence (Part 4)

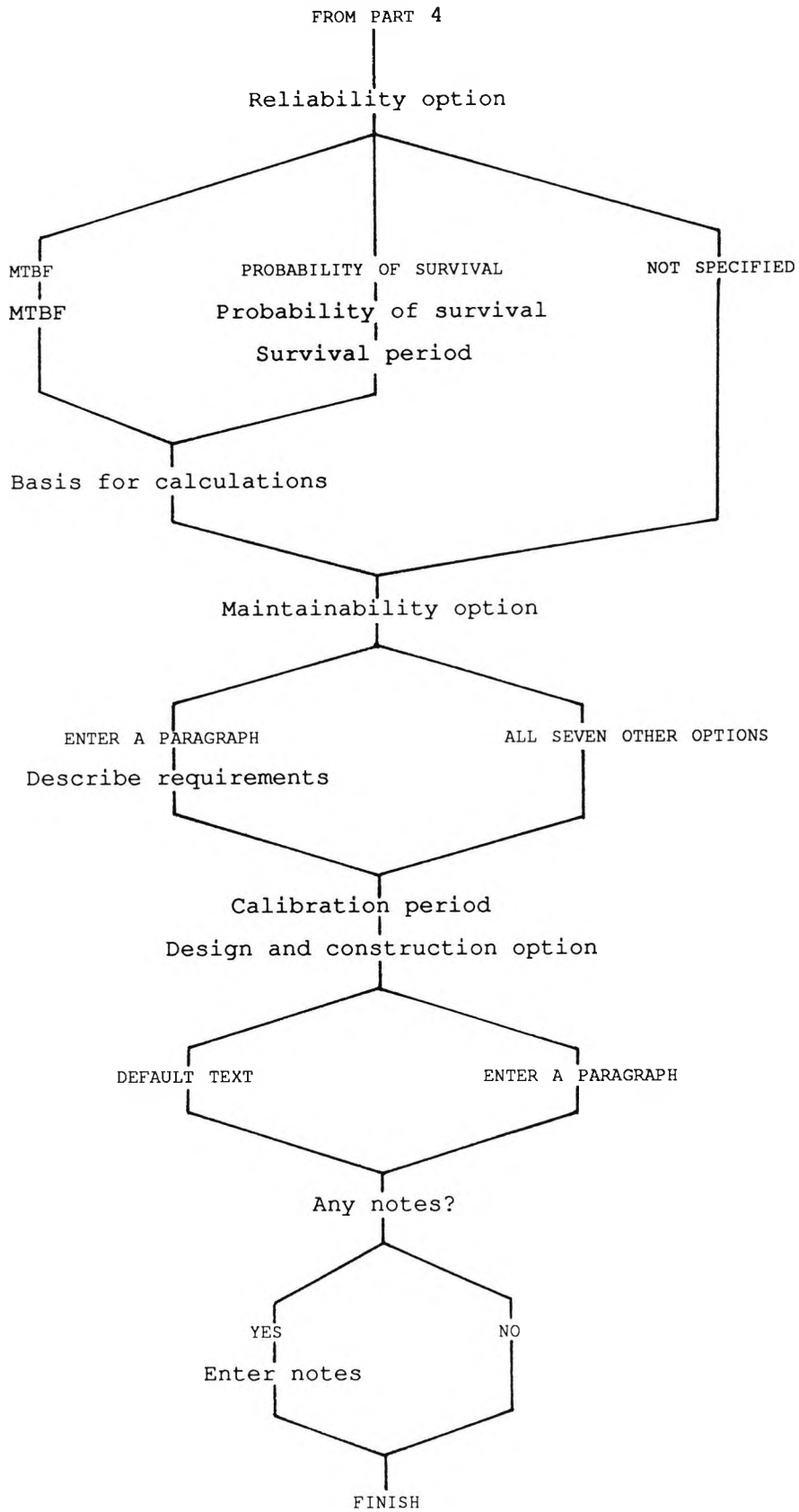
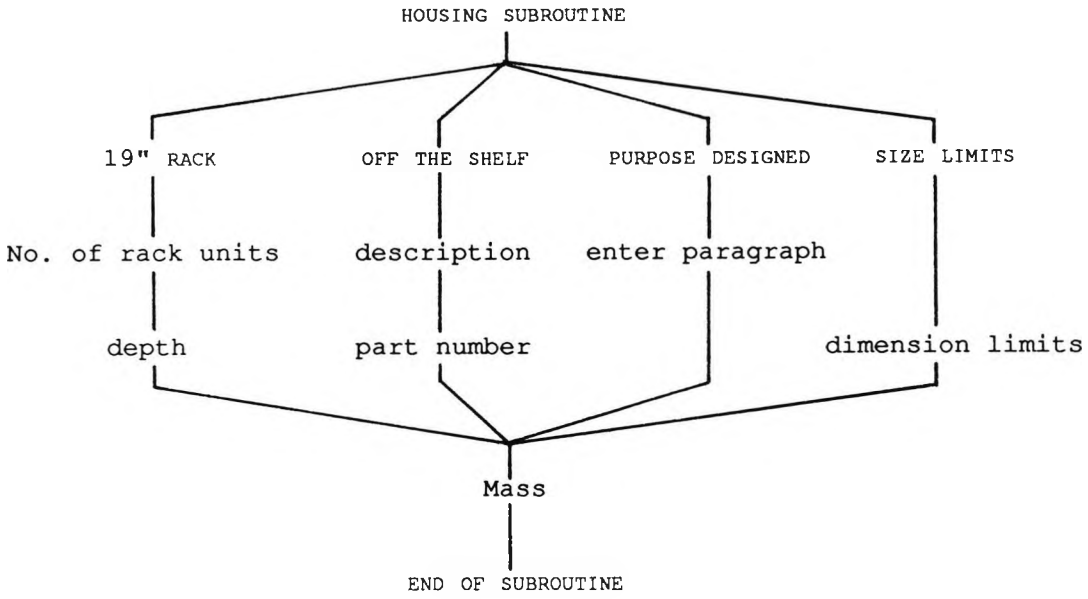
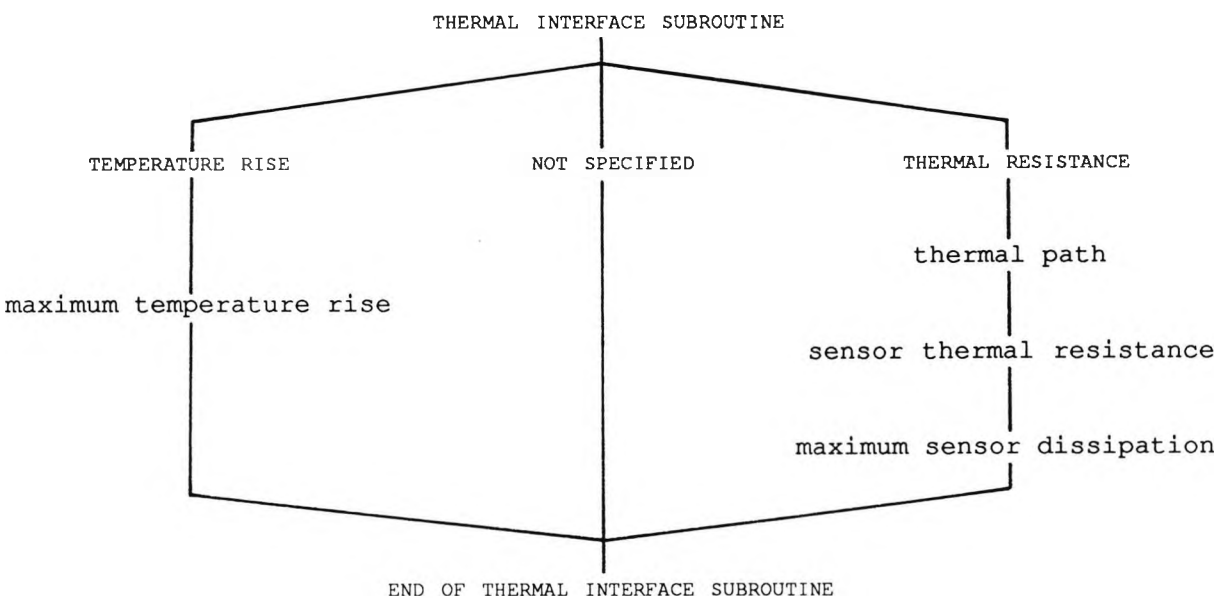


Figure 4-5 *Specriter 1* Question Sequence (Part 5)



(a) Housing Subroutine



(b) Thermal Interface Subroutine

Figure 4-5 *Specriter 1* Question Sequence (Subroutines)

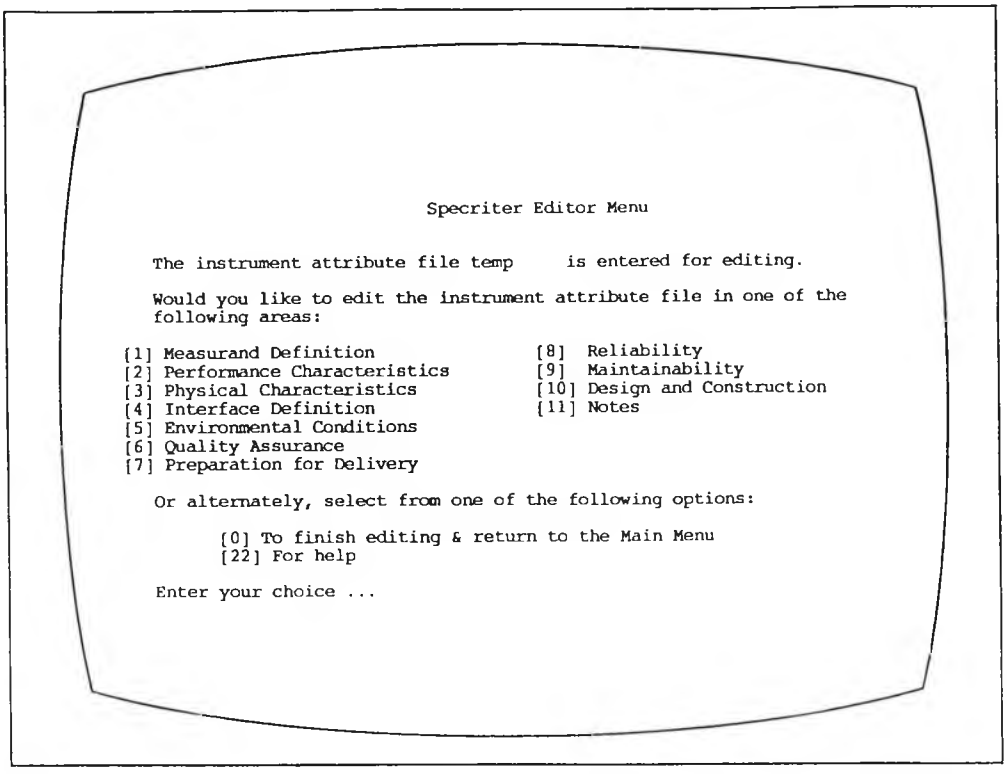


Figure 4-6 Editor Menu Screen

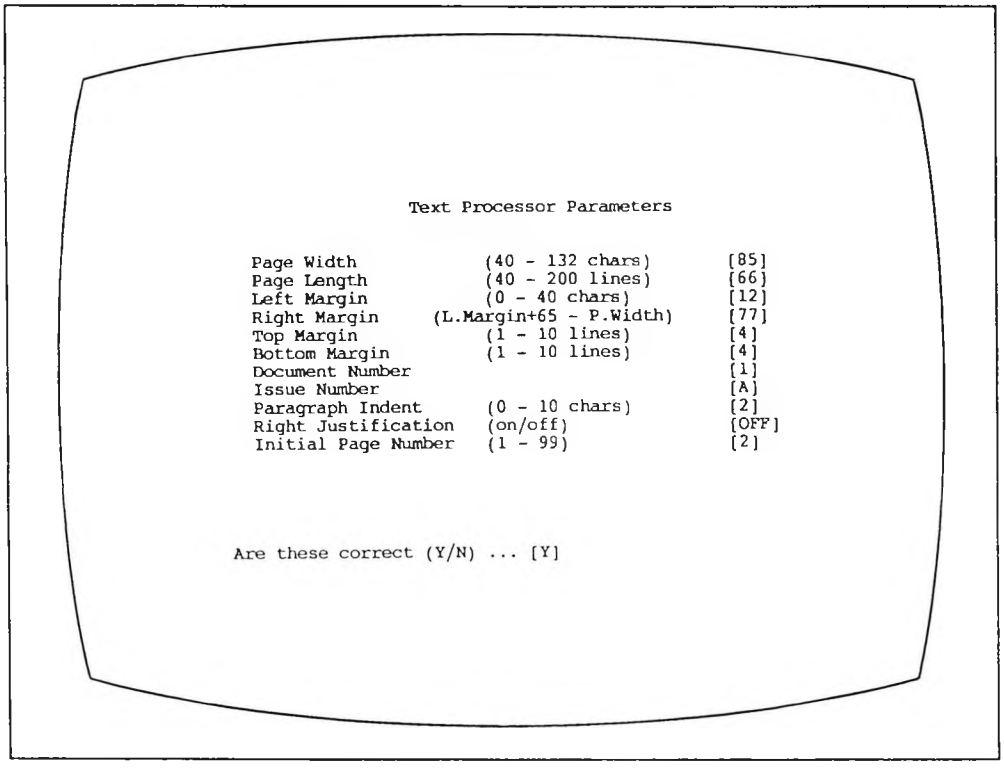


Figure 4-7 Text Format Menu

or more topics can be edited as many times as necessary in one editing session. Once the user is satisfied with the information entered, the attribute file is written to disk and control is returned to the main menu.

4.4.3.3 The Text Generation Programs

Text Generation is performed by selecting the DISPLAY or PRINT options on the main menu. Both these options call the text generation program *Textgen.* This program reads the workfile and the associated text format file. The latter file contains control information for the text formatter and can be altered by the menu which appears next shown in Figure 4-7. The values shown in the figure represent the default format derived from Cook (1986). Once the format is decided the construction of the text commences. The process comprises examining the user's responses, selecting which paragraphs should be used in the output text, and creating the final text from a combination of stored and user text strings. Small grammatical adjustments are performed in this process. Control codes for the target text formatter are then added prior to writing the raw text to an ASCII text file.

Three different formatters have been used successfully; *Runoff* when using VMS, *Nroff* when using Unix and finally, the finished *Specriter 1* uses a DOS program called *RUNOFF* (Blaise, 1986). Each of these is conceptually identical; in response to control codes the text is placed into justified paragraphs, headers and footers and page numbers are added, and the resultant text written to a disk file. Constants were used in *Textgen* to facilitate changes between operating systems.

If DISPLAY is selected, then the finished document is displayed on the screen using Borland's *Readme* program. Whereas, if the PRINT option is used, the finished text is sent to the DOS printer spooler utility *Print* to be printed out as a background task. Control would be returned to the main menu as soon as the spooler was loaded.

4.5 Discussion of *Specriter 1*

Specriter 1 is a complete working system that can assist in the preparation of requirements specifications for measuring instruments. If the user has all the information to hand, a complete document can be produced in around half an hour. The specification will be as good as the input provided and can be adequate for low-risk projects. A modest amount of editing by a manager could render it suitable for contractual use. *Specriter 1* was never intended to be a product; indeed it was not engineered as such. Rather, it was to be a research tool with which to demonstrate the viability of computer assistance in this non-numeric area. Other reasons to produce a demonstrator included the desire to identify research issues, to gain experience in human-computer interface techniques, and to derive the requirements for a knowledge-based approach.

The remainder of this section is divided into four parts. The first enumerates the contribution *Specriter 1* has made to knowledge and the support its success lends to the MISC program CAEINST. The second is concerned with the important lessons that can only be learned from observing an entity in action. The third considers areas identified in early chapters that this system did not address. And finally, the last sub-section sums up the knowledge gained from *Specriter 1* in terms of input to the design of the next generation *Specriter*.

4.5.1 Achievements of *Specriter 1*

Specriter 1 was the first known computer-aided engineering package to be produced to assist in the automated production of measuring instrument specifications. It demonstrated that the generalised specification method for measuring instruments, described in Chapter 3, is capable of being applied to a wide range of tasks and is fundamentally sound. The very fact that a complete system could be made to work from attribute entry, through editing, to data controlled document generation is, in itself, important.

Many of the items identified in the design aims both in this chapter and in

Chapter 3 were met to at least a limited degree. A comprehensive list of questions is provided as a aid to completeness. Quite reasonable specifications of measuring instruments can be produced quickly which represents a major improvement in efficiency and perhaps in quality as well. Initial defaults are provided for all the responses, even if rudimentary.

Specriter 1 was one of the first major components of CAEINST. MINDS was able to interface to *Specriter* by generating an instrument attribute file using the description given in Cook (1988b). Completion of the many items not determined by MINDS could then be performed using the Edit program.

From an implementation viewpoint, run-time performance vindicated the choice of the IBM family of personal computers as a platform. Portability has proved no problem to date; all IBM PC machines and compatibles have run *Specriter 1* satisfactorily.

4.5.2 Lessons Learned From *Specriter 1*

The first observation is that there is a need to limit the amount of detail the user is exposed to. Depending on the instrument, there can be nearly 100 questions to answer and considerable expert knowledge of instrumentation is required to do this satisfactorily. Users (Goldsmith, 1989; Sydenham 1989) suggested more elaborate help facilities be incorporated and a mechanism be provided to return to previous questions and change the response before completing the entry process.

From a software engineering viewpoint, many important points emerged. The programs are hard to maintain; to add another attribute requires changes to both *Edit* and *Textgen*. As the screen is produced by many separate statements, editing the screen layout and adding another question is not simple. To permit easier maintenance, this information needs to be held in data structures rather than program statements. Software tools could then be constructed to generate and maintain these screens. It was also clear that the string handling limitations stemming from the original decision to use ANSI Pascal were causing significant

degradation of the human interface and this needed to be rectified. Provision for very long strings coupled to line and paragraph editors was also identified.

4.5.3 Requirements Not Addressed by *Specriter 1*

Specriter 1 was never intended to be a product but rather a research tool to demonstrate concepts and help identify requirements. Accordingly, some requirements known to be necessary in a final system were not attempted. The foremost of these was consistency and reasonableness checking.

No attempt was made to extract any meaning from the specification generated or from the user text responses. At no point in the execution of *Specriter 1* is the knowledge about the instrument under consideration in a form which readily permits the contents of the specification to be reasoned about.

4.5.4 Design Directions for the Next *Specriter*

Part of the purpose of producing an early demonstrator like *Specriter 1* was to hone the requirements for a more comprehensive system. The question list itself has proved adequate but completing it required the user to possess too much domain knowledge. Additional knowledge would also be required to check the specification during and after preparation to ensure consistency, completeness and reasonableness. These requirements indicate that a knowledge-based approach would be required. The latter also indicated that the responses would have to be reasoned about. This is quite a special requirement for a knowledge-based system (KBS). What is usual, is to design the human interface to suit the KBS in particular its inferencing strategy. Instead, here the aim is to create a KBS which has to use the question and answer structure imposed by the text needed to complete a formatted specification. This has important implications and indicates that the adoption of an underlying formal specification for *Specriter* is not just desirable but perhaps essential if consistency and reasonableness checking are to be performed. This is elaborated in the next chapter.

Some of the limitations of the human interface need to be removed as already discussed. A more fundamental improvement that was identified, would be to allow greater flexibility over the order in which the questions could be answered. The need for comprehensive help facilities in all contexts was often requested by users.

There was no reason to change from the hardware platform as this had proved to be well suited to the task. There remained, however, the task of finding a programming language or environment, available for the IBM PC, that could be used to construct a knowledge-based system and also be used to produce a human interface driven from data structures as opposed to programs.

4.6 Conclusion

This chapter describes the software system *Specriter 1* which implemented the ideas developed in Chapter 3. In doing so, the computer specification generation process, the use of a specification template and the paragraph specification methods have all been shown to be viable. While it was intended to be no more than a concept demonstrator, *Specriter 1* was fully functional and robust.

The description of *Specriter 1*, started by explaining the background of the development which was largely directed by the available software and hardware. The limitations of employing a VAX were discussed and the reasons for transferring the development onto an IBM PC/AT style personal computer elaborated. This account tracks the evolution of the computing industry over the last five years which has led to the rise of the personal computer, and now the workstation, as the replacement for central mainframes supporting numerous terminals.

A description of the final version *Specriter 1* then followed. Each of the modules was described together with the concepts behind their design. (Installation and use of the software is covered in Appendix 1.)

The attainments of *Specriter 1* were then compared against the design aims

developed earlier. Most of the aims were in fact met. Valuable lessons were learned from building and using this software. The prime one was the need for more stored knowledge to reduce the load on the user. This is not an easy addition onto the procedural realisation adopted. Thus *Specriter 1.41* represents, the final development using conventional procedural programming.

Chapter 5 takes the knowledge gained from the work to date and discusses methods of designing an improved software tool which can meet all of the design aims. The largely minor deficiencies recognised in the discussion of *Specriter 1* are also considered there.

Specriter 1 is currently installed on a PC pool at the South Australian Institute of Technology. It has been surprisingly successful, over the last two years over 200 students, staff and clients have used *Specriter 1* in training exercises and design projects (Sydenham & Harris, 1990; Sydenham & Vaughan, 1990).

Chapter 5 - Underlying Concepts for a Knowledge-Based *Specriter*

5.1 Introduction

Chapter 4 showed what could be done, using conventional procedural programming, to meet the design aims for a Computer-Aided Engineering (CAE) tool intended to assist in the generation of measuring instrument specifications. The system described, *Specriter 1*, worked well and succeeded not only in meeting many of these design aims, but was also valuable in refining these aims and identifying aspects not adequately covered. The discussion of *Specriter 1*, concluded that knowledge-based techniques were indicated for such tasks as entry assistance, minimising the number of questions, and checking the specification for reasonableness and internal consistency. The latter requires that the specification under consideration be in a form suitable for automatic reasoning, which implies the use of a formal representation.

The chapter opens with a brief description of formal methods which is immediately followed by an appraisal of the potential this technique for the largely non-functional specification domain of *Specriter*. Given the candidate specification technique that emerges from the subsequent examination of possible specification paradigms, the search for a compatible knowledge representation technique is then pursued.

This chapter deals with the selection of basic concepts for a completely new specification generation tool. Design and implementation issues are the subject of the next chapter.

5.2 Formal Specifications

Formal methods have developed in response to the increasing complexity of systems, in particular, software. Berg et. al. (1982), state that formal methods seek to do for

programming what mathematics has done for engineering: provide symbolic methods whereby the attributes of an artifact can be described and predicted. Listov and Berzins (1986) define a formal specification as one written entirely in a language with an explicitly defined syntax and semantics. They go on to state that formal specifications have the advantage that they can be studied mathematically and can be meaningfully processed by a computer and in addition, certain forms of inconsistency or incompleteness can be detected automatically. Thus validation and system behaviour can be studied before the design stage commences rather than having to wait to observe the execution of the final system.

Cohen et. al. (1986) devote the first chapter of their book to the role of formal specifications in the design cycle. While their treatment is software orientated, they do show how formal specification can be used on a wider class of problem. Formal methods currently find use in software engineering as an aid to specification preparation, program design, and subsequent verification (Berg et. al., 1982; Cohen et. al., 1986; Lees, 1987; Guttag & Horning (1986)). Another related application is automatic programming. This term refers to the process of converting a user's requirement into an executable computer program (Blazer, 1985). This is achieved by first translating the user requirements into a formal specification and then invoking a compiler to generate the executable code. Other applications of formal specifications include automatic verification of communications protocols (Wibur-Ham (1987); Sunshine, 1982) and more recently, albeit in a limited way, the specification of measuring instruments (Delisle & Garlan, 1990).

5.2.1 The Case for Employing Formal Specification Techniques

The subject of this research project is the automation of the process of specifying measuring instruments. This is part of a larger research program CAEINST (Sydenham, 1987) which aims to automate the entire instrument development process from requirements analysis through to application of the finished instrument. The goal of going from a user requirement to a finished product is analogous to the aim of the automatic programming researchers. However, Blazer (1985) states that he considers that full optimisation will never be achieved for

automatic programming and that interactive translation of the high-level specification to a lower-level which can be automatically compiled, will always be necessary. In the measuring instrument domain, this equates to directing the selection of such things as the physical principle of operation and the choice of candidate designs.

From this it would appear that a reasonable expectation for the new *Specriter* would be to operate as an advisory system or intelligent assistant in the preparation of a formal specification. This formal specification would then be interrogated to create the final output text. The advantage of using a formal specification as an intermediate step is the same as in the automatic programming case; the possibility exists for the specification to be validated without recourse to building the item in question. Reasoning can be performed using an inference mechanism operating on the formal specification and associated knowledge bases. Future benefits to be gained include the possibility of using the specification to drive subsequent design, implementation, and manufacturing packages which could eventually create the finished product.

5.2.2 A Formal System for Measuring Instrument Specifications

It is important to consider the implementation of other aspects of the task when considering which of the formal methods to pursue. Knowledge of measurement science and specification practices has to be stored in a form which is appropriate for the reasoning mechanisms which need to operate on the specification. In addition, the user interface needs access to both entities to provide knowledge-assisted entry and context-sensitive help facilities. Finally the text generation program must be able to translate the specification into an English language equivalent.

Software behaviour can be specified as logical relations between inputs and outputs (Blackburn, 1989). However, measuring instruments can only be partially described in terms of a relation between inputs and outputs. Chapter 3 shows clearly that performance characteristics represent a fraction of the items which

need to be described when specifying an instrument. The introduction of two separate specification methods, one for the performance or functional characteristics and one for the rest would not only be cumbersome but would prohibit reasoning about performance in relation to all the other requirements, for example the operating environment.

Cohen et. al. (1986) discusses a range of formal specification techniques. These are abstracted below and later examined for suitability for the task in hand.

5.2.2.1 Model-Based Approach

In the model-based approach, specifications are explicit system models constructed out of well defined abstract or concrete primitives. The constituents of models are data objects representing the inputs, outputs, and the internal state of the system. This approach is well established and methods based on models have been applied to industrial problems successfully. Perhaps the best known model-based specification language is the Vienna Development Model (VDM) which was developed at the IBM Vienna Research Laboratories in the 1970's and is described in Chapter 5 of Cohen et. al. (1986). VDM is based on the use of mathematical abstractions such as sets and finite mappings. A newer approach based on set theory, called Z, has been developed by the Programming Research Group at the University of Oxford (Sufrin, 1986; Delisle & Garlan, 1990). Another example is Gist (Blazer, 1985), a model-based language developed by the Information Sciences Institute of the University of Southern California which has executable semantics and is intended to support automatic programming.

5.2.2.2 Property-Oriented Specifications

In the property-oriented approach, specifications are given in terms of axioms which define the relationships of mathematical operations to each other. Data types have no values defined and no explicit model is formulated, but logical manipulation of the axioms can be used to deduce interesting properties of the

specifications. So-called "algebraic" specifications are the best known.

5.2.2.2.1 Algebraic Specifications

An algebraic specification defines a mathematical object in terms of relations among the operations defined over the object. The specification comprises a list of operations, a list of equations which defines the meaning of these operations in terms of a collection of relations that exist among them together with declarations of constants, types, and the nature of the closure of the set of equations. The complete specification is made up of a hierarchy of modules descending down to basic primitives such as the Boolean object.

5.2.2.2.2 Non-Algebraic Approaches

The principal example of this axiomatic approach is logic programming, such as Prolog (Kowalski 1979) where a restricted form of logic known as Horn clauses is used as a specification language which can also be directly executed using a resolution theorem proving approach. Two other systems are given in Cohen et. al. (1986), a Japanese system IOTA where programs are specified in predicate logic and implemented in an Algol-like language and ANNA proposed for specifying Ada programs.

5.2.2.3 Selection of A Formal Technique

It is not clear how algebraic or model-based specification paradigms can be used to specify anything other than the relation between the input and output of the instrument. Also the complexity of the simple examples given in Cohen et. al. (1986) is somewhat daunting. None of the languages for these paradigms were available to investigate their extension for use on this project. As each of these example systems has been the subject of around a decade's research it was seen as far too large a task to attempt to produce a new formal specification language specifically for measuring instruments.

The approach selected was to write the specification in Prolog. Prolog which is a contraction for PROgramming in LOGic is a programming language which borrows its basic constructs from logic. A Prolog program comprises a finite set of facts and rules expressed as Horn clauses. Prolog is a declarative language and a Prolog program does not step through a sequence of steps but rather seeks to establish whether a goal is true or false. Procedural activities can, however, be achieved by making use of standard predicates which produce side effects in the process of proving the goal. Good descriptions of Prolog can be found in Kowalski (1979), Clocksin and Mellish (1987), Bratko (1986), and Sterling and Shapiro (1986).

Predicate calculus is the underlying formalism of Prolog. Ramsay (1988) examines first order predicate calculus and shows how it can be used to construct a formal system and how unification and resolution can be employed to prove whether an arbitrary formula expressed in that calculus is valid. In Chapter 10 of their book, Clocksin and Mellish (1987) compare Prolog to predicate calculus in order to ascertain just how far the results of logic can be applied to Prolog. They show that predicate calculus formulae can be rewritten in clausal form in terms of conjunction, disjunction and negation. They go on to describe how resolution can be applied to this clausal form to achieve inferencing of new propositions from the ones stated and hence achieve theorem proving. They state that in general Prolog matching will be equivalent to unification used in resolution. They acknowledge, however, that differences can arise because Prolog is a computer language and its design had to take into consideration execution efficiency and extralogical constructs necessary to perform a range of tasks. The offending additions are the built-in predicates which perform functions such as input/output and the "cut" which is used to control the procedural meaning of a Prolog program. In Prolog, it is possible to convert symbols to strings, convert structures to lists and convert structures to clauses. These operations violate the simple self-contained nature of predicate calculus propositions. What is more, the Prolog database can be altered during the conduct of a proof altering the set of axioms. This violates the principle that in logic each fact or rule states an independent truth,

independent of what other facts and rules there may be.

From the above, it can be surmised that Prolog can be used to construct a formal system provided that Prolog's extralogical clauses are avoided. This restriction cannot always be guaranteed and hence not all the properties of a formal system may be available in the new *Specriter*. For this reason, the term *limited formal specification* is most meaningful here. The great attraction of using a computer language rather than a specification language for implementing the underlying formal specification is that other parts of the overall task can be constructed using the same language and development environment. This presents an opportunity for integration of the specification, knowledge based systems and control mechanisms.

5.3 Knowledge Representation

5.3.1 The Case for Inclusion of Knowledge into *Specriter*

Traditional procedural programming is an efficient method of solving problems which can be expressed in an algorithmic form. The very existence of an algorithm indicates that the process to be modelled on the computer is well understood. However, when the problem domain is not well understood, or input data is incomplete or inconsistent, or the relationship between interacting factors cannot be well described, usually because of complexity, procedural techniques cannot be employed. This is often the case for complex real-world problems, which until recently could only be tackled by humans.

Knowledge-based systems are an attempt to duplicate the performance of the human reasoning process using computers. This approach is based on the concept of storing domain knowledge, as facts and relations between facts (in general rules), in a manner amenable to the chosen inferencing process. Figure 5-1 from Bratko (1986) illustrates the minimal structure of a knowledge-based system.

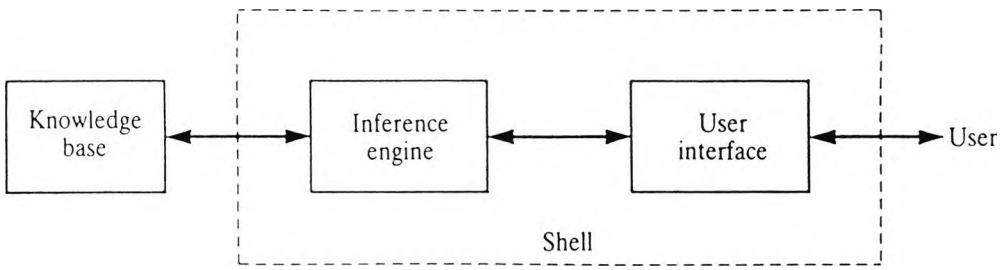


Figure 5-1 Minimal Structure of a Knowledge-Based System (from Bratko, 1986)

Such systems are often termed *expert systems*. This phrase has many definitions, see Simons (1985), but it is generally agreed that expert systems tackle problems requiring a considerable level of human expertise, the classic example being medical diagnosis, using stored knowledge and can produce an explanation of their reasoning. As there is no need for *Specriter* to meet all these restrictions the more general term Knowledge-Based System (KBS) will be used henceforth. As there are many books which describe the evolution of knowledge-based systems (for example, Michie, 1982; Hayes-Roth et. al. 1983; Winston, 1984), this standard discourse will not be reproduced here.

The task of preparing measuring instrument specifications requires a combination of intelligence and, in particular, experience derived from considerable exposure to the engineering industry. This point was strongly reinforced in Chapter 3. While it has been shown in Chapters 3 and 4 that certain aspects of the task can be conceived as an algorithm, within that algorithm are sub-tasks which require knowledge for their completion. The most important areas are user assistance and checking for consistency and reasonableness. The latter can only be achieved by a considerable depth of knowledge across the domains of measurement science, instrument engineering with emphasis on current industrial practice, and project

management including commercial factors. As discussed in Chapter 3, this is usually beyond the scope of a single person's competence and becomes a shared responsibility. Other topics which could benefit from KBS treatment could be text generation coupled to natural language understanding and the document format selection and enhancement. It was decided to concentrate on the first two as the last two areas are tolerably well handled using the techniques implemented in *Specriter 1*, see Chapter 4.

5.3.2 *Specriter* Knowledge Representation Requirements

Knowledge can be represented in a computer using techniques which vary from the purely procedural, through to constructs which support an emulation of common-sense reasoning. Reviews of established knowledge representation techniques can be found in Brachman & Levesque (1985), Ringland & Duce (1988) and Walters & Nielsen (1988). These works make it clear that the selection of a knowledge representation and the associated inferencing strategy is dependant on the application. Thus the representation requirements of the knowledge domains need to be established.

There are some general requirements common to all knowledge-based systems. In order to maintain and improve the knowledge base, it is desirable that it be completely separate from both the inferencing mechanisms and the human interface which together are often referred to as an *expert system shell* or just *shell*. This also allows the shell to be re-used for other similar problems.

The *Specriter*-specific knowledge representation requirements are unusual in that there are at least three definable knowledge domains to be considered. Each of these is dealt with in turn below but ideally they need to be unified into a single structure which can become the *Specriter* knowledge base. Another consideration to take into account is the desire to construct the KBS's using Prolog to allow ready interaction with the limited formal specification of the measuring instrument under consideration.

5.3.2.1 Instrumentation Knowledge Representation Requirements

This is a large field of knowledge which covers the domains of measurement science, instrument engineering and current industrial practice. In addition, for each measurand there will need to be a separate knowledge base. The potential magnitude of the knowledge base(s) indicates that the knowledge will have to be organised in some way to constrain the number of facts and rules considered at any one time. The reasonableness of an instrument specification can be divided into two components: whether the requirements are reasonable from a technical point of view, for example with respect to known operating principles and associated achievable performances; and whether the specification represents a reasonable potential task for a supplier to undertake. The latter would also need to take into account company objectives, contractual arrangements, and penalty clauses. Thus project management knowledge needs to be included.

5.3.2.2 *Specriter* Human Interface Requirements

There is a desire to provide knowledge-assisted entry to reduce the input entry effort and the knowledge needed to successfully specify an instrument. Such techniques as intelligent default generation, intelligent question list generation, intelligent option generation are all possibilities. The help information could not only be context-sensitive but also intelligently generated.

The finished text can be thought of as a translation, or view, of the internal limited formal specification. It can be conveniently considered as part of the human interface. It would be particularly useful to be able to view the finished text during an editing session. Hence the human interface system will need access to the questions, screen format information, the instrumentation knowledge bases, the specification knowledge base and whatever structure is used to hold the text fragments. This access would be necessary during both user interaction and specification generation.

5.3.2.3 Specification Generation Requirements

The knowledge of specification practices and specification methods for measuring instruments, described in Chapter 3, mapped well onto the procedural implementation described in Chapter 4. The sole criticism from an implementation viewpoint is that all the knowledge was held as program statements. Any changes necessitated recompilation. Adding an extra attribute, for example, meant amending both *Edit*, *Textgen* and the format of the instrument attribute file. This was not only time consuming but also led to configuration control problems as the programs can only work together as a set.

To overcome this problem and to provide the human interface functionality discussed above, it is clear that the entirety of the knowledge about the finished document structure, the specification methods, and the text fragments, needs to be held in some structure that can be accessed by the text generation engine.

5.3.3 In Search of A Knowledge Representation for *Specriter*

Quite a number of knowledge representation schemes have been proposed over recent years. Brachman and Levesque (1985) state that the notion of knowledge representation is essentially a simple one that has to do with writing down in some language, or other communication medium, descriptions that correspond in some salient way to the world or the state of the world. It is however, also necessary to consider the ways in which the representation can be manipulated and the uses to which it can be put. Ringland and Duce (1988) suggest that the first ingredient in the knowledge representation problem is to find a knowledge representation language, that is some formal language in which domains of knowledge can be described. The second is to have some component of the knowledge representation which can perform automatic inferences for the user, while the last concerns capturing the knowledge.

Prolog, the language already selected to represent the specification of the instrument under consideration, is widely known for its ability to represent

knowledge (Kowalski, 1979; Clocksin and Mellish, 1987; Bratko, 1986; Sterling and Shapiro, 1986; Ramsay, 1988). There is significant practical value in selecting this language for the knowledge representation tasks also. Prolog can be used for a variety of representational paradigms and examples of Prolog rule-based systems (Bratko, 1986; Schildt 1987; Smith, 1988; Marcellus, 1989), logic-based systems (Yin, 1987; Weiskamp & Hengl 1988), frame-based systems (Cuadrado & Cuadrado, 1986; Yin, 1987; Rosin 1988; Weiskamp & Hengl 1988), and semantic nets (Weiskamp & Hengl, 1988) can all be found. Thus the choice of language does not complete the task of choosing a knowledge representation. Each one of these representation techniques is examined below together with its reasoning strategy to decide the knowledge representation best suited to *Specriter*.

In addition to expressive adequacy and reasoning efficiency, Ringland and Duce (1988) include the following issues in their list of items to be considered when selecting a knowledge representation: *meta-representation*, the structure of the knowledge and the representation of knowledge about this structure; *incompleteness*, how can inferencing be performed over incomplete knowledge and how earlier inferences can be revised in the presence of more complete knowledge; and *real-world knowledge*, how can beliefs, desires and intentions be dealt with. These issues will also temper the selection of a knowledge representation technique.

5.3.3.1 Rule-Based Systems

5.3.3.1.1 Introduction

A classic way to represent human knowledge is the use of a series of production rules of the form:

IF <predicate> THEN <consequent>

The satisfaction of the rule antecedents contained within the predicate gives rise to execution of the consequent which performs some action.

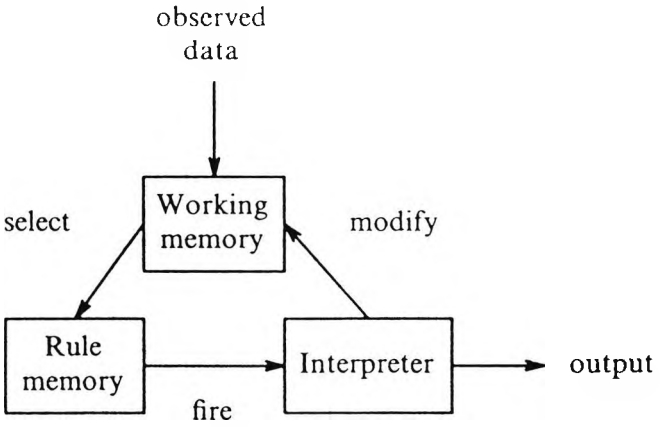


Figure 5-2 Production System Execution Cycle
(from Williams and Bainbridge, 1986)

Figure 5-2 reproduced from Williams & Bainbridge (1988) shows the architecture and execution cycle of a simple production system. Such systems have been used successfully to model human problem solving activity and adaptive behaviour. Commercial quality tools such as OPS 5 (Forgy 1982) have been developed to support applications such as R1 (McDermott, 1982). The pioneering medical diagnostic expert system MYCIN and its derivatives (Buchanan & Shortliffe, 1984) showed what could be achieved with rule-based systems.

Reasoning is performed by the inference engine which analyses and processes these rules in one of two ways: backward or forward.

5.3.3.1.2 Backward Chaining

In backward chaining, the inference engine works backward from the hypothesised consequent to locate known predicates that would provide support. The classic example of such a system is where the number of goals is small such as in a diagnostic system and the inference engine starts

with the first possible diagnosis and checks to see if the facts about the current fault or condition support the diagnosis. This is the inherent mechanism of Prolog and is achieved readily using the inbuilt unification algorithm.

5.3.3.1.3 Forward Chaining

In forward chaining, the inference engine works forward from known predicates to derive as many consequents as possible. The classic use of this inferencing mechanism is for planning or computer configuring where the solution space i.e. number of goals can be very large. Forward chaining can be implemented in Prolog but the semantics of the program are somewhat harder to grasp.

5.3.3.1.4 Suitability of Rule-Based Systems for *Specriter*

The biggest drawback of rule-based systems is the lack of structure in the knowledge base. As the number of rules increases, inferencing efficiency declines and maintenance become difficult, it is often not clear what effect the addition of a new rule will have on the system. In addition, because the rules are independent from each other and from the control strategy, it is impossible to determine rigorously the systems's behaviour by static analysis (Williams & Bainbridge, 1988). Hence system performance can only be ascertained by testing the system with the data of interest. Large rule-based systems cannot be exhaustively tested hence behaviour can never be verified.

A rule-based paradigm is not ideal for *Specriter* because of the potentially large number of rules needed to cover the various knowledge domains and the lack of structure and the potential maintainability problem. An additional consideration is that much knowledge, for example associated with the human interface and text generation, is procedural or just text strings, neither of which is well catered for by production rules.

5.3.3.2 Logic-Based Systems

There is dispute over what is meant by the term, *logic* when applied to artificial intelligence (Pavelin, 1987). He distinguished three definitions:

"(a) First Order Logic (FOL).

(b) Some development of FOL which maintains its notation, its notion of a formal language, a deductive proof theory and a well defined model theory.

(c) Any formally defined method of representing knowledge and making inferences about it."

Following Pavlin's example, this discussion of logic-based systems will only consider (a) and (b). In what follows, the term predicate calculus which is synonymous with FOL, will be used in preference because it is more common in AI literature.

In a logic-based system, the knowledge base consists of statements of fact expressed in predicate calculus or some extension of predicate calculus. Deductive reasoning is performed on these statements, i.e., sentences of predicate calculus in the knowledge base, to arrive at new sentences, which are in fact theorems.

The primary advantage of using logic to represent knowledge is that logic is precisely defined and has a widely understood notation and a model theory (semantics) which gives precision to the mapping between sentences of logic and some domain (Pavlin, 1988). Logic is also expressive. Moore (1985), observes that problems of reasoning and representation involving incomplete knowledge can typically only be tackled by systems of formal logic. Logic is free of the ambiguity of meaning often attributed to frames and semantic nets. Perhaps the most attractive property of a logic representation is the proof theory, in particular that the completeness theorem that everything that is true

in all models of a theory can be proved (Pavlin, 1988; Ramsay, 1988; Winston, 1984).

The most obvious language to implement a logic-based system is Prolog, which is in fact itself based on predicate FOL clauses as discussed earlier, because it has been shown that predicate calculus can be represented in Prolog (Clocksin & Mellish, 1987). Reasoning can be performed directly by Prolog using resolution and unification, and if more than one predicate of the same type is present, then conflict resolution is performed by Prolog's inherent capability based on declaration order. Because a logic-based system mirrors the structure of the selected computer language, Prolog, it can be very efficient and effective in operation.

The major difficulty with using logic in artificial intelligence applications is not in representation but in its role in reasoning. It has been argued that reasoning about the real world is not deductive McDermott (1987). Furthermore FOL is monotonic in that new axioms only add to the list of provable theorems and never cause any to be withdrawn. This property is incompatible with some natural ways of thinking because initial axioms cannot be revised to take new axioms into account (Winston, 1984).

Logic is also weak in its ability to represent certain types of knowledge. Winston (1984: Chapter 6) gives examples such as heuristic distances, state differences, the idea that one approach is particularly fast, or the idea that some manipulation works well but only if done less than three times. As this is the type of knowledge that would be held on instrumentation and perhaps other areas, a purely logic-based approach would be unsuitable for *Specriter*.

It is useful to note that in practice, Prolog overcomes many of the representational and reasoning limitations of first order logic by resorting to special built-in predicates and second order logic which considers sets and their properties rather than individuals (Sterling & Shapiro, 1986: Chapter 17). Notwithstanding, knowledge represented as a sequences of clauses in Prolog

suffers from the many of the same problems as rule-based representations. The knowledge base becomes hard to maintain and system behaviour hard to predict as the knowledge base grows.

5.3.3.3 Semantic Nets

Quillian (1968) is generally acknowledged to have been the first to apply semantic networks in the AI field, more specifically in the field of natural language translation and understanding. His idea was to capture the meaning of words in an encoding scheme similar to human memory. Semantic nets comprise objects denoted as *nodes* and relations between objects denoted as *links* or *arcs*. For example the semantic net of Figure 5-3, from Winston (1984), is a semantic net which signifies that BRICK12 is a BRICK and a TOY and that it is RED in colour. The links are unidirectional and it cannot be inferred that RED is a BRICK.

There are a number of important concepts arising from semantic networks that are of interest for *Specriter*. These are described clearly in Winston (1984). The first is the concept of inheritance which is based on the observation that when humans know the identity of something they adopt a list of assumptions about it. This can be modelled using a tree-like semantic net where the root is the most general level and the descendants become more specific. Information need only be explicitly stored at the most general level and can be inferred or inherited by any of that node's descendants. For example, in Figure 5-4 also from Winston (1984), asking for the shape of WEDGE18 yields TRIANGULAR and for BRICK12, RECTANGULAR. The inheritance algorithm given in Winston (1984) institutes a breadth first search in the event that the object does not possess the required link. Inheritance reduces repetition and enables some concept of hierarchical structure. Procedural knowledge has been successfully coupled into semantic networks. If the value for an object is not available then a procedure can be called to compute the value from information which exists within the database. This concept was interesting because it offered a

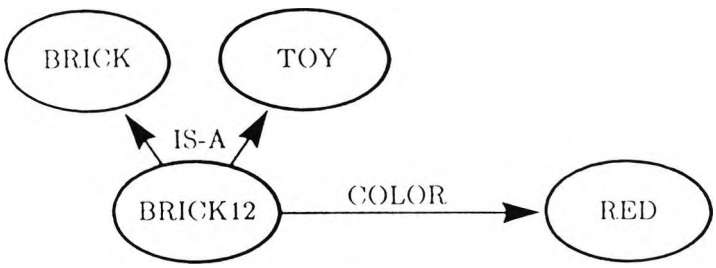


Figure 5-3 A Simple Semantic Net (from Winston, 1984)

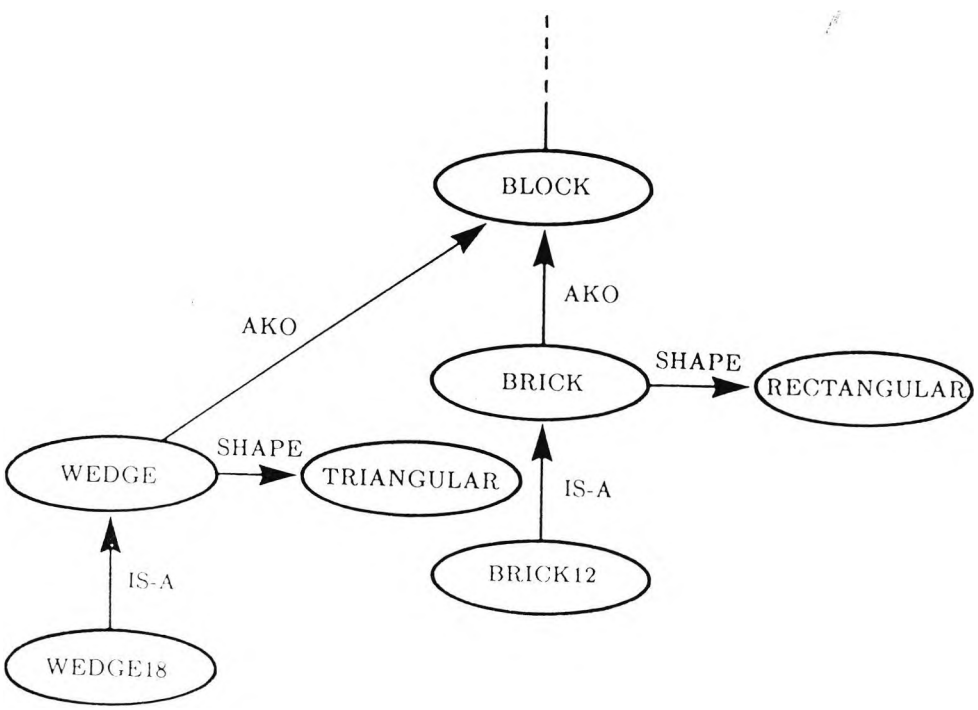


Figure 5-4 Semantic Nets and Inheritance (from Winston, 1984)

mechanism for driving the *Specriter* human interface and text generation from the knowledge base.

The assumptions that a human makes about an object requires the adoption of defaults. For example if a small part of a car is visible a human can often recognise the make of car, approximate year of manufacture and will assume the rest of the car is present and that it represents a mode of transport which uses petrol as a fuel and so on. This concept is very interesting for the specification problem because it could represent a method of capturing the myriad assumptions that are normally taken for granted in the interaction between customer and supplier.

Semantic nets have, however, been criticised for their inadequacies. Woods (1975) points out that links are used in two different ways: assertionally to establish relations between nodes as in the previous examples and structurally to construct part of the knowledge base structure. Brachman (1983) discusses how the IS-A link is used to describe a variety of relations which make the semantics of the net open to interpretation. He also points out that many of the features of semantic nets, for example, modality and defaults, raise severe difficulties for predicate calculus. Randal (1988) concludes by stating that semantic networks are not sufficient in themselves to be an adequate knowledge representation langauge, though they provide a powerful and flexible base on which more complex hybrid system can be built.

5.3.3.4 Frame-Based Systems

5.3.3.4.1 Description

The concept of frames as a knowledge representation technique was initiated by Minsky (1975) although he gives credit for many of the concepts to Bartlett (1932). Minsky sought to represent common sense thought and wished to combine AI with psychology. Minsky asserts that to enable the

performance of human mental activity, the unit of reasoning and the representation of language memory and perception should be larger and more organised than production rules or independent statement expressed in logic. He postulated that when a human encounters a new situation, we select from memory a structure he calls a *frame* which is a remembered framework to be adapted to fit reality by changing details as necessary.

A frame can be considered to be a data structure for representing a stereotyped situation. Minsky uses examples such as being in a certain kind of room or going to a child's birthday party. Attached to each frame can be several kinds of information, for example, how to use the frame, what can happen next in the situation described, what to do if the unexpected occurs, and all manner of details about the situation under consideration.

Each frame can be considered as a network of nodes and relations. It is customary to follow Minsky's suggestion that the top levels of the frame should be fixed and represent things that are always true about the situation. The lower levels have *slots* that must be filled by specific instances of the data. Each slot can specify conditions its assignment must meet and may take the form of a sub-frame. Collections of related frames are linked together into frame-based systems and important actions are mirrored by transformation between the frames of a system. Thus in visual systems, the different frames of a system can represent a scene from different viewpoints and the transformation between frames corresponds to the observer changing position.

The power of frames is based on the inclusion of expectations and other kinds of presumptions. These take the form of defaults placed in the frame's terminal slots. These defaults are loosely attached and can be replaced by new items that fit the current situation better. Defaults can also be generated automatically. In the information retrieval network for machine vision described by Minsky (1975), a matching frame is sought to describe the current scene. If an adequate match is not available, the network

provides another frame which possesses whatever information is appropriate and generates defaults to cover values which are not explicitly known.

Winston (1984) makes it clear that frames are an extension of semantic networks where each frame can comprise a portion of the semantic net. Thus all the desired properties of semantic nets such as inheritance, attached procedures, and defaults are available in frames. Figure 5-5 from Ringland (1988), illustrates a simple frame system comprising just two frames, MAMMAL and DOG. This figure also illustrates inheritance by showing that when reasoning about DOG, a new frame can be formed which combines all the information explicitly known about DOG with that inherited from MAMMAL.

Hayes (1979) considers stereotypical frames to be bundles of properties expressible in predicate calculus and particular instances as simply instantiations. He notes, however, that while the meanings appear to be the same, the inferences allowed by frames, because of their structure, may be different from those sanctioned by logic. Ringland's (1988) discussion of frames, which is based on the papers of Minsky and Hayes augmented by input from later researchers, concludes that frames are more than an alternative logic representation as many features such as the inherent structure of the frame system and the special properties of defaults, such as non-monotonicity and loose attachment are meta-logical.

5.3.3.4.2 Suitability of Frames for *Specriter*

In the context of *Specriter*, frames offer all the expressive power of semantic nets with the advantages that they can be largely modelled using predicate calculus and are easier to understand and maintain. The near equivalence of frames to predicate calculus indicates that Prolog would be well suited for implementing a frame-based system providing the meta-logical characteristics of frames can be handled by appropriate means.

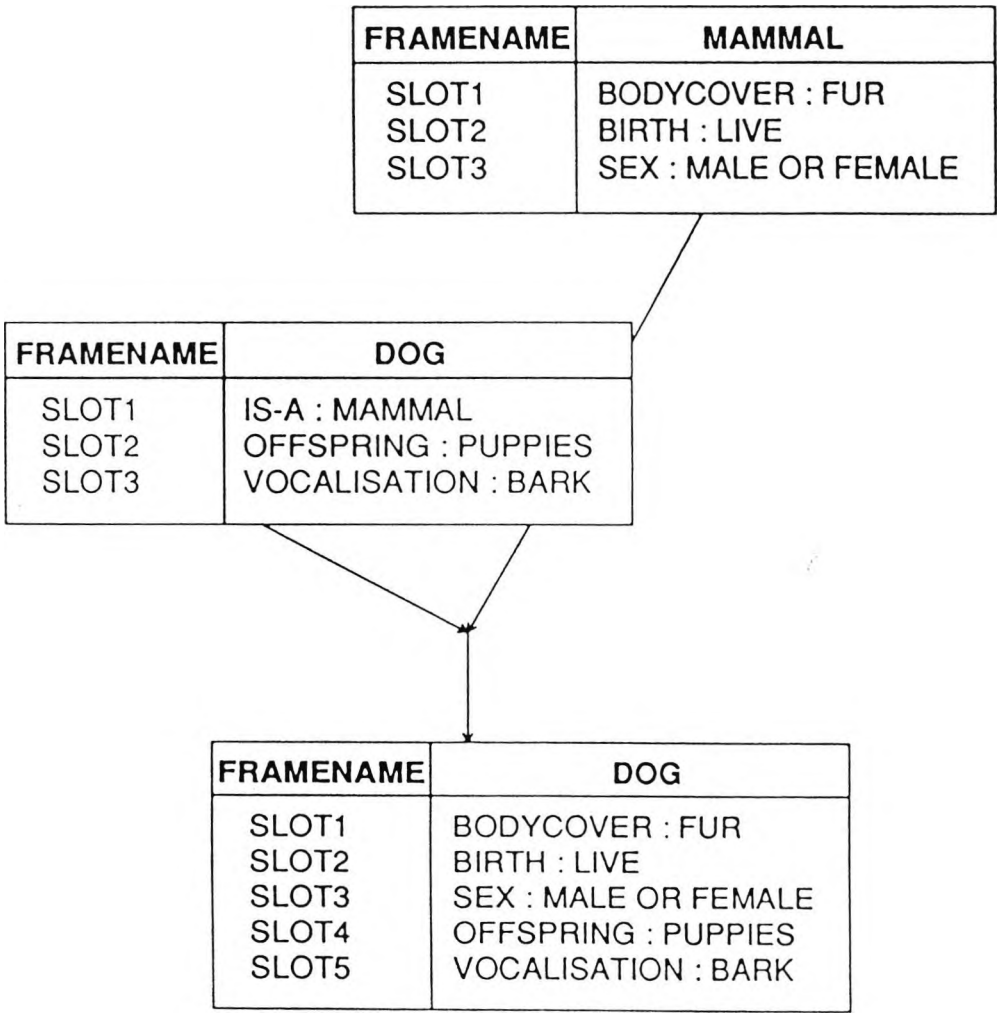


Figure 5-5 Frames and Inheritance (from Ringland, 1988)

Frames systems, like semantic nets possess an inherently hierarchical structure. To take advantage of a frame-based representation the problem domain needs to be structured in like fashion. It is easy to perceive the *Specriter* knowledge domains as devoid of any such structure and in fact this has often been mooted in private communications. It can certainly look that way from the instrument engineer's perspective because it is difficult to modularise or create a hierarchy of the heuristics needed to judge consistency and completeness. Indeed it can appear that it is necessary to reason about the specification as a whole. The specification generation process, already demonstrated to be a predominantly serial procedural process, also reveals little structure. It is from the perspective of the human interface that modularisation and the ensuing benefits of a hierarchical structure can be appreciated.

It has already been said that certain questions can be beneficially arranged into groups. The lengthy question list of *Specriter 1* was divided into eleven such groups as shown in Figure 4-1. This can be considered the first level of structure. The selection of options determines the question sequence. After a selection is made, the context of the topic changes; these alternative contexts become the next level of structure. Options within options represent further levels of structure.

This method of partition may initially seem superficial but far more than the question sequence is determined by option selection. Text generation, the behaviour of the checking system, most of the human interface parameters such as context sensitive help, function keys, and entry assistance are determined in this manner. Hence it could be proposed that every possible entry screen should be represented as a frame, as each can be considered a unique view of the specification under consideration.

5.3.4 Selection of a Knowledge Representation for *Specriter*

In the previous section, a number of commonly used knowledge representation

methods were examined for their potential to fulfil the *Specriter* knowledge representation requirements. The multiple problem domains and desired functionality posed problem for all candidate representations. The most promising candidate is some type of frame-based representation but the proposed screen-partitioned hierarchy needs to be examined in detail for viability.

Frames provide a useful method of modularising a knowledge base. In this case, a frame system is proposed which will have as many lowest-level or terminal frames as there are individual screens. Many of the properties of higher level frames could be inherited by the ones at lower levels in the interests of representational economy. The final obstacle to be overcome is the representation, within this frame system, of consistency checking knowledge. This is a highly heuristic area which has no apparent structure. However, it is clear that the knowledge base of the checker would need to take into account the options selected during the entry and editing processes. The method eventually proposed was to attach a fragment of the consistency checking knowledge base to each frame. That fragment would only be concerned with the values display on the current screen. The entire specification can be checked for consistency by simply checking each valid terminal frame in turn. Rudimentary completeness checking could also be performed at that time by validating that each terminal frame possesses a complete set of slot values.

The question remains on how to represent the checking information in the frames. Aikens (1983) solved a somewhat similar problem by combining the frames with production rules to form the CENTAUR system. Her task was to improve the implementation of a system called PUFF which was written to perform pulmonary function test interpretations. PUFF employed the generalised of MYCIN called EMYCIN, a subject covered by most texts on knowledge-based systems. Although PUFF's performance was satisfactory, there were difficulties with the knowledge representation, in common with many rule-based systems, such as: adding or modifying the rules, altering the order of the questions comprising the consultation, and representing prototypical knowledge. Aikens found that the unstructured rules in the PUFF knowledge base could in fact be grouped. These

smaller collections of rules could then be much more easily maintained and the interaction between rules better established. Aikens then overlaid a frame system over the groups of rules and assigned the rules to a designated slot type. Aikens' prototypes, which represent lung disease patterns are conceptually equivalent to the screen frames proposed for *Specriter,* but the problem solving strategy is quite different.

CENTAUR, in common with all diagnostic systems, seeks to establish a diagnosis given the responses to a list of questions. This is done by establishing which prototype best fits the observed situation, i.e., the frame of best fit. *Specriter*, in contrast, seeks to assist in the production of a specification. The frames are not matched to a given task but are used to determine the perspective when viewing the specification under construction. When a consistency check is called from a particular screen, only the concerns of that screen need be checked. In this proposal, the checking rules of each frame, in fact, comprise a small independent knowledge-based system.

A valid criticism of *Specriter 1* is that a large number of questions need to be answered. It was felt that many of these could be completed automatically in response to higher-level questions. This can be achieved within the proposed frame system using the following strategy. Provide a high-level screen which seeks the response to a small number of general questions such as measurand, environment type, instrument life, expected cost, etc. Assign a slot to each terminal frame which can hold rules to convert these high-level responses into intelligent defaults replacing the initial general defaults such as "not specified".

In assessing the proposed knowledge representation, it is worth examining it against the design aims for a knowledge-based *Specriter.* Reduction in entry effort can be achieved by the intelligent default generator just described. The frame structure can possess slots containing context-sensitive help messages, screen titles and other items necessary to assist users. Checking facilities can be provided on a screen by screen basis which will allow modular construction of this complex function. The proposed structure can contain everything about the

problem domain including all the necessary information about the screens to be displayed. The specification of the instrument under consideration can also be represented in the structure by instantiation of value slots with elements of the specification. In this way the knowledge base structure is complete and becomes the idealised separate knowledge base often discussed but rarely achieved in knowledge-based systems.

The production of the screens and the creation and maintenance of the knowledge base can be achieved using a purpose built tool. This can cope with the inherent tedium associated with constructing screen-based interfaces using cursor addressing.

Thus all the design aims identified for the knowledge-based *Specriter* have been addressed. A particularly useful feature of the frame-based approach is the simplicity in which an unusual situation can be dealt with. A frame representing a complete new screen at any level can be constructed which can cope with any unforeseen extension to *Specriter.*

In fact, as the shell is separate from the knowledge base, the entire knowledge base could be replaced by another which could be used to generate a completely different type of structured document.

5.4 Conclusion

This chapter commenced by examining the possibility of creating a formal specification of the instrument under consideration. It was decided that the most appropriate technique was to write the specification in Prolog and attempt to avoid the meta-logical extensions of the language which distinguish it from predicate calculus. The obvious attractiveness of a single language implementation led to the search for a Prolog knowledge representation suitable for the three knowledge domains to be embraced in a knowledge-based *Specriter.*

A review of established techniques exposed that a structure based on frames

associated with display screens appeared the most attractive and a hybrid representation was proposed which combined production rules and frames. This representation was compared against the design aims evolved in earlier chapters, and all desired features were shown to be possible.

Chapter 6 describes the design and implementation of this system which is named *Specriter 3*.

Chapter 6 - *Specriter 3*: A Knowledge-Based Specification Generation System

6.1 Introduction

This chapter describes the evolution, design and implementation of the Prolog software system, *Specriter 3*. The chapter commences by detailing the selection of the host computer and development software. The appropriateness of the selection is illustrated with examples of how aspects of the task could be tackled.

A description of *Specriter 2*, a Prolog re-implementation of *Specriter 1*, then follows. This development is worthy of discussion because it showed what could be achieved with the tools selected and because many of the concepts for the human interface were evolved during its construction.

Specriter 2, which was undertaken in the spirit of a training exercise, was conducted in parallel with the research recorded in Chapter 5. At around the time work on *Specriter 2* finished, the limitations of an unstructured knowledge representation were becoming obvious from both a theoretical and practical viewpoint. The next section shows how the important ideas developed in Chapter 5 flowed through into the design and implementation of a completely new system, *Specriter 3*.

The ensuing section describes the facilities offered by this new system. The numerous options available from the main menu are covered and greater detail about the design of the system is provided. A description of how *Specriter 3* can be used to generate a measuring instrument requirements specification is covered in the next chapter.

This chapter concludes with a description of the knowledge base editing facility, *Framedt*. The description covers design, implementation and use.

6.2 The Selection of a Host Machine and Development Software

6.2.1 The Selection of the Host Machine

Following the successful implementation of *Specriter 1* on the IBM series of personal computers, it was felt that this class of host would provide a good platform for future development. This choice is supported by the other considerations previously listed in Section 4.4.1, in particular, excellent software portability and fast screen updating.

6.2.2 The Selection of the Programming Language

The study of specification and knowledge representation, described in Chapter 5, had determined that Prolog was to be the language for further work. The initial choice of development language was the newly-released Turbo Prolog, Version 1.1, (Borland, 1986), primarily because of its high development and run time efficiency (Shammas, 1986). This product can behave both as a traditional Prolog interactive interpreter, with full execution trace facilities, and as a native code compiler. Version 1.1 was subsequently upgraded to Turbo Prolog Version 2.0 (Borland, 1988a&b) and further discussion of the language will refer entirely to this version.

The high compilation and execution speed of Turbo Prolog has been achieved by compromising some of the generality of traditional Prolog described by Clocksin and Mellish, (1987). The main differences are:

(1) In Turbo Prolog, programmable operators and meta-programming are not provided as built in functions, although they can be modelled.

(2) Turbo Prolog is a typed compiler.

Programmable operators enable the modification of Prolog syntax at run time by the declaration of infix operators. This feature is said to aid readability of the

code but as the user will not interact with the Prolog source code directly, this is of little interest. In any event, conventional Prolog prefix operators can always be used to implement the function normally achieved with programmable operators.

Meta-programming refers to the manipulation of the source code at run time. The ability to assert and retract clauses at run time is one of the features of traditional Prolog. However, if this facility is used, the Prolog code departs from predicate calculus as the set of axioms becomes non-monotonic. Hence, there is a desire to avoid this feature so that *Specriter* can maintain its links with first-order logic.

Borland, (1988b), in defence of their product, point out that realistic expert systems implemented in traditional Prolog usually require that an inference mechanism be modelled. They state that this is done because backward chaining inherent in Prolog is rarely adequate, and also because it is a method of separating the knowledge base from the inferencing and control mechanisms. Examples of this technique can be found in Sterling and Shapiro (1986: Chapter 19). Such an Inference engine would be an interpreted interpreter with corresponding speed penalties. Thus the concept of modelling an inferencing engine in a compiled language looks to be an advantage rather than a handicap. A stand alone inference engine is distributed with the compiler.

Turbo Prolog is a typed compiler which means that all relations and objects must be declared within the program. Typing enables compilation-time checking of the program which can detect many potential errors such as variables only used once in a predicate, missing predicates and erroneous type clashes. Borland (1986b: Appendix K) also state that it enables functors of compound objects to be converted into single-byte tokens resulting in very fast execution and minimal memory consumption. Typing also eliminates some of the departures of traditional Prolog from predicate calculus, such as the ability to unify variables with structures (Clocksin and Mellish, 1987: Section 10.6).

Another feature of Turbo Prolog which was attractive, was the ability to link to other programming languages such as C, Fortran, Pascal and assembler. It is

generally considered that Prolog is unsuitable for procedural tasks such as human interface, file handling, and algorithm encoding. In the event that these tasks proved difficult to implement in Prolog, the provision of a lifeline to familiar languages was very appealing.

6.2.3 Text Handling and Human Interface Tools

Section 4.5.2 listed some lessons learned from the implementation of Specriter 1. It was pointed out that the constrictions on string length and the lack of true screen editing needed to be attended to. Turbo Prolog string types allow variable length strings and this facility immediately solves the problem of internal manipulation of text. The only limitation on string length is set by the internal 16 bit segment registers of the IBM PC's processor, thus the limit is 2^{16} characters.

User entered paragraphs can be dealt with equally as easily using other features of the language. The text of the paragraph can be assigned to a single string. Editing is performed by opening a window, loading the previous text into it, and invoking the Turbo Prolog editor. At the termination of the editing session, the corrected text is stored. Thus, a large purpose-written Pascal module was replaced by two lines of code. Furthermore, a primitive line-replacement editor has been replaced by a fully-featured editor with in-built context-sensitive help.

Another recommendation from Section 4.5.2 was that screen descriptions should be held in text files rather than the source code and that tools should be used to create and maintain them. The creation of screen layout tools was abandoned upon the release of the Turbo Prolog Toolbox (Borland, 1987). This inexpensive package includes examples of all the necessary tools such as line input drivers, pull down menus, and screen handlers. These products have the capability to cope with strings much longer than the screen area reserved for their display. This *virtual* screen capability frees applications from arbitrary limits on string length.

The high-level built-in predicates of Turbo Prolog, coupled with the human

interface tool archetypes available in the Turbo Prolog Toolbox, cover the *Specriter* human interface requirements completely.

6.3 *Specriter 2*

6.3.1 Introduction

In parallel with developing the ideas for the desired knowledge and specification representation, work began on becoming familiar with Prolog and the numerous tools in the toolbox. After the standard examples were mastered, work began on progressively converting *Specriter 1* to Turbo Prolog to form *Specriter 2*. The structure of the software remained the same as the intention was to simply translate the program adding only limited extra functionality.

6.3.2 Main Menu

The first module tackled was the main menu. Turbo Prolog's "system" predicate was used to call the various compiled modules avoiding the need to generate and execute batch files. The menu itself was implemented using the "pulldown" tool from the toolbox. Each major function was displayed along the top line of the screen. Selection was performed by moving the cursor to the required option with the arrow keys and pressing Enter. For options with sub-choices, a column menu, generally referred to as a pull down menu appeared. The desired command could then be selected using Enter in the same way once again. This is the type of menu Borland use in all their compilers and is widely used in PC software.

The new menu worked well with the executable modules from *Specriter 1.41 and* the complete system was designated version 1.5. This main menu was carried over into *Specriter 3* and will be described in greater detail there.

6.3.3 Attribute Editing Program

The *Edit* program was tackled next. Each of the topics shown in Figure 4-1 was

assigned to a single screen. The screens were then laid out using the *Scrdef* program from the Toolbox. The new *Edit_2* comprised 11 programs each of which used the *Scrhnd* module from the toolbox to display the screen. The use of these tools permitted the screen to be designed by writing the questions and response fields directly onto the screen in the place they were to appear. Also various function keys were enabled to provide help, display of options, editing of responses, invocation of checking facilities, and the means to signal completion and return to the main menu. These screens were also carried over to Specriter 3 largely unchanged, and will be described later.

The text generation program translated readily in Prolog and completed the re-implementation. The human interface limitations of Specriter 1 were now overcome and more importantly, familiarity with Turbo Prolog and its capabilities, and limitations, established. Some checking facilities were incorporated to show that it was a straightforward business to reason about the attribute values.

6.3.4 Lessons Learned From *Specriter 2*

There were a number of important lessons learned from building *Specriter 2*. The first is that the powerful human interface programs were so large that a separate program was needed for each screen to avoid exceeding the memory limitations of the computer. Another reason for the size of the programs was that every field needed to have its own set of predicates. One was needed to describe the actions to be performed during the editing of the attribute value. This covered calling a line input driver, performing input checking, executing attribute specific routines, and finally, attribute storage. A second was then needed to display each attribute in its field, and further predicates were required to handle certain functions associated with a field, in particular, paragraph editing fields.

Another major problem was that the amount of time needed to add another attribute was nearly as long as with the Pascal version. The process comprised updating the relevant screen using *Scrdef*, adding an action and display predicate into the appropriate screen handling program, and adding checking information,

adding appropriate text into the text generation program. Worst of all, the global database declaration needed to be updated every time a new attribute was added, necessitating recompilation of every module.

The third important lesson learned was that while it is easy to represent knowledge directly in Prolog, the knowledge then becomes part of the source code with all the disadvantages this entails. The principal problem is that because the human interface, control, and knowledge are all combined, good familiarity with the entire program suite is needed before new knowledge can be added. Additional problems can arise from the way certain predicates deal with the attributes held in the global database. These attributes which hold the information about the instrument under consideration, can be modified from anywhere in the program and, consequently, changing a rule in one part of the program can have unexpected effects elsewhere.

Although none of the above problems were insurmountable, it was clear that the ad hoc knowledge representation technique and the human interface program could be improved. Work stopped on *Specriter 2*, and the search for a structured knowledge representation proceeded in earnest. The outcome of this task, which is documented in Section 5.3, became for basis for the design of *Specriter 3*.

6.4 *Specriter 3* General Description

Specriter 3 is a knowledge-based advisory program which is designed to assist users create specifications for measuring instruments. This version, in contrast to its predecessors, adds heuristic knowledge to the procedural knowledge previously employed to provide improved performance and functionality.

The best way to consider the conceptual design of *Specriter 3* is by portraying it in a layer diagram akin to the ISO Open System Interconnect (OSI) seven layer reference model (Nussbaumer, 1990) or the popular operating system onion skin model (Lister, 1984). Figure 6-1 shows the six layers that comprise the total software environment of *Specriter 3*. In this model, the lower layers are closest to

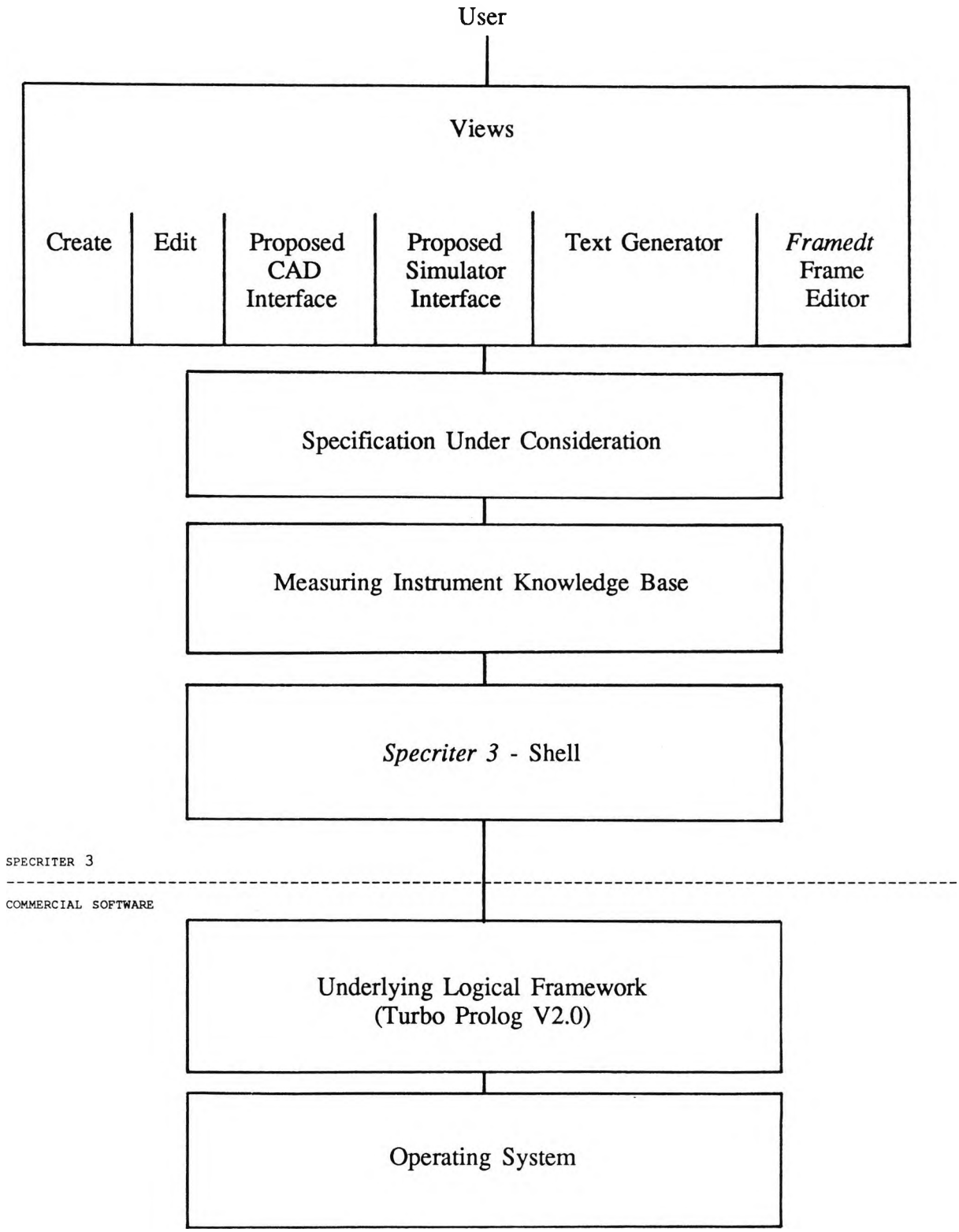


Figure 6-1 The Layering of *Specriter 3*

the hardware of the machine and perform low level functions, such as interaction with the machine's hardware, while the higher layers provide the higher level functions such as human interface and elements of intelligent behaviour.

The design of the entire system is predicated on the knowledge representation scheme selected so this is described first. In the ensuing discussion on the content of the layers the extent to which this external knowledge base controls program operation, can then be appreciated.

6.4.1 *Specriter 3* Knowledge Representation

6.4.1.1 Background

In order to present the minimum number of questions to the user, *Specriter* systems have altered their question list in reaction to previous question responses. In the procedural *Specriter 1*, this was achieved by using conditional branching in the question sequence. In *Specriter 2* the screens changed to suit the response to certain options. Thus for each topic there was a number of different states the screen could take and this was handled by asserting and retracting the necessary screen fields.

In both systems, the final text was produced by a text generation program which read the state of the question list in conjunction with the values held in the associated attributes and selected the appropriate text fragments to assemble.

After observing that the state of the set of questions or screens is the fundamental determiner of the behaviour of the system, it was decided to investigate the use of this information to partition the knowledge base. Hence, this information, combined with other knowledge representation considerations detailed in Chapter 5, led to the scheme proposed in Section 5.3.4.

The concepts proposed were entirely incorporated into *Specriter 3*. Thus all the knowledge used in *Specriter 3*, be it procedural or heuristic, implicit or

explicitly described, is held in a hierarchical frame-based structure where each of the terminal frames of the tree represents a displayable screen.

The knowledge base contains:

(1) All the information required to create the various screens that comprise the human interface to the specification of the instrument under consideration. This includes context-sensitive help information and function key definitions.

(2) Knowledge of the domain of instrumentation to provide consistency checking and intelligent default generation.

(3) Knowledge of specification and document generation techniques. This is held as the list and organisation of the attributes comprising the screens, and the associated text fragments.

The first level of partitioning of the knowledge base is the revised list of topics from Figure 4-1 which are shown in Figure 6-2. Options within those topics which change either the text fragment needed or the list of attributes displayed on the screen, spawn succeeding levels of partitioning. A frame tree for the measuring instrument knowledge base is illustrated in Figure 6-3. The value of these key options determines which leaf frame is displayed, or otherwise viewed, by the application programs. Changing any of these options causes an alternative frame to be loaded.

6.4.1.2 Implementation

There are many ways to implement frames in Prolog (Cuadrado & Cuadrado, 1986; Weiskamp & Hengl, 1988; Keller, 1988). The concept of frames is very general, and about the only thing which appears universal is that a frame should comprise a set of slots. There are no limitations about what these slots may contain. The fundamental decision was whether to have a single large

Units, measurand, and instrument name
Physical characteristics
Electrical interface
General and static performance
Dynamic performance
Quality assurance
Reliability
Maintainability
Design and Construction
Preparation for delivery
Notes
Applicable Documents
High-level requirements

Figure 6-2 List of Specification Topics

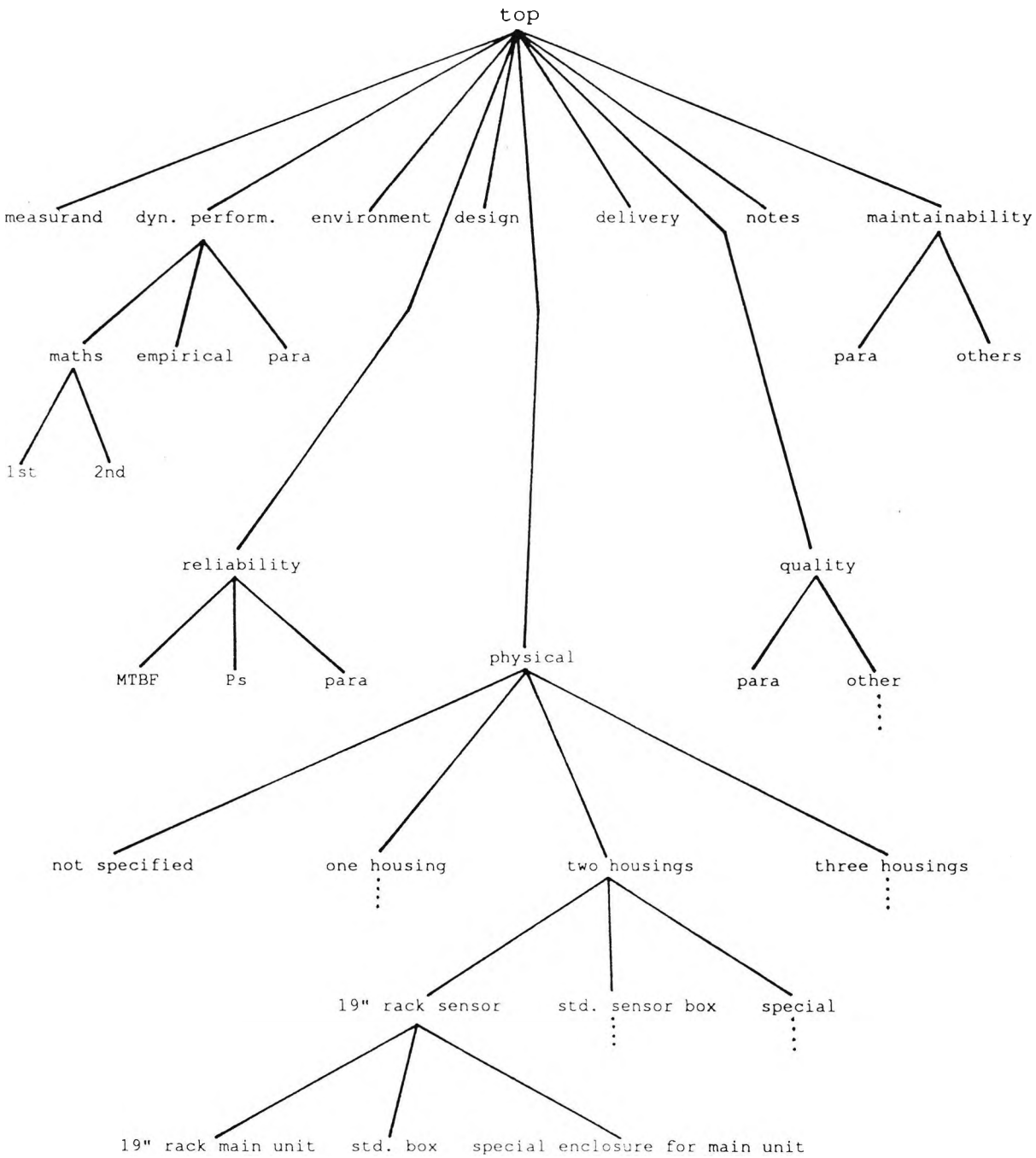


Figure 6-3 Measuring Instrument Specification Frame Tree

predicate for each frame or use more than one. The latter was adopted because Turbo Prolog is a typed compiler which mandates that each predicate to be asserted into the database must conform to a pre-declared format. Addition of further slots at a later date, would have proved problematic had a single predicate been used. In addition, it is generally quicker to search through a list of small predicates rather than through a list of lists. Thus it was decided to have one predicate for each different type of slot. Adding another type of slot is no problem with this implementation. Figure 6-4 lists the slot types supported, and Figure 6-5 expands the information on the active fields. These fields are the ones displayed on the screen which have actions associated with them. The Prolog representation of each of these predicates can be found in Appendix 2.

6.4.2 Description of Each Layer

A brief description of the functions residing in each layer follows. The design and implementation of the modules which implement the functions, is covered later.

6.4.2.1 Proprietary Software

6.4.2.1.1 Operating System

The bottom layer is the standard operating system of the IBM Personal Computer (PC), *DOS*. Development and demonstrations were performed on machines loaded with DOS versions 3.2 and 3.3 but as minimal use is made of the operating system, no problem with other versions is anticipated.

6.4.2.1.2 Logical Framework

The logical framework which supports the *Specriter 3* shell is provided by Turbo Prolog Version 2.0 (Borland 1988a&b).

SLOT NAME	DESCRIPTION
Frame description[1]:	A four-tuple which contains the frame name, the frame level (top = 0), the parent frame name, and a string containing a screen title to be displayed in the top centre of the window.
Output text[1]:	A single string suitable for use by the text generation utility. Any variables in the string will be replaced by their current value held in the knowledge base.
Help message[2]:	A string containing all necessary control characters for help display utility.
Function key list[2]:	A formatted string which holds the function key labels.
Screen text[3]:	A set of text strings that will appear on the screen.
Consistency rules[1]:	A set of Prolog terms. Can cover the entire knowledge domain.
Default generation rules[4]:	A set of Prolog terms used to generate intelligent defaults from "High_level" frame.
Active fields[3]:	Description of the active fields which permit user interaction with the knowledge base and specification. Types include: Standard entry Predefined list entry Database assisted entry User paragraph entry Key option selection

[1] Non-inheritable slots.
[2] Optional. If absent, slot value from next highest frame inherited.
[3] Optional and all slot values from higher frames are always inherited.
[4] Optional. Level 1 frames primarily.

Figure 6-4 *Specriter 3* Frame Description

FIELD SLOT TYPE & SUB-SLOTS	DESCRIPTION
Standard entry:	Line editor-based string entry.
Screen position:	Row, starting column and length.
Default value:	Pre-defined value used when a new specification is created. (Not intelligent default.)
Minimum value:	Lower range checking limit.
Maximum value:	Upper range checking limit.
Prompt:	Prompt message which appears in line input window.
Actual value:	Specified value.
Predefined list entry:	Selection from a predefined list. Used for control.
Screen position:	Row, starting column and length.
List title:	A string which will be displayed top centre of list.
Default selection:	Preferred selection when new specification created.
Option list:	A formatted list, separated by commas.
Prompt:	Prompt message which appears in line input window.
Actual value:	Specified selection.
Database-assisted Entry:	Selection from a database of options. Can be used as a standard entry.
Screen position:	Row, starting column and length.
List title:	A string which will be displayed top centre of list.
Default value:	Predefined value used when a new specification is created.
Prompt:	Prompt message which appears in line input window.
Actual value:	Specified value.
User paragraph entry:	Invokes a screen editor to create or edit a paragraph
Screen position:	Row, starting column and length.
Field title:	A string which overlays the field.
Edit window position:	Starting row & column, number of rows and columns.
Window title:	A string which will be displayed top centre of edit window.
Default paragraph:	Paragraph loaded when a new specification created.
Actual paragraph:	Specified paragraph.
Key option:	Entry from a predefined list. Each entry represents a frame.
Screen position:	Row, starting column and length.
Default selection:	Preferred selection when new specification created.
Actual value:	Specified selection.

Figure 6-5 *Specriter 3* Active Field Description

6.4.2.2 Purpose Written Software

6.4.2.2.1 The *Specriter 3* Shell

The shell is a domain independent program which provides the human interface drivers, local text generation, and an inference engine to interpret attribute checking rules and intelligent default generation rules.

6.4.2.2.1.1 The Human Interface Drivers

The behaviour of the human interface can be limited to a pre-defined set of actions, hence a conventional data-driven system can be used. Thus as the size of the editor screens and the help windows, and the nature of the line input drivers were all pre-determined, the variables held in the slots of the knowledge base, provide all the necessary information to generate the human interface. Inspection of Appendix 2 provides information on the options available to the human interface drivers from the knowledge base.

6.4.2.2.1.2 Local Text Generation

Local text generation is limited to displaying the output text associated with the current screen. As text is held in each frame, and a frame is already being viewed, searching is limited to collecting the text fragments from parent frames. The local text generator assembles these text fragments into a single string, inserts the value of attributes where indicated, and displays the result in a window.

6.4.2.2.1.3 The Inference Engine

In order to permit maximum generality it was decided to represent the heuristic rules directly in Prolog. The rules could then be interpreted

using the inherent backward chaining mechanism of Prolog or, if necessary, by any other inferencing mechanism modelled in Prolog, for example forward chaining.

The interpretation of rules directly requires meta-programming. Although meta-programming is not available as a standard feature of Turbo Prolog, a meta-interpreter can be built using the language. This is the subject of Appendix K of the Reference Manual, (Borland, 1988b). Extensions to the interpreter were necessary to permit operations on real numbers and the use of special data handling predicates.

6.4.2.2.2 The Knowledge Base

The knowledge base has already been described. It can then be thought of as sitting on top of the shell and accessing it to provide the functions demanded by the layers above such as screen display and reasoning. Details of the implementation of the knowledge base can be found in Appendix 2.

6.4.2.2.3 The Specification

The specification under consideration is a list of Prolog terms held in a data file. During loading, the specification is used to fill the values of the slots in the knowledge base. There is no requirement for the specification to be complete and the prolog terms can be stored and retrieved in any order. This flexibility allows ready integration of Specriter to other programs which may wish to create, manipulate, or use all or part of the specification. Any slot values which are not loaded from the specification file are filled with a default value held in the database.

Because the specification and knowledge base are merged at run-time, the Views effectively see a seamless knowledge representation.

6.4.2.2.4 The Views

The outermost layer of *Specriter 3* is the called the views. These are the programs which view, or interrogate, the entire system. As the shell contains most of the low-level predicates and all the screen handlers, the various view programs tend to be quite small.

6.5 *Specriter 3* Design and Function

The purpose-written software which comprises *Specriter 3*, is formed from a number of modules which are linked together to form a single large executable program. The partition of the modules was directed by implementation issues, as discussed below, rather than conceptual design.

Turbo Prolog has powerful debugging facilities but these are only available when the program is compiled into memory as opposed to a stand alone executable file. Prolog can be particularly difficult to debug at any time, and there are many instances were debugging facilities are essential. An example would be when optimising highly recursive predicates by applying techniques such as tail-recursion elimination. Thus, each of the modules was designed to be a stand alone program to provide access to these valuable tools. Conversion between stand alone modules and a linked executable is achieved through use of conditional compilation directives.

Specriter 3 divides readily into four modules:

(1) The main module, *SPEC3*, which creates the main menu, accesses the on-line manual and calls all the other modules.

(2) The specification creation and editing facility, *Edit3*.

(3) The *Specriter Inference Engine*, *SIE*, which performs reasoning tasks on the specification, based on the contents on the knowledge base.

(4) *Textgen3* the text generation facility which converts the specification held in the knowledge base into a document.

Figure 6-6 illustrates the connection between the four modules and shows which ones access the various disk files. Each will now be described in turn from an operational viewpoint.

6.5.1 *Spec3*

6.5.1.1 General Functions

The primary functions of *Spec3* are the initialisation routine which sets up the system, the main menu system, and the straightforward menu command routines. File accessing is also concentrated in this module and the extensive file checking routines reside here.

6.5.1.2 Initialisation

On invocation of *Specriter 3*, the main module, *Spec3* runs. Its first task is to re-create the environment previously stored in disk files. The first file loaded is "SETUP.DAT" which contains the screen colours, names of the knowledge base files, name of the specification file and the last frame examined. (The latter is used by *Framedt,* the knowledge base editor, see later.)

There are three knowledge base files. Fragmenting the large knowledge base was necessary because the text editor in *Framedt*, in common with many DOS editors, can only operate with files up to 2^{16} or 64 K bytes in length.

The order of loading the knowledge bases is not crucial because of the modular nature of the knowledge base and the independence of Prolog terms. In fact it is possible to load only one file should this be desirable for any reason. The files which comprise the measuring instrument specification knowledge base are called "SPEC3_1.KBA", "SPEC3_2.KBA" and "SPEC3_3.KBA". The first

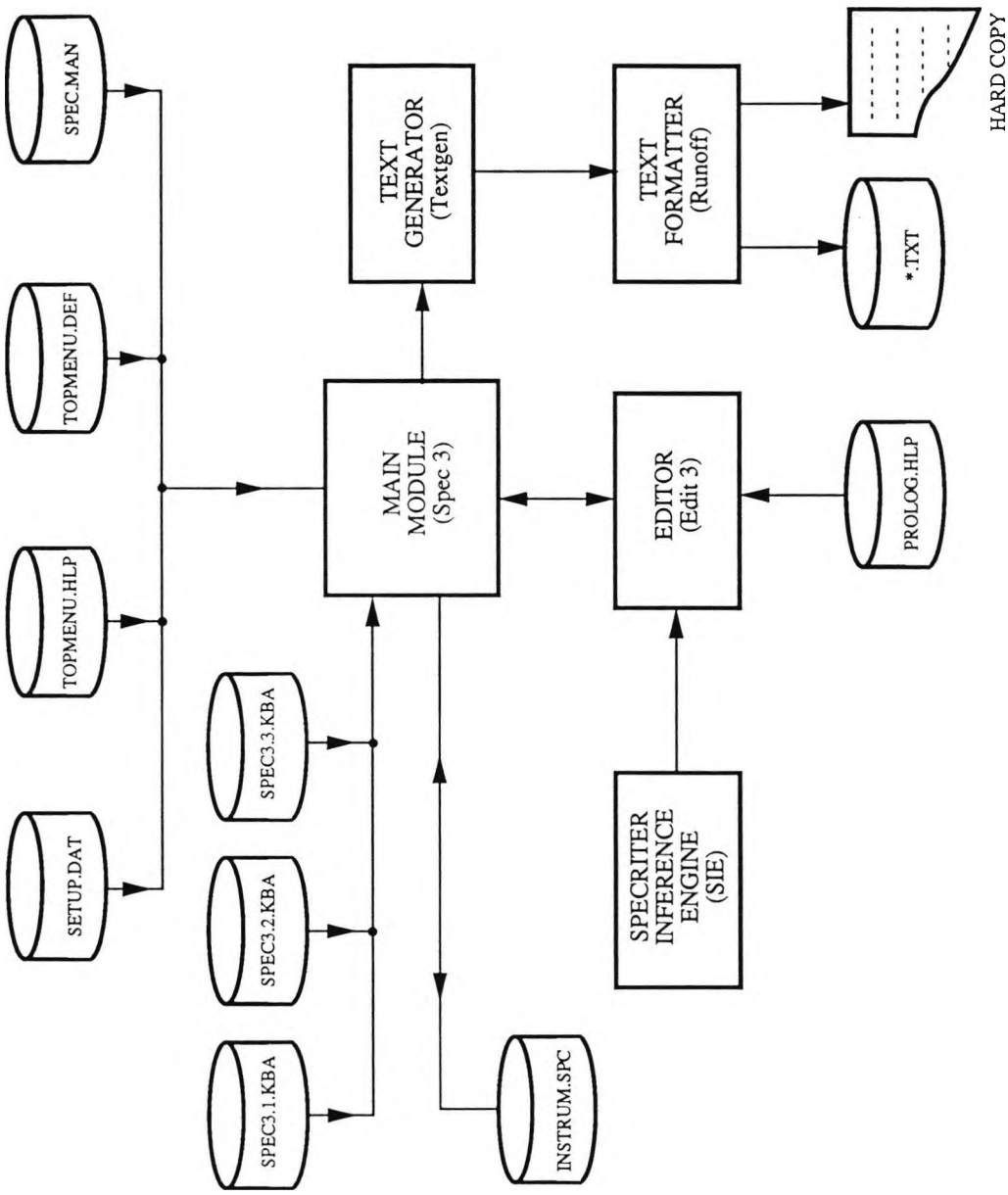


Figure 6-6 *Specriter 3* Structure

contains frames for measurand establishment and physical characteristics. The second contains electrical interface, environmental characteristics and performance while the last contains all the other topics listed in Figure 6-2.

The workfile concept from *Specriter 1* is retained in this version of the software. The system retains the name of the last specification under consideration in the setup file so that work can continue from where it was left off. This specification file is loaded next. This file comprises a list of predicates named "att" with two arguments; the attribute or slot name and a string representing its value. A process is then called to load these values into the knowledge base value sub-slots. If an unknown attribute is encountered it will be ignored and the process will continue with the next predicate. There is no need for the specification file to cover every slot; if a slot is missed the default value will be used. The benefit of this flexibility is that if the knowledge base is updated and frames and slots added or deleted, specification files created with earlier versions of the knowledge base will still be useable to the extent they are still applicable.

The knowledge base loaded with the specification of current interest, represents the limited formal specification of that measuring instrument.

The main menu help system files are now loaded. These files "TOPMENU.HLP" and "TOPMENU.DEF" contain the help message text and indexing information respectively. These files were created and can be edited with the Toolbox facility *Helpdef* (Borland, 1987). There is one help screen for each main menu topic.

Specriter 3 is now ready for operation and the main menu is called.

6.5.1.3 The Main Menu

6.5.1.3.1 Overview

Figure 6-7 is a screen image of the *Specriter 3* main menu with the Edit

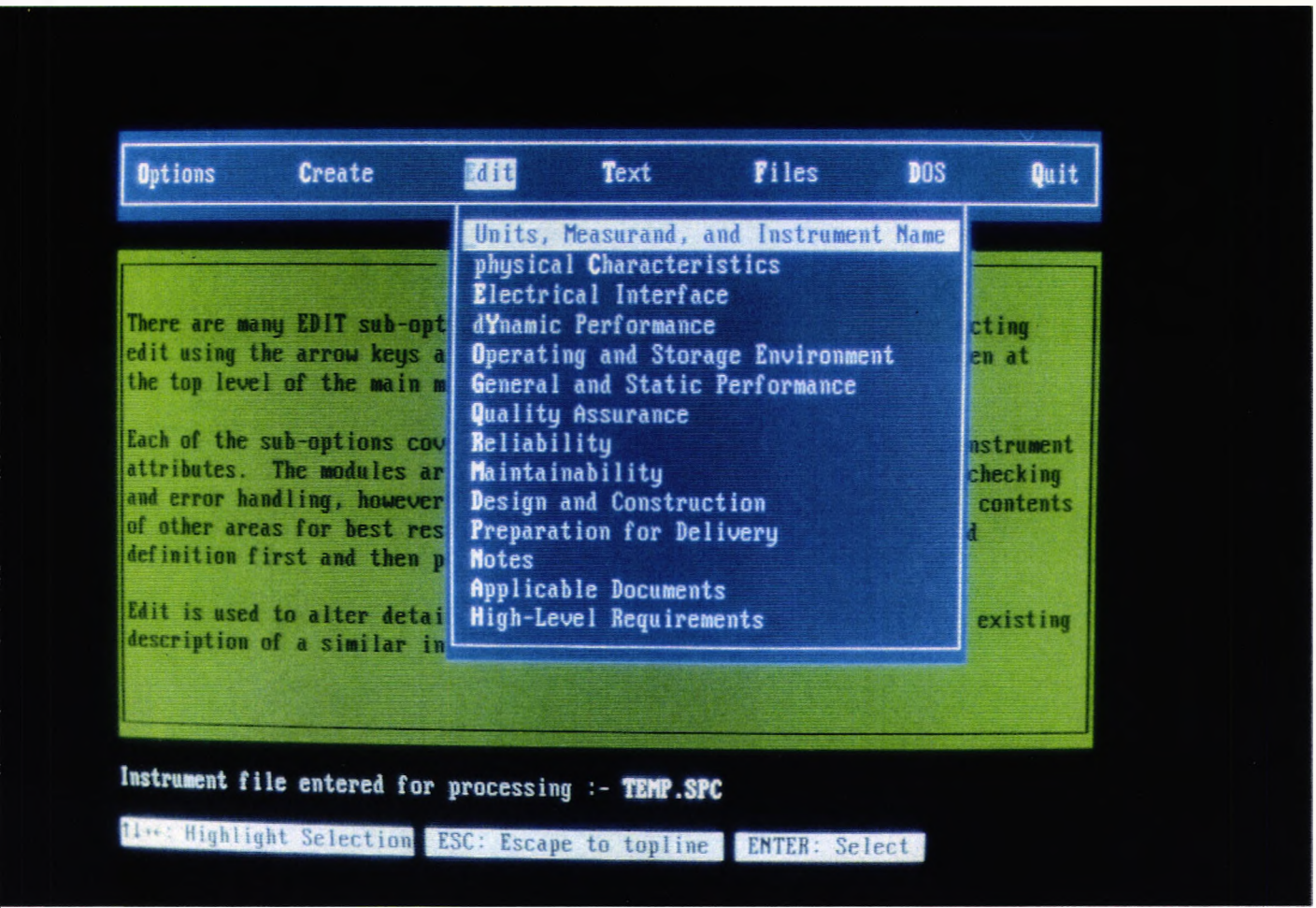


Figure 6-7

Specriter 3 Main Menu - Edit Sub-Menu Open

option pull-down menu exposed.

The main menu is made up of four windows. The first is the blue horizontal menu which lists the major options. There is no cursor, instead the active selection appears in inverse video. There are two ways to select an option and execute its function. The first is to press Enter which will execute the active selection. The menu item is chosen by moving the active selection with the arrow keys. The alternative is to type the high-lighted first letter of the menu topic required.

The next window is the help window, shown in green in Figure 6-7. The text is automatically updated when a new menu item is selected with the arrow keys. Below the edit window is a one-line borderless window which displays the current specification file name.

At the bottom of the screen is the function key window. The active keys are displayed in inverse video. Care has been taken throughout *Specriter 3* to ensure that the function key display line is applicable to the current human interface context. When the main menu is displayed, the status line shows that the arrow keys permit movement around the menu, the Escape key can be used to return to the top line, and Enter can be used to select the menu option currently highlighted.

6.5.1.3.2 Menu Options

There can be more than one level in the menu. Vertical sub-menus appear whenever an option is selected that possess sub-options. All the options will be discussed from left to right and from top to bottom. Whenever a task is complete, control is returned to the main menu.

6.5.1.3.2.1 Options

Specriter 3 starts with Options highlighted. Upon pressing Enter, the two

sub-options are displayed; On-line manual and Colours. The on-line manual option creates a full-screen window over the main menu and invokes the Turbo Prolog screen editor in Display mode and loads the manual file "SPEC3.MAN". In this mode, the editor allows the user to read the file but no changes are permitted. The function key line is updated to show the keys available in Display mode. Of particular value are the text searching functions which significantly reduce the time taken to find information shown in the index.

The Colours sub-option permits a user to change the colour of the windows to suit himself. A further sub-menu will appear when Colours is selected with the six windows in the following order: Menu, Help, Message, Error, Edit, and Status. The colour changing mechanism uses the Turbo Prolog "colorsetup" predicate which open a window which displays the 128 foreground and background text colour combinations displayable on an IBM PC. The current colour is boxed and a new colour can be selected simply by moving the box with the arrow key and pressing enter when finished.

6.5.1.3.2.2 Create

Create has no sub-options. This option has the same basic function as the Create option in Specriter 1. It is used to create a specification of a measuring instrument from scratch. Create first looks for a Level 1 frame called "High_level". If it is present, as it will be when the measuring instrument knowledge base is loaded, *Edit3* will be called and this frame displayed with all values set to "not specified". The user completes this screen and presses F10 to signal this. Create then invokes *SIE* to generate defaults for all other aspects of the instrument specification from the information given and the default generation rules. In instances where an attribute is not covered by these rules, then the default stored in the knowledge base is used. With many of the attributes sensibly completed, the user is now presented with the screen for each

topic in turn. These can be edited in exactly the same way as they can when in Edit. Upon completion of this sequence of screens, the first draft of the specification is complete.

6.5.1.3.2.3 Edit

The Edit option permits each specification topic to be edited. When the Edit option is selected, a sub-menu is produced which displays a list of Level 1 frames. It is important to note that this sub-menu is not hard programmed but is in fact read from the knowledge base. The values of the attributes in each frame can be edited including those in High_level. The intelligent default process can be re-run from the High_level frame. This option is covered in greater detail later during the discussion of *Edit3* and the use of *Specriter 3*.

6.5.1.3.2.4 Text

Text has three sub-options: Display, Print, and Format. When Display is selected, *Textgen3* is invoked and the finished document created and displayed using the Display facility in the same way as the on-line manual is displayed. The Print sub-option directs the document to the DOS printer spooler for printing rather than displaying it on the screen. The last sub-option, Format, permits the format of the finished text to be altered by editing the format control options in exactly the same way as *Specriter 1*.

6.5.1.3.2.5 Files

The Files option has five sub-options: Load, Directory, Copy & Load, Rename, and Erase. The Load sub-option is intended to be used whenever a previously created specification is to be edited. On selection of Load, the current specification file will be saved and a small window is opened prompting for a file name with the ".SPC" extension. The file

to be loaded can be entered here with or without the extension, or alternatively, if Enter is pressed without a file name being given, a directory of suitable files is displayed and the required file can be selected using the arrow keys and Enter. Checking is provided to prevent non-existent file names being accepted. The Directory option simply produces a directory of all the instrument specification files in the current directory. Copy & Load allows the selection of the required file using the same method as Load and then prompts for a file name to copy it to. The name offered is thoroughly checked to verify that it is a valid DOS file name before the file is created and loaded. This option is intended to be used to copy existing files, in particular future library specifications, to the workfile for customisation. Rename simply changes the name of a given specification file while Erase can be used to delete unwanted specification files.

A full range of file handling options is provided so that users do not have to resort to using the operating system for these tasks. All file operations support full directory paths so that it is possible to access files not in the current directory or indeed the default disk drive. This means that the specification files can be held on a floppy disk and need not become immersed in the numerous *Specriter 3* run-time files.

6.5.1.3.2.6 DOS

The DOS option creates another copy of DOS over the top of *Specriter 3* and places user control there. *Specriter 3* is still loaded and can be returned to at any time by typing EXIT at the DOS command prompt. This option allows users to execute DOS commands and run small external programs without incurring the time delay associated with exiting *Specriter 3* and subsequently re-loading it to continue work.

6.5.1.3.2.7 Quit

This option saves the status of the current session in the specification and setup files and then terminates the program.

6.5.2 *Edit3*

6.5.2.1 General

The *Edit3* module is the heart of *Specriter 3*. It comprises the Edit and Create views, the *Specriter 3* Shell, and all the human interface functions for specification editing. The necessary environment for *Edit3* is made available by *Spec3*. Thus when *Edit3* is called, the knowledge base is already in place and loaded with the attribute values for the instrument under consideration.

6.5.2.2 Initialisation

Edit3 is passed the name of the Level 1 frame to be edited. However, it is the terminal or leaf frame which characterises the topic and this must be determined before going further. This frame is found by a predicate which traces the branching options until the leaf frame is found. An additional function performed during this search is to assert into the database the list of parent frames for the frame to be displayed. This frame list is used repeatedly by the Prolog backtracking mechanism to achieve inheritance of parent slot information.

Editing screens are composed of text messages and active fields, one for each attribute to be specified. The search for screen text starts first. These text strings are inheritable and as such it is necessary to collect them not only from the current leaf frame but all the parent frames as well, and place them in the screen database for display. This is achieved using Prolog's inherent backtracking mechanism. Next, the list of active fields is extracted and placed into the screen database in the same way. The distinct advantage in using

inheritance in this application is that all the information contained in the parent frames need not be repeated. Providing some fields have been found, *Edit3* goes on to create the function key list. This slot is also inheritable but the search through the frame hierarchy ceases when the first function key definition slot is found. A utility predicate is then called which takes the function key list as a parameter and creates the status line at the bottom of the screen. The frame placed at the root of the frame tree, called "top", has some function keys specified, so there will always be a function keys list to display.

The screen handler sub-module is invoked next. This complex program is a heavily modified version of *Vscrhnd* from the Turbo Prolog Toolbox (Borland, 1987). It commences by reading the screen database loaded earlier, and proceeds to write the text strings and the active fields to the screen. The screen handler is a continuous process which continues until a key is pressed.

There are five distinct types of active fields: entry, list, database-assisted, option, and edit (See Appendix 2). To aid the user, each is displayed in a unique colour. For the first four, *Edit3* looks up the appropriate colour and creates a field on the screen in that colour containing the field value. If the string containing the value is longer than the space available, the undisplayable portion is simply truncated.

Edit slots take more effort. A small window is displayed on the screen to show the contents of user entered paragraphs. If these are re-written continuously, screen update is very slow and key response sluggish, hence a flagging arrangement was adopted. The first time the screen is displayed, the display window is created and the text entered, after which that paragraph is flagged as "displayed".

The screen handler now loops, updating the screen until a key is pressed. This continuous activity permits time-dependant strings to be displayed such as a clock, should the need arise. Figure 6-8 is an editor screen chosen to show the use of a diverse selection of fields.

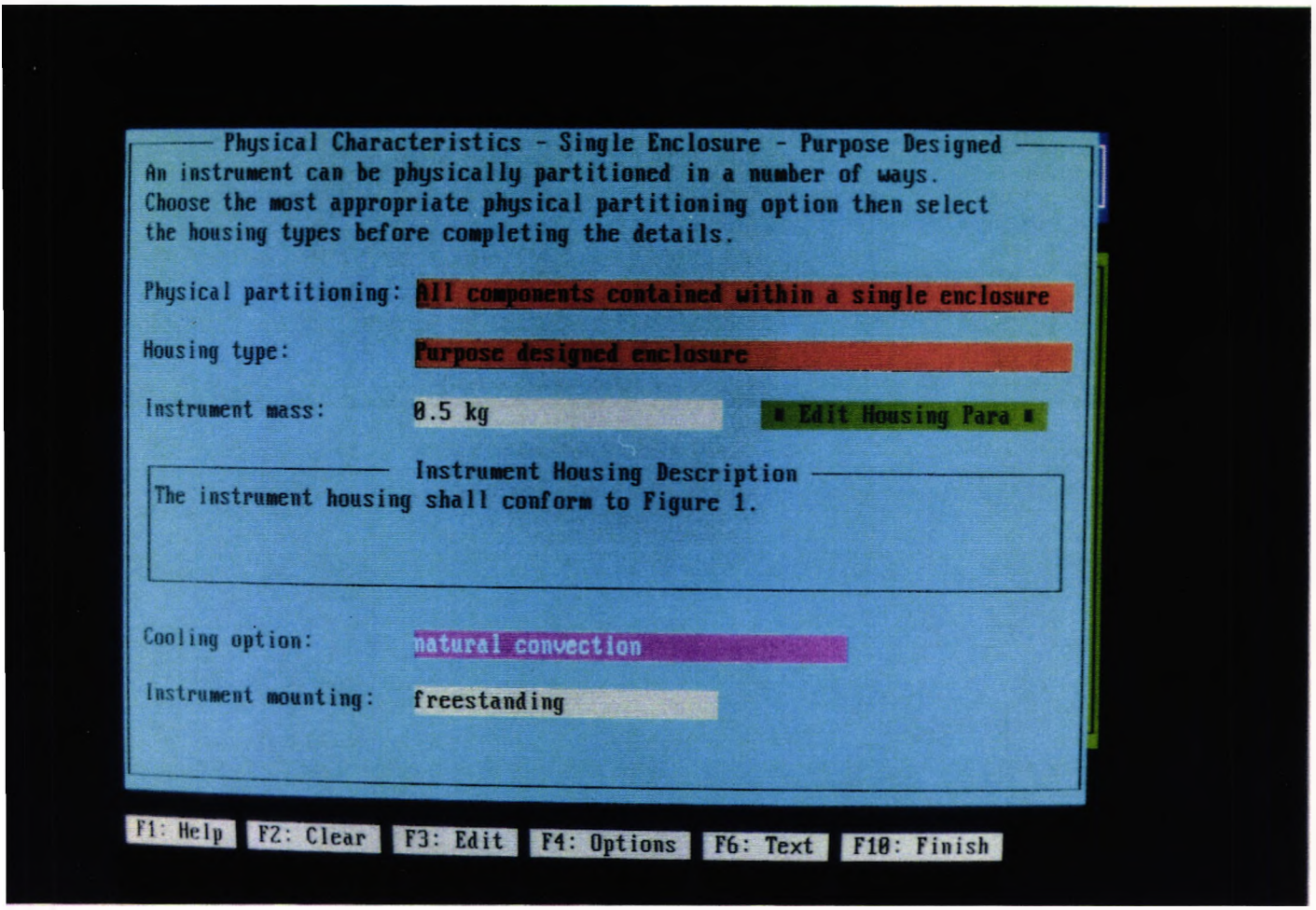


Figure 6-8

A Typical *Specriter 3* Editor Screen

6.5.2.3 *Edit3* Functions

When a key is pressed, the looping process is suspended and the key handler commences and executes the required action. Actions can be divided into two broad classes: attribute editing, the act of changing the value of an attribute, and function key handling which covers a range of tasks.

Some of the complexity of the screen handler is caused by the virtual screen capability built into it. This facility permits much larger screens to be built than can be displayed at any one time. When operating in this mode, the screen handler is analogous to a window which can slide over a large picture. Virtual screens are used extensively to specify physical characteristics as the large number of items to specify could not be made to fit in a single screen.

6.5.2.3.1 Function Keys

6.5.2.3.1.1 Cursor Movement Keys

The cursor can only reside on the first space of the active fields. It can be moved to the field of interest by the arrow keys. Other keys which move the cursor are Home and End which takes the cursor to the top and bottom of the virtual screen respectively, and Page Up and Page Down which move to the top and bottom of the screen being displayed. All fields can be found, however, by scrolling the screen with the arrow keys.

6.5.2.3.1.2 Function Key F1

The function key F1 calls up a help message. When F1 is pressed *Edit3* first creates a message display window over the top of the screen being edited. It then proceeds to search the database for a help message. The search uses the same inheritance method employed by the function key status line handler in that it searches the current frame first and backtracks through the parent frames until one is found. The first

message found terminates the search. Once again, as there is a help message in the top frame so there will always be one to display. The Display facility used for the on-line manual is used to display this message, so the length of the message is not constrained by the window size.

6.5.2.3.1.3 Function Key F2

F2 is used to set the values of all the attributes on the current screen to the empty string with the exception of the branching options which determine which screen is being displayed. This key is useful when making large changes to a specification and it needs to be clear which attributes have been changed and which have not.

6.5.2.3.1.4 Function Key F3

F3 opens an attribute for editing. It is useful when wishing to enter an uncatered-for response for database-assisted and list fields. This is because the Enter key displays the database or list as this is the preferred entry mode, for these field types.

6.5.2.3.1.5 Function Key F4

F4 calls up the options available for database-assisted, list and option fields. It is not strictly necessary as this can be done with the Enter key. However, it is included to indicate that option selection fields are present in the screen.

6.5.2.3.1.6 Function Key F5

F5 invokes the Specriter Inference Engine (*SIE*) using the checking rules for the frame. A dialogue window is created over the top of the screen being displayed to permit interaction with the user, which is usually just

display of messages. *SIE* is described later.

6.5.2.3.1.7 Function Key F6

F6 invokes the local text generation process. The process starts by collecting the text fragments for the current frame and all the parent frames using the same inheritance mechanism employed for screen text and active field collection. The text fragments are then placed in a single string starting with the parents first, as this is the logical way a specification reads, i.e. the more general statements come first. This string is then searched for attribute names. As the string has to be searched character by character for the escape character '#', which surrounds the attribute names, the performance of this process is crucial on a finite-memory machine such as an IBM PC. If the process stacks for every character, the limit of around three to four thousand stack frames can easily be exceeded and memory overflow result. By careful design, it has been possible to employ tail recursion elimination which means that each recursive loop of the process can use variables held in registers and stacking is avoided.

After the replacement process, a window is created over the screen being displayed and the resultant text displayed, once again using the "display" predicate. The user can then check instantly whether the responses given will fit into the stored text for that frame. This technique of using the user to perform text checking, neatly avoids the complexity of natural language processing. To keep local text generation fast and simple, there is no paragraph sorting, justification, or other tiding up performed.

6.5.2.3.1.8 Function Key F10

F10 terminates *Edit3* and returns control to the main menu. The specification is left in the state displayed on the screen. In general, pressing F10 can be interpreted as "finish gracefully, accepting changes"

whereas Escape can sometimes abort changes, for example during attribute entry.

6.5.2.3.1.9 Escape

Escape is used to exit processes such as sub-menus, text display, attribute editing etc. As stated above it is better to use F10 to guarantee that changes will not be lost.

6.5.2.3.1.10 Enter

The Enter key is used to initiate and terminate processes such as line entry and option selection. It is the most natural way to use Specriter as the default function, for Enter is always the preferred option in any circumstance.

6.5.2.3.2 Attribute Editing

6.5.2.3.2.1 Standard Entry Attributes

A standard entry attribute can be recognised by its characteristic field colour; black writing on a white background. To enter a new attribute simply position the cursor in the field and start typing. On pressing the first key, an input window containing a prompt, will open on the screen over the field position. To edit an attribute, press Enter or F3. The input window will now contain the current entry after the prompt. In either case, the area occupied by the previous entry operates as a text editor. The arrow keys enable movement along the string and Home and End can be used to find the beginning and end of the text. To insert new characters into the string, press Insert before typing, otherwise the line editor operates in overstrike mode. The line input editor can accept strings up to 2^{16} characters long so there is no artificial barrier to adequate expression. To cater for this, the line input driver is another

virtual length process which just displays a portion of large strings.

6.5.2.3.2.2 List Entry Attributes

List attributes, identified by their white on purple appearance, can be edited in two ways. The first is to use free form entry as described above. This is invoked by F3. The preferred method is to select one of the items in a predefined list stored in the knowledge base. Selection from the list will not only provide a sensible answer to the question but will also enable automatic reasoning to take place if there is any checking or default generation associated with that attribute. To display the list, press Enter or F4. The list will be displayed with the previous value highlighted in inverse video. A new choice can be selected either by positioning the highlighted region over the desired choice with the arrow keys and pressing Enter, or alternatively using the "hot key" highlighted in the desired option.

6.5.2.3.2.3 Database-Assisted Entry Attributes

Database assisted entry attributes, identified by their black on orange colour, can be modified in exactly the same way as list entry attributes. The only difference being that because the list is derived from an external database, hot key selection is disabled as many options may have the same first letter. If the list is long, it will extend beyond the limits of the window created to display the list. Once again, the list selection utility resorts to a sliding window technique to overcome this potential problem. The database entries are sorted into alphabetical order to aid selection.

6.5.2.3.2.4 Option Attributes

Option attributes are the gateway to the *Specriter 3* frame hierarchy structure. When the value of one of these options is changed, the leaf

frame changes. A new option is selected in the same manner as a list attribute option. After a selection is made, the screen handler terminates. It is quite possible to change an option which is part-way up the tree, therefore, it is necessary to search for the correct leaf frame and then repeat the initialisation process and re-start the screen handler to display the selected frame.

6.5.2.3.2.5 Edit Attributes

Edit attributes are changed by placing the cursor on the black on green edit field and pressing either Enter or F3. The display window will turn into an edit window and the control is passed to the Turbo Prolog editor. At this stage it is usually useful to enlarge the edit window to fill the screen using F5, as directed by the status line. Help on the numerous editor commands is available by pressing F1. Complete information is available in the Turbo Prolog User's Guide (Borland, 1988a) but this should not prove necessary. Completion of editing, and incorporation of the result into the knowledge base, is achieved by either pressing F10 or Escape.

6.5.3 The Specriter Inference Engine (*SIE*)

It is impossible to predict the nature of all the possible rule-based systems which may be needed, so in order to permit maximum generality it was decided to represent the rules in Prolog. The rules could then be interpreted using the inherent backward chaining mechanism of Prolog or, if necessary, by the wide range or other inferencing mechanisms which can be built in Prolog, for example forward chaining. The interpretation of rules directly requires *Specriter 3* to support meta-programming.

Although meta-programming is not available as a standard feature of Turbo Prolog, a meta-interpreter can be built using the language. This is the subject of Appendix K of the Reference Manual, (Borland, 1988b). Extensions to the

interpreter were necessary to permit operations on real numbers, to extract numeric values from strings, and to interact with the attribute values of the knowledge base.

Figure 6-9 is a complete list of the predicates and operators supported. The interpreter was further modified to look for its input code in slots in the knowledge base as opposed to user control. The resulting module has been named the *Specriter Inference Engine (SIE).*

Using this module it is possible to extract the value of any attribute stored in the knowledge base, perform processing operations available from both Prolog and the purpose written predicates, (such as extracting the first real number from a string), reason about the set of values extracted, write messages to a window, and perform modifications to the knowledge base. In fact, because the interpreter can interpret a whole language rather than just rules in a pre-defined format, its potential uses go well beyond what is required in this application. If it should be necessary, it is possible to modify the interpreter to gain access to any of the predicates in Turbo Prolog or to linked functions written in assembler or C.

6.5.4 *Textgen3*

Textgen3 is invoked from main menu with either of the Text sub-options; Display or Print. Textgen commences by compiling a complete list of applicable paragraphs from the knowledge base. This is achieved using Prolog backtracking as follows. The text collection algorithm finds a Level 1 frame from the knowledge base, finds the active leaf frame by searching through the options in the same way *Edit3* does to determine the frame to display, then collects all the text fragments up through the sub-trees in an identical manner to the local text generation. Once all the text is collected for the topic, the algorithm backtracks, and finds another Level 1 frame until all the paragraphs are collected. This compendium is then scanned to find and replace attributes with their values using the identical optimised algorithm previously described under local text generation.

PREDICATE	FUNCTION
true	Success
fail	Prolog fail
repeat	Succeeds forever
write(Term*)	Writes a list of arguments
nl	Outputs a carriage return and line feed
display(Term*)	Outputs a functor in prefix notation
read(Term)	Read a term
readln(Line)	Read a line into a string
readchar(char)	Read a character
retract(Term)	Retract a term
tell(Filename)	Redirect output to a this file
telling(Filename)	Return the current output file
told	Close the current output file
see(Filename)	Redirect input to this file
seeing(Filename)	Return the current input file
seen	Close the current input file
term =.. list	Prolog univ; conversion between a term and a list
arg(N,Term,Argn)	Unify Argn with the nth argument of Term
functor(Term,Functor,Arity)	Return functor and arity of Term or builds a new one
clause(Head,Body)	Returns clauses from the database
concat(string,string,string)	Concatenation of strings
str_int(string,int)	Conversion between a string and an integer
str_real(string,real)	Conversion between a string and a real
strg_real(string,real)	Extracts first real number from string
str_atom(string,Atom)	Conversion between a string and an atom
Integer is Expression	Evaluation of expressions
Term == Term	Testing for true equality
Term \== Term	Not true equality
Term = Term	Unify terms
Term \= Term	Test whether terms unified
Term < Term	Less than (real numbers)
Term > Term	Greater than (real numbers)
Term =< Term	Less than or equal (real numbers)
Term >= Term	Greater than or equal (real numbers)
Term >< Term	Different evaluated values (real numbers)
integer(Term)	Is Term an integer?
var(Term)	Is Term a free variable?
novar(Term)	Is Term bound?
time(Hour,Min,Sec,Hundreds)	Returns the system time
scr_char(Row,Col,char)	Print a character at a selected position
char_int(char,int)	Conversion between characters and integers
consult(Filename)	Consult named file
reconsult(Filename)	Reconsult named file
save(Filename)	Save a file
op(Priority,Assoc,Op)	Returns operators or changes operators
Goal, Goal	And
Goal; Goal	Or
not(Goal)	Negation
!	Cut
call(Goal)	Call
assert(Rule)	Asserts rule into database
asserta(Rule)	Asserts rule at front of database
assertz(Rule)	Asserts rule at rear of database
value(Attribute,Value)	Extracts the value to the measuring instrument attribute
change(Attribute,New_value)	Assigns a new value to a measuring instrument attribute

Figure 6-9 Standard Predicates Implemented in the *Specriter Inference Engine*

The text is collected in any order. Hence there is a need to sort the ensemble into paragraph order. The need for a paragraph sort algorithm has the advantage of permitting non-contiguous paragraphs to be handled in one frame. To permit easy sorting, each paragraph is stored as a list commencing with the paragraph number.

The sorting algorithm used to place the paragraphs in order is a modified quicksort. The algorithm design is complicated by the need to take into account the significance of the dots as well as numeric order, for example, 3.3.20.8 must appear before 3.4.2.6.

Final text processing in *Specriter 3* is still performed by *Runoff* (Blaise 1986) primarily because it has been a low priority to incorporate its function into a Prolog program. *Runoff* takes a formatted input file complete with control codes and creates a processed output file. Thus, *Textgen3* opens a disk file named "WORD." into which it writes heading information, the document title, and the processed text. *Textgen3* concludes by calling *Runoff* to process WORD. via an external system call.

The resultant text file "WORD.DOC" is copied to a file bearing the specification name and the ".TXT" extension, for example, "FLOW.TXT". This file is either displayed or printed depending on the main menu option used to start the text generation process.

6.6 The *Framedt* Facility

6.6.1 Background

Any knowledge-based system that has the desirable characteristic of having a distinct and separate knowledge base, must possess a mechanism for creating and maintaining that knowledge base. There are three main methods used, and each will be discussed below.

The first is to create a knowledge base language and describe the knowledge base using this langauge. The knowledge base can then be edited using a conventional text editor. The knowledge base source code is then synonymous with the program source code written in a conventional computer language. In order to load such a knowledge base into the system, it is usual to employ tools similar to those found at the front end of a compiler, namely a scanner to break the input stream into recognisable tokens and then a parser to extract the symbols and variables of the language and hence the semantic content of the input. This is the technique employed by Rosin (1988) to load his frame-based vision system. It is suitable for knowledge bases that are changed infrequently. However, construction of the tools is an extended task, especially if comprehensive error trapping is to be incorporated. Such error detection is quite necessary as it is very easy to introduce errors into the knowledge base during the creation and editing processes. In any event, this technique is not all that useful for *Specriter* because the screen layouts will have to be done on paper beforehand. In fact the knowledge base may just as well be written in Prolog and compiled with the rest of the programs.

The second knowledge base creation and maintenance technique, is to incorporate an interactive module into the run-time system. This method is commonly used in proprietary expert system products. The knowledge base can then be readily altered and the system run immediately to observe its behaviour. *Specriter*, however, seeks to create, edit, and print, specifications not knowledge bases. Hence there is little reason to combine the knowledge base editor with the run-time system.

The last alternative is to build a stand-alone tool to create and maintain the knowledge base. The primary advantage of this technique is that the number of errors introduced during editing can be greatly reduced. Low-level syntax errors in the knowledge base can be completely eliminated simplifying the run-time system. Another advantage of a completely separate tool is that it does not compete for memory, a valuable commodity on an IBM PC.

Of the three options, the approach selected was to develop a stand alone system

with an integral screen layout tool.

6.6.2 *Framedt* General Description

The *Specriter 3* knowledge base editing facility is a stand alone program called Framedt for FRAMe EDiTor, reflecting its frame by frame operation. Conceptually *Framedt* is a top-level view of the layered *Specriter 3* system shown in Figure 6-1. In practice, Framedt is completely separate from Specriter 3 so the underlying layers are duplicated to permit interaction with the active knowledge base. Figure 6-10 illustrates the structure of *Framedt.*

The main module creates the Framedt environment, and specifically handles the interaction with the knowledge base, with the exception of the displayable fields. There are two control screens; the main screen shown in Figure 6-11 and the frame editing screen Figure 6-12. The *Framedt* screens were created, and can be edited, using the Turbo Prolog Toolbox *Scrdefn* program (Borland, 1987). *Framedt* uses the same human interface concepts as *Specriter 3.*

ScreenDef, the screen definition module permits the displayable fields to be entered and moved. This module is capable of working on screens which extend beyond the physical limits of the video display unit by using the sliding-window virtual screen technique described earlier.

SIE, the *Specriter Inference Engine,* is identical in every respect to the module used in *Specriter 3.*

6.6.3 Using *Framedt*

Framedt starts up the same way as *Specriter 3*; the setup file is read to find the names of the knowledge base files, and the last frame accessed by *Specriter 3.* Then the three knowledge bases are scanned and only the one containing the frame is loaded. This automatic loading feature aids frame development by speeding the edit and test cycle. The main screen is then displayed.

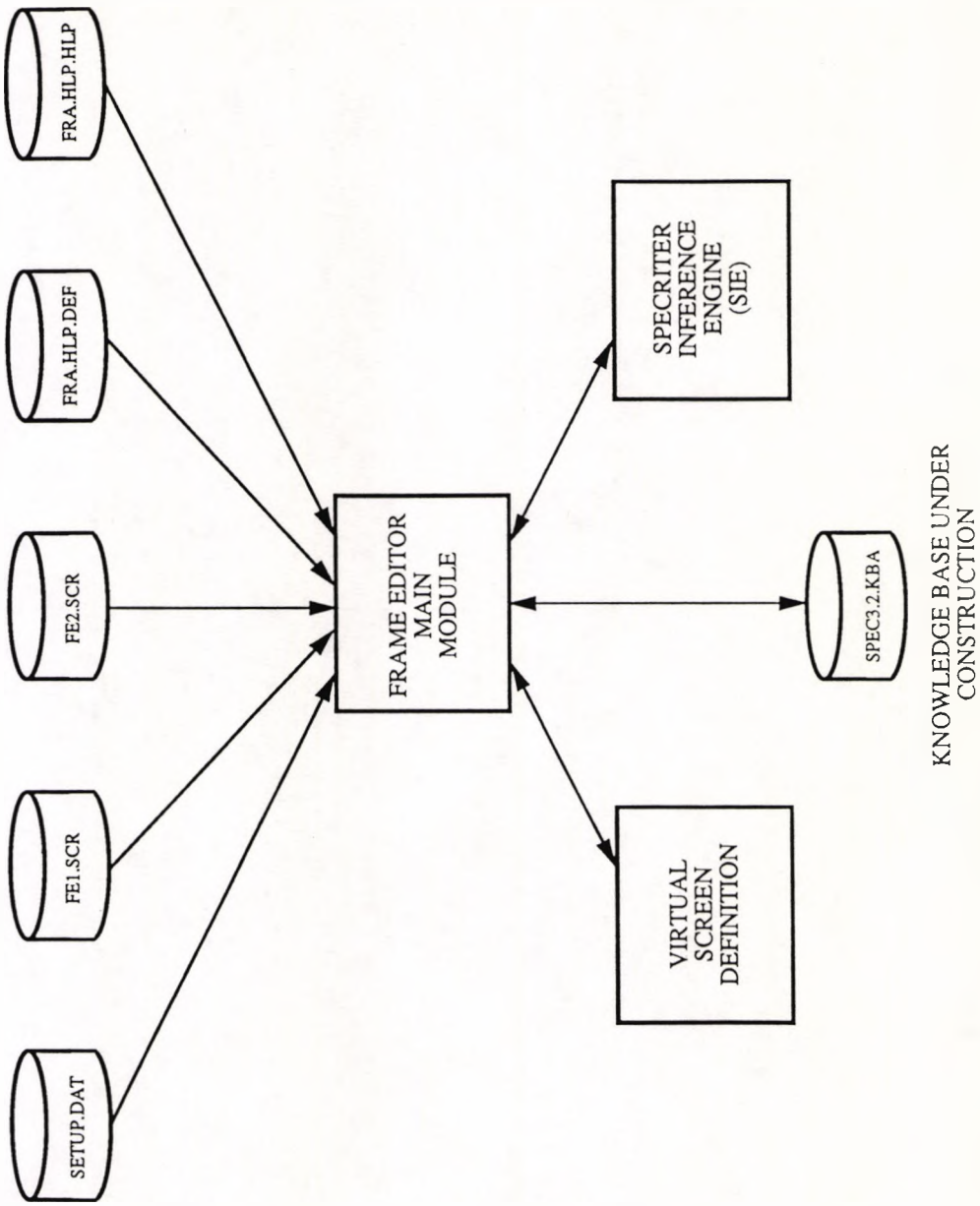


Figure 6-10 The Structure of *Framedt*

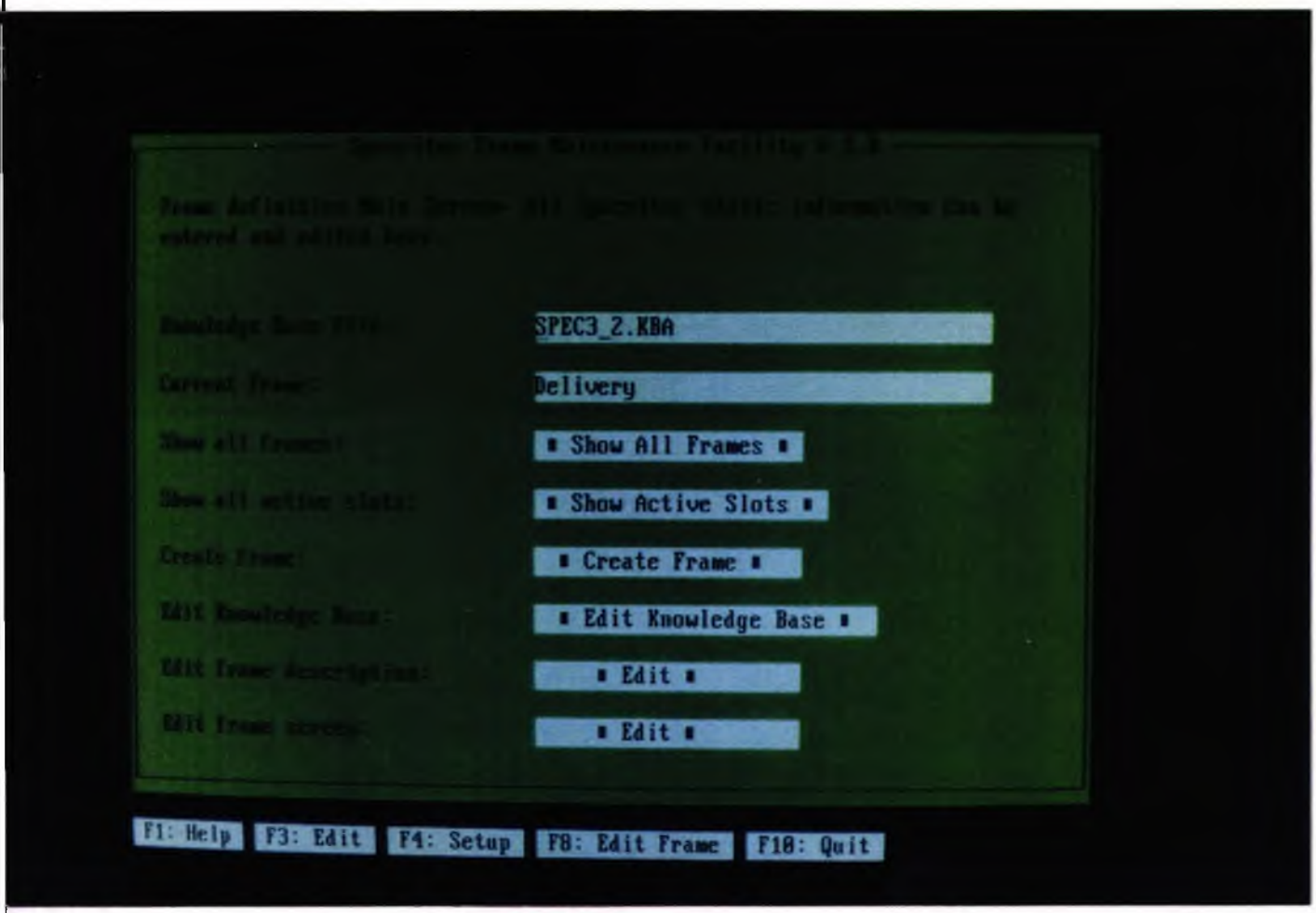


Figure 6-11

The *Framedt* Main Screen

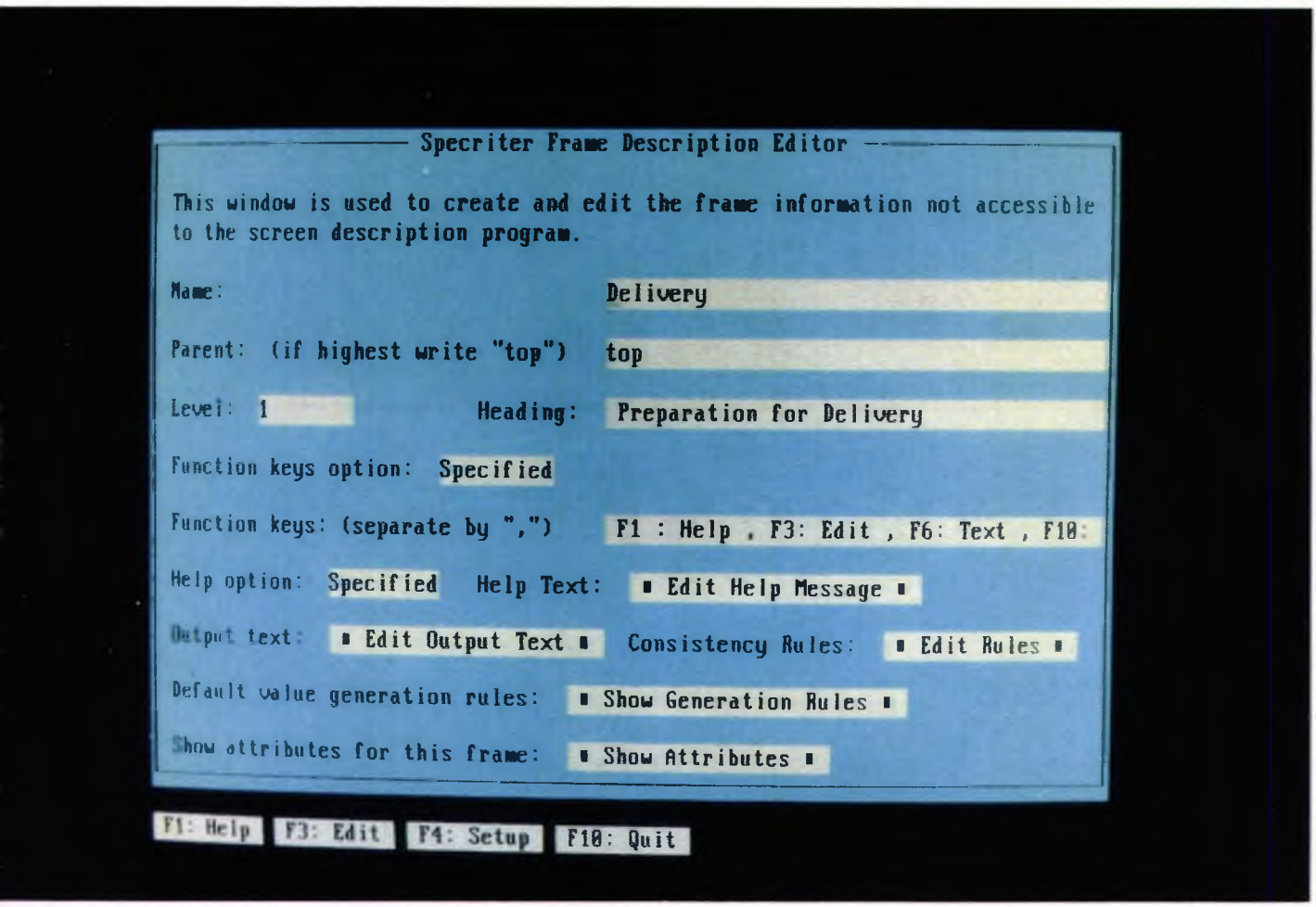


Figure 6-12

The *Framedt* Frame-Editing Menu

The main screen shown in Figure 6-12, provides the following functions:

(1) Knowledge base loading using the same file loading routine developed for *Specriter 3*.

(2) Frame selection from a list created from all the frames contained within the knowledge base.

(3) Frame display showing parent frame.

(4) Display of all slots in the knowledge base showing parent and slot type.

(5) Level 1 frame creation facility.

(6) Text editing of the knowledge base file.

(7) Editing of the non-displayable slots of the current frame using the frame editing screen.

(8) Editing of the displayable slots using the Virtual Screen Definition (VSD) module.

The user is free to use any of the main module functions in any order. A typical operation would be to correct, say, a spelling mistake on the current frame. To do this, the cursor is moved to the last field of the screen and the Enter key pressed. VSD is called, a window opened, and the current frame fields displayed. None of the frames from either parent or child frames are displayed, these can only be edited when they are nominated as the current frame. The spelling error is amended by editing the text fields as if they were in a line editor, i.e., by over typing, or if Insert is pressed, inserting new characters. New attribute fields can be added by positioning the cursor on the beginning of the proposed field and pressing F3. The field is then drawn on the screen with the arrow keys. Pressing Enter completes that process and then a sequence of windows appear requesting

the information needed to specify all the information for the type of field requested. See Appendix 2 for details.

When displayable field editing is complete, pressing F10 returns control to the main menu. Pressing F8 displays the Frame editing menu in place of the main menu, see Figure 6-12. All non-displayable fields can be edited from here and the functions available can be ascertained from the menu.

Editing the heading can be achieved simply by changing the string shown.

Function keys and help messages can be inherited from a parent frame or specified in each frame. The function key and help option fields enable slots, once defined, to be removed if required. The function key definition is entered as a single string separated by commas. This is converted into the internal list structure when the slot is saved.

The help message is straightforward to edit. On selecting this function, a window identical in size to the one displayed in *Specriter 3* is opened with the Turbo Prolog editor activated. The user has the full power of the editor available to create or modify the message, which can extend beyond the borders of the window in both dimensions. The output text is edited in the same way.

The "show attributes" facility displays the attributes from the current frame and its parents, i.e. all those that would be displayed in the *Specriter 3* editor screen. This is handy, as these names need to be referenced by the output text and the generation and checking rules.

The generation rules are edited by placing the cursor on the appropriate field and pressing Enter. A window opens, with the editor enabled, containing the existing rules. The generation rules are intended to be used to create intelligent defaults given the information available from the "High_level" screen. There is however, no restriction on what can be undertaken and all the predicates of Figure 6-9 are available. After editing, SIE is called, which executes the rules, and displays the

result to the user. The editing and execution cycle can be repeated until the desired function is achieved. Consistency rules are edited in the exactly the same way.

Control is returned to the main menu with F10 and pressing F10 a second time will save the edited knowledge base and exit the program. The results of the editing session can then be observed by running *Specriter 3*.

6.7 Conclusion

This chapter has described the evolution, design and implementation of the structured specification generation system, *Specriter 3*. The system is based on a novel application of frames where the entire context of the topic being specified is described by the leaf-frame in use. The numerous attributes which characterise the specification are represented as slot values combined into the frames, for all functions other than storage. This represents a complete implementation of the knowledge representation proposed for the new-generation *Specriter* in Chapter 5.

Specriter 3 can be thought of as being analogous to an expert system shell. It is domain independent and contains all the necessary components to solve a range of structured documentation generation problems. It has been characterised by the addition of a measuring instrument knowledge base to create requirements specification for measuring instruments.

A comprehensive knowledge base editing tool was constructed to create and edit knowledge bases for *Specriter 3*. This fully functioning facility, named *Framedt*, has been described. It is noteworthy that *Framedt* was, in fact, used to create the measuring instrument knowledge base.

The next chapter shows how the *Specriter 3* combined with the measuring instrument knowledge base can be used to generate a requirements specification for a measuring instrument. That chapter contains a review of the achievements of *Specriter 3*.

Chapter 7 - Using *Specriter 3* to Produce a Measuring Instrument Specification

7.1 Introduction

One of the features of *Specriter 3*, is that the nature of the documents it produces is completely characterised by the knowledge base. This means that the description of *Specriter 3*, given in the last chapter, will have given little insight into how well the design aims for a specification generation tool, enumerated at the beginning of Chapter 4, have been met. This chapter addressees that issue by discussing the performance and use of *Specriter 3* when loaded with the measuring instrument knowledge base.

The chapter commences with a brief discussion of how the human interface chosen for *Specriter 3*, helps to meet the design drivers derived from the design aims.

The bulk of the chapter describes the process of creating and editing an example measuring instrument requirements specification using the Create and Edit facilities of *Specriter 3*. In this discussion, the content of the knowledge base and the degree of automation provided is also covered. This descriptive method of dealing with the knowledge base was thought to be more effective than providing a listing of the complete knowledge base as an appendix of around 200 pages.

Finally, the last section is devoted to comparing the software described in this and the last chapter, to the original design aims. The effectiveness of the third generation of the software in overcoming the design and implementation limitations of the first two generations also receives attention.

7.2 Human Interface Concepts

Any large software entity takes some time to learn to use effectively. One of the aims of the research project was to design a system for infrequent specifiers. Hence it became a design goal to produce a system that takes minimal time to learn and is simple to use. The key to ease of use is the design of the human interface and the quality of the help available (Galitz, 1985; Borland, 1987). It is also preferable to limit the need to access the user's guide. This ideal is hard to achieve and most software products have manuals that extend to hundreds of pages. It is, however, believed that it should be possible to use *Specriter 3* without having to undergo the type of training course usually necessary for modern word processors, spreadsheets and computer modelling and simulation packages.

The main menu is the user's introduction to *Specriter 3*. Users familiar with PC software will adapt quickly to the horizontal menu which is similar in concept to Borland language compilers and commercial products, such as the Lotus 123 spreadsheet. All the screen designs follow the guidelines of Galitz (1985). The colours, which are customisable using the Options/Colours command, are used to help the user follow the context of the task. Thus menus, edit screens, help screens, status lines, error messages, and warning messages all appear in different colours. Similarly, the fields of the edit screens are colour coded. The default colours were chosen to provide good contrast and to be easily distinguishable from one another.

7.3 Generating a Measuring Instrument Specification with *Specriter 3*

This sub-section traces through the process of creating, editing and printing a measuring instrument specification using *Specriter 3*. The measuring instrument chosen to illustrate the process is a thermometer intended to be part of the thermal control sub-system of an interplanetary scientific satellite.

7.3.1 Specification Creation

In this example, a new specification is to be created rather than an existing one modified, so the Create facility is used. Accordingly, this option is selected from the main menu. Create starts by saving the current specification and then sets all the attribute values in the knowledge base to the pre-defined standard defaults. Subsequent functions are described in greater detail below.

7.3.1.1 File Name Elicitation

The Create utility's next action is to open a window and request a file name for the specification to be created. A comprehensive checking routine is employed to ensure that the name will be acceptable to the DOS operating system. This means that the name given must start with a letter and can be no more than twelve characters, where the last four characters must be the predetermined extension ".SPC" which denotes a specification file. This extension will be added automatically as required. If the name offered is not correct, the user is given further instructions and re-prompted for a another.

7.3.1.2 The High Level Requirements Screen

The screen called "High_level" is loaded next. This screen is shown in Figure 7-1 with the intended operating environment field open for selection. The attributes to be completed in this screen are not intended to be used directly in the finished specification. Their purpose is to capture the high-level user requirements which would normally be included in other contractual documentation or taken for granted. This information can then be used to provide intelligent default generation for many attributes and input for consistency checking. As nothing is known about the instrument at this stage, each of the high-level attributes is set to "not specified". Like all *Specriter 3* screens, this one can be completed in any order, but for convenience, the discussion will run down the list in the order displayed.

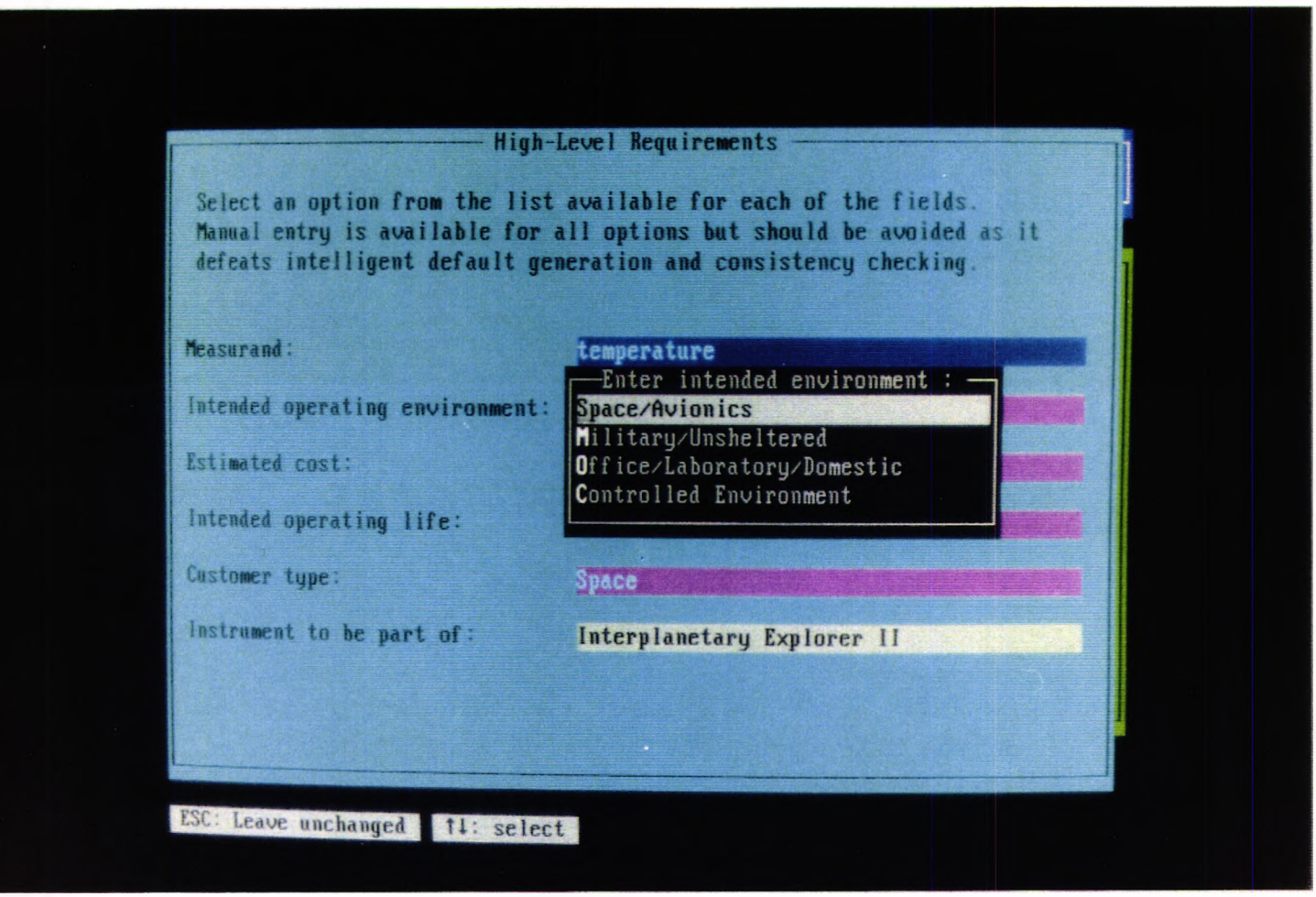


Figure 7-1

High - Level Requirements Screen

The intended measurand field comes first. This is a database-assisted field and the preferred mode of completion is to select an option from the database. Upon pressing Enter or F4, the Shell looks up the database name from the knowledge base, reads the file, in this case "NAME.DAT" and displays a list of the attribute nominated, in this case measurands. The virtual list handler is needed in this case to deal with the list as it is longer than the number of screen rows available. The list is sorted into alphabetical order so the desired measurand, "temperature", is found by scrolling the list up with the PgDn and down arrow keys. Selection is completed by pressing Enter when the desired measurand is reached.

It also possible to enter a measurand name not in the list, by using the standard attribute line entry facilities available by pressing F3. However, if the measurand is not in the database, then intelligent default generation and many checking functions will be hampered.

The next field encountered in the list, is "intended environment". This is a list attribute and pressing F4 or Enter will display the list available for selection, see Figure 7-1. The intended environment selection is used to generate intelligent defaults for numerous detailed attributes, in particular those relating to the operating and storage environment. To make use of this facility, it is necessary to chose one of the four options offered even if it is not a good match. Details can be amended later. For the example specification, the obvious choice is "Space/Avionics".

Intended cost field can take one of six value representing the order of magnitude cost. Expected cost is rarely included in a specification explicitly as it is a contractual, as opposed to technical, matter. However, it impacts most aspects of an instrument development or purchase and hence the reason for its inclusion here. Default generation rules exist to specify a number of topics based at least partially on cost, for example, quality assurance, reliability verification, and maintainability philosophy. All space craft equipment is expensive, and a cost estimate for the thermometer could be expected to be,

say, twelve thousand pounds. Hence, the "10 000 to 100 000" option is used.

Intended life has been divided into four coarse divisions: less than one year, one to five years, five to twenty years, greater than twenty years. The option selected helps the intelligent default generation process specify such topics as reliability, maintenance philosophy, and housing type. The spacecraft has a goal life of nine years so the option "5 - 20 years" is selected.

The last list attribute to specify is the customer type. Four options are provided: space, military, industrial, and consumer. Once again, the best match should be chosen to take advantage of intelligent default generation and consistency checking facilities. In this case the selection is simple: "Space". The customer type helps define quality assurance, applicable documents, maintainability, and physical characteristics.

The standard attribute at the bottom of the screen requires the name of the super-system, if any, to be entered, i.e., what the instrument may be part of. For the space craft thermometer of the current example, the entry is "Interplanetary Explorer, Thermal Sub-system". This is used in descriptive parts of the specification for the information of the reader.

Any of the entries, in this or in fact any other *Specriter 3* screen, can be edited any number of times after initial completion. Once the entries are satisfactory, the user exits this screen in the normal way by pressing F10.

7.3.1.3 Intelligent Default Generation

At this stage, the knowledge base is in a known state as all the attributes values have been set to their default values. Thus, the intelligent default generation function needs only alter those attribute values which can be better specified using values inferred from the high-level requirements using the default generation rules stored in slots in the knowledge base.

A search for default generation rules slots starts the process. It commences with the Level 1 frames and works down through each tree. In the measuring instrument knowledge base, these rule slots are confined to the Level 1 frames. However, the exhaustive search facility is provided for unforseen use.

When a default generation rule slot is encountered, the Prolog rules are extracted and passed to the *Specriter Inference Engine (SIE). SIE* produces a dialogue window and executes the rules until completion, whereon the user is asked to press the space bar. This manual intervention enables the user to read the messages output by *SIE.* After the key is pressed, Control is returned to the Create utility and the search continues. Whenever a default generation slot is found, *SIE* is called again. This sequence proceeds until all the default generation rules have been executed.

The search algorithm will follow the same path as the one used to create the Edit sub-menu, hence the default generation rule sets will be found and executed in that order.

The operation of each default generation rule set is described topic by topic below.

7.3.1.3.1 Units, Measurand, and Instrument Name

The "measurand" attribute in this topic is not the same as "intended measurand" completed earlier in the high-level requirements screen. It was decided to keep these separate to permit the future possibility of processing the intended measurand string and converted into, say, one of the pre-defined measurands. The current system simply sets the measurand to the intended measurand.

Next, the database file which holds the relationship between measurands, instrument names and measuring units is consulted and loaded into the *SIE* database. This database is then searched for the intended measurand. The

measurand "temperature" is present and hence the corresponding values for the other two attributes are extracted and automatically written into the specification. Hence, measuring units are set to "degrees Celsius" and the instrument name becomes "thermometer".

7.3.1.3.2 Physical Characteristics

The standard default physical arrangement is the provision of two enclosures, one to house the sensor and the other to house the remaining functions, nominally the data processor and display. There is no information available in the high-level menu to improve on this default so the default generation rules are limited to selecting the cooling option and the mounting methods.

The default generation rules use the fact that space or avionic instruments will generally possess a special mounting face which also acts as a heat path. Hence this knowledge is used in the current example. Had an industrial customer been specified, rack mounting would have been specified together with convective cooling. Instruments destined for the consumer market would have been catered for by the standard defaults of purpose designed enclosures and convective cooling.

The most appropriate default for military instruments was difficult to determine because of their diverse physical characteristics. It was eventually decided not to alter the standard defaults.

7.3.1.3.3 Electrical Interface

There is little that can be extracted from the high-level frame to help to specify the electrical interface. The customer type is used to define the power source. A space instrument will often be powered from 28V D.C. so this is used for the current example, whereas many military systems operate from 115V, 400Hz A.C. These will, of course, often be inappropriate but they can be readily changed.

Military and space instruments will have defined electromagnetic compatibility requirements, so for these classes of product, the general requirements of MIL-STD-461 are cited.

7.3.1.3.4 General and Static Performance

There are no default generation rules for this topic.

7.3.1.3.5 Dynamic Performance

The standard default for this topic the mathematical description using a first order response with a time constant of 20 ms. This time constant is somewhat arbitrary and was chosen to provide a response which is effectively instantaneous to a human observer without unnecessary bandwidth. There is no information available to improve on this default so there are no default generation rules for this topic.

7.3.1.3.6 Operating and Storage Environment

The intelligent default rules for this topic generate the many detailed environment attributes from the intended environment selection in the high-level requirements menu. It is worth noting that the operating environment is usually modified by a platform and this has been taken into account when deciding the intelligent defaults. The following default values are used for the example instrument because it is intended for a space environment:

Temperature range:	5 to 50 degrees C
Pressure limits:	0 to 105 kPa
Humidity limits:	0 to 95%
Vibration range:	not applicable
Max. vibration:	not applicable

The upper pressure and humidity limits reflect the fact that spacecraft need to be tested in the integration and test facility before launch. The temperature of most space craft electronic boxes is controlled to within a few degrees of a nominal 20 degrees Celsius to ensure good reliability and freedom from thermal stress, hence the operating temperature range boundary is narrower than might have been expected. There is generally little or no vibration in a spacecraft after deployment, and thus the default vibration attributes are set to "not applicable".

The non-operating environment for a spacecraft is dominated by launch and airfreight considerations. The values applied were derived from MIL-STD-810E (1989) and AEL (1987), except for the storage time which is a customary default for scientific spacecraft launched by NASA:

Temperature range:	5 to 50 degrees C
Pressure limits:	0 to 105 kPa
Humidity limits:	0 to 100%
Vibration range:	10 to 500 Hz
Max. vibration:	3mm from 10 to 50 Hz decreasing at 6dB per octave until 500 Hz

Had the instrument been intended for a military environment, then the following default operating environment extracted from MIL-STD-810E (1989) and Cook (1983) would have been used:

Temperature range:	-31 to +55 degrees C
Pressure limits:	73 kPa to 105 kPa
Humidity limits:	0 to 100%
Vibration range:	5 to 55 Hz
Max. vibration:	1.5 mm from 5 to 15 Hz, 1 mm from 15 to 25 Hz, and 0.5 mm from 25 to 55 Hz.

The figures correspond to "basic cold" as encountered in Europe, and

operation to altitudes of up to around 3,000 metres. The vibration values relates to what could be expected on a "non-specific mobile platform" and encompasses things like ships and ground vehicles driven on roads.

The non-operating environment for this option is drawn from the same reference documents. The low pressure limit is the minimum cargo hold pressure encountered in transport aircraft.

Temperature range:	-40 to +55 degrees C
Pressure limits:	57 kPa to 105 kPa
Humidity limits:	0 to 100%
Vibration range:	5 to 55 Hz
Max. vibration:	1.5 mm from 5 to 15 Hz, 1 mm from 15 to 25 Hz, and 0.5 mm from 25 to 55 Hz.

Selection of the more benign sheltered indoor environment would have set the following default operating environment:

Temperature range:	2 to 40 degrees C
Pressure limits:	95 kPa to 105 kPa
Humidity limits:	0 to 95%
Vibration range:	not applicable
Max. vibration:	none

The non-operating environment would be dominated by the airfreight environment:

Temperature range:	-19 to +55 degrees Celsius
Pressure limits:	57 kPa to 105 kPa
Humidity limits:	0 to 100%
Vibration range:	5 - 100 Hz
Max. vibration:	0.25 mm

Finally, in the event that the controlled environment option was selected, as found in, say, a computer room, the following values from MIL-STD-810E would have been set:

Temperature range:	21 to 25 degrees Celsius
Pressure limits:	95.45 kPa to 103.05 kPa
Humidity limits:	45 to 55%
Vibration range:	not applicable
Max. vibration:	none

In this instance, the non-operating environment would be unchanged from the last option.

7.3.1.3.7 Quality Assurance

Quality assurance provisions are specified by selecting one of three options, good commercial practice, 100% testing of performance characteristics, or rigorous verification of every paragraph of the specification. The choice of the one to use will usually be decided by the expected cost of the instrument and the type of customer. For instruments costing less than 1000 pounds destined for commercial customers, the first option is selected. For instruments whose expected cost is between 1000 and 100 000 pounds, that are intended for other than the space industry, the second option is used. Finally, for instruments costing more than 100 000 pounds and/or bound for the space industry, the expense of a full quality assurance program is indicated. The latter category will have been automatically selected for the current example because of the instrument's intended use on a spacecraft.

7.3.1.3.8 Reliability

The default generation rules are required to chose between specifying reliability by probability of survival, mean time to failure, or a user supplied paragraph. The rules examine the customer and use this parameter as the

main switch. Consumer instruments are assigned no formal reliability criterion but instead a sentence is placed in the user paragraph to the effect that the instrument should be designed with its intended life taken into account. Instruments intended for industrial customers are specified by mean time between failure with the MTBF set to the upper end of the life expectancy. The military or the space industry usually wish to specify reliability by probability of survival. As a starting point, the probability is set to 0.99 and the survival period the upper end of the intended life. Thus for the current instrument, the latter method and values are used.

7.3.1.3.9 Maintainability

The standard default for this topic is to specify that the instrument should be returned to a service organisation for repair. This covers most situations but this is changed by the default generation rules, in some situations. For example, if the instrument costs less than 100 pounds then a throw-away philosophy is specified and repair is by replacement. At the other end of the scale, full maintenance support will be specified for very expensive instruments. If, as in the case of the current example, the intended operating environment is space flight, maintenance will be impossible so this option is selected.

7.3.1.3.10 Design and Construction

This topic is generally left unspecified for development specifications and accordingly there are no default generation rules. This is an area for further research.

7.3.1.3.11 Preparation for Delivery

A useful, general purpose default paragraph is provided. This is adequate for most situations and no default generation rules are provided.

7.3.1.3.12 Notes

Notes do not form part of the specification and there are generally few, if any. Hence there is no need for default generation rules.

7.3.1.3.13 Applicable Documents

The number of applicable documents should always be as small as possible, hence the standard default is an empty list. Certain of the default generation rules, however, cite reference documents and hence it is useful to include these in the list, as necessary. In the case of the current example, some of the basic reference documents for a space flight instrument would have been automatically written into the applicable documents paragraph by the intelligent default generation rules.

7.3.1.4 Completion of Entry

At the completion of the above process, each of the editor screens will be displayed in turn. The user will need to examine each of the default attributes and make amendments as necessary. (The specification methods for each topic are fully described in Chapter 3.) This completes the specification creation process.

For the sake of combining the current example with a description of the editing process, it is assumed that no changes were made to the example specification during the pass through the sequence of editor screens.

7.3.2 Editing a Specification

If the current specification is not the one to be edited, the first action will be to locate the required specification, make a copy, and load the new file. This can be readily accomplished using the Files/Copy & Load command. If the user presses Enter in place of supplying a source file name, a directory of file names is

displayed to select from.

After the file is loaded, editing can begin by selecting Edit from the main menu. The list of Level 1 frames is then displayed, see Figure 6-7. Each of these menu items can access the full compliment of leaf frame options in that topic. The editing process can take place in any order as there is no dependency between the items shown in the list menu, however, the description which follows will start from the top.

The functions of the edit process will be illustrated using the specification of the spacecraft thermometer created earlier. Because no attributes were altered during the pass through the edit screens at the end of the create process, that work will need to be done now.

7.3.2.1 Units, Measurand, and Instrument Name

This screen contains just three database-assisted fields, one for each of the items of its title. All three have been completed by the intelligent default generation rules for this frame. The measurand is set to "temperature", the units to "degrees Celsius", and the instrument name to "thermometer". On pressing F5, the checking process commences. The consistency checking rules for this frame are extracted from the knowledge base and passed to *SIE* for execution. Consistency checking for this frame consists of searching the nominated database "NAME.DAT" to see if a set comprising the specified measuring instrument, measurand and measuring units can be found. The matching process is case independent, but no attempt is made to perform natural language processing to match strings with identical meaning but different expression. This is a topic for further research. Dialogue with the user is via a window opened by *SIE* for that purpose. In the case of the example, the user is informed that the three attributes form a set known to the system. This is to be expected as they all came from the database.

The example instrument is a component of a larger system and a more specific name is required for identification purposes. The user moves the cursor to the instrument name field and presses enter to examine the list held in the database. The correct name is not found so it is necessary to return to the editor screen without altering the previous value using the Escape key. The desired name is entered by pressing F3 to open the line editor and then typing "Type 601 Spacecraft Thermometer" followed by Enter.

This screen is now complete and F5 is pressed again. This time the checking process reveals that the set of items is not known to the system. This is already known and can be ignored. The approach taken in the *Specriter 3* measuring instrument knowledge base, is to keep the checking rules simple and flag potential errors and let the user decide on the action to be taken. Much more could be done in this area and this would be a good candidate for further research.

F10 closes this screen and returns control to the Edit sub-menu.

7.3.2.2 Physical Characteristics

Physical characteristics, which covers physical partitioning, thermal interface, and mechanical interface, is the next topic on the list. These items are strongly interrelated so were combined into one frame tree. There are four combinations of physical arrangements and three enclosure specification methods. To cater for all of these, around 50 leaf frames are required. There is a useful amount of knowledge held in the frame structure and, wherever possible, automatic defaults are written directly into the text in preference to presenting another attribute. For example, if a 19 inch rack mounted enclosure is specified, it is known that mounting is achieved via the pre-defined mounting flanges and cooling is convective.

The default physical arrangement is to specify two enclosures, one for the sensor and another for the remaining functions, nominally data processing and

display. The intelligent default generation process used the fact that the intended environment for the example specification was space or avionics to specify purpose designed enclosures as these are customary in this field. Additional intelligent default rules specified conductive cooling and mounting via bolts through the expected mounting flange.

Of the nineteen possible attributes spanning the range of leaf frames, as a result of the intelligent default generation process for this topic, the user is presented with only 12. Three of these are correctly completed option fields and a further four have sensible defaults deduced from the high-level requirements. As there are more options to complete than could comfortably fit on a single screen, the virtual screen handler has been used for this topic. Figure 7-2 shows the first half of the attribute list under discussion. The remaining attributes can be scrolled up, to permit observation and editing, using the arrow keys, Page Down, or End.

The attributes can be completed in any order. The mass limits for each enclosure will already be known as these will have been determined in the system design phase. The main unit mounting method is correct so this needs no changes, however the one for the sensor had to be changed to "attached to the structural member nominated in IE-ICD-MI-005 using heat conductive epoxy adhesive type AD-185".

The housing description paragraphs need to be completed to specify only those enclosure restraints necessary. This is best done by referencing an interface control document, or a drawing. The former is used for both enclosures needed for the current example specification.

The last field to complete is the maximum sensor temperature rise. As the instrument under consideration is a thermometer, this parameter is not particularly applicable as it is intended for sensors concerned with other measurands. In a space flight application where thermal design cannot be left to chance, it is worth placing a limit on it nonetheless, so 0.1 degree Celsius is

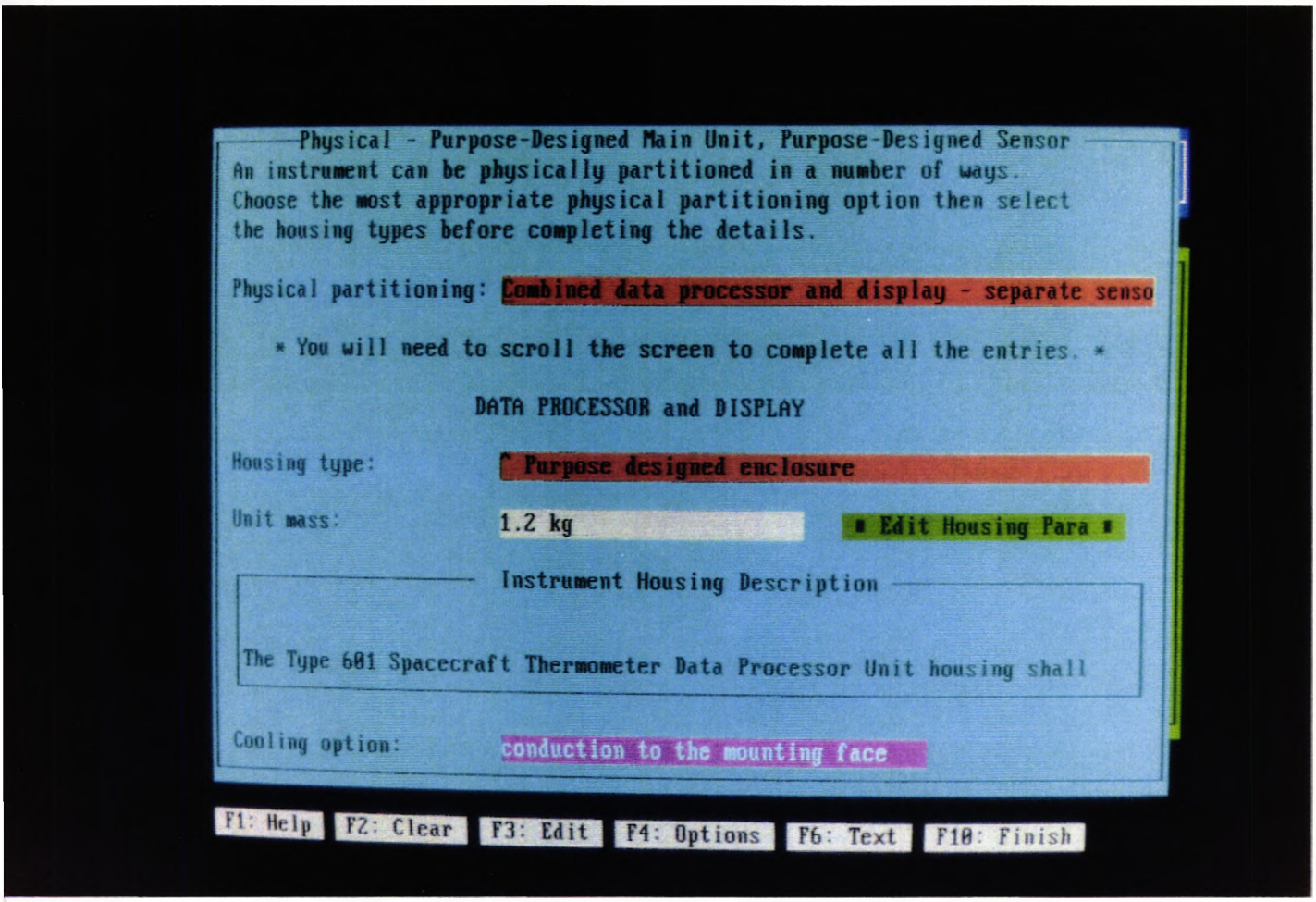


Figure 7-2

Physical Characteristics Screen

entered. When entering this value, a warning message is displayed saying that the value entered is lower than the nominal lower limit of 1 degree Celsius. In this case the warning is ignored.

7.3.2.3 Electrical Interface

Electrical interface is specified using a single screen with no options. There are six standard entry fields covering power, communication interface and electromagnetic compatibility (EMC). The power source of 28 V DC was correctly specified by the intelligent default rules for the frame. The power connector and the peak current need to be entered next. Once again, these will be available from earlier system design activity. The communication interface is indeed an RS-232-C and this standard default is left unaltered. The EMC limits are described in the spacecraft Electrical Interface Control Document and hence these entries are changed to reflect that.

7.3.2.4 General and Static Performance

This screen contains nine standard entry fields and a single user paragraph for special requirements, see Figure 7-3. These were all set to "not specified". Each has to be completed in turn.

The values entered are checked for consistency with themselves and the rest of the specification by pressing F5. Consistency rules produce warnings if the total measuring error is not achievable given the values of the other error sources, the power consumption inconsistent with peak current limits and thermal interfacing.

The help information is particularly useful in assisting the completion of this frame.

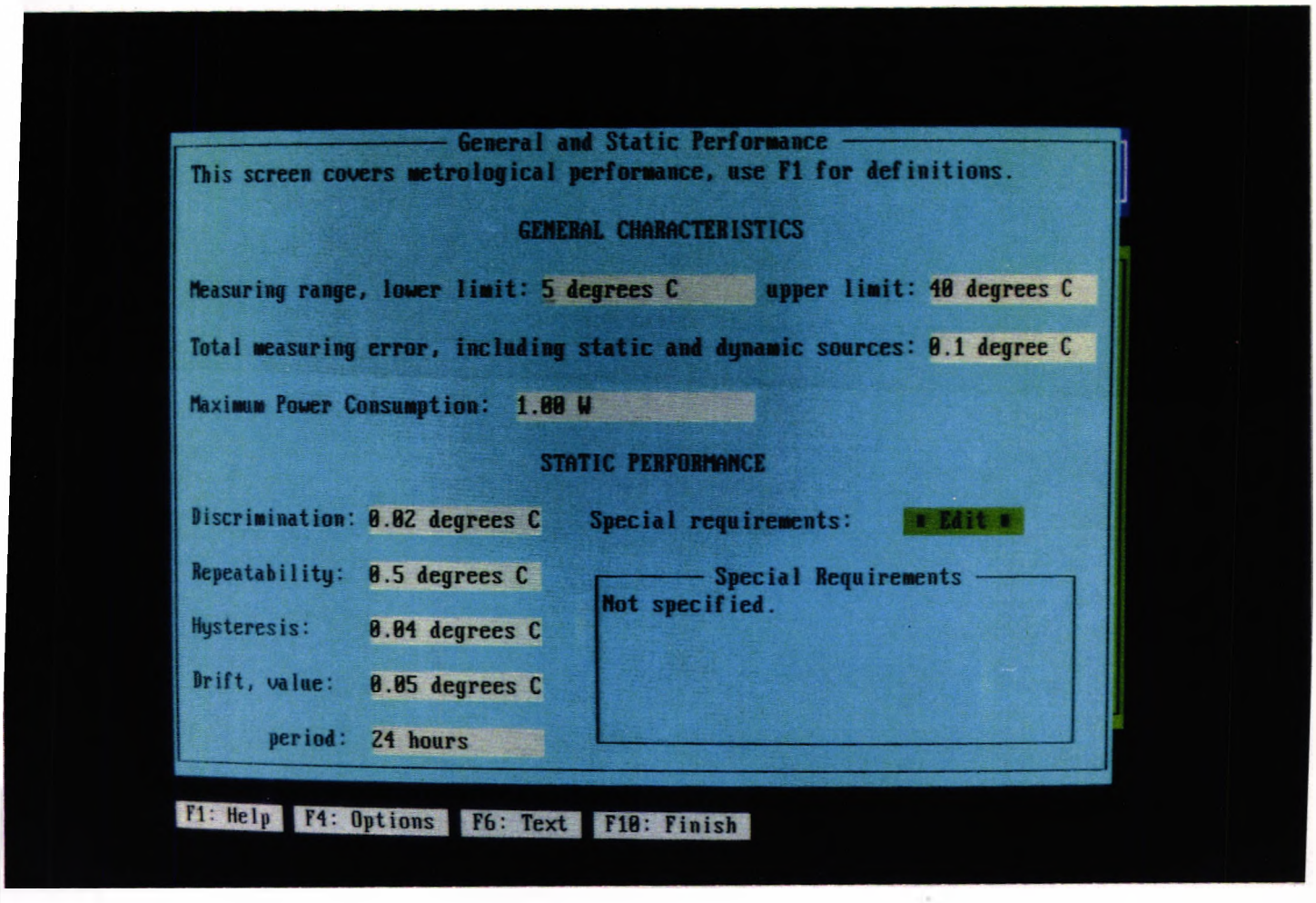


Figure 7-3

General and Static Performance Screen

7.3.2.5 Dynamic Performance

This topic contains two option fields which give rise to a total of five leaf frames to chose between. The standard default of first order response is the most appropriate for the instrument under consideration. However, the time constant needs to be changed to suit the instruments's role as a component of the closed loop control systems which maintains the spacecraft's structure at a constant temperature. The value mandated by the thermal control loop design is entered in the normal way.

7.3.2.6 Operating and Storage Environment

This environment screen contains 19 standard entry screen and two user paragraphs, see Figure 7-4. The intelligent default generation process has set the standard entry fields to the values listed earlier for space flight instruments. These require few changes for this particular instrument, but there is a need to specify ionising radiation dose limits using the special operating environment paragraph. Acoustic noise is specified using the special environment paragraph for the storage and transport environment.

7.3.2.7 Quality Assurance

The intelligent default of specifying rigorous quality assurance provisions is ideal for the spacecraft thermometer. The consistency checking rules for this topic compare the selected option against the high-level requirements of expected cost, intended environment, and intended customer type. As would be expected, an option chosen by the intelligent default generation rules produces no warning messages when the checking key, F5, is pressed.

7.3.2.8 Reliability

The Reliability screen shows that the probability of survival method has been chosen. The intelligent default of 20 years needs to be changed to reflect the

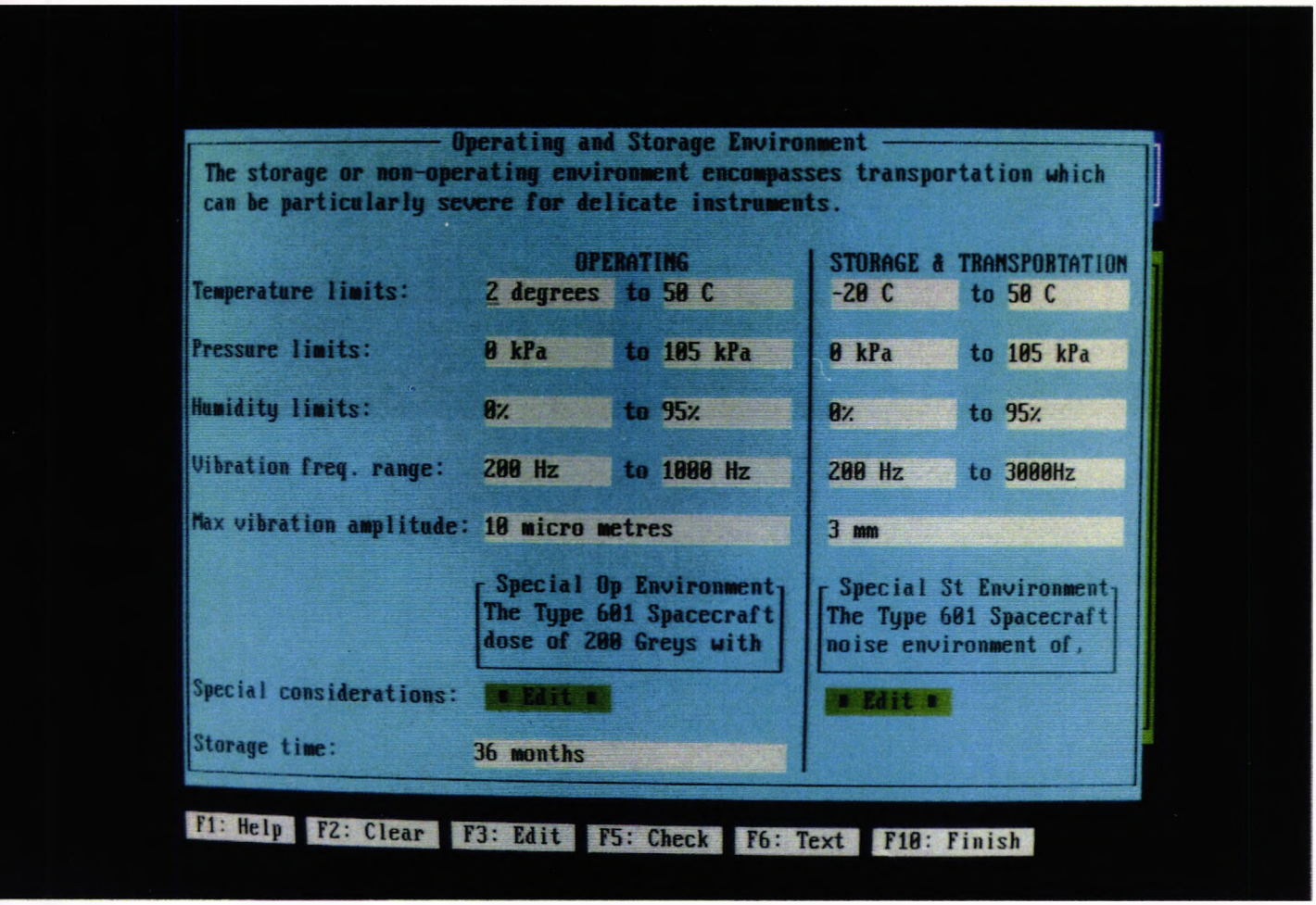


Figure 7-4

Operating and Storage Environment Screen

mission lifetime of 10 years and the probability of survival of this relatively minor part of the spacecraft needs to be increased to a more respectable 0.998.

7.3.2.9 Maintainability

The application environment does indeed prohibit maintenance, thus the intelligent default is unchanged.

7.3.2.10 User Paragraph Screens

The last four topics are specified solely by user paragraphs. The default text can be edited to suit the instrument in question and the nature of the specification, i.e., whether it is a development specification where considerable flexibility is permitted particularly on design and construction or a product specification where interchangeability is paramount.

7.4 Text Generation

Now that the specification has been edited and consistency checked, is time to generate the first draft requirements specification. This is done by selecting the Text/Display command from the main menu. *Textgen3* is called and after performing its functions, as described in Chapter 6, the final text appears on the screen. If checking reveals that alterations are required, then the editor can be called as many times as desired to perform the necessary changes.

The Text/Print Option can be used to print the finished document on a printer attached to the standard printer port on the computer. Another alternative is to exit *Specriter 3* and read the output text file, in this case, "TEMP.TXT", into a word processor program and proceed from there.

7.5 Finishing a *Specriter 3* Session

After the completion of the printing process, or indeed at any other time, selection of

the Quit option from the main menu will terminate the session. The specification under consideration will be saved automatically. Restarting the program will return the user to exactly the same state as just before exiting; there is no need to explicitly load the specification file.

7.6 Achievements of *Specriter 3*

Chapters 1, 2, and 3 generated a set of requirements for a tool to assist in the production of measuring instrument specifications. These were listed in the introduction of Chapter 4, Section 4.1. This discussion commences by examining the degree to which each of these is met by the final system *Specriter 3*.

The first aim was to improve the efficiency of generating specifications compared with manual processes. This was achieved with *Specriter 1* and has been improved upon since. The fact that a reasonable printed draft can be produced in less than an hour confirms this.

The second aim was to provide knowledge-based assistance to help in the generation of the specification, covering areas such as specification format and use of language. This knowledge has been incorporated into the knowledge base, and is embodied in the numbered paragraph headings, the stored text fragments, the text generation routines, and the default text format parameters.

The third aim was to provide a user interface suitable for both novice and experienced specification writers. *Specriter 3* possesses a sophisticated user interface which combines a high degree of functionality with ease of use. User entry is always handled by either a virtual-line editor or a full screen editor. Either can cope with pages of input should it ever be necessary.

There are two command modes available. Infrequent users can access the system using just the arrow keys and Enter. They can choose items from lists and menus with the arrows and select them with Enter. Users more familiar with the system, can use the "hot keys" present in most menus. These can be pre-typed to save time.

For example, after *Specriter 3* is called from the operating system, if 'e' and 'o' are pressed the system will commence with the operating and storage environment screen of the editor. The on-line manual and the context-sensitive help, which is available for almost all situations, further assists users.

The fourth aim was to provide some mechanisms for completeness, consistency and reasonableness checking. *Specriter 3* allows consistency checking to be incorporated into every leaf frame. Checking takes the form of executing a sequence of rules written in Prolog. Thus a powerful open-ended mechanism exists for this function. Consistency checking rules exist for many situations and more can be added without difficultly, using the knowledge base editor *Framedt.*

Completeness checking is also provided but in a different form. Cohen et. al. (1986) and Blackburn (1989) define a formal specification to be complete when the implementation of all the (abstract) objects is fully defined. As *Specriter 3* is a limited formal system, a specification contained within the system will be complete whenever each topic has a leaf frame defined and the values of the attributes which comprise that frame, and its ancestors, are defined. All specifications produced with the Create process conform to this requirement, and in that sense can be considered to be complete. From a practical perspective, it is of course possible to have many attributes not fully defined because they are assigned the value "not specified", or something similar. Checking for this situation can be done by the simple expedient of inspecting each editor screen in turn.

The fifth aim was to provide sensible defaults preferably generated from information already entered. This has been successfully achieved with the intelligent default generation process.

The sixth requirement was to concentrate on the incorporation and structuring of existing knowledge and techniques. This has been the approach adopted throughout. Novel specification techniques have not been pursued.

The seventh design aim was to incorporate project management aspects in addition to

technical ones. This has been achieved by including many contractual items in the high-level requirements screen, such as expected cost and intended life. This information is available to the system's reasoning mechanisms. In addition the format of the specification produced and its comprehensiveness are a definite aid to project management. It would be possible to construct a view into the knowledge base from a management system should this be desirable.

The eighth and final aim was to be aware that the resulting software may later form part of a larger entity. This requirement has always been observed and any DOS program, whether written in Turbo Prolog or not, can gain direct access to the specification and knowledge base files used by *Specriter 3*. Simple integration of disparate computer-aided design software entities could be achieved through these files. The format of these files is described in Appendix 2.

7.7 Conclusion

This chapter has built on the description of the *Specriter 3* system given in Chapter 6 by discussing how a measuring instrument requirements specification can be produced using the knowledge base compiled for that task. This account was used as a vehicle to discuss the heuristic knowledge contained in the default generation and consistency checking rules. Appendix 5 includes a copy of the finished specification generated in the course of illustrating the *Specriter 3* specification creation, editing, checking, and text generation facilities.

The last section of this chapter critically examines *Specriter 3* against the design aims established, some four years ago, before the commencement of any programming work. Each has been fulfilled.

A point which was not made in that review is the future potential of *Specriter 3*. The measuring instrument knowledge base can be readily expanded with both more options and more rules through use of the *Framedt* facility. Completely new structured document generation applications can be built by replacing the existing domain-specific knowledge base with another.

Relatively small modifications to the *Specriter* shell, such as the inclusion of slots containing attached procedures, would permit a range of frame based tasks to be performed. *Specriter 3* could then be extended with extra views which could interrogate the specification to simulate the instrument's behaviour or to pass information onto design assistance programs.

Chapter 8 - Conclusions and Suggestions for Further Work

8.1 Conclusion

This thesis describes the first scientific attempt to apply the growing understanding of the specification process to the important field of measuring instruments. A human specification generation algorithm, uncovered during the investigation, has been shown through the application of knowledge engineering and artificial intelligence, to form a suitable basis for a computer tool for assisting the process of creating requirements specifications for measuring instruments.

Writing requirements specifications for measuring instruments is a difficult task. The writer needs to have a clear idea of what the measuring instrument is to do, what forms a consistent and reasonable specification, the specification format and type most appropriate for the task, and finally the writing style to employ. Specifications form a part of contractual documentation between a customer and a supplier, and hence the adequacy of a specification is a matter of considerable importance to both organisations. For these reasons, and others fully covered in Chapter 3, it can be seen that there is a need to automate the specification generation process.

It was not apparent at the commencement of the research programme, some five years ago, what could be achieved in this field. The approach adopted was to examine the human specification generation process, as practised in industry, in an effort to identify some structure, and possibly an algorithm, which could be used as the basis for automation. A human specification generation algorithm was discovered and this was translated into a computer generation process. This process, documented in Chapter 3, requires extensive use of stored knowledge concerning the specification format to be used and the methods used to specify the individual paragraphs. The uncovering of this knowledge and its description in a single work, represents a useful contribution to instrument engineering.

Specriter 1 was the first software system designed and built to implement the computer generation process. This system is capable of producing a complete printed specification in response to a list of questions. Editing, file handling and text generation facilities were all integrated into a single entity. The specification methods and the format of the specification are taken care of, permitting a user to concentrate on the instrumentation aspects of the task. *Specriter 1* has been very successful and has been used by staff, clients and students at MISC to produce a wide range of measuring instrument specifications.

It became clear that extending the architecture *Specriter 1* to contain instrumentation knowledge in the form of reasonableness and consistency rules would not be appropriate. A formal specification representation was indicated, one possessing a defined syntax and semantics that could permit direct reasoning about the contents of the specification. Chapter 5 presents a review of such methods. The method adopted was to write the specification in Prolog because of its expressive power and because the specification can be interpreted directly using the reasoning mechanisms inherent in the language.

An adequate knowledge representation is a pre-requisite for a successful knowledge-based system. Three different knowledge domains were identified for inclusion in the new system: instrumentation knowledge, human interface, and specification generation knowledge. A review of knowledge representation techniques revealed that a frame-based system could be used to contain all of these and also the specification, provided the entire system were constructed in Prolog. The key concept behind the chosen representation is that each individual view or perspective on the specification is represented by a unique frame. Thus each screen displayed to the user corresponds to a frame held in the knowledge base. Where there are options in the specification process for a given topic, a hierarchy of frames is used. Knowledge common to more than one frame is inherited from a parent frame thus reducing needless repetition. Reasoning about the content of the specification is performed on a frame by frame basis thus avoiding the verification and maintenance difficulties associated with large unstructured knowledge bases.

The formal specification and knowledge representation concepts were incorporated into *Specriter 3*, a completely new system written in Prolog. The major advantages this system has over its predecessor are: the inclusion of heuristics in the form of rules written in Prolog, the complete separation of the knowledge base from both control and inferencing mechanisms, and its potential for expansion to cater for additional requirements. *Specriter 3* can be thought of as a specialist expert system shell dedicated to the creation and editing of standard documents or forms. Its behaviour is totally characterised by its external knowledge base. The single knowledge base currently in existence has been referred to as "the measuring instrument knowledge base" but, in fact, it contains all three knowledge domains identified earlier. A purpose-built editing tool named *Framedt*, was built to create, edit and extend this or any future knowledge base.

The functionality of the system resulting from loading *Specriter 3* with the measuring instrument knowledge base, completely eclipses the earlier *Specriter 1*. Specification of the roughly one hundred attributes is greatly assisted in *Specriter 3* by the adoption of intelligent default generation. From the user's response to a few high-level questions, such as intended environment, expected cost, and intended customer, rules held in the knowledge base select the most appropriate specification options for every topic and deduce roughly half of the detailed attributes. Considerable assistance is offered to the user during editing, through context-sensitive help messages and highly-capable line and paragraph text editors. After completion, the consistency of the specification can be examined, topic by topic, using consistency checking rules. These can alert the user to a range of potential problems in the specification.

8.2 Suggestions for Further Work

Specriter 3 breaks new ground in the preparation of specifications for physical equipment. There are, understandably, many areas which need to be developed to maturity.

The *Specriter 3* shell is well advanced, the most obvious need is to extend the

knowledge base. For example, extension could be provided for instruments which measure in more than one dimension, for example vision systems. The specification template could be extended to cover topics which are more specific to product specifications such as design and construction, logistics, documentation, and safety. The intelligent default generation and consistency checking rules could be augmented.

Of greater interest, is to use the *Specriter 3* knowledge representation paradigm and the imbedded instrument specification, as the central core for a complete measuring instrument design system such as CAEINST at SAIT or that evolving at City University. Design concept generation and detailed design modules could then interact with a development specification created by *Specriter 3* to convert it into a product specification from which an instrument could be built, perhaps later via computer aided manufacture.

Appendix 1 - *Specriter 1* User's Guide

SPECRITER USER'S GUIDE

S.C. COOK, APRIL 1988

CONTENTS

1 DESCRIPTION AND BACKGROUND

1.1 Background

Specriter 1.4 is an intermediate output from research into Computer-Aided Engineering of measuring INSTruments (CAEINST) being conducted by The Measurement and Instrumentation System Centre (MISC) at The South Australian Institute of Technology (SAIT). CAEINST aims to provide the necessary knowledge for a user to specify, create, and apply capable measuring and control systems.

Specriter's primary function is to extract information from the user regarding the measurement problem to b e solved, and from this, generate a written specification for a measuring instrument. The output text closely follows the MIL-STD-490 specification format. This format or derivatives of it are widely used throughout the world for specifying military, aerospace, and professional equipment.

This version of Specriter was originally developed in VAX Pascal running on the VMS operating system and then converted to ANSI PASCAL to run under UNIX to finally be ported to Turbo Pascal for use on IBM PC or compatible computers. To permit portability, few language extensions over ANSI Pascal were used. This heritage explains the appearance of the screens; each was designed for a ANSI terminal and hence none of the advanced features available for the PC have been employed. String handling and input robustness have all been built from the ground up; this approach, while necessary because of the inadequacies of ANSI Pascal, has resulted in high degree of input checking and hopeful freedom from abrupt exits due incorrect user actions.

A consequence of designing the programs for portability is that all operating system calls are contained in a single element; a command language program for the VMS version and a batch file and small associated programs for MS-DOS. This necessitates a certain overhead and sub-optimal implementation in DOS. For this reason and because the programs are large and use overlays, execution from floppy drives is not practical.

1.2 Structure

Figure 1-1 illustrates the structure of Specriter 1.4. The structure has been heavily influenced by the use of ANSI Pascal and the desire to permit portability across operating systems. Specriter uses the concept of work files. There is an Instrument Attribute File for each instrument that held in the Specriter directory. The file name of the current instrument being worked on is displayed in bold in the main menu. Disk space is the only limit to the number of

Instrument Attribute Files that can exist permitting substantial libraries to be built.

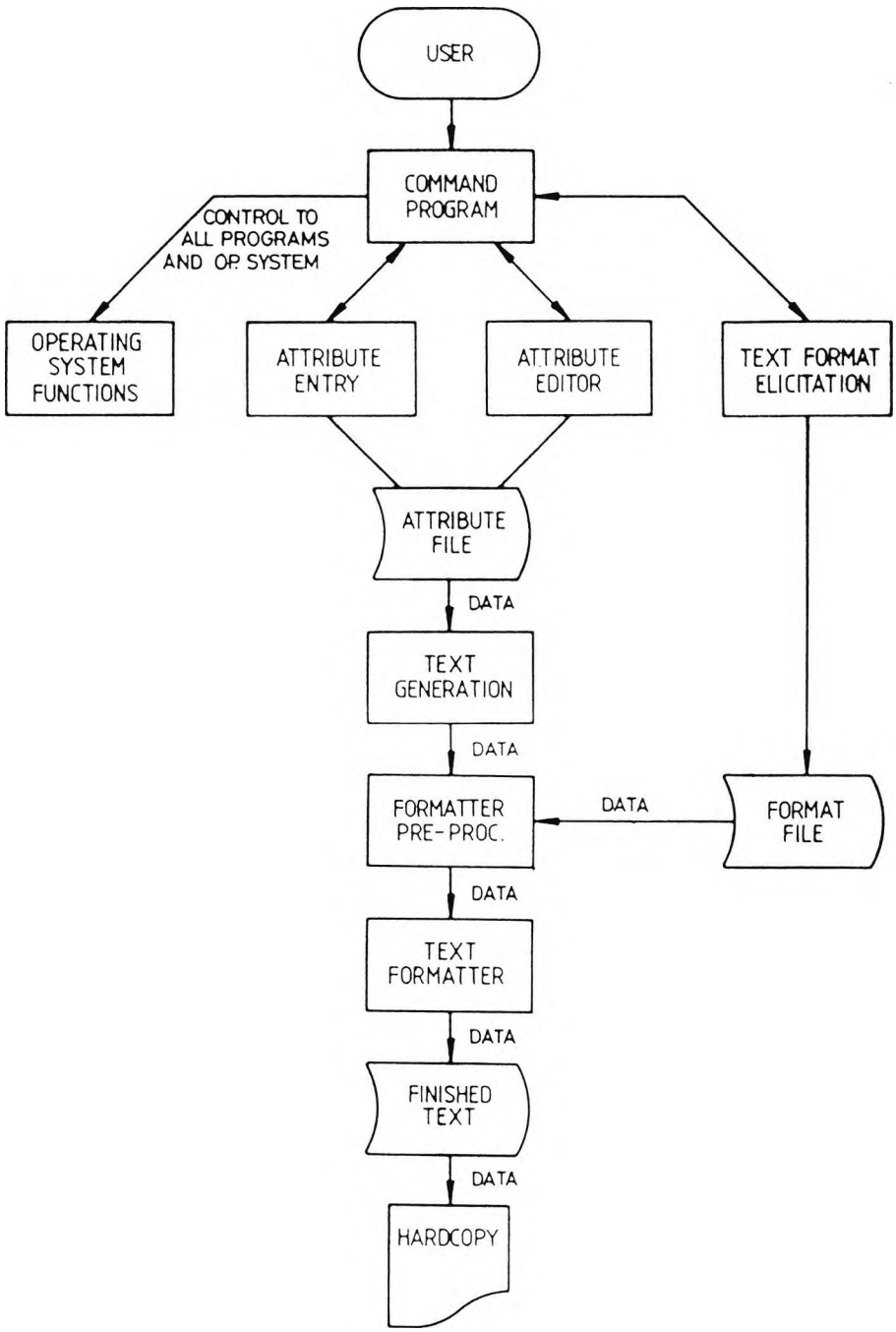


Figure 1-1 Structure of Specriter 1.4

1.2.1 Command Programs

The command program is in fact a group of programs; two for VMS and UNIX and three for DOS, only the latter implementation will be discussed here. DOS has a very limited command language which means that many tasks originally handled in the command language needed to implemented by a Pascal program.

Invoking Specriter runs a program called "COM". This program generates the banner and produces the main menu. A correct menu selection updates the file "CONTROL" which contains the work file name, and the menu option selected. "COM" then terminates and the Specriter batch file now starts the DOS specific program "CONVERT" which interprets "CONTROL" and generates a batch file called "NEXTTASK". "NEXTTASK" runs next performing the function selected from the main menu, e.g., running the Editor program. When this program terminates, control is returned to the "SPECRITER" batch file which restarts "COM". This program re-displays the main menu and awaits another selection.

This method of performing operating system calls indirectly from an applications program, although rather cumbersome, does permit portability with the change of just one program.

1.2.2 Attribute Entry & Editing

Knowledge elicitation from the user and subsequent editing of the information gathered, is performed by a program called "SPEC". This program is set to either edit an existing Instrument Attribute File or create a new one dependant on the main menu selection which invokes it.

Turbo Pascal version 3, the language used to implement Specriter, has a maximum code size of 64K. As "SPEC" is considerably larger than this, 10 overlays have been used, one for each of the major areas of the specification. (This is another reason why floppy disk operation is so slow.) The essence of the instrument described by the user is written to an Instrument Attribute File given the current work file name. Whenever the Editor program is invoked and the Instrument Attribute File changed, the corresponding text file is deleted because it will now be out of date.

1.2.3 Text Format Elicitation and Text Formatting

Whenever the display or print main menu options are selected, the Specriter command program searches for the text file, and if absent, invokes the text generation program "SPECOUT". This program reads in the formatting data contained in "FORMAT". There is only one format file for each Specriter installation because it was felt that most of the entries, e.g., page length, margins, and indent rate would not change once set up. The consequence of this is that the user has to be careful to ensure that the document and issue number are current when the work file is changed. "SPECOUT" uses the format control information to generate a suitable input file for the Runoff text formatter. This program reads the data file output by "SPECOUT" and generates another file which later becomes named as the current work file name with the extension ".TXT". The text formatter produces a finished right justified document comparable in presentation to a word processed document. The only blemish being that it is not able to attach text to paragraph headings very well.

1.2.4 Text Display

The Borland program "README.COM" is used to display the text. This program offers a range of facilities and provides fast scrolling when the Page Up and Page Down keys are pressed.

1.2.5 Text Printing

Printing is performed by using the DOS Print command which is assumed to be available on the machine.

2 INSTALLATION

2.1 Hardware Requirements

Specriter is written in Turbo Pascal and also uses MS-DOS commands to perform operating system calls. Any IBM PC or compatible machine which can run Turbo Pascal in a DOS environment should experience no problems. The Specriter programs are of sufficient size to crowd out a standard 360K double density disk used to distribute the software. For this and the other reasons mentioned earlier, Specriter needs to run from a hard disk. Temporary files are generated during execution hence be sure to have at least 500K bytes free before loading Specriter. Minimum memory requirements have not been established. Certainly 512K is sufficient and probably as little as 256K may suffice.

2.2 Software Requirements

Development was performed using MS-DOS Version 3.2. Other recent versions should pose no problem. Specriter calls many DOS commands hence it will be necessary for the DOS to be installed on the hard disk and the DOS directory included in the path statement. (This is normally the way IBM PCs are set up.)

2.3 Installation

Create a directory for Specriter called SPEC using:

```
c:> MD SPEC
```

Next copy the contents of the distribution disk to this directory using:

```
c:> COPY A:*.* SPEC
```

To start Specriter set the default directory to SPEC using:

```
c:> CD SPEC
```

Check that at least the following files are present:

```
SPEC.COM      - Entry and Editor Programs
SPEC.OOO      - Overlay for Spec.com
SPECOUT.COM   - Text Generation Program
COM.COM       - Main Menu Program
CONVERT.COM   - Batch File Generation Program
RUNOFF.COM    - Text Formatter
README.COM    - Text Display Program

APPDOC.       - Applicable documents file
CONTROL.      - Control file holds work file & menu option
FORMAT.       - Text formatter set-up file
ATT.          - Current work file
HELP.MSG      - Main menu help file

SPECRITE.BAT - Start-up batch file
```

by typing:

```
C:\SPEC> DIR
```

then type the single command:

```
C:\SPEC> SPECRITER
```

The introductory banner should now appear shortly followed by the main menu. Specriter should now be installed. The best way to verify correct operation is to generate a specification, edit it, display and then print it.

3 HOW TO GENERATE A SPECIFICATION USING SPECRITER

There are two ways to use Specriter to generate specifications:

(a) Create a new specification from scratch
(b) Modify an existing specification.

3.1 Creating New Specifications

Start Specriter as described in Section 2. When the main menu appears, select the CREATE option by entering 1 and pressing enter. A file name will be requested; enter an appropriate name following the directions displayed on the screen. Once an acceptable name has been entered the Entry program starts. Follow the directions given and complete the list of questions, don't worry about mistakes they can be corrected later. Eventually the questions will end and your responses will be written into the file bearing the name you entered earlier and the extension ".STD". A new specification now exists in a computer readable form which can be manipulated by other programs in the Specriter system, i.e., edited, displayed or printed.

3.2 Creating New Specifications From Old

If you have a library of specifications available from previous use of Specriter, the quickest method of generating a specification for new instruments is to modify an existing relevant Instrument Attribute File. Identify the most suitable file using either the finished output as displayed or printed, or the EDIT program. Quit Specriter and copy the Instrument Attribute File to a suitable new file name. For example if a thermometer is described by TEMP1.STD and the another thermometer of slightly different performance needs to be specified then type the following:

```
C:\SPEC> copy temp1.std temp2.std
```

The original file is still available and work can begin on the new specification. Start Specriter as described in Section 2 and use the RECALL utility to load TEMP2.STD into Specriter. See Section 7 for details but the process is straightforward. Proceed to alter the information contained within this file using the EDIT program as described in the following section.

4 SPECIFICATION EDITING

Instrument Attribute Files which contain the information entered by the user about particular instruments, can be edited using the EDIT program. This program is invoked by selecting option 2 from the main menu. The work file (highlighted in the main menu), is loaded into the program and after the header page the editor menu appears. The questions asked by the create program are divided into ten separate areas each of which can be edited individually. After selecting a topic using the appropriate number, editing begins. Editing takes the form of replacing the previously stored response with a new one. If the enter key is pressed without entering any characters then the previous response is untouched.

When editing is complete, enter option 0 from the editor menu. An opportunity is provided to scrap changes made in the editing session but the default is to replace the old Instrument Attribute File with that just created.

5 THE DISPLAY UTILITY

The Display main menu option automatically generates the text of the document and displays it on the screen. If the text already exists, the user is prompted to see if either the existing text should be displayed to save time, or if the generation process should be invoked to allow alteration of the print format. The generation process first prompts the user to provide details on the print format. The defaults given are for the standard 66 line per page fan fold computer paper. A4 paper can be catered for by selecting 70 lines per page. It will be necessary to change the document and issue numbers to suit the current work file.

Once the print format is satisfactory, the text is generated and automatically displayed. Use of the display program is self explanatory and additional information is available by pressing the help key, F1. Use the ESC key to leave the text viewing program and subsequently return to the main menu.

6 THE PRINT UTILITY

The Print main menu option uses the DOS print command (PRINT) to print the specification documents generated by Specriter. Print will generate the text if necessary but it is highly recommended to use Display to perfect the text before printing it. Print uses no special printer commands and hence places no special requirements on the printer.

7 FILE HANDLING

7.1 General

Instrument description data entered by the user will be stored in Instrument Attribute Files which commence with the name entered by the user and have the extension ".STD". Specriter uses the concept of a work file such that the options selected from the main menu will be performed on the file highlighted near the top of the screen. Text files generated will have the name of the work file and the extension ".TXT".

ANSI Pascal does not requires all file names to be declared at compilation time. Consequently, predefined file names are used in the programs and DOS batch programs are employed to copy the files related to the current work file to these when required. Hence the intermediate files "ATT." and "WORD." and "WORD.DOC" can be found in the specriter directory after using Specriter and these represent the current Instrument Attribute File, the output from the text generation program "SPECOUT" in a form ready for text formatting, and the finished document for the current work file, respectively. It was decided not to delete these files at the end of a session because they are useful as diagnostic tools. For those who like to minimise disk usage add the following lines to the end of "SPECRITE.BAT":

```
DEL ATT.
DEL WORD.
DEL WORD.DOC
```

It has been decided to write the Instrument Attribute Files to the same directory which holds the Specriter programs. This is somewhat messy but does avoid adding additional path statements to the "AUTOEXEC.BAT" file and being wary of duplicating executable file names. Those who use Specriter in earnest may wish to create sub-directories for the Instrument Attribute Files - this is left to them.

7.2 The List Utility

The List utility is available as an option on the main menu. It calls the DOS directory (DIR) command without leaving Specriter and displays all the instrument attribute files. To return to the main menu simply press any key.

7.3 The Recall Option

Recall enables the user to select a new work file from the Instrument Attribute Files that exist in the Specriter directory. Incorrect or non-existent files are not accepted and the current work file can be retained by simply pressing Enter without entering any other characters.

8 TIDYING UP

Inevitably, anyone using another person's software will want to change the defaults. The documents produced by Specriter are in standard ASCII format and can be edited by any ASCII editor or read into most word processors. Small changes should present no problem but deletion and especially creation of new lines will need to be performed carefully. If the number of lines on the page are changed, then the document page breaks will no longer line up with the page length selected.

Keen users who wish to make substantial changes to the output text without having to fiddle with the page breaks may wish to exit Specriter and edit the file which contains the raw source text, "WORD.". This file is filled with text formatter commands but it should be possible to successfully make substantial changes without difficulty. To generate the finished document, simply type:

```
C:\SPEC> runoff word.
```

The resultant text will be found in "WORD.DOC".

Appendix 2 - *Specriter 3* Knowledge Base Technical Description

1 Introduction

Specriter 3 is a frame-based structured document generation tool which is characterised by external knowledge bases stored as disk files. It is not intended that these knowledge bases should be created and maintained using ASCII text editors. An interactive frame editing tool *Framedt* is available for this purpose and should always be used as it eliminates syntax errors which would prevent *Specriter 3* loading the knowledge base. The purpose of this document is to provide the detail necessary to permit interested programmers to interface to a *Specriter 3* knowledge base.

The first application of *Specriter 3,* and the reason for it construction, was the generation of measuring instrument specifications in accordance with established military practice. Hence there will be many references to instruments in the text to follow which could just as easily be references to monthly financial statements.

2 General Description

Specriter 3 comprises a domain-independent compiled executable program and up to three separate knowledge bases. The program contains inferencing mechanisms, the human interface drivers and text generation modules. It has been designed to produce structured documents from user-supplied responses to question options. The knowledge base characterises the behaviour of the *Specriter 3* and contains:

(1) All the information required to create the various screens that comprise the human interface to the document, say specification, under consideration. This includes context sensitive help information and function key definitions.

(2) Knowledge of the domain of interest, for example instrumentation, to provide

consistency checking and intelligent default generation.

(3) Knowledge of the appropriate document format techniques, for example specifications, held as the list of attributes comprising the screens, the help text and the final text fragments.

The knowledge is organised into a hierarchy of frames as shown in Figure 1 which illustrates the measuring instrument specification knowledge base used in *Specriter 3*. Each frame at the leaf of the tree represents a displayable screen. The totality of the information available to each frame is that contained within the frame and that which is inherited from parent frames. The parent frames contain information applicable to more than one screen and thus inheritance reduces repetition. The frames are divided into slots and these are defined in Figure 2. The types of inheritance employed depends on the slot type:

(1) No inheritance; used for slots which contain information unique to the frame or alternatively, control information. Slots of this type are Frame description and Default generation rules.

(2) Conditional inheritance; used when the inheritance process terminates when the first slot of the required type is found, be it the leaf frame or a parent. Used in Help message, Function key list, and Consistency rules.

(3) Complete inheritance; used when all the information contained within the parent slots of the same type is to be inherited. Used in Output text, Screen text, and Active fields.

The top frame contains no screen information but does contain function key definitions and a help screen which can be inherited if needed. The Level 1 frames start the frame trees for each topic to be specified. The leaf frames emanating from each, represent the diversity of screens needed to cover the range of options supported. The frame tree branches on certain options which change the document context or the output text. The appropriate leaf frames can be selected using these

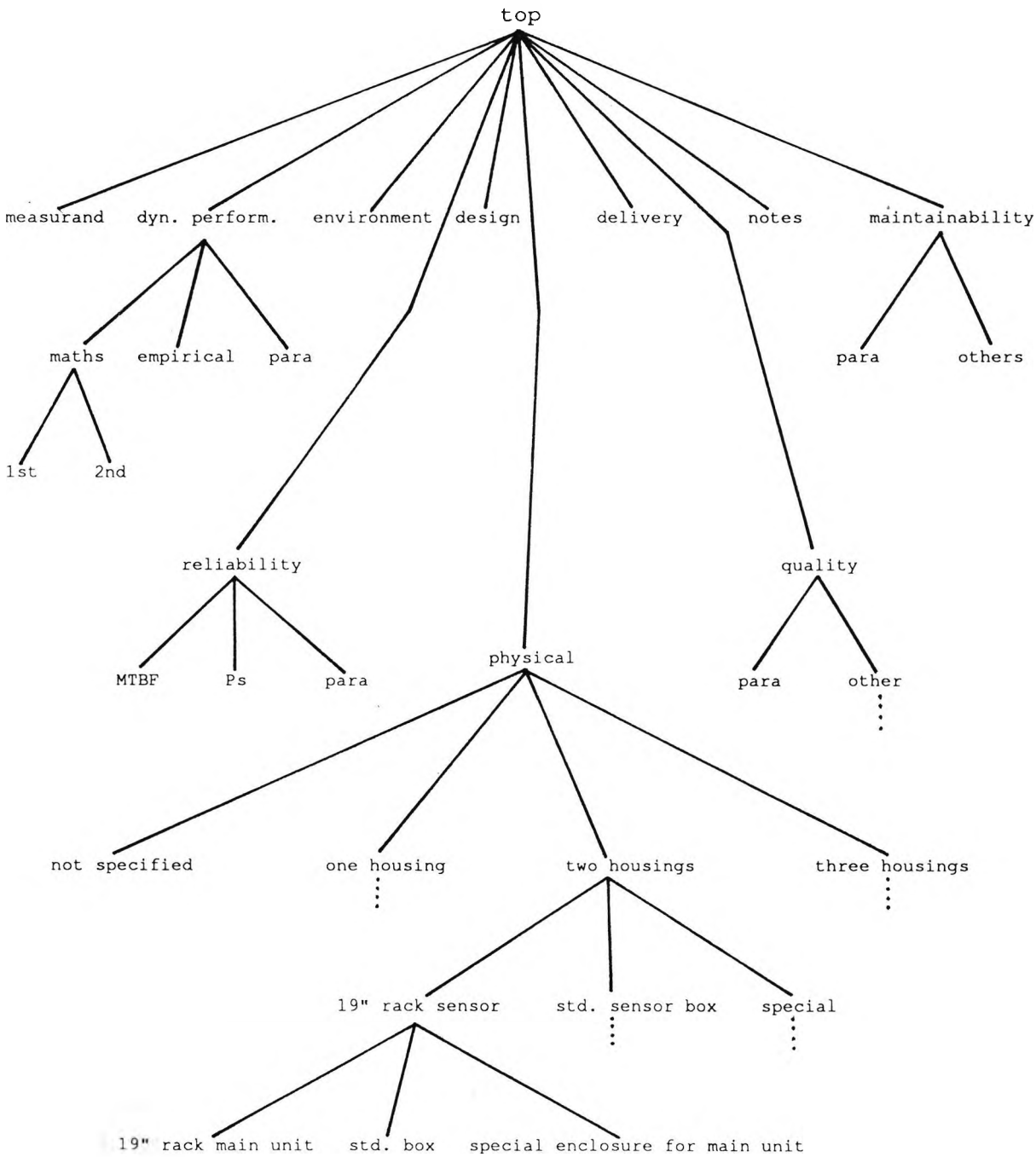


Figure 1 - Measuring Instrument Specification Frame Tree

SLOT NAME	DESCRIPTION
Frame description[1]:	A four-tuple which contains the frame name, the frame level (top = 0), the parent frame name, and a string containing a screen title to be displayed in the top centre of the window.
Output text[1]:	A single string suitable for use by the text generation utility. Any variables in the string will be replaced by their current value held in the knowledge base.
Help message[2]:	A string containing all necessary control characters for help display utility.
Function key list[2]:	A formatted string which holds the function key labels.
Screen text[3]:	A set of text strings that will appear on the screen.
Consistency rules[1]:	A set of Prolog terms. Can cover entire knowledge domain.
Default generation rules[4]:	A set of Prolog terms used to generate intelligent defaults from "High-level" frame.
Active fields[3]:	Description of the active fields which permit user interaction with the knowledge base and specification. Types include: Standard entry Predefined list entry Database assisted entry User paragraph entry Key option selection

[1] Non-inheritable slots.
[2] Optional. If absent, slot value from next highest frame inherited.
[3] Optional and all slot values from higher frames are always inherited.
[4] Optional. Level 1 frames primarily.

Figure 2 *Specriter 3* Frame Description

options. Only the questions associated with the leaf frame will be presented rather than all the possible questions relating to the topic. In this way, the human interface of *Specriter 3* can be thought of as an intelligent form which can modify itself in response to the form completion process.

3 Detailed Description

3.1 General

The knowledge base is a series of Prolog terms. The order is not important. However, as Specriter 3 is written in Turbo Prolog, which is a typed compiler, only the predefined predicate types described below can be used. Any others will cause consultation errors at run-time.

The frames are described by a number of predicates, one for each slot type. This technique has the advantage over having a single predicate for each frame, in that it makes access to the slots in the knowledge base much quicker as no list processing is needed to extract the slots. It also helps to keep the predicates to a manageable size and hence makes maintenance via a text editor much easier. Should be necessary at a later date, it would be a simple matter to declare another slot predicate type and add examples to an otherwise unchanged knowledge base. Figure 3 is an example frame definition for the measuring instrument knowledge base. It is the Level 1 frame for entering measurand, instrument name, and measuring units. The only change has been the addition of carriage returns, line feeds and spaces to assist readability because otherwise the predicates are single lines.

The only special frame in the knowledge base is the one bearing the reserved name "High_level". This frame is used as the first screen of the *Specriter 3* Create module from which intelligent defaults can be generated in response to a few high-level questions. If this frame does not exist, then the *Specriter 3* Create command will be inoperative and all document manipulation will have to be performed using the Edit command.

```
frame("Meas","top","1"," Units, Measurand, and Instrument Name ")


outext("Meas","1. Scope",
       "\n This specification establishes the performance, design, development,
        \n and test requirements for a #instrument_name#.")


help_slot("Meas",
          "The MEASURAND is the quantity the instrument is to measure.
        \n The MEASURING UNITS are the units used to express this measurement,
        \n Specriter knows about the international system of metric units.
        \n The INSTRUMENT NAME is the name used to refer to the instrument.
        \n
        \n The measurand, instrument name and measuring units form a set.
        \n Specriter contains a database on such sets which can be used to
        \n complete this screen.
        \n
        \n To use this facility, first clear all the fields with F2 and then
        \n position the cursor on the field you wish to start with and press F4
        \n to display the options from the database.  The list is sorted into
        \n alphabetical order and extends beyond the borders of the window.  The
        \n arrow keys and PgUp, PgDn can be used to find and select the desired option.
        \n
        \n Move to another field.  Repeat the process.  This time the options are
        \n limited to those consistent with the previous entry.  Similarly, complete the final field.
        \n
        \n In the event that the database does not contain the desired set, enter
        \n them as you would any other attribute. The set may be included in
        \n the database if you wish by selecting the appropriate option after F10
        \n is pressed.")


fkeys("Meas",[" F1:Help "," F2:Clear "," F3:Edit "," F4:Options "," F5:Check "," F6:Text "," F10:Finish "])


txt_slot("Meas",1,2,68,"Complete the screen in any order.  There is a database of measurands")

txt_slot("Meas",2,0,68,"available to assist completion, use F1 to obtain instructions.")

txt_slot("Meas",4,2,54,"These attributes will be used throughout the document.")

txt_slot("Meas",9,2,10,"Measurand:")

txt_slot("Meas",11,2,16,"Instrument Name:")

txt_slot("Meas",13,2,16,"Measuring Units:")
```

Figure 3 Example Frame for Measuring Instrument Knowledge Base (Part 1)

```
cons_rules("Meas",
          "run:- consult(\"name.dat\"),!, write(\"Consulted measurand database\"),nl,nl,
        \n       write(\"If no further messages appear, measurand, instrument name and the\"),nl,
        \n       write(\"measuring units do not form a set known to Specriter.\"),nl,nl,
        \n       assign_name(\"#measurand#\",\"#instrument_name#\",\"#units#\"),
        \n       write(\"OK - Set found.\"),
        \n       nl,nl.")


gen_rules("Meas",
         "run:- write(\"Measurand Establishment\"),nl,nl,fail.
       \n
       \n run:- change(\"measurand\",\"#intended_measurand#\"),
       \n      write(\"Set measurand to intended measurand.\"),nl,fail.
       \n
       \n run:- consult(\"name.dat\"),!, write(\"Consulted measurand database\"),nl,
       \n       assign_name(\"#intended_measurand#\",Name,Units),
       \n       write(\"Measurand found in the database.\"),nl,
       \n       change(\"instrument_name\",Name),
       \n       change(\"units\",Units),
       \n       write(\"Instrument name and units set to the first option in the database.\"),
       \n       nl.
       \n
       \n run:- write(\"Measurand not found in the database.  You will need to change the\"),nl,
       \n       write(\"instrument name and the measuring units later.\"),nl.      ")


slot("instrument_name",s("Meas",select,a,inherit),f(11,24,31),
     p(" ","","","Enter instrument name: ",43),
     ["name.dat","assign_name(_,X,_)","assign_name(_,_,_)"," Select Instrument Name "],
      "liquid crystal thermometer")

slot("units",s("Meas",select,a,inherit),f(13,24,31),
     p(" ","","","Enter measuring units: ",43),
     ["name.dat","assign_name(_,_,X)","assign_name(_,_,_)"," Select Measuring Units "],
     "Celsius")

slot("measurand",s("Meas",select,a,inherit),f(9,24,31),
     p(" ","","","Enter measurand: ",37),
     ["name.dat","assign_name(X,_,_)","assign_name(_,_,_)","Select Measurand "],
      "temperature")
```

Figure 3 Example Frame for Measuring Instrument Knowledge Base (Part 2)

Line feed/carriage return combinations are signalled using "\n". In the descriptions below, all symbols are literals except the arguments enclosed in the meta-symbols "<" and ">" which represent variables. A definition for each slot follows. It is essential that the correct types are used for each argument.

3.2 Frame Description

The frame description takes the form of a 4-tuple of strings:

frame(<frame name>,<level>,<parent>,<screen title>)

For example:

frame("Dynamic","top","1"," dYnamic Performance ")

The frame name is the name cited by all the slots which comprise the frame. Frame names must be unique. The frame level is a positive integer, but as a string type has been declared it must be enclosed in quotation marks. The top frame is Level 0 and the main topics are Level 1, succeeding levels are 2,3,4 ... etc. The level number is used during execution to make lists of frames at the same level. The Edit pulldown sub-menu uses this facility.

In this context and throughout this document, the parent frame refers to the next frame up in the hierarchy, not the ultimate parent. The screen title is used in two ways depending on the position of the frame in the hierarchy. In the case of leaf frames, it is used as the screen title of the editor screen. Whereas, for Level 1 frames, the titles are used as menu items in the Edit pulldown sub-menu. The order of this menu is determined by the order the Level 1 frame definition predicates are loaded into the database from the knowledge base files. This feature permits the knowledge engineer control over the order. The main menu handler uses hot keys which are highlighted. These will be the first capital letter in the title or the first letter should there be no capital letters. To enable hot keys to function correctly, title for Level 1 frames should avoid starting with a

previously used capital letter. This can be done in numerous ways; note the curious appearance of the title in the example.

3.3 Output text

The output text predicate has two arguments in addition to the frame name. They are all strings, thus:

outtext(<frame name>,<paragraph number>,<paragraph text>)

For example:

outext("Probability of survival","3.2.3 Reliability",
"\nThe probability of the #instrument_name# surviving for\n#surv_period# shall be greater than #prob_surv#.\nThe probability of survival shall be determined using\n#rel_basis2#.")

The paragraph number includes the title of the paragraph. This is separated from the paragraph text to simplify text sorting in the text generation modules. The paragraph text includes all the text for the frame in one string, including sub-paragraphs. The values assigned to the active fields can be substituted into the text by using the *Specriter 3* slot name substitution mechanism. Wherever a slot name appears in the output text surround by the escape character '#', the *Specriter 3* text generation modules will substitute the current value.

3.4 Help Message

Help messages are held in a double string-argument predicate:

help_slot(<frame name>,<message>)

For example:

help_slot("Notes","\nThe notes section contains information of a general or explanatory\nnature, but no requirements and nothing which is contractually binding.\nThe possible contents of the notes section in the recommended order of\nappearance are:\n\n Intended use\n Ordering data\n Preproduction sample\n Definitions\n Qualification provisions\n Cross reference classifications\n Miscellaneous notes.\n")

The message will be display in a window 72 characters wide so it is useful to restrict the line length. Not all frames have help_slots as they are optional. The help facility searches up the frame tree until a slot is found. It is useful to place a general message in the top frame so there is always a message to display from any frame in the system.

3.5 Function Key List

Function key definitions are held in a list in a double argument predicate, where the first argument is a string and the second a list of strings, thus:

fkeys(<frame name>,<function key list>)

Where:

function key list = [<first key>,<second key>, ... ,<last key>]

For example:

fkeys("App_docs",[" F1: Help "," F3: Edit "," F6: Text"," F10: Finish "])

The list can have from one to as many key definitions as will fit within the 79 character limit. This slot is optional so if no definition is needed the entire predicate is omitted, in which case, a key definition from a parent frame will be

used. If a key list is placed in the top frame, there will always be a key list displayed.

Specriter 3 extracts the key definitions from the list and displays them as a status line on the bottom of the screen in inverse video, black on white, separated by black spaces. To surround the key definitions by spaces, as is the practice in measuring instrument knowledge bases, it is necessary to include the space characters in the key definitions. This method provides greatest flexibility.

3.6 Screen Text

The text to be displayed on the screen is defined using any number of the following predicates:

txt_slot(<frame name>,<row>,<starting column>,
<length>,<text string>)

For example:

txt_slot("High_level",11,0,15,"Estimated cost:")

The first and last arguments are strings and the remaining three numbers are integers and must not be surrounded by double quotation marks.

Specriter 3 uses a virtual screen handler so it is possible to specify row, column and length combinations which lie outside the 21 row, 77 column active display window. These will be displayed automatically upon the user pressing the cursor movement keys initiating a search for an active field beyond the boundary of the physical display window. It is important to include a message on any screens where the virtual screen handler is employed, for the benefit of the user.

A significant side benefit of the use of this type of advanced screen handler is that errors in positioning will not give rise to fatal run-time errors.

3.7 Consistency Rules

Consistency rules are stored as a single string in the second string argument of the double argument predicate:

cons_rules(<frame name>,<rule text>)

The rule string can cover many lines; an example of this predicate can be seen in Figure 3. The rules must be written in Prolog. A list of standard predicates is given in Figure 4. Values can be extracted from any slot in the knowledge base using either the substitution mechanism of surrounding the slot name with hash symbols, for example, "#measurand#" or by using the value predicate. The later method can be used at run time. New values can be set into the knowledge base using the change predicate. It must be remembered that the slot names and values are handled as strings and must be surrounded by double quotation marks. External files can be consulted to make dynamic extensions to the database, Figure 3 shows an example of this technique where a database held in "NAME.DAT" is accessed to provide a list of relations between measurands, instrument names, and measuring units.

The intention is that consistency checking rules should alert the user to possible problems with the values of the slots in the current frame. Access is available to other frames but checking is intended to be modular although it need not be so. The potential exists for these rules to provide automatic error correction by modifying clearly faulty entries.

3.8 Default Generation Rules

Default generation is performed using the same inference engine as that employed for consistency checking. The rules are held in the double string-argument predicate below:

gen_rules(<frame name>,<default generation rules>)

PREDICATE	FUNCTION
true	Success
fail	Prolog fail
repeat	Succeeds forever
write(Term*)	Writes a list of arguments
nl	Outputs a carriage return and line feed
display(Term*)	Outputs a functor in prefix notation
read(Term)	Read a term
readln(Line)	Read a line into a string
readchar(char)	Read a character
retract(Term)	Retract a term
tell(Filename)	Redirect output to a this file
telling(Filename)	Return the current output file
told	Close the current output file
see(Filename)	Redirect input to this file
seeing(Filename)	Return the current input file
seen	Close the current input file
term =.. list	Prolog univ; conversion between a term and a list
arg(N,Term,Argn)	Unify Argn with the nth argument of Term
functor(Term,Functor,Arity)	Return functor and arity of Term or builds a new one
clause(Head,Body)	Returns clauses from the database
concat(string,string,string)	Concatenation of strings
str_int(string,int)	Conversion between a string and an integer
str_real(string,real)	Conversion between a string and a real
strg_real(string,real)	Extracts first real number from string
str_atom(string,Atom)	Conversion between a string and an atom
Integer is Expression	Evaluation of expressions
Term == Term	Testing for true equality
Term \== Term	Not true equality
Term = Term	Unify terms
Term \= Term	Test whether terms unified
Term < Term	Less than (real numbers)
Term > Term	Greater than (real numbers)
Term =< Term	Less than or equal (real numbers)
Term >= Term	Greater than or equal (real numbers)
Term >< Term	Different evaluated values (real numbers)
integer(Term)	Is Term an integer?
var(Term)	Is Term a free variable?
novar(Term)	Is Term bound?
time(Hour,Min,Sec,Hundreds)	Returns the system time
scr_char(Row,Col,char)	Print a character at a selected position
char_int(char,int)	Conversion between characters and integers
consult(Filename)	Consult named file
reconsult(Filename)	Reconsult named file
save(Filename)	Save a file
op(Priority,Assoc,Op)	Returns operators or changes operators
Goal, Goal	And
Goal; Gaol	Or
not(Gaol)	Negation
!	Cut
call(Goal)	Call
assert(Rule)	Asserts rule into database
asserta(Rule)	Asserts rule at front of database
assertz(Rule)	Asserts rule at rear of database
value(Attribute,Value)	Extracts the value to the measuring instrument attribute
change(Attribute,New_value)	Assigns a new value to a measuring instrument attribute

Figure 4 Standard Predicates Implemented in the Specriter Inference Engine

These rules will be executed whenever the "generate" function is invoked from the "High_level" screen, either by the user in the *Specriter 3* Edit mode, or automatically during Create mode.

Figure 3 shows an example of simple default generation rules from the measuring instrument knowledge base. This example is interesting because the values asserted are extracted from an external database. In fact further external rules could be consulted in the same way. Such techniques greatly extend the capabilities of the checking system and obviate the need to include all such options in the rules in the main knowledge base files. The external database can be created and maintained using an ASCII editor.

3.9 Active Fields

There are five types of active fields and each will be discussed below. They each use a single predicate and the type of the field is selected by the constant string held in "field type". The general form is as follows:

slot(<attribute name>,
s(<frame>,<field type>,<status>,<inheritability>),
f(<row>,<starting column>,<length>),
p(<lower bound>,<upper bound>,<default>,
<prompt>,<prompt length>),
[<list>],
<current value>)

where:

<attribute name>, <frame>, <lower bound>, <upper bound>, <default>,
<prompt>, and <current value> are strings;
<field_type> is one of the set symbols [att,select,option,edit,list];
<status> is one of the set of symbols [a,p];
<inheritability> is one of the set of symbols [inherit,no];
<row>, <starting column>, <length>, and <prompt length> are integers;

and <list> is a list of strings.

The attribute name is the name of the slot. It is used internally and the name is never seen by the user. The "s" functor includes the following: the name of the frame the slot is a part of, the type of field, and two arguments which were included for special control. The field types are covered individually later. Status refers to whether the field is active or not. In the measuring instrument knowledge base, this is always set to "a" to indicate that the frames are active and have actions associated with them. The alternative option, "p", for passive, will disable any action associated with the field but the field will still be displayed. Inheritability determines if a slot can be inherited by a child frame. Inheritance is enabled with "inherit" or disabled with "no". The measuring instrument slots all have inheritability set to "inherit".

Descriptions of how this general active-field predicate is used for each of the five field types is the subject of the next five sub-sections.

3.9.1 Standard Entry

Standard entry attributes are defined with the following format:

slot(<attribute name>,s(<frame>,att,a,inherit),f(<row>,<column>,<length>),
p(<min value>,<max value>,<default>,<prompt>,<prompt length>),
[],<current value>)

For example:

slot("disp_rack_depth",s(" 19\" Rack Unit",att,a,inherit),f(16,24,24),
p("100 mm","600 mm","450 mm","Enter rack depth: ",38),[],"450 mm")

The "p" functor is concerned with the elicitation of responses and holds limits and default values. *Specriter 3* when the attribute is edited a prompt window appears which displays the prompt message and the current value. On completion of

editing, indicated by pressing Enter or F10, the first real number is extracted from the string holding the proposed new value and compared against that of the pre-defined limits. If the value lies within the limits it is immediately accepted and loaded into the current value sub-slot. If not, a window is created which states this fact and displays the limits. The user can then elect to accept the new value or not. The list is not used for standard entry slots.

3.9.2 List Entry

List entry fields can be used in the same way as standard attribute entry if F3 is pressed when the cursor is on the field. However, the preferred mode of use is to use Enter or F4 to display a predefined list held in the knowledge base and then select one of the options. List entry slots are defined as follows:

```
slot(<attribute name>,s(<frame>,list,a,inherit),f(<row>,<column>,<length>),
    p(<min value>,<max value>,<default>,<prompt>,<prompt length>),
    [<option list>],<current value>)
```

For example:

```
slot("sensor_cool_select",s("Standard off-the-shelf sensor enclosure",list,a,inherit),
    f(38,24,24),p("","","natural convection",
                "Enter cooling option (manual entry defeats checking): ",70),
    ["radiation","conduction to the mounting face","natural convection","forced
    convection","unspecified means"],"natural convection")
```

In the example, no limit values are given. A blank string, or indeed any which does not contain a number, will return a zero from the numeric extraction routine. Zeros for both limits disables checking. When the entry is expected to be descriptive, as in the example, blank limit strings are the appropriate entry.

The list can contain up to 20 options. If more are required database-assisted entry fields should be used as they can cater for a very large number. The length of

the entries is not constrained but strings longer than 78 characters will be truncated. The width of the menu is automatically adjusted to accommodate either the longest entry or the menu title.

3.9.3 Database-Assisted Entry

These active fields function in the same way as the list entry fields but instead of having the options held in the knowledge base, they are held in an external database file. Database-Assisted entry fields are defined thus:

```
slot(<attribute name>,s(<frame>,select,a,inherit),f(<row>,<column>,<length>),
    p(<min value>,<max value>,<default>,<prompt>,<prompt length>),
    [<file name>,<findall argument>","",<menu title>],<current value>)
```

For example:

```
slot("measurand",s("Meas",select,a,inherit),f(9,24,31),
    p("","","not specified","Enter measurand: ",37),
    ["name.dat","assign_name(X,_,_)","assign_name(_,_,_)",
     " Select Measurand "],"flow")
```

In this case, the file name is a database file which contains the desired predicate. The findall argument is the second argument of the Turbo Prolog standard predicate "findall" which makes a list from matching facts in a database. Using the example, the executed program statement would read "findall(X,assign_name(X,_,_),List)" where "X" is an unbound fact, and "assign_name" is the functor which holds the relations between measurands, instrument names and measuring units. The resulting, "List", is then input to the list menu handler. A virtual length list handler is employed to display and select items from the list. Thus lists much longer than the number of lines available on the screen, can be handled.

3.9.4 User Paragraph Entry

User paragraph entry is quite different from all other entry possibilities. An edit window is provided which enables the user to enter free format text. This text is intended to take the place of the entire paragraph text (although this is dependant on the way the output text is written). There is no checking or intelligent default generation available for user entered paragraphs. They are defined as follows:

slot(<attribute name>,s(<frame>,edit,a,inherit),f(<row>,<column>,<length>),
p("",<field overlay>,<default paragraph>,<paragraph title>),
[<start row>,<start column>,<number of rows>,<number of columns>],
<paragraph text>)

For example:

slot("disp_housing_para",s(" Purpose Designed Enclosure",edit,a,inherit),
f(12,51,18) ,p("," Edit Paragraph ","," Display Housing Paragraph ",0),
["15","2","5","75"],"unspecified")

The field overlay is placed over the green paragraph entry field in black letters. In the measuring instrument knowledge base, the convention of surrounding the overlay string with square blocks, (ASCII 254), has been adopted. The paragraph title appears in the top centre of the edit window. The list contains the information needed to specify the window position, namely, the top left hand row and column coordinates and the number of rows and columns (including border). The paragraph text will often encompass many lines. The control sequence "\n" will give carriage return/line feed combinations to allow easy reading in the editor or in local text display.

3.9.5 Option Entry

Option entry slots are used when the selected response can alter the context of the document and hence the frame to be used to describe this option. Each of the

options available from these fields spawns a new branch of the frame tree. Option entry slots are defined as follows:

slot(<attribute name>,s(<frame>,option,a,inherit),f(<row>,<column>,<length>),
p("","",<default option>,"",0),
[<option list>],<current value>)

For example:

slot("sensor_housing_option",s("19 Inch Rack Unit",option,a,inherit),f(30,24,52),
p("","","Purpose designed sensor enclosure","",0),
["19\" Rack Mounted Sensor (Unusual)","Standard off-the-shelf sensor enclosure","Purpose designed sensor enclosure"],
"Purpose designed sensor enclosure")

The control mechanisms search through these options to find the active leaf frame, hence it is essential the current value be one of the available options. For the same reason, no input numeric checking is needed and empty strings are provided in place of the limits. Line input is not needed and thus no prompt is needed. The prompt length can set to any integer value, nominally zero to satisfy the knowledge base syntactic checker.

Appendix 3 - *Specriter 3* Technical Reference

1 Introduction

Specriter 3 is a knowledge-based system which assists in the preparation of measuring instrument specifications. It has a separate knowledge base which is described in the *Specriter 3* Knowledge Base Technical Description document.

2 Installation of Run-Time *Specriter 3*

2.1 Hardware Requirements

Specriter 3 is written in Turbo Prolog version 2.0 and can only run on an IBM PC or compatible computer. The programs and data files fit onto a single high density 1.2 M bytes 5 1/4" floppy disk. It is possible to run the system from this disk but as there is considerable disk access, operation from a hard disk is highly recommended. Temporary files are generated during execution so it is worthwhile to ensure 2 M bytes are available on the hard disk before loading *Specriter 3*. Memory requirements are 640 K bytes minimum. It will be necessary to remove terminate-stay-resident utilities to run *Specriter 3* successfully as at least 500 K bytes of free memory are required.

2.2 Software Requirements

Development and testing have been conducted on MS-DOS versions 3.2 and 3.3. No problems are anticipated with other versions. *Specriter 3* does call some DOS commands so it will be necessary for DOS to be available from the Specriter 3 directory through a suitable path setup.

2.3 Installation

Create a directory for *Specriter 3*, called SPEC3, using:

```
C:> MD SPEC3
```

Next copy the contents of the distribution disk to this directory using:

```
C:> COPY A:*.* C:\SPEC3
```

Check that at least the following files are present:

S3.BAT	- *Specriter 3* start up batch file
SPEC3.EXE	- *Specriter 3* executable program
TEXTGEN3.EXE	- Stand-alone text generator
TOPMENU.DEF	- Main menu help definition file
PROLOG.ERR	- Error message file (just in case)
PROLOG.HLP	- Help file used by screen editor
TOPMENU.HLP	- Main menu help text
SETUP.DAT	- Setup data file
*.SPC	- Specification files
TEXT_CON.DAT	- File holding text generator control data
SPEC3_1.KBA	- Knowledge base file 1
SPEC3_2.KBA	- Knowledge base file 2
SPEC3_3.KBA	- Knowledge base file 3
SPEC.MAN	- *Specriter 3* on-line manual text file
RUNOFF.COM	- Text formatter
NAME.DAT	- Database file

The following files comprise the frame editing facility *Framedt*:

FRAMEDT.EXE	- *Framedt* executable program
FE1.SCR	- *Framedt* frame-editing screen definition

FE2.SCR	- *Framedt* main screen definition
FRA_HLP.DEF	- *Framedt* help definition file
FRA_HLP.HLP	- *Framedt* help text

The following files will be created at run time:

LISTS.DAT	- Frame or attribute list, temporary file.
WORD.	- Unformatted document produced by the text generator
WORD.DOC	- Formatted file produced by RUNOFF.COM
NEXTTASK.BAT	- Batch file used to chain the text generator
*.TXT	- Finished document files

3 Executing *Specriter 3*

To run *Specriter 3,* set the default directory to SPEC3 using:

```
C:> CD SPEC3
```

and call the batch files which starts the system using:

```
C:SPEC3> S3
```

This batch file contains only two commands; SPEC3 and NEXTTASK. The idea is that if the internal text generation system senses there is insufficient memory available, for whatever reason, the external text formatting program is used instead. To do this, a batch file called NEXTTASK is generated which lists two commands; the external text generation program TEXTGEN3.EXE followed by SPEC3. *Specriter 3* then terminates and as NEXTTASK is the next command in the batch file which started the process, this new batch file is loaded and run. The external text generator then runs and displays the finished text. *Specriter 3* is then reloaded and runs once again. This rather messy sequence frees up about 200K of memory for knowledge base expansion, terminate-stay-resident utilities, or network software. If *Specriter 3* terminates using the Quit command, NEXTTASK is cleared, resulting in

a clean return to DOS.

4 The *Specriter 3* Development Environment

Specriter 3 is a sizeable software development and the files were held in a number of directories. Firstly, A Turbo Prolog Version 2.0 compiler will need to be installed on the machine according to the directions contained in the Turbo Prolog User's Guide.

The entire set of *Specriter 3* files are contained on three 1.2 M byte High Density floppy disks. The first contains the executable programs and support files, as described above, and these are loaded into a directory named C:\SPEC3. The second disk has two sub-directories: LINK and NOT_LINK which contain the sources for *Specriter 3* and the stand-alone modules respectively. The last disk has three sub-directories: SIE, FRAMEDT, and TOOLS which contain the inference engine (in stand-alone form), the frame editor, and software tools respectively. Corresponding directories should be created on the target machine's hard disk and the files loaded. If the programs are to be re-compiled it will be necessary to create sub-directories below each of the program directories named OBJ to hold the object files.

The following files should be present:

C:\SPEC3\TOOLS:

HELP.SCR	- Screen definition for HELPDEF
HELPDEF.EXE	- Help screen updating tool; used for main menu and *Framedt* help
SCRDEF.EXE	- Screen definition facility; used for editing the *Framedt* screens

C:\SPEC3\SIE:

SIE.PRO	- Inference engine main module

SIE_S.PRO - Stand-alone inference engine main module
SIE.SCA - Inference engine scanner include file
SIE.PAR - Inference engine parser include file
SIE.INF - Inference engine inferencing include file
SIE.HLP - Inference engine on-screen help include file
SIE.OUT - Inference engine screen writing include file
GLOBS_L.PRO - *Specriter* global declarations
UTILS_L.PRO - *Specriter* utilities include file
PROLOG.ERR - Turbo Prolog error message file
PROLOG.SYS - Turbo Prolog configuration file

C:\SPEC3\LINK:

SPEC3.PRJ - *Specriter* project (linking list) file
SPEC3_L.PRO - *Specriter 3* main module
EDIT3_L.PRO - *Specriter 3* editor module
TXTGEN_L.PRO - Text generation module, linkable
SIE.PRO - Inference engine main module
SIE.SCA - Inference engine scanner include file
SIE.PAR - Inference engine parser include file
SIE.INF - Inference engine inferencing include file
SIE.HLP - Inference engine on-screen help include file
SIE.OUT - Inference engine screen writing include file
GLOBS_L.PRO - *Specriter 3* global declarations
UTILS_L.PRO - *Specriter 3* utilities, linkable
LGMENU_L.PRO - Virtual menu handler, linkable
LIST_L.PRO - List handling utilities, linkable
HELP_S3L.PRO - Help system, linkable
LINEIP_L.PRO - Virtual line input driver, linkable
FNAME_L.PRO - Filename elicitation driver, linkable
TPREDS_L.PRO - Toolbox standard predicates, linkable
PROLOG.ERR - Turbo Prolog error message file
PROLOG.SYS - Turbo Prolog configuration file

C:\SPEC3\NOT_LINK:

TEXTGEN3.PRO	- Stand-alone text generation main module
GLOBS_L.PRO	- *Specriter 3* global declarations
LIST.PRO	- List handling utilities include file
TPREDS.PRO	- Toolbox standard predicates include file
UTILS.PRO	- *Specriter 3* utilities include file
PROLOG.ERR	- Turbo Prolog error message file
PROLOG.SYS	- Turbo Prolog configuration file

C:\SPEC3\FRAMEDT:

FRAMEDT.PRJ	- *Framedt* project (linking list) file
FRAMEDT.PRO	- *Framedt* main module
SCRDEF_L.PRO	- Screen definition module
SIE.PRO	- Inference engine main module
SIE_S.PRO	- Stand-alone inference engine main module
SIE.SCA	- Inference engine scanner include file
SIE.PAR	- Inference engine parser include file
SIE.INF	- Inference engine inferencing include file
SIE.HLP	- Inference engine on-screen help include file
SIE.OUT	- Inference engine screen writing include file
GLOBS_L.PRO	- *Specriter* global declarations
UTILS_L.PRO	- *Specriter* utilities include file
FILENAME.PRO	- Filename elicitation driver include file
HELP_S3.PRO	- Help system include file
LINEIP1.PRO	- Virtual line input driver include file
LIST.PRO	- List handling include file
LGMENU.PRO	- Virtual line input driver include file
SCRHND_F.PRO	- Screen driver include file
TPREDS.PRO	- Toolbox predicates include file
PROLOG.ERR	- Turbo Prolog error message file
PROLOG.SYS	- Turbo Prolog configuration file

Automatic generation of measuring instrument specifications

S. C. Cook

Engineering Manager, Vision Systems Ltd, Innovation House West, Technology Park, The Levels, Adelaide, South Australia 5095, Australia

Equipment specifications are tools used in the engineering industry to ensure equipment designed and/or supplied matches the requirements of the customer. A computer-aided engineering software package, named Specriter, is being developed to automate the specification compilation process. The progress and future direction of this research is discussed. The rationale behind the decision to transfer to a knowledge-based approach is outlined.

Keywords: CAD, Systems engineering, Heuristic programming, Instrument specifications

1 Introduction

Specriter is a part of the Computer-Aided Engineering of Instruments software package (CAEINST) under development by the Measurement and Instrumentation Systems Centre (MISC) of SAIT (Sydenham, 1987). CAEINST aims to provide the necessary knowledge for a user to specify, create, and apply capable measurement and control systems.

Specriter fits near the beginning of the CAEINST process. Its primary function is to extract information from the user regarding the measurement problem to be solved, and from this, generate a written specification for a measuring instrument. The other major function is to make the information which defines the instrument, ie, the essence of the written specification, available as direct input to programs which can produce a detailed design.

2 The need to automate the specification generation process

Defining a product to be designed or purchased is an important task in all fields of engineering. This definition usually takes the form of a written specification. Regardless of the form chosen, the purpose of the specification is to convey the requirements identified by the customer to the provider of the goods or services. A good specification is essential to ensure that the equipment, or service, to be provided will fulfil the customer's actual requirement.

Where the requirement can only be satisfied by developing a new product, the stakes are much higher. Product developments commenced with inadequate specifications inevitably lead to expensive product iterations and, all too often, disappointing final results. Expressed in commercial terms, poorly controlled development programmes equate to cost overruns and uncompetitive products. The larger system engineering and product development companies well recognise this situation, and establish procedures to endeavour to control engineering tasks (Burgess, 1969). All such procedures emphasise that the first step in the design process is to clearly identify the requirements for the items to be developed.

With so much depending on the adequacy of specifications, it is disturbing that the traditional methods of specification preparation are far from ideal. Not only does the generation of a document usually consume many man-hours, but the result is often incomplete and marred by inconsistencies and ambiguities. Another fundamental limitation is that the knowledge contained within a specification is limited to a subset of the knowledge held by the authors. If this were not bad enough, documents, however precisely written, are always open to interpretation. It is not surprising, therefore, that more thorough specifications are often called for. Hence the need for tools in this area.

Specriter aims to permit an inexperienced engineer or scientist to produce a specification for a measuring instrument which is at least as good as one produced by an experienced specification writer. Furthermore, Specriter aims to complete the task in an hour or so rather than days.

3 Specriter implementation

3.1 Specriter 1 design requirements

At an early stage, it was determined that the intended users of Specriter would be engineers or scientists who had only modest computing skills and no specialist training or experience in instrumentation. This baseline firstly establishes the need for a sophisticated human interface optimised for simplicity of use, and secondly, necessitates the incorporation of the measurement science and specification writing knowledge lacking in the targeted users.

The latter requirement indicates the use of artificial intelligence techniques. However, processing knowledge as opposed to numerical data is a relatively recent development of computer science and is a current research topic. To gain an understanding of what was truly required in both the computing and measurement

science fields, it was decided that research should be combined with the production of a demonstrator program written in Pascal. Specriter 1 is the outcome of this activity.

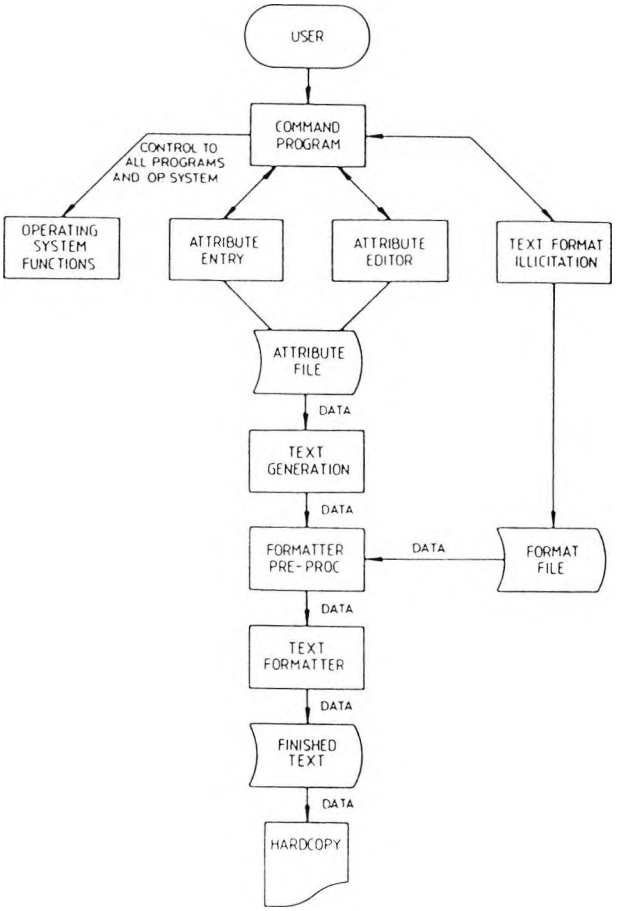


Fig 1 Specriter 1 structure

Fig 1 is a block diagram of Specriter 1 illustrating the connectivity between the programs and files. Instrument attribute data files are used to store the information needed to characterise a particular instrument. These files are created by the entry program and can be modified by the editor. The text generation programs operate on the data stored in these files to produce the final printed text. To enable minimal effort portability across computers and their operating systems, all system calls are handled by a single command program which is written in the operating system command language. Installation on a particular computer requires an ANSI-compatible PASCAL compiler, a text formatter and a purpose-written command program. To date, versions exist for VAX/VMS and MS-DOS.

Even for the first demonstrator, considerable thought has gone into ensuring that the targeted users are able to use Specriter easily. Once invoked, there is no need to leave the Specriter environment until the job is finished. Tasks normally performed by operating system commands such as running programs, copying files, and printing text are all handled from within Specriter. In order to get started, all the user is required to know is how to log on to the computer and the single command to start Specriter.

3.2 The form of the specification produced

Organisations charged with producing national and international standards have compiled documents to describe the format and content of equipment specifications. One of the most comprehensive and widely used formats is that described by MIL-STD-490 written by the United States Military.

1 SCOPE
2 APPLICABLE DOCUMENTS
3 REQUIREMENTS
3.1 Instrument Definition
3.1.1 General Description
3.1.2 Interface Definition
3.1.2.1 Electrical Interface
3.1.2.1.1 Power Interface
3.1.2.1.2 Communications Interface
3.1.2.1.3 Electromagnetic Compatibility
3.1.2.2 Mechanical Interface
3.1.2.2.1 Sensor
3.1.2.2.2 Data Processor
3.1.2.2.3 Display
3.1.2.3 Thermal Interface
3.1.2.3.1 Sensor
3.1.2.3.2 Data Processor
3.1.2.3.3 Display
3.2 Characteristics
3.2.1 Performance
3.2.1.1 Range
3.2.1.2 Discrimination
3.2.1.3 Repeatability
3.2.1.4 Hysteresis
3.2.1.5 Drift
3.2.1.6 Dynamic Response
3.2.1.7 Measuring Error
3.2.1.8 Power Consumption
3.2.2 Physical Characteristics
3.2.2.1 Sensor Physical Characteristics
3.2.2.2 Data Processor Physical Characteristics
3.2.2.2.1 Data Processor Housing
3.2.2.2.2 Data Processor Mass
3.2.2.3 Display Physical Characteristics
3.2.2.3.1 Display Housing
3.2.2.3.2 Display Mass
3.2.3 Reliability
3.2.4 Maintainability
3.2.5 Environmental Conditions
3.3 Design and Construction
3.4 Documentation
3.5 Logistics
3.6 Personnel and Training
4 QUALITY ASSURANCE
5 PREPARATION FOR DELIVERY
6 NOTES

Fig 2 Paragraph headings for a measuring instrument specification

Using this format, it is possible to write a generic equipment specification for measuring instruments covering a wide range of measurands and measuring situations. Fig 2 gives the paragraph headings for a document of this type, which is specifically tailored to describe instruments with a physically separate sensor, data processor and information display. Clearly, not all the topics will be applicable to each example but this represents no problem as 'not applicable' can be entered

or the paragraph removed as appropriate. Fig 3 is a typical paragraph produced by Specriter.

For most paragraphs the information to be written will be either a numeric boundary – for example, 'Power dissipation shall be less than 2 W' – or one of a limited number of pre-defined alternatives. Armed with this knowledge, it has been possible to produce computer programs which extract details from the user and produce a formatted specification written in English.

3.1.1 General Description

The pressure gauge comprises three modules: viz, the pressure sensor, the data processor and a display. Each of these modules is separate and is to be selected from standard catalogue lines wherever possible to minimise cost.

The sensor shall receive any power required from, and output data to, the data processor in unprocessed form. The data processor shall process the input stream to provide the necessary information for display to the user. Compensation for temperature and influence effects may be performed at the sensor, the data processor or a combination of both.

Fig 3 A typical paragraph output from Specriter

4 Producing formatted specifications using Specriter 1

4.1 The main menu

Specriter is invoked from the computer operating system by a single command: for example, in the MS-DOS version, "SPECRITER". Following the presentation of a banner giving version details, the command program generates the main menu. The user has the choice of creating a new specification or working on one created previously. A list of attribute files held in the computer is available by selecting one of the menu options. The current instrument attribute file under consideration (i.e. the workfile) is established by either selecting one from the list or by selecting the option which invokes the attribute entry program. Further options are available to enable editing of the workfile, generation of the text from the information contained within the workfile, and printing of the final text.

4.2 Information entry

The entry program is selected from the main menu by a single keystroke. The user is then prompted for a file name to identify the instrument specification about to be produced. Once a name has been entered, the information entry process commences.

Information entry is accomplished by a question and answer format. The order of the questions is based upon the concept of extracting a high proportion of the information in early questions with the detail following. This technique helps the user to build a mental image of the instrument at a functional level before the detail is tackled. Fig 4 shows this order and this can be compared with Fig 2.

1 Measurand definition
2 Performance characteristics
3 Interface definition
4 Physical characteristics
5 Environmental conditions
6 Quality assurance provisions
7 Preparation for delivery
8 Reliability
9 Maintainability
10 Design and construction

Fig 4 Order of tackling topics

Requested responses fall into one of the following categories:

(a) A numeric limit to a particular parameter.
(b) Selection of one of a given set of alternatives.
(c) A user-entered descriptive paragraph.

Entry of a user-defined paragraph is available as an override option to many questions. This provision caters for unusual instruments not covered by the internal knowledge base used to source the questions and the range of response alternatives. It is expected that this facility will be needed infrequently.

Once invoked, the information entry program proceeds through the questions until all the necessary information has been extracted. Users are prompted to enter "TBD" (to be determined) for parameters which cannot be entered during the current session but will be defined later. If no entry is made, the program inserts "TBS" (to be specified) to highlight the fact. Use of these two forms is common in draft specifications compiled in industry. Their use enables feedback from engineering staff, or a contractor, to be obtained to the bulk of the document during the period when the remaining details are being defined.

Considerable thought has been given to the presentation of the questions. Rather than have the dialogue with the user roll up from the bottom of the screen, a cursor-addressable terminal is used which permits defined placement of characters on the screen. Major topic heading appears in inverse video at the top of the screen and subtopic headings appear similarly highlighted, directly above the relevant questions. Information is given on the screen to aid question answering, and help facilities are incorporated. This type of screen layout has proven to be particularly effective for the targeted users.

The minimum number of questions are presented to the user. This is achieved by interpreting the responses to key questions to select the appropriate question sequence. Although conceptually the information entry program is equivalent to completing a form, efficiency gain is achieved by question list truncation and the explanations given. Help facilities should improve productivity further.

After completion of information entry, the essence of the information gathered is written to the instrument attribute data file named at the start. Control is then returned to the main menu. The user may wish to generate and print the document at this stage or go directly into the editor to correct errors and omissions that may

have come to mind during the course of the entry session.

4.3 Information editing

The editor is invoked directly from the main menu. The instrument attribute file to be edited is the workfile shown in this menu, which will either be the file just created by the entry program or one previously selected from the library. Editing takes the form of altering the previous responses to a question. Rather than sequence through all the questions again, the questions are divided into the 10 topics shown in Fig 4 and a single topic at a time is edited by selecting it from the editor menu.

For each topic, the user is presented with the original questions followed by the previous responses enclosed in square brackets. The user can then elect to leave an entry unchanged by simply typing "RETURN", or enter a different response. User-entered paragraphs can be edited on a line by line basis.

One or more topics can be edited as many times as necessary in one editing session. Once the user is satisfied with the information entered, the attribute file is written, the editing session terminated, and control returned to the main menu.

4.4 Alternative information entry

Instead of answering the complete list of questions needed to specify a new instrument, a user can nominate an existing instrument attribute file which describes a broadly similar instrument and alter it to meet his exact requirements. This method provides inexperienced specifiers with even greater assistance, as default responses will exist for a similar measurement situation. A documented library of such files is being compiled for Specriter. Each time a new instrument attribute file is created using the Specriter entry program, another file is added to the library. A facility to help choose the most appropriate file for a particular application will be added as the number of files grows.

An extension of this idea, which is being investigated, is to partition the attribute file into a number of smaller files. For a given number of these smaller files it would then be possible to more closely match a measurement situation. This idea is analogous to using a 'photofit' for human identification.

4.5 Text generation

Text generation is performed by selecting the appropriate option on the main menu. This program reads the specified instrument attribute workfile and proceeds to construct the text of the specification. This process comprises merging the user's responses into stored paragraphs, deleting unwanted paragraphs, and selecting the most appropriate paragraph when there is a choice.

The output from this program is a text file in a format suitable for input to the text formatter used to produce the final printable text. A future refinement will be to add a pre-processor for the text formatter to provide user selectable text formats. Such details as page length, column width, indenting of paragraphs, and headers and footers (containing document number, issue number and date) will be attended to by this program.

Control of the text pre-processor will be achieved via a text format specification which would normally be set to the default shown in Fig 5, which is that adopted by Vision Systems Ltd (Cook, 1986) and is similar in concept to the standard document format of many companies. User-defined text format specifications, to suit either a company's format or the printing hardware, will be entered using special menu.

5 Limitations of the current approach

The demonstrator program suite, Specriter 1, has shown that computer-aided specification generation is both feasible and of value. Pursuing the method of producing formatted specifications outlined, through to a finished product, would yield a useful tool for instrument engineers who need to specify instruments.

Nevertheless, it is felt that such an approach only partially fulfills the goals of the research program. Although a user needs to know nothing about how to write a formatted specification or even the normal content of such a document to produce a specification using Specriter 1, he is still required to be familiar with instrument performance characteristics and the effects of operating environments on these. In short, to specify an instrument well with Specriter 1, an instrumentation expert is still required.

Further study of knowledge engineering techniques and evaluation of some software tools indicates that Intelligent Knowledge Based Systems (IKBS) could be applied to further reduce the level of user knowledge

Page width:	210 mm (A4)
Page length:	296 mm (A4)
Left margin:	30 mm
Right margin:	20 mm
Top margin:	20 mm
Bottom margin:	20 mm
Centre header:	Page number
Right header:	Document number
Centre footer:	Issue number
Paragraph indenting:	Two spaces per paragraph level
Miscellaneous:	Right justification enabled
	Paragraphs separated by blank lines
	Single line spacing
	Two blank lines before major sections
	Major paragraph headings capitalised

Fig 5 Page format default

necessary to generate an instrument specification. A possible development is to use one or more IKBS to process the response to higher-level questions and produce the instrument attribute file used by the existing Specriter system. The provision of limited natural language understanding has been considered to avoid the regimentation of multiple choice selection.

However, this is only a partial solution as a fundamental deficiency still remains in that the current software, even enhanced as described above, does not explicitly address the meaning of the specification and of the information comprising it. This makes it difficult to ensure that the resulting specification is well formed. Furthermore, to enable a CAD system to design the instrument, a more suitable method of representing and extracting the knowledge contained within the specification is required. For these reasons it became clear that a better method of knowledge representation than that employed to date would be preferable.

6 The way ahead

Recent developments in software engineering, in particular the field of automatic programming (Balzer, 1985; Borgida, 1985; Alexander, 1985), provide an alternative approach which overcomes the limitations of the above and is altogether more suited to the task.

Automatic programming research endeavours to convert a user's requirement for a computer program, expressed in human terms, into an executable program. The heart of such a system is the formal specification which is generated either directly by the user or from requirements stated to the computer in an English-like form. In this context a formal specification is a set of requirements expressed in an unambiguous notation that allows the entity described to be reasoned about. Fig 6 illustrates the conceptual construction of the next generation of Specriter. At the innermost level is the underlying logical framework of the system, which will be based on that of Prolog as this language is both available and appropriate for the task (Subrahmanyan, 1985; Bratko, 1986). The next level will be an instrument description language which will define the syntax, rules of semantics and rules of the domain of the next level. It will be optimised for the domain of measuring instruments. Above this resides the set of rules in the instrument description language which model or specify a particular instrument.

These elements comprise a formal specification. Construction and interrogation of the specification are achieved via a set of programs known as views. Entry of the specification can be achieved by similar methods to those already in use. The output of the entry program is now not a data file but a set of rules. Alternative entry programs encompassing IKBSs, as previously proposed, would be able to generate the whole or a portion of this set from the responses to higher-level questions. Production of a formatted specification is performed by interrogating the model and, for compatability with the software already developed, production of a suitable attribute file. Similarly, editing facilities can be provided at both the attribute level and the higher level of human entry using developments of the existing program. CAD interface is achieved by a purpose-written interrogation program.

It is believed that the proposed concept has the potential to fulfill the aims set down for Specriter. The important point is that the nexus of the system is to be a model of the measuring instrument under consideration, which shall permit the meaning of the model (i.e, the specification) to be reasoned about.

Extensions and enhancements to the model which defines the instrument can be achieved by the inclusion of additional rules in the model description language.

The entry program can be written to prevent the generation of specifications which are not well formed by reasoning about the information entered to date in conjunction with the current information presented to the model before it is accepted. Establishing what constitutes a well formed specification for a measuring instrument is the key to the success in this area and is currently a research topic.

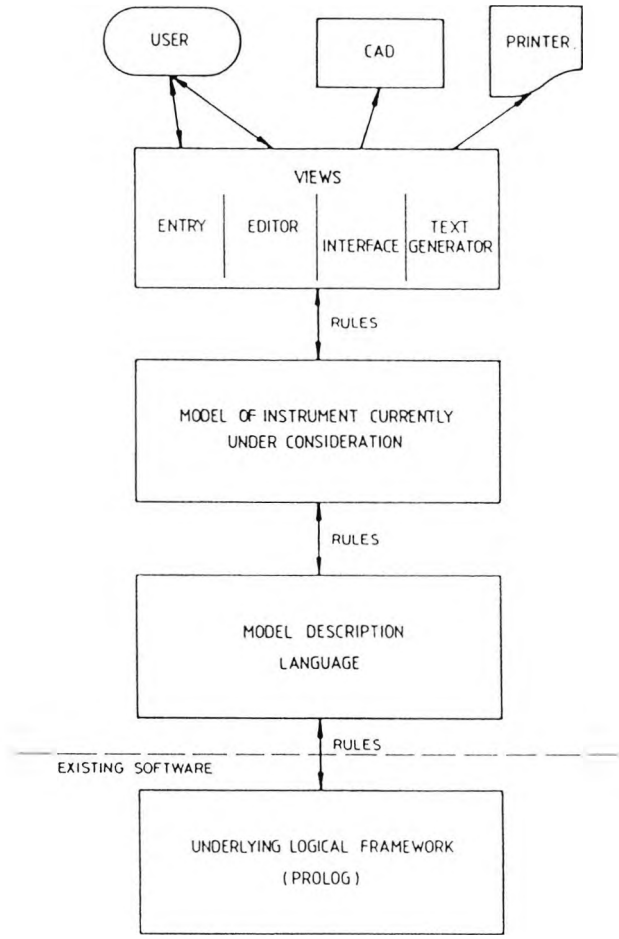


Fig 6 Specriter 2 structure

7 Conclusion

A demonstrator program suite, named Specriter 1, has been produced which is capable of producing formatted specifications for instruments. Requirement entry and editing is performed via a screen-addressable computer terminal, and the philosophy of minimal typing has been implemented. A fully developed version of this program will be a useful aid to the generation of specifications for instruments, but requires the user to be skilled in measurement science to ensure the result

is meaningful. For this reason, Specriter is best considered as an efficient alternative to the manual method of specification generation currently used in industry.

Research is continuing into more fully automating the process to enable a scientist or engineer, who is not an instrumentation expert, to enter a requirement into a computer and obtain a written specification for equipment to fulfil it. A CAD interface program will be developed to interrogate the information stored in the computer to generate input to an instrumentation CAD suite.

8 Acknowledgements

The author wishes to thank his joint research supervisors, Professors L. Finkelstein and P. Sydenham, for their direction and encouragement from the commencement of the research project. Dr A. Finkelstein, who introduced the concepts of formal specifications to the project, also deserves special mention.

References

Balzer, R. 1985. 'A 15-year perspective on automatic programming', *IEEE Trans on Software Eng*, **SE-11** (11).

Borgida, A. *et al.* 1985. 'Knowledge representation as the basis for requirements specifications', *IEEE Computer*, April.

Bratko, I. 1986. *Prolog programming for artificial intelligence*, Addison-Wesley.

Burgess, J. A. 1969. 'Organising design problems', *Machine Design*, November 27.

Cook, S. 1986. *Document preparation guidelines*, Vision Systems Ltd.

Subrahmanyan, P. 1985. 'The "Software engineering" of expert systems: Is Prolog appropriate?', *IEEE Trans on Software Eng*, **SE-11**(11).

Sydenham, P. 1987. 'Computer-aided engineering of measuring instrument systems', *Computer-aided Engineering Journal*, **4**(3), 117–123.

Knowledge-Based Generation of Measuring Instrument Specifications

Mr. **S. Cook,** London/GB

Abstract

The task of writing specifications for measuring instruments requires knowledge from many fields, including, specification writing practices, measurement science and current instrumentation practice. It is a specialist task which is time consuming and difficult to do well. This paper describes the derivation of a knowledge-based system, Specriter 3, which produces specifications of measuring instruments by user interaction. The representation paradigm for the domain knowledge and how it is used is discussed, and the potential and use of the software outlined.

1. Introduction

Specriter is a computer-aided engineering tool which assists engineers and scientists in the production of requirements specifications for measuring instruments. The knowledge incorporated into Specriter enables a user, who has little experience or training in measurement science and instrumentation, to produce a document which rivals that of instrumentation specialist but in much less time. Furthermore, the internal computer representation can be used to provide a degree of consistency and reasonableness checking and a valuable interface to other CAE packages.

Specriter is a part of the Computer-Aided Engineering of Instruments software package (CAEINST) under development by the Measurement and Instrumentation Systems Centre (MISC) of SAIT /1/. CAEINST aims to provide the necessary knowledge for a user to specify, create and apply capable measurement and control systems. The session commences by running MINDS /2/ to determine appropriate measurands and then Specriter is called to prepare specifications for the instruments identified, in both computer usable form and in printed form. Additional programs, currently under development, at SAIT and City University, London /3/ will digest the computer usable specification and attend to the design, implementation, and application of the resulting instruments.

2. Specriter 1 - A Problem Definition Exercise

Automatic generation of measuring instrument specifications was first demonstrated by a Pascal program suite entitled Specriter 1 /4/ which produced a written equipment specification in the well known U.S. military format /5/.

This software asked a sequence of questions and used the responses to direct the construction of a printed specification. Intelligence was limited to removing

questions from the sequence if a previous responses had rendered their asking redundant. Once the initial entry process was completed, the questions and associated responses were grouped into ten categories, thus permitting editing without sequencing through all the questions. On the completion of editing, the text generation programs produced a document in the military style according to a physical layout predefined by the user.

Specriter 1 was surprisingly successful, and has been used at MISC to produce specifications for many measuring instruments. It is also used routinely at SAIT in the undergraduate teaching program.

More importantly, Specriter 1 succeeded in defining the scope of the task through the identification of over 100 measuring instrument attributes requiring specification. Another important result was the identification of short lists of alternative specification methods for many topics, for example maintainability, reliability, quality, and dynamic response. Finally, it demonstrated that the automation of specification generation was viable starting from attribute entry, through editing to data-controlled document generation.

Although Specriter 1 captured a useful amount of knowledge about specification generation, it lacked the expertise of the instrumentation engineer. This is needed to provide adequate responses to the many questions and conduct consistency checking. The classically procedural implementation of Specriter 1 made the addition of this largely heuristic knowledge difficult.

Furthermore, Specriter 1 also served to establish the limitations of a written specification in the design process; the only thing that can be done with such a document is to give it to a human to read. Hence it became clear that progress in computer assistance hinged on representing the specification of the instrument in a form that permits computer reading and subsequent reasoning. Facilities such as consistency and reasonableness checking, high-level entry and interface to other CAEINST packages can then be much more easily supported.

Hence, it was decided to take the knowledge gained from the software, the user comments regarding the need for context-sensitive help and improvements to the user interface /6,7/ and seek a better representation for both the specification itself and the knowledge needed to produce it.

3. The Introduction of Formal Methods

Formal methods, as applied to software engineering, automatic programming and protocol verification /8,9/ were investigated as a possible specification representation technique. Automatic programming research endeavours to convert a user's requirement for a computer program, expressed in human terms, into an executable program. The heart of such a system is the formal specification which is generated either directly by the user or from requirements stated to the computer in an English-like form. In this context, a formal specification is a set of requirements expressed in an unambiguous notation that allows that entity to be reasoned about. Reasoning is performed using an inference mechanism operating on the formal specification and an associated knowledge base.

It was important to consider the implementation of other aspects of the task when considering which of the formal methods to pursue. The knowledge of measurement science and specification practices has to be stored in a form which is appropriate for the reasoning mechanisms which need to operate on the specification. In addition, the user interface needs access to both entities to provide knowledge-assisted entry and integrated context-sensitive help features.

Clearly, a representation which combines these requirements would be ideal. Prolog has been advocated for property-oriented non-algebraic formal specifications which can also be directly interpreted using a resolution theorem proving approach /8/. Prolog is also widely used for implementing knowledge-based systems, hence it was decided to examine the potential of this language for the task. This decision avoids the need to construct a specification language for measuring instruments and write an interpreter. Borland's Turbo Prolog Version 2.0 was selected to investigate the production of a limited formal system because of its library of screen handling predicates and good program development efficiency.

4. Specriter 3 Description

Specriter 2 was a Prolog re-implementation of the Pascal software and primarily constructed as a learning exercise in Prolog and screen design. Specriter 3, however, is truly knowledge-based. It employs a knowledge representation technique which encompasses the entire domain. In fact, the editor program can be thought of as a shell because it is domain independent and could be used to assist in the creation of a range of structured documents.

4.1 Knowledge Representation

The knowledge representation requirements for Specriter are somewhat unique in that domain is broader than that encountered in many knowledge-based systems. Two specialist areas, specification generation to MIL-STD-490A and instrumentation had to be included along with most of human interface control and a portion of the program control. The aim was to remove all the domain specific information from the compiled programs and place it in external files. This would then permit a complete separation of user interface, inference engine and control.

When tackling such a large domain it is useful to try and identify an underlying structure in the knowledge. Specriter 1 successfully demonstrated that editing could be performed on the mutually independent question groupings below:

Measurand Definition
Performance Characteristics
Physical Characteristics
Interface Definition
Quality Assurance
Design and Construction
Reliability
Maintainability
Preparation for Delivery
Operating and Storage Environment

These were used as the first level partition for the knowledge base. User feedback identified that it was desirable to display on one screen the entire grouping of between 3 and 15 questions. This necessitated that the sequential nature of question asking and the associated control mechanism be abandoned. Instead, a set of screens was proposed, one for each question grouping. Redundant question removal would then be handled by expanding each question grouping screen into a set of slightly different screens. The actual screen being displayed from this set would then be dependant on responses to option questions. After changing a key option, screen re-selection would occur. It was noted that not only does the screen need to change to display the appropriate question list, but the context for reasoning is also changed. This then is the rationale for the second and subsequent partitions of the knowledge base. Thus a tree structure can be created. A particular instrument would be described by the responses to the questions on the screens represented by active leaves of the tree.

Having defined the requirements for the knowledge base, it remained to identify a suitable knowledge representation technique. From the structured techniques, a variation on the concept of frames introduced by Minsky, /11/ was selected to represent all the domain knowledge. In this case, a frame represents a user interface screen to which is coupled the following: related questions and answers, the active function keys, the help message, the output text, and heuristic rules. Figure 1 illustrates the slots which make up a frame using the format from reference 12.

A child frame is called whenever a user entered option is selected which requires that the screen be changed. This child can inherit all but the output text from the parent reducing duplication of knowledge. Progressively lower-level child frames can be called from other option fields within the displayed frame. The knowledge base thus becomes modular and the rule-based consistency checking facility can be a manageable size.

The instrument under consideration is represented by the instantiation of the set of values of the instrument attributes. These are incorporated into the knowledge base structure and are available to the reasoning mechanisms inherent in Prolog as directed by the contents of the knowledge base.

4.2 Inclusion of Heuristic knowledge into Specriter 2

Heuristic knowledge comprises the rules of thumb and intuition humans use so successfully to solve problems for which there exist no problem solving algorithm or other rigorous technique. By their very nature, heuristics tend to be unstructured and are best represented by production rules and facts. However, rules can be grouped to solve sub-problems. Specriter 3 uses these observations to extend the concept introduced by Aikins /13/ of combining production rules with frames. This has the advantage of retaining the frames structure which ideally suits the screen-based user interface whilst utilising the convenient representation of heuristics as rules. The modular nature of the knowledge base, helps keep the number of rules to be dealt with at any one time within practical limits. There are two types of rules used in Specriter:

Figure 1 - Specriter Frame Description

Way of Specifying:	Higher-level frame (parent).
Frame Level:	1 onwards.
Screen Heading:	Screen title.
Screen text:	Many individual text fields can be used on each screen.
Help message:	A single string suitable for use by the screen display utility.
Output text:	A single string suitable for use by the text generation facility. Any variables in the string are replaced by their value currently held in the knowledge base.
Function key list:	A formatted string which holds the function key labels.
Consistency rules:	A set of Prolog terms.
Generation rules:	A set of Prolog terms.
Active fields:	Fields displayed that user interacts with.
Standard entry:	Line editor-based string entry
Question prompt:	
Default value:	
Default minimum value:	
Default maximum value:	
Actual value:	
Screen position:	
Predefined list entry:	Selection from a predefined list - used for control
List title:	
Default selection:	
Actual value:	
Screen position:	
Option list:	
Database assisted entry:	Selection from database held in a file or alternatively, normal standard entry.
Screen position:	
Rules for assistance:	
Actual value:	
Control:	Used to invoked user paragraph entry.
Procedure to execute:	
Key options:	Entry from a predefined list. Forces change to another frame.
List title:	
Default selection:	
Actual value:	
Option list:	
Screen Position:	

(a) Those which test the values in a particular frame for consistency and reasonableness;

(b) Those used to generate default values, and default limits from a small number of high-level questions.

The rules are held in Prolog syntax and are directly executable. It is thus possible to construct whatever form of knowledge-based system in best suited to each screen. To date, the backward-chaining paradigm inherent in Prolog has been employed.

4.3 Human Interface and Program Control

The human interface comprises a main menu together with pull-down sub-menus which are used to call the various program modules. These modules can be partitioned into two types: utilities and program specific functions. The first category covers file handling of instrument specification files, access to the on-line manual, access to the operating systems and facilities to tailor the Specriter environment. These features, in common with most modern software products, obviate the need for detailed familiarity with the operating system.

The program specific functions will be described in order of use. CREATE is used whenever there is no suitable instrument specification in the library which can be modified to suit the current requirements. It begins by invoking the high-level question screen which calls for such information as the measurand, physical arrangement, intended use, intended life, operating environment, expected cost etc. When this screen is complete, the generation of detailed attribute values and the numeric default limits commences. The program then searches for areas not satisfactorily dealt with and prompts the user to complete these screens. Upon completion of this process, the user is directed to use the EDIT program to view and modify, as necessary, the detailed values.

The heart of Specriter 3 is the EDIT program. The main menu edit pull-down window displays the list of level 1 frames, i.e., the first level question groupings. A topic is selected by positioning the cursor over the required option and pressing the Enter key. This causes the lowest level child frame defined by the values of the internal options, to be displayed. The screen comprises text and five types of colour-coded active fields as described in figure 1. These fields permit values to be selected from lists, entered as free text using a line editor or entered as paragraphs using a screen editor. Assistance is available through context-sensitive help and each numeric input is compared to default reasonableness values to help identify errors. The consistency checker looks at the values entered in the screen as a set and attempts to identify problems. From within the editor, it is possible to display the output text relevant to the current screen. This allows the user a convenient opportunity to modify responses to suit the text stored in the knowledge base.

Once editing is complete, the complete text can be displayed or printed using the TEXT facility. This program extracts the paragraphs from the active leaves of the knowledge tree structure and sorts them according to their paragraph number.

Page numbers, headers and footers, indenting and date stamping are then included. Finally, the text is formatted using a traditional text formatter to produce the finished document.

4.4 The Frame Editor

The traditional method of entering frame-based knowledge is to write it out in a language specifically designed for the task and then employ a purpose-built interpreter to load in into the knowledge-based system. Since the entire knowledge base is held as Prolog terms, it is relatively easy to enter it directly in this form using an ASCII editor. However, to make the task of creating and maintaining this rather substantial knowledge base easier, a frame editing program was written. This program allows the user to create and edit frames using a variety of tools. A similar user interface to that developed for Specriter is employed for all the non-displayed knowledge, whilst the actual screen can be crafted using a screen definition program. Thus it is a straightforward task to edit the knowledge base and add new frames.

5. Conclusions

A new program suite which can generate measuring instrument specifications has been described. The core concept employed is to represent the requirements specification for the instrument under consideration, as a formal specification written in Prolog. This specification can then be reasoned about using sets of rules held within a knowledge base. A domain independent shell is used to perform the specification generation, and limited checking under control of the contents of the knowledge base. The final text is generated using another program which views the knowledge base and the instantiated values which represent the instrument under consideration. The knowledge base can be easily updated using a separate frame editing facility.

Specriter 3 goes a long way to fulfilling the aim of producing a user-friendly knowledge-based specification generation tool. It is certainly easy to use and can produce a useful specification,
however, the knowledge base would benefit from further research.
It is proposed that the internal Prolog representation of the specification be used as the backbone for the CAEINST package. Then not only can the design, implementations and performance simulation programs use the specification as input, but it opens the way for output from each stage of the design process to be automatically verified against the specification. The latter can be used to evaluate bottom-up detailed designs which can often be more efficient solutions.

6. Acknowledgements

The author would like to acknowledge the guidance and support of his supervisors, Professors P. Sydenham and L. Finkelstein throughout the research program. The informative discussions with Dr. A. Finkelstein which led to the development of the unified knowledge base are gratefully acknowledged. The author is a Postgraduate Fellow employed by the Electronics Research Laboratory, Defence Science and Technology Organisation, Salisbury, South Australia.

References

/1/ Sydenham P.H.: Computer-aided engineering of measuring instrument systems. Computer-Aided Engineering Journal, June 1987.

/2/ Sydenham P.H., Harris D.D., Hancock N.H.: MINDS - A software tool to establish a measuring system requirement. To be published in Measurement 1990.

/3/ Mirza M.K., Neves F.J.R., Finkelstein L.: A knowledge-based system for design-concept generation of instruments. Measurement Vol 8 No1, Jan-Mar 1990.

/4/ Cook S.C.: Automatic generation of measuring instruments. Measurement Vol6 No4, Oct-Nov 1988.

/5/ MIL-STD-490A Military Standard - Specification Practices. United States of America Department of Defence, 4 June 1985.

/6/ Sydenham P.H.: Private communication, 1989

/7/ Goldsmith K.: Private communication, 1989

/8/ Cohen B., Harwood W.T., Jackson M.I.: The Specification of Complex Systems, Addison-Wesley, U.K. 1986.

/9/ Balzer R.: A 15 Year Perspective on Automatic Programming. IEEE Transactions on Software Engineering Vol SE-11, No 11 November 1985.

/10/ Borgida A.: Greenspan S., Mylopoulos J.: Knowledge Representation as the Basis for Requirements Specifications. IEEE Computer, April 1985.

/11/ Minsky M.: A framework for representing knowledge, in P. Winston (Ed.), The Psychology of Computer Vision. McGraw-Hill, New York, 1975 pp 211 -277.

/12/ Barr A., & Feigenbaum E.: Handbook of Artificial Intelligence Volume 1. Addison-Wesley USA, 1981.

/13/ Aikens J.S.: Prototypical Knowledge for Expert Systems. Artificial Intelligence 20 (1983) 163-210.

Appendix 5 - Example Specification Produced by *Specriter 3*

This document was produced by

SPECRITER 3.A

from

The Measurement and Instrumentation Centre (MIC)

of

The City University, London

Written and Copyright by Stephen C Cook, 1990

Specification Details

Specification Name: TEMP

Generation Date: November 28, 1990 3:05 PM
Document Number: ERL-TR-23456
Issue Number: 1

REQUIREMENTS SPECIFICATION

for a

TYPE 601 SPACECRAFT THERMOMETER

1. Scope

This specification establishes the performance, design, development, and test requirements for a Type 601 Spacecraft Thermometer.

2. Applicable Documents

The documents listed hereunder form a part of this specification to the extent invoked by specific reference in other parts of this specification. If a specification is listed but not referenced in any specific paragraph, then the specification is applicable as a design guideline.

AD-185	Requirements Specification for Thermally Conducting Adhesives for Spacecraft Use.
AD-409	Requirements Specification for Low-Outgassing Potting Compound - Space Grade.
IE-ICD-MI-005	Interplanetary Explorer Thermal Sub-system Interface Control Document.
IE-ICD-EI-601	Interplanetary Explorer Thermal Sub-system Electrical Interface Control Document.
NHB 1007	Design and Construction Standards for Spaceflight and Ground Equipment associated with Scientific Spacecraft.
MIL-HDBK-217E	Reliability Prediction Data for Electronic Components.

3. Requirements

The measuring instrument described by all the requirements of this section shall pass the examinations, analysis and tests specified in Section 4.

3.1 Instrument Definition

3.1.1 General Description

The purpose of the instrument is to measure temperature in a Space/Avionics environment. The Type 601 Spacecraft Thermometer is housed in two enclosures; one for the sensor and the other for the remainder of the functions, nominally the data processor and display.

The Type 601 Spacecraft Thermometer forms part of Interplanetary Explorer II.

3.1.2 Interface Definition

The Type 601 Spacecraft Thermometer is to interface to the system being measured, power supplies, equipment housings and the operator.

3.1.2.1 Electrical Interface

3.1.2.1.1 Power Interface

The performance of the Type 601 Spacecraft Thermometer described herein shall be met when connected to 24 +/- 2V DC. Connection to the power source shall be achieved by one or more flexible power leads. Plugs conforming to Cannon Royal D, 7 pin shall be fitted to these leads. The peak current drawn by the instrument shall not exceed 50 mA.

3.1.2.1.2 Communications Interface

Processed and corrected measurement data shall be available to external equipment via an interface port conforming to RS-232-C.

3.1.2.1.3 EMC

The Type 601 Spacecraft Thermometer shall meet the emission requirements of IE-ICD-EI-601 and the susceptibility requirements of IE-ICD-EI-601.

3.1.2.2 Mechanical Interface

3.1.2.2.1 Data Processor and Display Unit Mechanical Interface

The Data Processor and Display Unit shall be attached via bolts on mounting flanges surrounding the mounting face.

3.1.2.2.2 Sensor Mechanical Interface

The Sensor shall be attached to its mounting by multiple bolts.

3.1.2.3 Thermal Interface

3.1.2.3.1 Data Processor and Display

The Data Processor and Display Unit shall be cooled via conduction to the mounting face.

3.1.2.3.2 Sensor Thermal Interface

The Sensor shall be cooled by conduction to the mounting face such that the temperature rise of the sensing element above ambient, when operating, shall be constrained to less than 0.1 degrees C.

3.2 Characteristics

3.2.1 Performance

The Type 601 Spacecraft Thermometer shall meet the performance requirements specified herein when operated in the environment specified in section 3.2.5.

3.2.1.1 Range

The instrument shall be able to measure temperature from 5 degrees C to 40 degrees C with at least the performance described by the remaining paragraphs of Section 3.2.1.

3.2.1.2 Total Measuring Error

The error of measurement including systematic error, random error, reading error, repeatability error, hysteresis error, and dynamic error shall be less than 0.7 degrees C.

3.2.1.3 Static Performance

3.2.1.3.1 Discrimination

The Type 601 Spacecraft Thermometer shall have a discrimination of 0.02 degrees C or better.

3.2.1.3.2 Repeatability

Any two measurements performed by the Type 601 Spacecraft Thermometer on any given identical value of temperature within the range specified in paragraph 3.2.1.1, shall indicate values within 0.5 degrees C.

3.2.1.3.3 Hysteresis

Hysteresis effects owing to alteration of the direction of the temperature change, including any dead band, shall be less than 0.04 degrees C.

3.2.1.3.4 Drift

Drift resulting from aging effects shall be less then 0.05 degrees C over 24 hours.

3.2.1.3.5 Special Requirements

The transfer function of the instrument shall be monotonic to better than 0.01 degrees C.

3.2.1.4 Dynamic Performance

The Type 601 Spacecraft Thermometer shall exhibit first order response. The time constant shall be 0.5 seconds.

3.2.1.5 Power Consumption

The power consumption of the Type 601 Spacecraft Thermometer shall be less than 1.00 W.

3.2.2 Physical Characteristics

3.2.2.1 Enclosures

3.2.2.1.1 Data Processing and Display Unit Enclosure

The Type 601 Spacecraft Thermometer Data Processor Unit housing shall observe the mounting face requirements and dimension limits specified in IE-ICD-MI-005.

3.2.2.1.2 Sensor Housing

The sensor shall be enclosed by potting compound conforming to AD-490. The mounting face and overall dimensions shall conform to IE-ICD-MI-005. Any necessary machining shall be performed to approved procedures.

3.2.2.2 Mass

3.2.2.2.1 Data Processing and Display Unit Mass

The mass of the Data Processor and Display shall be less than 1.2 kg.

3.2.2.2.2 Sensor Mass

The mass of the Sensor shall be less than 120 gms.

3.2.3 Reliability

The probability of the Type 601 Spacecraft Thermometer surviving for 10 years shall be greater than 0.998. The probability of survival shall be determined using MIL-STD-217E.

3.2.4 Maintenance

Once installed in its intended operating environment, the Type 601 Spacecraft Thermometer is not accessible. There is no maintainability requirement.

3.2.5 Environmental Conditions

The environment the instrument is to withstand is divided into operating and non-operating environments. The latter includes both transportation and storage. The Type 601 Spacecraft Thermometer shall meet the performance characteristics specified in section 3.2.1 when operated in any of the combination of environmental conditions defined in the operating environment. The Type 601 Spacecraft Thermometer shall survive the storage environment for 36 months without deterioration.

3.2.5.1 Operating Environment

3.2.5.1.1 Operating Temperature Range

The operating temperature range of the Type 601 Spacecraft Thermometer shall be from 5 degrees C to 50 degrees C.

3.2.5.1.2 Operating Humidity Range

The humidity range over which the Type 601 Spacecraft Thermometer is to operate shall be from 0% to 95%.

3.2.5.1.3 Operating Vibration Range

The performance of the Type 601 Spacecraft Thermometer shall be within the bounds specified in 3.2.1 when subjected to vibration over the range 5 Hz to 200 Hz of amplitudes up to 0.1 mm.

3.2.5.1.4 Operating Air Pressure Range

The instrument shall operate over an atmospheric pressure range of 0 kPa to 105 kPa.

3.2.5.1.5 Special Operating Environment Characteristics

The Type 601 Spacecraft Thermometer shall withstand an ionising radiation dose of 200 Greys without damage or loss of calibration.

3.2.5.2 Non-Operating Environment

The instrument is required to survive the non-operating environment and be able to meet its performance characteristics without re-calibration when environmental stress falls within the operating environment limits.

3.2.5.2.1 Non-Operating Temperature Range

The non-operating temperature range of the Type 601 Spacecraft Thermometer shall be from 5 degrees C to 50 degrees C.

3.2.5.2.2 Non-Operating Humidity Range

The non-operating humidity range of the Type 601 Spacecraft Thermometer shall be from 0% to 100%.

3.2.5.2.3 Non-Operating Vibration Range

The Type 601 Spacecraft Thermometer shall survive vibration over the range 10 Hz to 500 Hz of amplitudes up to 3mm from 10 Hz to 50 Hz decreasing at 6dB per octave until 500 Hz.

3.2.5.2.4 Non-Operating Air Pressure Range

The non-operating atmospheric pressure range of the Type 601 Spacecraft Thermometer shall be from 0 kPa to 105 kPa.

3.2.5.2.5 Special Non-Operating Environment Characteristics

The Type 601 Spacecraft Thermometer shall withstand the launch acoustic noise environment of 150 dBA from 200 Hz to 5000 kHz falling at 6dB per octave beyond those frequencies, without damage or loss of calibration.

3.3 Design and Construction

The materials and workmanship of the Type 601 Spacecraft Thermometer shall conform to the "Space" requirements of NHB 1007.

4 Quality Assurance

4.1 General

4.1.1 Responsibility for Tests

The vendor shall be responsible for performing the tests, analysis or inspections specified in Figure 4.1 the Verification Matrix. Any non-compliancies encountered during the examinations or tests shall cause rejection of the instrument.

4.1.2 Test Reports and Certificates

A test report shall be prepared following the testing of the instrument. A copy of this report shall be available on request. A calibration certificate shall be included with the instrument when despatched.

4.2 Quality Conformance Verification

The requirements for, and the methods used, to verify the design and performance requirements of section 3 will be satisfied, are tabulated in the Verification Matrix, Figure 4.1.

4.2.1 Test Sample

Each instrument delivered shall be inspected and acceptance tested.

4.2.2 Test Sequence

Acceptance inspection and tests shall consist of the examinations and tests in the following sequence.

(1) Physical Examination
(2) Functional Test
(3) Environmental Tests
(4) Post Environmental Functional Test

4.2.2.1 Physical Examination

Examination of the instrument shall be performed prior to functional testing.

4.2.2.2 Functional Testing

Functional tests shall be performed prior to, during and where appropriate, following environmental testing.

4.3 Verification Methods

4.3.1 Inspection

Inspection shall consist of physical examination of the product, engineering drawings and/or other documentation to determine conformance to the requirements. The Type 601 Spacecraft Thermometer shall be inspected to determine conformance to the physical characteristics specified in paragraph 3.2.2 and the applicable drawings and specifications.

4.3.2 Analysis

The analysis method shall consist of review of analytical data resulting from analyses performed by generally recognised techniques for requirements that cannot readily be demonstrated through conventional testing techniques. Computer simulation, is the preferred method, where appropriate.

4.3.3 Testing

4.3.3.1 Functional Tests

Verification that the Type 601 Spacecraft Thermometer performs as specified herein shall be achieved by performing functional tests specified in figure 4.1.

4.3.3.2 Environmental Tests

Testing of performance shall be performed at both extremes of the range of each environmental characteristic. It shall not be necessary to combine tests of more than of environmental characteristic in a single test, i.e. high temperature and low pressure. Instrument performance in such situation shall be verified be analysis as indicated in figure 4.1.

4.4 Test Procedures

All tests shall be performed in accordance with approved, released procedures.

4.5 Rejection and Retest

If a failure occurs during test, testing shall be discontinued until an analysis is performed to determine whether the condition warrants continuation of the tests or discontinuation of the tests for more detailed failure analysis. The test procedure shall be repeated until completed successfully. If corrective action substantially affects the significance of results of previously completed tests, such tests shall be repeated also.

5. Preparation for Delivery

The instrument shall be packaged in such a way that its performance shall not be impaired by storage for periods up to the storage time specified in Section 3.2.5. The packing shall be designed to support the instrument during transportation and storage and provide absorbent protection from mechanical shock. The packing shall not contaminate the instrument during storage and shall keep it free from scratches, dust and chemical attack. All containers or parcels which comprise the instrument shall be marked with a minimum of the manufacture's name and the product name. In addition, where appropriate, labels detailing special handling requirements shall be affixed, for example, "Fragile Device", "Do not expose to X-rays", "This Way Up", "Anchor during Transport", etc.

6. Notes

The Type 601 Spacecraft Thermometer is intended to be used for a number of forthcoming missions in the next ten to fifteen years.

References

Adole (1985). *Postscript Reference Manual*, Addison-Wesley, U.S.A.

AEL (1987). *System Functional Requirements Specification for a Laser Airborne Depth Sounder for use by the RAN in Hydrographic Surveying Applications*, Advanced Engineering Laboratory, Adelaide.

Aikens J.S. (1983). "Prototypical Knowledge for Expert Systems." *Artificial Intelligence*, Vol. 20, p163-210.

Amstadler B.L. (1971). *Reliability Mathematics*, McGraw-Hill, New York.

AS 1514, Part1 (1980). *Glossary of terms Used in Metrology, Part 1 - General Terms and Definitions*, Standards Association of Australia, Sydney.

AS 1821 (1985). *Supplier's Quality Systems for Design, Development, Production and Installation*, Standards Association of Australia, Sydney.

Auspace (1984). *Starlab Instrument Package - Electronics Subsystem Specification*, Auspace, Canberra.

Barr A. & Feigenbaum E. (1981). *Handbook of Artificial Intelligence Volume 1*, Addison-Wesley, USA.

Bartlett F.C. (1932). *Remembering*, Cambridge University Press, Cambridge.

Balzer R. (1985). "A 15 Year Perspective on Automatic Programming." *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 11, p1257-1268.

Berg H.K., Boebert W.E., Franta W.R. & Moher T.G. (1982). *Formal Methods of Program Verification and Specification*, Prentice-Hall, New Jersey.

Blackburn M.R. (1989). "Using Expert Systems to Construct Formal Specifications." *IEEE Expert*, Spring, p62-74.

Blaise (1986). *RUNOFF - The programmers Text Formatter, User Reference Manual*, Blaise Computing Inc, Berkeley.

Bobrow D.G. (1985). "If Prolog is the Answer, What is the Question? or What it takes to Support AI Programming Paradigms." *IEEE Transactions in Software Engineering*, Vol. SE-11, No. 11.

Borgida A., Greenspan S. & Mylopoulos J. (1985). "Knowledge Representation as the Basis for Requirements Specifications." *IEEE Computer*, April, p82-91.

Borland (1986). *Turbo Prolog Owner's Handbook*, Borland International Inc., U.S.A.

Borland (1987). *Turbo Prolog Toolbox User's Guide and Reference Manual*, Borland International Inc., U.S.A.

Borland (1988a). *Turbo Prolog User's Guide - Version 2.0*, Borland International Inc, U.S.A.

Borland (1988b). *Turbo Prolog Reference Guide - Version 2.0*, Borland International Inc, U.S.A.

Borgida A. (1985). "Knowledge Representation as the Basis for Requirements Specifications." *IEEE Computer*, p82-91.

Bosman D. (1978). "Systematic design of instrumentation systems", *J. Phys. E.: Scientific Instruments*, Vol. 11, p97-105.

Brachman R.J. (1983). "What IS-A is and isn't: an analysis of taxonomic links in semantic networks." *IEEE Computer*, Vol. 16, No. 10, p30-36.

Brachman R.J. & Levesque H.J. (Ed.) (1985). *Readings in Knowledge Representation,* Morgan Kaufman Publishers Inc, Los Altos.

Bratko I. (1986). *Prolog Programming for Artificial Intelligence*, Addison Wesley, U.K.

BS 4778 Pt1&2 (1987). *Quality Vocabulary*, British Standards Institution

BS 5750 (1979). *Quality Systems, Pt1*, British Standards Institution.

Burgess J.A. (1969). "Organizing Design Problems." *Machine Design*, November 27, p120-127.

Buchanan B.G. & Shortliffe E.H. (1984). *Rule-Based Expert Systems: THe MYCIN Experiments of the Stanford Heuristic Programming Project*, Addison-Wesley, London.

Clocksin W.F. & Mellish C.S. (1987). *Programming in Prolog,* 3rd Edition, Springer-Verlag, Berlin.

Cohen B., Harwood W.T. & Jackson M.I. (1986). *The Specification of Complex Systems*, Addison-Wesley, U.K.

Cook S.C. (1983). *Design Aim for Dynamic Access Memory PCA, Part Of Avionics Fault Tree Analyzer (AFTA)*, Doc: 20033, British Aerospace Australia, Adelaide.

Cook S.C. (1986). *Document Preparation Guidelines*, Vision Systems Limited, Adelaide.

Cook S.C., Thomas G. & Leslie B.A. (1988). *Quality Manual*, Vision Systems Limited, Adelaide.

Cook S.C. (1988a). "Automatic generation of measuring instruments." *Measurement*, Vol. 6 No. 4, p155-160.

Cook S.C. (1988b). "Specriter Instrument Attribute File, Technical Description." Unpublished research note.

Cook S.C. (1990). "Knowledge-Based Generation of Measuring Instrument Specifications." In *IMEKO International Symposium on Knowledge-Based Measurement*, 1st, Karlsruhe, FDR, Sept.

Cuadrado J.L. & Cuadrado C.Y. (1986). "AI in Computer Vision." Byte, January 1986.

Delisle N. & Garlan D. (1990). "Applying Formal Specifications to Industrial Problems: A Specification of an Oscilloscope." *IEEE Software*, September.

Doebelin E.O. (1983). *Measurement Systems*, 3rd Edition, McGraw-Hill, Singapore.

Donker J. C. & Kat P.J. (1987). *NEXT: An Expert System Development Tool to Support Engineering Design, NLR MP 86070 U*, National Aerospace Laboratory NLR, The Netherlands.

Draper C.S., McKay W. & Lees S. (1952). *Instrument Engineering*, McGraw Hill, New York.

Dunham C.W., Young R.D. & Bockrath J.T. (1979). *Contracts, Specifications and Law for Engineers*, McGraw Hill, New York.

DSMC (1983). *System Engineering Management Guide*, Defense Systems Management College, Fort Belvoir, Virginia, U.S.A.

Edmonds E.A. & Guest S. (1979). *A Man-Computer Dialogue System*, Leicester Polytechnic.

ESA (1983). *Hipparcos System Specification*, European Space Agency.

Forgy C.L. (1982). "A Fast Algorithm for the Many Pattern/Many Object Match Problem." *Artificial Intelligence*, Vol. 19, No. 1, p17-37.

Finkelstein A.C.W. & Finkelstein L. (1985). "A review of instrument system design automation." In *Acta IMEKO X, New Measurement Technology to Serve Mankind*, Ed. Striker G., OIKK, Budapest, p79-85.

Finkelstein L. & Finkelstein A.C.W. (1983). Review of design methodology. *IEE Proceedings*, Vol. 130, Pt. A. No. 4, p213-222.

Finkelstein L. & Leaning M.S. (1984). "A review of the fundamental concepts of measurement." *Measurement*, Vol. 2, No. 1, p25-34.

Finkelstein A., Maibaum T. & Finkelstein L (1990). "Engineering-In-The-Large: Software Engineering and Instrumentation", *Proceedings of UK IT 1990.*

Galitz W.O. (1985). *Handbook of screen format design*, QED Information Sciences, Wellesley.

Gardner P. (1985). *The Use of Microcomputers for the Mathematical Modelling of Instruments*, Ph.D. Thesis, City University.

Goldman A.S. & Slattery T.B. (1964). *Maintainability: A Major Element of System Effectiveness*, John Wiley & Sons, New York.

Goldsmith K. (1989). Private communication.

Guttag J.V. & Horning J.J. (1986). "Formal Specifications as a Design Tool." In *Software Specification Techniques*, Ed. Gehani N. & McGettrick A.D., Addison-Wesley.

Hayes F. (1979) "The Logic of Frames", in *Frame Conceptions and Text Understanding*, ed. Metzing D., Walter de Gruyter & Co, Berlin.

Hayes-Roth F., Waterman D. & Lenat D. (Ed.) (1983). *Building Expert Systems*, Addison-Wesley.

Hill P.H. (1970). *The Science of Engineering Design*, Holt, Rinehart and Winston Inc., New York.

IEC 914. *Specification for Rack Mounted Equipment.* International Electrotechnical Commission.

ISO 8402 (1986). *Quality - Vocabulary*, International Standards Organisation.

ISO 9001. *Model for quality systems for design/development production, installation and servicing*, International Standards Organisation.

Keller R. (1987). *Expert System Technology*, Prentice-Hall Inc., New Jersey.

Kowalski R. (1979). *Logic for Problem Solving*, North Holland, Oxford.

Lees A. (1987). The Development of Non Critical Systems Using Abstract Specifications, *Journal of Electrical and Electronic Engineering Australia*, Vol. 7, No. 1.

Leech D.J. (1972). *Management of Engineering Design*, John Wiley and Sons, London.

Liskov B.H. & Berzins V. (1986). "An Appraisal of Program Specifications." In *Software Specification Techniques*, Ed. Gehani N. & McGettrick A.D., Addison-Wesley.

Lister A.M. (1984). *Fundamentals of Operating Systems*, Macmillian Education, London.

Lucas T.A. (1981). "Standards for Instrumentation." *Measurement & Control*, Vol. 14, June.

Kowalski R. (1979). *Logic for Problem Solving*, North-Holland.

Marcellus D.H. (1989). *Expert System Programming in Turbo Prolog*, Prentice Hall, New Jersey.

McDermott J. (1982). "R1: A Rule-Based Configurer of Computer Systems." *Artificial Intelligence*, Vol. 19, No. 1, p39-88.

Mead D.W., Mead H.W., & Ackerman J.R. (1956). *Contracts, Specifications and Engineering Relations*, McGraw Hill, New York.

Michie D. (1982). *Machine Intelligence Related Topics*, Gordon & Breach Science Publishers, London.

Metric Conversion Board, Australia (1972). *Metric Handbook: SAAMH1-1972*, Standards Association of Australia.

MIL-S-83490 (1968). *Military Specification - Specifications, Types and Forms*, United States of America Department of Defense.

MIL-STD-461. *Military Standard - Electromagnetic Compatibility*, United States of America Department of Defense.

MIL-STD-490 (1968). *Military Standard - Specification Practices*, United States of America Department of Defense.

MIL-STD-490A (1985). *Military Standard - Specification Practices*, United States of America Department of Defense.

MIL-STD-499A (1974). *Military Standard - Engineering Management*, United States of America Department of Defense.

MIL-STD-721B. *Definitions of Effectiveness Terms for Reliability, Maintainability, Human Factors and Safety*, United States of America Department of Defense.

MIL-STD-810E (1989). *Military Standard - Environmental Test Methods and Engineering Guidelines*, United States of America Department of Defense.

MIL-Q-9858A *Quality Program Requirements*, United States of America Department of Defense.

Minsky M. (1975). "A framework for representing knowledge." *The Psychology of Computer Vision*, Ed. Winston P.H., McGraw-Hill, New York, p211-277.

Mirza M.K., Neves F.J.R. & Finkelstein L. (1990). "A knowledge-based system for design-concept generation of instruments." *Measurement*, Vol. 8, No. 1, p7-11.

Moore R.C. (1985). "The Role of Logic in Knowledge Representation and Commonsense Reasoning." In *Readings in Knowledge Representation*, Ed. Brachman R.J. & Levesque H.J., Morgan Kaufman, Los Altos.

M'Pherson P.K. (1980). "Systems Engineering: an approach to whole system design." *The Radio and Electronic Engineer*, Vol. 50, No. 11/12, p545-558.

M'Pherson P.K. (1981). "A framework for systems engineering design." *The Radio and Electronic Engineer*, Vol. 51, No. 2, p59-85.

Nii H.P. (1986). *Blackboard Systems*, Knowledge System Laboratory Report No KSL 86-18, Stanford University. (Also appeared in AI Magazine, Volumes 7-2,7-3.)

Nussbaumer H. (1990). *Computer Communication Systems Vol 1.*, John Wiley & Sons, Chichester.

O'Conner P.D.T. (1985). *Practical Reliability Engineering*, John Wiley & Sons, Chichester.

O'Neill J.L. (1987). "Plausible Reasoning." *The Australian Computer Journal*, Vol. 19, No. 1.

Oxford English Dictionary (1989). *The Oxford English Dictionary*, Ed. Simpson J.A. & Weiner E.S.C., Oxford University Press, Oxford.

Pavelin C. (1987). "Logic in Knowledge Representation." In *Approaches to Knowledge Representation*, Ed. Ringland G.A. & Duce D.A., Research Studies Press, Letchworth.

Proakis J.G. & Manolakis D.G. (1988). *Introduction to Digital Signal Processing*, Macmillan Publishing Company, New York.

Quillian M.R. (1968). "Semantic Memory." In *Semantic Information Processing*, Ed. M. Minsky, MIT Press, Cambridge, MA.

Ramsay A. (1988). *Formal Methods in AI.* Cambridge University Press, Cambridge.

Randal D.M. (1988). "Semantic Networks." In *Approaches to Knowledge Representation*, Ed. Ringland G.A. & Duce D.A., Research Studies Press, Letchworth.

Ringland G. (1988). "Structured Object Representation - Schemata and Frames." In *Approaches to Knowledge Representation*, Ed. Ringland G.A. & Duce D.A., Research Studies Press, Letchworth.

Ringland G.A. & Duce D.A. (Ed.) (1988). *Approaches to Knowledge Representation*, Research Studies Press Ltd, Letchworth.

Roseman M.A., Coyne R.D. & Gero J.S. (1987). "Expert Systems for Design Applications." In *Proceedings of the 2nd Aust. Conf. on Applications of Expert Systems*, Sydney, May.

Rosin P. (1988). *Model Driven Image Understanding: A Frame-Based Approach*, Ph.D. Thesis, City University 1988.

SAA (1985). *QA1 - Guidelines for the Preparation of Quality Manuals*, Standards Association of Australia.

Shammas N.C. (1986). "Turbo Prolog", *Byte*, September

Schildt H. (1987). *Advanced Turbo Prolog: Version 1.1*, Osborne/McGraw Hill, U.S.A.

Simons G.L. (1985). *Expert Systems and Micros*, NCC Publishers, Manchester.

Smith D.J. (1985). *Reliability and Maintainability in Perspective*, 2nd edition, Macmillian.

Smith P. (1988). *Expert Systems Development in Prolog and Turbo Prolog*, John Wiley & Sons, Chichester.

Stebbing L. (1987). *Quality Assurance - The route to efficiency and competitiveness*, 2nd Edition. John Wiley & Sons, Chichester.

Sterling L. & Shapiro E. (1986). *The Art of Prolog: Advanced Programming Techniques*, The MIT PRess, Cambridge, Ma.

Subrahmanyam P.A. (1985). "The Software Engineering of Expert Systems: Is Prolog Appropriate?" *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 11.

Sufrin B. (1986). "Formal Specification of a Display-Oriented Text Editor" In *Software Specification Techniques*, Ed. Gehani N. & McGettrick A.D., Addison-Wesley.

Sunshine C.A., Thompson D.H., Erickson R.W., Gerhart S.L. & Schwartz D. (1982). "Specification and Verification of Communications Protocols in AFFIRM Using State Transition Models." *IEEE Transactions in Software Engineering*, Vol. SE-8, No. 5, p460-489.

Sydenham P.H. (1982). "Standardisation of measurement fundamentals and practices" In *Handbook of Measurement Science Volume 1*. Ed. Sydenham P.H., John Wiley and Sons, Chichester.

Sydenham P.H. (1983a). "Static and steady-state considerations." In *Handbook of Measurement Science Volume 2*, Ed. Sydenham P.H., John Wiley and Sons, Chichester.

Sydenham P.H. (1983b). "Measurement system dynamics." In *Handbook of Measurement Science Volume 2*, Ed. Sydenham P.H., John Wiley and Sons, Chichester.

Sydenham P.H. (1984). "CAD-BASED Instrument Design." In *Conference on Measurement, Instrumentation and Digital Technology*, The Institution of Engineers Australia, Melbourne.

Sydenham P.H. (1985a). "Structured understanding of the measurement process - Part 1, Holistic view of the measurement system." *Measurement*, Vol. 3, No. 3 p115-120.

Sydenham P.H. (1985b). "Structured understanding of the measurement process - Part 2, Development and implementation of a measurement process algorithm." *Measurement*, Vol. 3, No. 4, p161-168.

Sydenham P.H. (1986). *Mechanical Design of Instruments.* ISA, USA.

Sydenham P.H. (1987). "Computer-aided engineering of measuring instrument systems." *Computer-Aided Engineering Journal*, Vol. 4, No. 3, p117-123.

Sydenham P.H. (1989). Private communication.

Sydenham P.H., Harris D.D. & Hancock N.H. (1990). "MINDS - A software tool to establish a measuring system requirement." To be published in *Measurement.*

Sydenham P.H. & Harris D.D. (1990). "Knowledge-Based Measurement Systems Design and Operation." In *IMEKO International Symposium on Knowledge-Based Measurement*, 1st, Karlsruhe, FDR, Sept.

Sydenham P.H. & Vaughan M.M. (1990). "Sensor design using rules of knowledge required", *Measurement*, Vol. 8, No. 4, p180-187.

Voelcker J. (1988). "Flex In Specs: A License to Innovate?" *IEEE Spectrum*, November, 1988.

Walter J. & Nielson N.R. (1988). Crafting Knowledge-Based Systems. John Wiley & Sons, New York.

Wheeldon R.W. (1974). "Specifications - the identifying facts." In *Introduction to Stathmology*, Ed. Sydenham P.H., Dept. of Continuing Education, Univ. of New England, N.S.W., Australia.

Weiskamp K. & Hengl T. (1988). *Artificial Intelligence Programming in Turbo Prolog.* John Wiley & Sons Inc, New York.

Wilbur-Ham M.C. (1987). "PROTEAN: A Tool for Verifying Protocol Specifications." *Proceeding of IREECON International 1987*, Sydney, p579-581.

Williams T. & Bainbridge B. (1988). "Rule Based Systems." In *Approaches to Knowledge Representation*, Ed. Ringland G.A. & Duce D.A., Research Studies Press, Letchworth.

Winston P.H. (1984). *Artificial Intelligence*, 2nd Edition. Addison Wesley.

Woods W.A. (1975). "What's in a link: foundations for semantic networks." In *Studies in Cognitive Science*, Ed. D.G. Bobrow and A.M. Collins, Academic Press, New York.

Yin K.M. & Soloman D. (1987). *Using Turbo Prolog*, Que Corporation.

Acronyms & Abbreviations

AI	Artificial Intelligence
CAD	Computer-Aided Design
CAE	Computer-Aided Engineering
CAEINST	Computer-Aided Engineering of INSTruments
FOL	First Order Logic
KBS	Knowledge-Based System
ISO	International Standards Organisation
MIC	Measurement and Instrumentation Centre, City University
MINDS	Measurement INterface Design System
MISC	Measurement and Instrumentation Systems Centre, SAIT
MTBF	Mean Time Between Failure
MTTR	Mean Time To Repair
OSI	Open Systems Interconnect (networking model)
PC	Personal Computer
SAIT	South Australian Institute of Technology
TBD	To Be Determined
TBS	To be Specified
VAX	Series of super-mini computers made by DEC
VMS	Virtual Memory System (operating system for VAX computers)
VSD	Virtual Screen Definition