A SPECIFICATION LANGUAGE

FOR DIGITAL SYSTEMS

by

PETER BLACKLEDGE

A thesis submitted in fulfillment of the requirements

for the degree of Doctor of Philosophy.

INTERDISCIPLINARY HIGHER DEGREES SCHEME

THE UNIVERSITY OF ASTON IN BIRMINGHAM

SEPTEMBER, 1982

The University of Aston in Birmingham

SUMMARY

A SPECIFICATION LANGUAGE FOR DIGITAL SYSTEMS

Peter Blackledge

Submitted for the Degree of Ph.D.
1982

The work reported in this thesis is concerned with the selection of a formal language for practical use in industry for writing specifications of systems containing both hardware and software. The aims of using such a language are to improve the communication of requirements and to increase the number of errors detected at an early stage of the design process. Due to the size of the audience of writers and readers of these specifications, one additional aim is to minimise the amount of training which will be required by these people. Apart from its formality, the language must therefore be able to describe large and complex systems in a comprehensible manner.

Criteria for the evaluation of candidate languages are derived from these needs and then used in a review of a large number of languages from published sources. All those reviewed were found to be deficient in some respect, so a new language was designed to fulfill the criteria. This language was named ASL, being an acronym for "A Specification Language"; it is suitable for use in specifying all information-processing systems where the received and transmitted information can be treated as discrete (i.e. digital) signals.

In order to confirm the suitability of ASL, a number of practical trials of the language were carried out. Although these were of limited size, they did cover both hardware and software systems and personnel. The results of these trials, including suggestions from the participants for improvements to ASL, are discussed as part of the evaluation of the success of the project.

KEYWORDS: Specification Requirements

ACKNOWLEDGEMENTS

The author gratefully acknowledges the financial support of GEC Telecommunications Ltd. and the SERC, the moral support of his supervisory team (Prof. J.E.Flood, Mr. R.K.J.Ford, Dr. N.W.Horne and Mr. G.A.Montgomerie), and the practical support of the participants in the trials (Mr. P.J.Briggs, Mr. R.Caberwal, Mr. G.V.Geiger, Mr. P.W.Gray, Mr. D.Simblet, Mr. H.B.Taylor, Mr. D.R.Thompson and Dr. J.C.Woodcock).

LIST OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1. The Problem

The project reported here is concerned with the introduction of disciplined methods into the design process, and particularly with the use of formal languages for system specification. However, in order to place the work in context, this chapter starts with a discussion of the underlying problems to be solved. This then leads to consideration of how the results of the project contribute to the required solution. Due to the number of words which are used with a particular technical meaning, a glossary of terms is included in Appendix G.

This project relates to the development of digital systems, and the word "system" is used throughout to refer to the intended output of some design project. Such a system is expected to be a purposeful information-processor which enters into some communication with its environment to fulfill that purpose (Ashby, 1969). However, the system may be designed in the form of words (e.g. instruction manuals or software), physical assem-

blies (hardware) or integrated cicuits (hardware and/or firmware) and so the word "system" has been used in order to avoid implying any particular physical embodiment of the information-processing entity. Advances in technology, especially in the field of computing, have led to a rapid growth in the complexity of systems. Public awareness of these advances and of the decreasing cost of computers provides a continual pressure to extend the capabilities of existing products. In the telecommunications industry this takes the form of new services (e.g. Prestel) and new facilities (e.g. subscriber-controlled redirection of calls to other numbers); but, as in other industries, these additions have proved difficult and expensive to develop despite the theoretical capabilities of the underlying technology.

Three main factors have been proposed to account for this difficulty :

(a) as the complexity of a system increases, the documentation describing the required behaviour is not increased in proportion (Jones, 1979) with the consequence that it is incomplete and the resulting systems often fail to meet their objectives,

(b) the scale of the projects concerned requires the involvement of large groups of people, so that organisational and communication difficulties often hamper progress (Brooks, 1975),

(c) when the system involves significant amounts of software or custom VLSI, there are currently no recognised methods for producing prototypes of the design.

Hence, design or specification errors are often not detected until late in the project timescale; thus their correction is likely to result in a failure to meet completion dates (Jones(b), 1980; Losleben, 1980).

Figures have been published showing the magnitude of the consequent wastage of resources (Alberts, 1976; Lehman, 1979). These problems become even more important when the systems being developed will take some responsibility for human safety or privacy and the cost of error may not be solely financial.

1.2. Analysis of the Problem

1.2.1. The Common Factor

The factors (a), (b) and (c) in Section 1.1 above have a common basis, in that all derive from communication problems :

factor (a) relates to the difficulty of achieving concise and precise descriptions in English or any other method based upon a natural language,

factor (b) is the result of communication difficulties between groups of people, especially if the groupings are based upon differing technical specialities,

factor (c) is a consequence of the lack of accepted intermediate forms of documentation to bridge the large gap between a specification written in English and the final design written in a programming language or

in logic diagram form.

The most commonly proposed type of solution to the problem is therefore based upon improved methods of communication (e.g. Ross, 1977). The validity of such a solution can best be demonstrated by considering first the type of design process currently in use, and then a new form which attempts to ensure improved communication between those groups of people involved in the project.

1.2.2. The Current Situation

Figure 1.1 depicts a simplified version of the design process which is typical of practice in British industry. The stages of the process are:

(a) specification, where the customer and supplier devise an agreed statement of the behaviour required of the system,

(b) design, where the supplier decides upon the logical and physical structure to be used to construct the system,

(c) the physical construction of the system,

(d) testing, where the completed system is subjected to a selected set of stimuli in an attempt to detect any undesirable behaviour.

The terms "customer" and "supplier" are used to indicate the roles of the respective parties involved. However, in many cases, both may be part of the same organisation and there is unlikely to be an explicit legal contract raised to cover the development of the system.

FIGURE 1.1 CURRENT DESIGN PROCESS

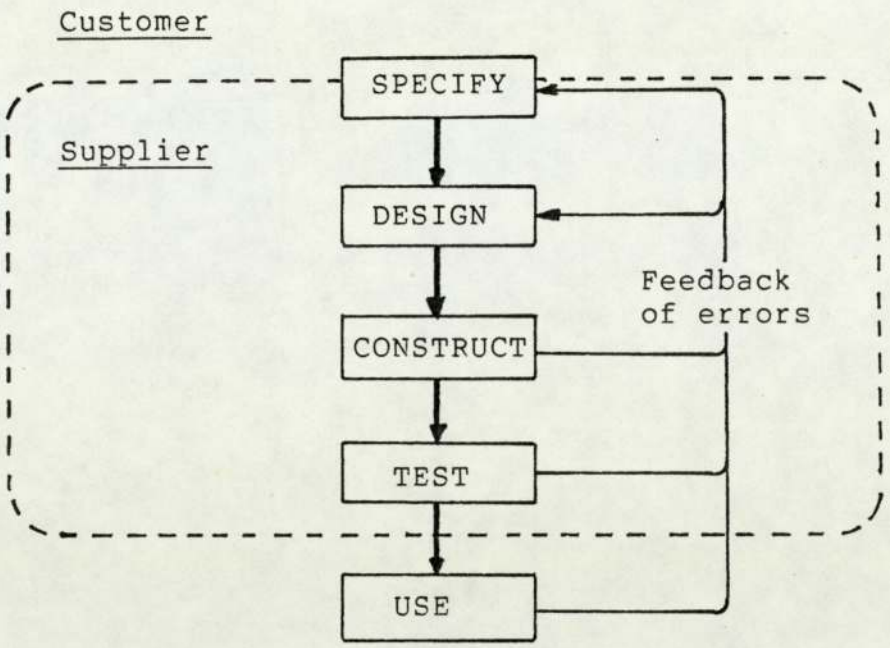
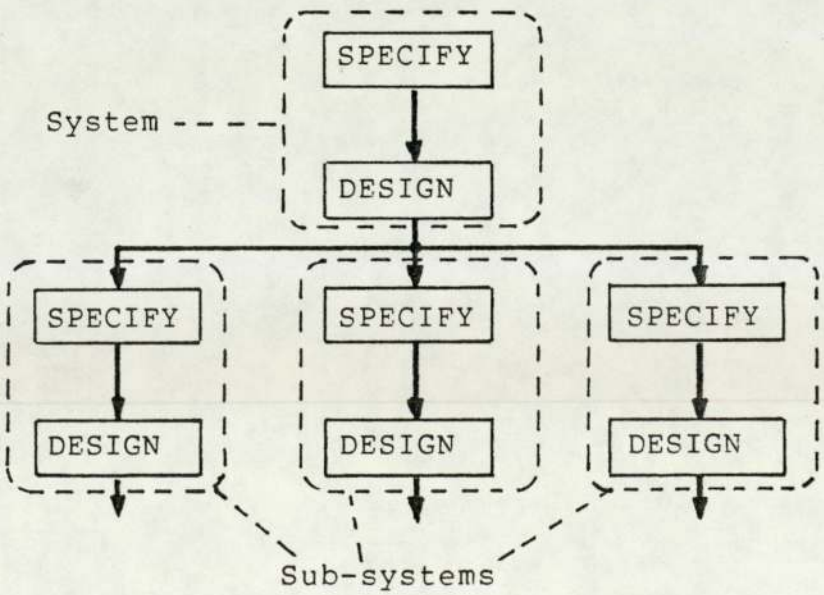


FIGURE 1.2 THE DESIGN HIERARCHY



The process depicted in Figure 1.1 can be repeated a number of times within one project as the complete system to be developed is divided into smaller and smaller sub-systems until a level is reached where each sub-unit of the system represents an acceptable unit of work for a small group of people. This is the approach of "top-down" or "structured" design (Yourdon & Constantine, 1979); it results in an hierarchically-structured description of the design as indicated in Figure 1.2.

In most engineering disciplines one early result of the design activity is a prototype or scale model of the proposed design, but this has not been common practice in the design of information-processing systems. As depicted in Figure 1.1, suppliers have tended to work from a specification which was (presumably) accepted by the customer, but with no checks upon the correctness of the interpretation of this document or upon the adequacy of the design until the testing stage. As testing only occurs after the construction of the system, the response time of the design process when viewed as a feedback system is very long in relation to the overall timescale of any design project. Hence, the correction of deficiencies detected during testing can require a large proportion of the design and construction activities to be redone, with the consequence of prolonged delays before the corrected system becomes available (Alberts, 1976). Lehman (Lehman, 1979) and Brooks (Brooks, 1975) have both noted the effects of this in large software systems, and Lehman (op. cit.) provides some estimate of the waste of resources

which results from such errors.

1.2.3. Proposed New Design Process

In recent years there have been numerous proposals for new design methods (or "methodologies" as they are often called in papers by American authors). Initially, these were mainly related to software (e.g. Baker, 1972; Naur & Randell, 1969) which was seen to be lagging behind the engineering disciplines in the use of agreed methods and notations. More recently, however, there has been growing interest in the use of such methods in computer hardware design, as the use of LSI and VLSI techniques has indicated that existing methods may no longer be adequate (e.g. Losleben, 1980). Recognition of the scale of the problems has now led to the creation of a number of national programmes backed by the governments of various countries, in an attempt to hasten the development and introduction of new methods (e.g. DoI(a), 1981; Redwine et al, 1981).

The basis of most of these proposals is a modified design process of the type shown in Figures 1.3 and 1.4. The principal aim of the modifications is to make the process more responsive and better-controlled by introducing rigorous checking between each pair of stages, so that the length of the feedback loop is only one stage rather than up to three (as shown in Figure 1.1). In the design hierarchy (see Figure 1.4) this means that at each level the equivalence of the design and the specification

FIGURE 1.3 NEW IDEAS ON THE DESIGN PROCESS

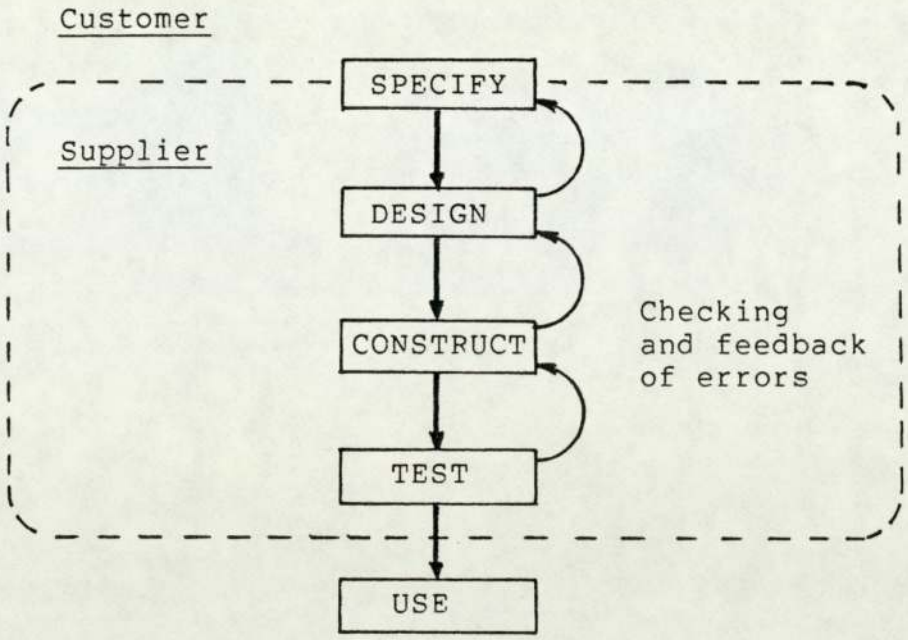
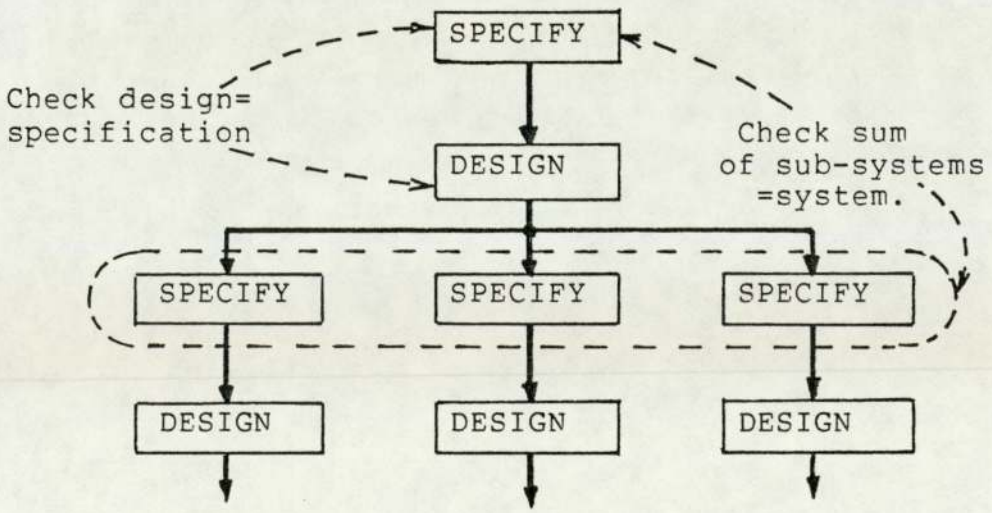


FIGURE 1.4 NEW IDEAS ON THE DESIGN HIERARCHY



is checked, whilst between levels it is ensured that the conjunction of the specifications of the sub-systems is equivalent to the specification at the level above. This approach should result in errors being detected and corrected at the earliest possible stage. Alford (Alford, 1979) provides figures showing that detecting an error one stage earlier can reduce the cost of correcting that error by an order of magnitude.

Amongst the implications of this new type of design process are the following.

(a) A larger proportion of the timescale for a project is to be spent in the more abstract stages of the design process (i.e. in specification and design activities) as these stages will involve more documentation and checking than is currently undertaken (Aron, 1976). As relatively few people are involved in these early stages (Alberts, 1976), this does not represent a significant increase in manpower costs.

(b) The proportion of the total timescale spent on the abstract parts of the design process is increased. However, the total timescale should be shorter, as the emphasis on checking should lead to a reduction in the overall time required to obtain a correct product.

(c) In order to achieve reliable checking throughout the design process, it is necessary to introduce standard methods of presenting information. Otherwise, different designers working on different levels of the system (see Figure 1.4) may produce incompatible

documents, making the checking activity impracticable (Lehman, 1981; Ramamoorthy & So, 1978). The use of such standard methods of presentation has been described as the introduction of engineering discipline and professionalism into areas which currently rely upon individual creativity (ASTG, 1981). Standardisation appears to be an essential feature in tackling large problems, where more than 5 or 6 people are involved in the project (Weinberg, 1971).

1.2.4. Long-term Prospects

The adoption of a new more-disciplined design method not only has the immediate benefit of reducing the total time required to develop new products, but it could provide additional benefits in the long term.

(a) The use of rigorous notations at all stages of the design process plus strict checking between stages will allow the amount of testing of the final product to be reduced (Mills, 1975). Any finite amount of testing can never demonstrate the total absence of errors in a complex system (Dijkstra, 1972), so it is much better to expend effort on reducing the number of errors likely to be present.

(b) Certain design stages may be delegated to computer-based design facilities (i.e. CAD), given that the input to these stages is expressed in a formal notation. Such systems are under development for automatic programming (Wood, 1980) and automatic layout

of VLSI (Lauther, 1979), but they are not yet ready for use in a commercial environment.

(c) The notations used for writing specifications may form a suitable input to a simulation system, allowing the system's behaviour to be demonstrated to the customer at each stage of development, thereby further reducing the risk of error (Cohen, 1981; Lehman, 1981).

(d) The acceptance tests for the product can be derived directly from such a specification (Alford, 1977).

All such benefits are dependent upon the full implementation of the new type of design process and the associated disciplines. This must therefore be seen as the prime task, to be undertaken before any of the benefits are obtained, but with some of the effects being apparent only in the long term.

1.3. The Purpose of the Project

1.3.1. Background

The project reported here was undertaken within the Telephone Switching Group of GEC Telecommunications Ltd., and therefore reflects some bias towards the particular problems of the British telecommunications industry. These are not however unique to that industry (see e.g. DoI(a), 1980), and the results reported here are of wider applicability. Discussion of problems specific to that industry are therefore kept to a minimum in this and sub-

sequent chapters, with the exception of Chapter 6 which covers work undertaken within the Company.

Although over the past few years the British telecommunications industry has introduced a significant number of standards relating to the documentation of product designs (e.g. System X, 1981), this has not been sufficient to gain the benefits mentioned in Section 1.3.3 above. This is largely because the documentation standards still rely upon the unregulated use of English to communicate meaning, which has proved to be unsatisfactory for the very large specification and design documents concerned (e.g. the specification for a large System X telephone exchange covers approximately 300 pages of A4-sized paper (POR 3231, 1976)).

With the continued increase in complexity of telephone systems, it was recognised that new methods would be necessary to avoid corresponding increases in the number of problems caused by poor communication. This project is one of a number of efforts which the Company is making in this direction.

1.3.2. Initial Scope of the Project

All the previous developments in methods within the Company had centred around the design, construction and testing stages, so the present project was intended to take a different view. The (chronologically) first step in the design process, that of specification, was selected as the starting point for the project, in ac-

cordance with the ideals of "top-down" design (e.g. Ross, 1977).

The terms of reference for the project were set as follows:

(a) to investigate methods of specification and languages used for writing specification documents (It should be noted that "languages" was taken to include any form of notation used in writing specifications, whether based on text or diagrams.),

(b) to propose which, if any, of these languages were adequate and suitable for use by the existing staff of the Company,

(c) if no existing language was found to be adequate, to design a new and more-suitable language,

(d) to introduce the chosen language and any essential support facilities into the Company.

Stages (a) to (c) were undertaken as a project under the Interdisciplinary Higher Degrees scheme at the University of Aston in Birmingham and are the subject of this thesis.

1.3.3. Final Scope of the Project

An investigation into the content of typical specification documents within the telecommunication industry showed that they contained:

(a) descriptions of the desired behaviour of the product, including such things as response times and maximum capacities,

(b) constraints upon the physical construction of the product, including power consumption, heat dissipation, weight and size,

(c) the required behaviour under conditions of overload or faults,

(d) relevant standards which must be met, such as documentation rules and health and safety standards.

This represents a mixture of information relating to different levels and stages in the design process, but the structure of the documents did not identify which part of the information was appropriate to each individual stage. It was decided that the "top-down" viewpoint which the project was intended to take would best be served by concentrating upon that information which is relevant to a "black box" specification (Ashby, 1969). Hence, all the information which forms constraints upon the design (such as constructional standards and power dissipation) would be considered to be outside the range of the investigation, and of any specification language. This decision appears to have been taken by almost all authors of articles on specification language (e.g. Abrial, 1980; Alford, 1977; Balzer & Goldman, 1979; CCITT, 1980; Goguen, 1979; Hemdal, 1973), although few make an explicit statement to this effect.

Additionally, it was decided that all information-processing systems could be described adequately without having to consider the detection and decoding of analogue signals. Thus, all signals can be treated as the instantaneous receipt of a packet of information, without ref-

erence to the physical encoding by which this information is represented as a physical waveform. This results in considerable simplification of the specification by separating the behaviour caused by each signal from details of physical representation; it is therefore possible to write a specification for a system without having to define the physical form of any signal, leaving such decisions to be taken by the designer.

The following definition therefore summarises the view of specifications taken by the project.

'A specification is a statement of the required behaviour of a system when that system is viewed as a "black box". It is expressed in terms of the responses which the system will make to external stimuli, and both the stimuli and responses take the form of instantaneous events, although there may be delay between a stimulus and the consequent response. Such a specification will contain information about the speed of operation of the system, any limits upon its capacity to respond and its behaviour when overloaded; however, it should not introduce any unnecessary constraints upon the design of products to meet that specification.'

Subject to this definition, the terms of reference in Section 1.3.2 (a) to (d) were otherwise unchanged.

1.4. The Direction Taken by the Project

1.4.1. General

The developments in technology which have taken place since the start of the project have confirmed the importance of behavioural specifications (sometimes called "requirements specifications" (Lehman, 1981) or "functional specifications" (Mackie, 1981)) in large systems, especially when an existing product is to be reconstructed using some new technology. Where no such specification existed, it has sometimes been found necessary to create it before commencing the design of the updated system (e.g. Henninger, 1979), in order to ensure compatibility between the old and new versions.

However, over the same period of time the majority of published work on specification languages has concentrated upon the use of formal mathematical languages and the techniques of theorem proving (e.g. Abrial, 1980; Goguen et al, 1978; Musser, 1979; Neumann et al, 1980). This project has taken a different approach for the following reasons.

(a) A requirements specification cannot be proved correct by mathematical methods, as it can only be compared with the customer's mental model. Although a specification can later be used in a proof that the system design is correct, it is much more important to ensure that it is fully understood and accepted by the customer.

(b) The main difficulty in commercial organisations is to obtain the requirements information. Most specifications are incomplete in some parts for much of the duration of a project; it is therefore essential to accept and record incomplete information, allowing the specification to be created incrementally.

(c) As a consequence of (b), axiomatic methods (e.g. Goguen et al, 1978) may be impracticable, as they require a complete understanding of the system being specified. Their form also makes the incremental creation of a specification more difficult, as they achieve brevity by combining information about separate parts of the system behaviour.

(d) A specification forms the main communication link between the customer and the supplier; thus, it should aim above all else to be comprehensible to both parties. Mathematical elegance and tractability, often seen as advantages by the proponents of the more mathematical specification languages, do not necessarily bear any relation to comprehensibility (Green, 1977).

Hence the objectives of this project, which are outlined in the next section, are based upon the adoption of a simple model for specifications which sacrifices mathematical tractability, wherever that becomes necessary, in order to retain comprehensibility. In particular, the aim has been to minimise the number of concepts which would be unfamiliar to the staff of a telecommunications manufacturer and its customers.

1.4.2. ASL (A Specification Language)

A review of existing specification languages (reported fully in Chapter 2) did not result in the identification of one which was considered adequate; a new language was therefore designed in accordance with item (c) of the terms of reference (see Section 1.3.2). In order to avoid confusion when discussing the relationship between this language and other languages it was given the name "ASL", an acronym for "A Specification Language". ASL is described in detail in Chapters 3 and 4, but its main objectives can be summarised as follows:

(a) A simple model of systems.

ASL uses the stimulus-response ("black box") model of systems as described in Ashby (Ashby, 1969). This maps directly onto the physical realisation of information-processing systems, provides a discipline which assists in the detection of omissions, and helps to avoid a number of problems of semantics. (These points are discussed in detail in Chapter 3).

(b) A limited number of simple primitive operations.

In ASL there are two basic operations: the sending and receipt of messages. To offset this extreme simplicity, a message is allowed to contain an arbitrarily large amount of information.

(c) Implicit specification of data transformations.

Transformations on information (i.e. functions in the mathematical sense) do not have to be specified as algorithms which achieve the desired result; they can

be expressed directly in terms of the required relationship between the input and output values. This is consistent with the "black box" view of systems, and results in simple, comprehensible descriptions.

(d) Direct reference to past events.

There is no reason for a specification to be concerned with the methods (and economics) of information storage. For simplicity and comprehensibility, the specification writer should be allowed to refer directly to all the events (i.e. messages sent and received) which represent the history of the system. This is in contrast to system models such as those based upon finite-state machine theory (e.g. Parnas, 1972), where past events are summarised as if stored in a limited number of accumulators in the memory of a computer, with consequent loss of comprehensibility.

(e) Tolerance of incompleteness.

Incomplete specification is permitted in ASL by allowing any part of the specification to be stated to be "undefined". In this way the specification contains an explicit marker against every incomplete portion so that these can easily be identified by anyone reading the document.

Superficially, ASL has been kept simple; it uses English words (e.g. "send", "receive") rather than special symbols, in order to reduce the amount of training required to be able to read (rather than write) ASL specifications. The syntax of the language (see Chapter

5.2) is suitable for the simplest type of recursive-descent analysis (Davie & Morrison, 1981). This reduces the complexity of the support facilities (see Chapter 5) and may also be more acceptable to the users of the language than a more complex grammar (Green, 1980).

1.4.3. Expected Benefits

In the initial stages of the introduction of ASL, it is unlikely that any of the expected reduction in the total timescale of a project will be achieved due to the additional time taken to train personnel in the correct use of the language. It might even lead to an increase in the time taken for the first project on which any particular group of people uses the language, as the concept of formal requirements specifications will be new to them. However, even in these initial projects, it should be possible to detect a reduction in the number of errors which are not identified until after the construction stage.

Due to the long timescales for the types of projects undertaken by GEC Telecommunications Ltd. it is not possible to report any significant evidence of such improvements in this thesis, as the true value of formal specifications will only become apparent over a period of years. The few reports from organisations which have been using formal methods for a number of years (e.g. Alford, 1979; Lattanzi, 1981) indicate that around 50 percent of errors may be detected at the specification and design

stages due to the use of such methods. Although these figures relate to the production of large software systems outside the telecommunications industry they are indicative of the scale of the possible improvement.

Taking results from such sources together with some figures from within the Company, it is possible to arrive at an extremely approximate estimate of the benefits which might be obtained. Because of the degree of approximation involved, and the mixture of sources of the figures, every attempt has been made to take a conservative view. Only the System X projects within the Company have been included, as these are the only ones for which the costs and numbers of changes per annum can easily be obtained.

These figures are as follows:

(a) Current number of changes per year. 5000
(This is the number of change notes issued on the System X project in the year 1981.)

(b) Average cost of each change on System X within the Company (Dawkins, 1982). £280
(This represents the cost of engineering effort and documentation in processing a change, but does not include the cost of rectification on existing equipment. The cost is given at 1980 price levels.)

(c) Percentage of errors due to poor specification (Jones, 1979). 15
(Analysis of large software projects in the U.S.A..)

(d) Percentage of specification errors detected by formal methods (Alford, 1979). (Report on use of formal methods for software development in T.R.W. Inc..) 50

(e) Estimated possible saving per year ((a) x (b) x (c)/100 x (d)/100). £105000

It is quite possible that the elements for which estimated savings could not be obtained (e.g. the rectification of existing equipment, and changes on products other than System X) represent a potential benefit many times greater than the total shown above. Some managers in the Company who have been involved in the development of System X consider the savings shown above to be a gross underestimate of the likely effect; their experience indicates that a very large amount of effort is wasted due to incompleteness in the present specifications. However, it was considered that the figures presented should be well-justified, representing the minimum savings to be expected in practice.

1.5. Structure of the Thesis

The subsequent chapters follow the general development of the project in chronological sequence, with Chapter 2 covering the review of existing specification languages and Chapters 3, 4 and 5 describing the development of ASL. In Chapter 3 the fundamental decisions behind the design of the language are explained,

then Chapter 4 covers the detailed definition of the language and Chapter 5 describes the support which can be provided by computer facilities. The initial trials of the language are reported in Chapter 6 and evaluated in Chapter 7, then Chapter 8 provides some conclusions and proposals for further work.

Due to the volume of supporting material (e.g. tables of comparisons between languages, formal definitions of ASL) much of the detail appears as Appendices.

CHAPTER 2

REVIEW OF POSSIBLE CANDIDATE LANGUAGES

2.1. Introduction

A wide variety of notations have been proposed for use as specification languages, and in this Chapter a representative sample are reviewed. With such a large range to evaluate it is essential to have an objective basis for the assessment, so the first section of the Chapter is concerned with the development of criteria, which are then used in the evaluation. One major guiding factor in this review has been the suitability of the languages for use by existing personnel without the need for extensive retraining; this is reflected in the choice of criteria used in the evaluation. An earlier and less detailed version of this review appeared in (Blackledge(a), 1981).

2.2. Requirements of a Specification Language

2.2.1. The Starting Point

When viewed in the context of its intended purpose, a good specification can be seen to be one which is:

(a) comprehensible, to both the authors and the readers,

(b) testable, with all statements in the specification being measureable attributes of the final product,

(c) adequate, in that it contains all the appropriate information,

(d) maintainable, with a structure which facilitates the introduction of amendments.

However, these are not suitable criteria for an evaluation of specification languages as they are compound attributes, and can only be assessed subjectively. It is therefore necessary to determine a set of objective criteria which equate to the achievement of the above aims. This can only be done on the basis of the available evidence, which is limited and fragmentary (e.g. Green et al, 1981), so that the final list of criteria must be seen as a partial test, to be complemented by subjective assessment.

The final list was the result of an iterative process, where each item in the list was replaced by those more detailed items which contribute to its achievement, until a stage was reached where all items in the list were amenable to objective evaluation. The level of objectivity demanded was that it should be possible to identify clearly the presence or absence of the appropriate feature in a language; none of the criteria are sufficiently quantifiable to permit the languages to be placed in order.

Figure 2.1 shows, in the form of a hierarchy, the stages by which the final criteria were reached; the ini-

tial aims appear at the top, connected by pointers to the items by which they were replaced. In the figure each item appears only as a brief title, but the following paragraphs provide an explanation for each stage and for the titles.

2.2.2. Comprehensibility

Comprehension of text or diagrams is enhanced by good organisation of the material, using means such as those listed below.

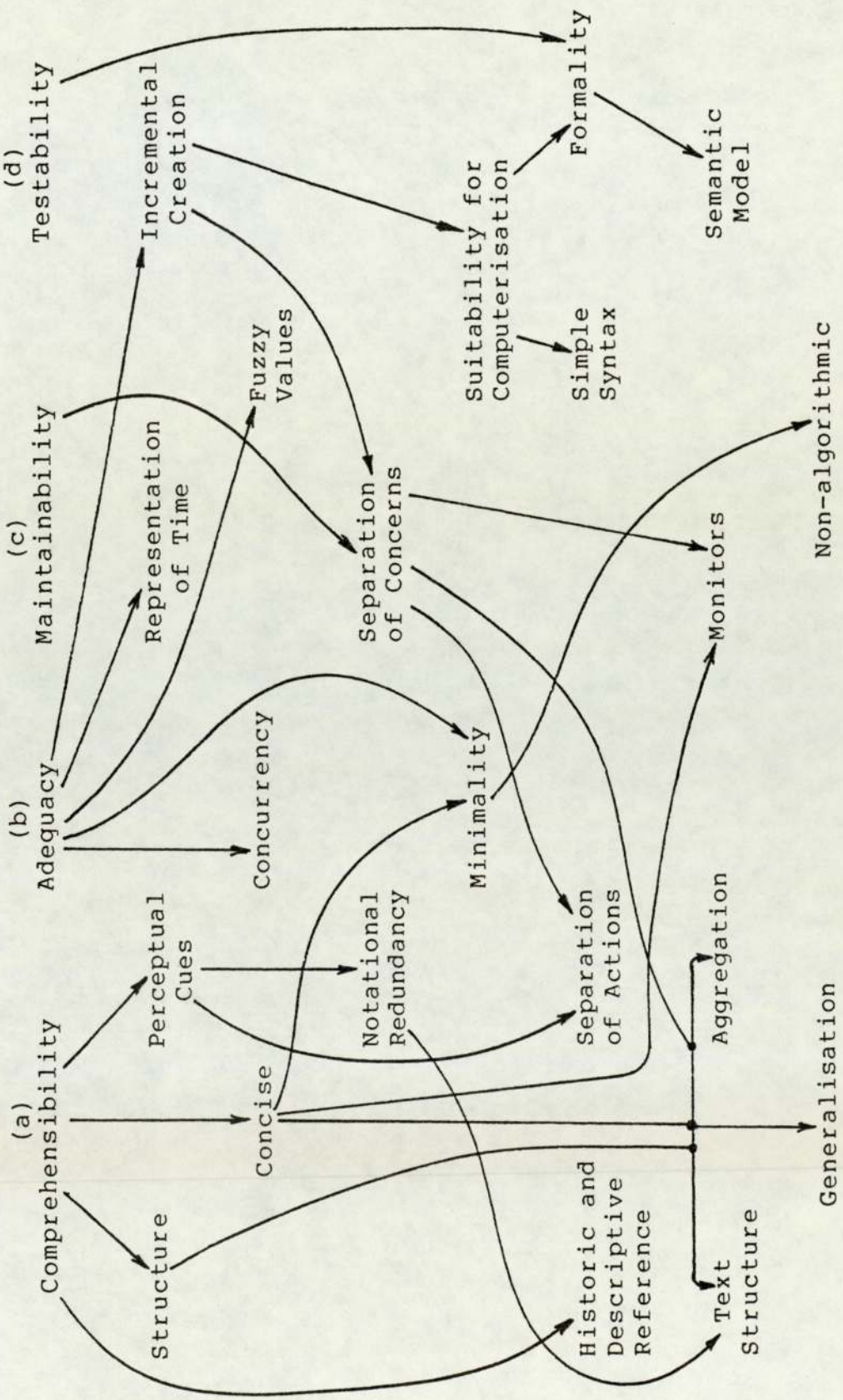
(a) Structure, such as paragraphs (Mills & Walter, 1978) and appropriate sequencing of the content (Posner & Strike, 1976), which are discussed in Section 2.2.6.

(b) Conciseness (Liskov & Zilles, 1978), which is discussed in Section 2.2.7.

(c) Perceptual cues, such as headings, which direct the reader's attention (Green et al, 1981; Hartley & Burnhill, 1977; Thomas & Carroll, 1981), discussed in Section 2.2.8.

(d) Descriptive and historical reference (Balzer & Goldman, 1979), rather than the modes of reference available in programming languages, where past information must be explicitly saved and cannot be accessed as "the last..." (Nylin & Harvill, 1976; Schueler, 1977), and must be mentioned by name rather than as "the ... with ...".

FIGURE 2.1 DEVELOPMENT OF THE SELECTION CRITERIA



2.2.3. Adequacy

This covers those features which make a language practical in a commercial environment on large projects with large project teams.

(a) Minimality, in permitting description of the required behaviour without demanding any unnecessary details (see Section 2.2.9).

(b) Recognition of concurrency. Most large systems involve actions occuring in parallel, and it is therefore appropriate to be able to represent this directly (Kornfeld & Hewitt, 1981; Petri, 1979).

(c) Representation of time. Although most properties of a system can be analysed using only the concept of sequence in time (Peterson, 1981), the omission of time delays and time limits leads to inadequate specifications (Winograd, 1979).

(d) "Fuzzy" values. Despite the need for quantitative statements which can be tested, there are likely to be many values which cannot be stated as a single, precise figure; if the author of the specification has to make an arbitrary choice of a single figure, this may result in unnecessary difficulties for the designer (Estrin, 1978). The language should therefore allow imprecise information to be stated, but in a way which indicates its nature (Balzer & Goldman, 1979).

(e) Incremental creation. For a large specification, with perhaps hundreds of pages, it is impractical to ex-

pect that all the necessary information will be available at the time when the specification is first written. Specifications are usually elaborated in discussion between customer and designer (Malhotra et al, 1980), and the document should at all times represent the latest information, even though this may be incomplete (Hewitt et al, 1979).

2.2.4. Maintainability.

Two identifiable factors which aid in the introduction of amendments are:

(a) separation of concerns, so that unrelated information is physically separated in the specification, as discussed in Section 2.2.10,

(b) computer-based support, to aid in locating the information to be changed and also in checking that the changes are made correctly and uniformly throughout the specification (see Section 2.2.11).

2.2.5. Testability.

The use of a formal language, which does not allow purely qualitative statements, is a major contribution to ensuring that the requirements are testable (Alford, 1977; Balzer & Goldman, 1979; Davis & Vick, 1977; Wasserman & Stinson, 1979).

2.2.6. Structure.

Apart from the physical structure of the text, there is also the organisation of the information for presentation. Two extremely useful forms of this type of structure are "generalisation" and "aggregation" (Smith & Smith, 1977). "Generalisation" is the use of a general object to represent the common characteristics of a collection of specific objects, e.g. the use of the word "dog" to represent the common features of a large set of individual animals. "Aggregation" is the introduction of a descriptive name for a group of associated objects, e.g. an "address" is made up of a house number, a street, a town and a postcode. These forms of structure provide a significant reduction in the amount of information in a specification, by allowing the use of the general names and aggregate names as abbreviations.

2.2.7. Conciseness.

"Conciseness" refers to features which help to produce short specifications. Text structure, generalisation, aggregation (see 2.2.6) and minimality (see 2.2.9) all contribute to the removal of unnecessary repetition of information; another feature which helps is the use of "monitors" (also called "demons" in Artificial Intelligence programs , e.g. Winston, 1976). A "monitor" is a statement of some condition (e.g. an error condition) and the action to be taken when that condition

occurs. It can be thought of as watching over the system, monitoring everything which happens to see if its condition occurs; and when it does then the monitor performs its action and afterwards returns to monitoring. An example of the analogous form in English is "When you feel hungry go and eat.".

2.2.8. Perceptual Cues.

Green et al (Green et al, 1981) point out the importance of visual cues to assist the reader.

(a) Text structure, such as paragraphs, headings, etc., to break the text into logical blocks (see 2.2.2(a)).

(b) Some redundancy in the notation, such as headings (Hartley & Burnhill, 1977; Thomas & Carroll, 1981), indentation of paragraphs (Green et al, 1981) and the use of a notation which avoids extreme terseness (Miller, 1967).

(c) Separate description of each action (Cleaveland, 1980), as this is more comprehensible than the intermingled form appearing in axiomatic descriptions (e.g. Guttag, 1977).

2.2.9. Minimality.

"Minimality" does not refer to the size of the specification document, but to the ability of a language to express exactly the required behaviour and no more (Liskov & Zilles, 1978). To achieve this the language

must not force the inclusion of unnecessary information, such as:

(a) an algorithm (detailed sequence of steps) for producing the required result, or

(b) a definition of the data to be stored within the system.

Both of these can be avoided, by using non-algorithmic languages for data transformations (e.g. Jones(a), 1980; Guttag, 1977) and historic reference to data (see 2.2.2(d)), and in this way the specification does not introduce unnecessary constraints upon the designer.

2.2.10. Separation of Concerns.

Correct use of text structure (Section 2.2.2(a)), including generalisation and aggregation (Section 2.2.6), monitors (Section 2.2.7) and the separate description of each action (Section 2.2.8(c)) result in a specification where each item of information occurs the minimum number of times, and only in appropriate places (Balzer & Goldman, 1979). This reduces the likelihood of some occurences of an item remaining unaltered when a change is introduced.

2.2.11. Suitability for Computerisation.

Goguen (Goguen, 1979) and Gerhart and Yelowitz (Gerhart & Yelowitz, 1976) note the prevalence of trivial errors in formal specifications, of types which can eas-

ily be detected by computer-based checking systems (e.g. Alford, 1977; Davis & Rauscher, 1979; Goguen, 1979; Teichrow & Hershey, 1977). For a language to be suitable for this kind of computer-based support it must be:

(a) simple syntactically, so that it is amenable to efficient, well-understood language processing techniques (e.g. Gries, 1971). Despite considerable progress in the processing of natural language (e.g. Bobrow et al, 1977), there are still many difficulties in applying these techniques in practice (James, 1981),

(b) formal, so that every possible statement in the language has a well-defined meaning. This is discussed further in Section 2.2.12 below.

2.2.12. Formality

A language with a well-defined syntax is not necessarily "formal", as without a sound semantic basis it is still ambiguous or meaningless (Lewin, 1977). From the point of view of this evaluation there are two types of semantic model which could be used.

(a) Operational models, where the meaning of statements in the language is "defined" by the operations which result when it is processed by a particular computer program (its "compiler").

(b) Theoretical models, where the meanings are defined in terms of some abstract, mathematical model, independent of any particular implementation on any particu-

lar computer.

Languages with theoretical models are preferable (Demuynck & Meyer, 1979), as any computer support facilities can use these theoretical models as an integral part of the checking procedures, rather than having to rely upon the integrity of a previous implementation.

2.2.13. Summary.

The final list of thirteen criteria, which provide the necessary level of objectivity, are :

(a) block or paragraph structure,

(b) generalisation,

(c) aggregation,

(d) separate description of each action,

(e) monitors,

(f) historic and descriptive references,

(g) non-algorithmic description of transformations,

(h) representation of time duration,

(i) recognition of concurrency,

(j) acceptance of fuzzy values,

(k) notational redundancy,

(l) simple syntax,

(m) a well-defined semantic model.

Although this list is not a complete set of criteria, due to the nature of the problem, the following sections will show that it does provide a sufficiently stringent test to indicate deficiencies in all the languages reviewed.

2.3. The Types of Specification Language

There are well over one hundred different languages which have been put forward as suitable for use in writing specifications, but by choosing a single language to represent groups which differ only slightly this has been reduced to eighty-eight in the review. Even with this reduction there is a need for some categorisation scheme which permits common failings and strengths to be identified. The categories which have been used are listed below, and divide the languages on the basis of their conceptual background, i.e. the source of the basic structure of the language. Where there appeared to be some choice over the appropriate category for any language, it was placed with the group which represents the major influence in its design. Many languages intended for other stages in the design process (Ramamoorthy & So, 1978) have been omitted; some of these, which are called "specification languages" by their authors, are much more concerned with the design of systems than with their required behaviour.

The categories are covered in Sections 2.4 to 2.17, as listed below, and then Section 2.18 summarises the evaluation.

Section	Category
2.4	Universal Languages
2.5	Computer Hardware Description Languages
2.6	New Programming Languages
2.7	Derivations from Programming Languages

Section	Category
2.8	Flow Charts
2.9	Hierarchic Description Methods
2.10	Finite State Machine Languages
2.11	Static Description Languages
2.12	Pre- and Post-condition Languages
2.13	Event-triggered Languages
2.14	Specification Analysers
2.15	Sequence Description Languages
2.16	Petri Nets
2.17	Languages using Axiomatics

Each of these categories is explained in more detail in the appropriate section and is briefly evaluated against the criteria given in Section 2.2.13. Tables showing the full evaluation of the eighty-eight languages against the thirteen criteria appear in Appendix A.

2.4. Universal Languages

The term "universal" is intended to indicate that these languages were not specifically designed for use in writing specifications, but have been used for that purpose.

2.4.1. Natural Languages

English and other natural languages have been used successfully as specification languages for many years, but with the increasing size and complexity of the sys-

tems now being developed their disadvantages have become more apparent (Alford, 1979; Jones(a), 1980; Lehman, 1981). The main problems relate to ambiguity (e.g. Hill, 1972) and the extensive use of implicit reference (Hobbs, 1977), which cannot always be fully resolved even in dialogue between the author and a supporting computer system (Balzer et al, 1978). Careful use of a natural language can produce good results (e.g. Naur, 1960), but the consequent need to give clear and complete definitions of all terminology can lead to verbose documents (e.g. Holbeck-Hanssen et al, 1975) without ensuring the removal of ambiguity.

2.4.2. Programming Languages

A number of proposals for specification languages make direct use of computer programming languages, such as APL (Jones & Kirk, 1980), and similar notations, e.g. PDL (Caine & Gordon, 1975). This takes advantage of the formal nature of these languages, with their simple syntax and defined semantic model (although many programming languages have only operationally-defined semantics - see Section 2.2.12), to produce precise, unambiguous specifications.

However, with few exceptions (see below) these languages are totally algorithmic, requiring detailed descriptions of the method for producing transformations on data, and fail to meet criterion (g). They also have been designed to operate efficiently on current computer

hardware, and so do not provide historic and descriptive reference (criterion (f)) or monitors (criterion (e)), cannot accept fuzzy values (criterion (j)), and provide no direct representation of time (criterion (h)). Prolog (Clocksin & Mellish, 1981) and SETL (Schwartz, 1973) both suppress almost all algorithmic detail, leaving their interpreter programs to organise the flow of control, and therefore satisfy criterion (g), but they share the other deficiencies mentioned above and fail to meet criteria (e),(f), (h) and (j).

2.5. Computer Hardware Description Languages

Computer Hardware Description Languages (CHDLs) are also known as Register Transfer Languages (RTLs) because they model digital circuits at the level of physical binary registers. Examples are AHPL (Hill & Peterson, 1973), DDL (Duley & Dietmeyer, 1968), HARTRAN (Bown, 1978), ISPS (Bell & Newell, 1971) and TEGAS6 (Szygenda, 1980). They are all extremely algorithmic as they describe in terms of a detailed design; at least one language (Bell et al, 1973) has been complemented by actual hardware modules, so that statements in the language can be directly translated into a design. Apart from the failure to permit non-algorithmic descriptions (criterion (g)), these languages are also restricted by their representation of all data as registers of bits, and therefore provide inadequate facilities for generalisation (criterion (b)) and aggregation (criterion (c)).

2.6. New Programming Languages

As a consequence of growing interest in formal proofs of correctness of computer programs (e.g. Mills, 1975) a number of programming languages have been developed which incorporate both the imperative features required to perform operations on a computer and non-imperative statements in which to make assertions about the intended correct behaviour of the program (e.g. Hantler & King, 1976). The languages Ada (Ichbiah et al, 1979), Alphard (Wulf et al, 1976) and Gypsy (Ambler & Good, 1977), and the Gamma program development system (Falla, 1981) all contain such features; however, as Krieg-Bruckner and Luckham (Krieg-Bruckner & Luckham, 1980) point out, they only provide sufficient features to verify the design, not to act as a complete specification. Even the extensions proposed by Krieg-Bruckner and Luckham (op. cit.) fail to satisfy criteria (e), (f), (h), (i) and (j).

2.7. Derivations from Programming Languages

In a number of cases a specification language has been derived from a programming language, either by the relaxation of the syntax rules to allow informal descriptions instead of algorithms or by the addition of features such as a representation for time. As examples, RLP (Davis & Rauscher, 1979) is based upon PL/I (IBM, 1976) with added block structuring, Delta (Holbeck-Hanssen et al, 1975) and Epsilon (Jensen et al, 1979) are based upon

SIMULA (Dahl & Nygaard, 1966) with the addition of non-algorithmic constructs. SMSDL (Frankowski & Franta, 1980) is also based upon SIMULA but uses informal descriptions of processes rather than additional formal statements. Others such as DDN (Riddle et al, 1979), SPECLE (Biggerstaff, 1979) and SREM (Alford, 1977) have similar forms, although they are not as closely modelled on any one programming language. As a group these languages still retain a large degree of the algorithmic nature of programming languages (see Section 2.4.2), even those which provide some non-algorithmic constructs; all represent data as stored variables rather than having descriptive and historic reference (criterion (f)).

2.8. Flow Charts

Flowgrams (Karp, 1978), progression charts (System X, 1979) and flow charts (Wayne, 1973) all take the form of diagrams containing boxes of various shapes connected by directed arcs. The boxes represent processes and decisions, and these charts are normally used to display the structure of a computer program or similar level of process (e.g. Sleight & Kossiakoff, 1974). The SX/1 system (Corker & Coakley, 1976) makes practical use of this by automatically producing computer program text from flow chart input. Apart from SX/1, which is comparable to the other programming languages in the evaluation (see Section 2.4.2), the flow charts all use informal, unstructured text as labels for their boxes and arcs. They

therefore fail to meet criteria (l) and (m), as they do not have a defined syntax or semantics for these labels.

2.9. Hierarchic Description Methods

One method for describing large systems which is often suggested in manuals on technical writing (e.g. Mills & Walter, 1978) is that of repeated subdivision into smaller and smaller elements until a level is reached when each element can be described in a few sentences. The work of Miller (Miller, 1967) on the number of "chunks" which can be stored in human short-term memory was taken as supporting this type of method, and a number of hierarchic specification languages appeared. These ideas also form the basis of various "structured programming methodologies" (e.g. Structured Systems Analysis (Gane & Sarson, 1979)).

CORE (Mullery, 1979), HIPO (Stay, 1976), SADT (Ross, 1977) and Structured Systems Analysis all use block diagrams to depict the hierarchy, with unstructured natural language text to describe processes. They therefore fail to satisfy criteria (l) and (m); also, because of their origins in commercial data processing, they have no representation of time or concurrency (criteria (h) and (i) respectively). In contrast, CADIS (Bubenko & Kallhammer, 1971), HOS (Hamilton & Zeldin, 1976) and PSL (Teichrow & Hershey, 1977) are based upon restricted languages with simple syntaxes and are provided with extensive computer-based support. However, PSL does not have any way of

describing data transformations, and all three fail to satisfy criteria (h) and (i).

2.10. Finite State Machine Languages

State transition diagrams and finite state machine theory have been used in the design of electrical cicuits for many years (e.g. Moore, 1956), but interest in their use for system specification appears to be more recent (Kawashima et al, 1971). The initial proposals (e.g. Hemdal, 1973; Kawashima et al, 1971) were based upon the use of state transition diagrams with informal labelling in natural language, and were therefore little more than special forms of flow chart (see Section 2.8); even some recent languages (e.g. Braek, 1979) have still retained this level of informality.

Other languages have been fully formal, so that they could be checked by computer and even in some cases automatically transformed into computer programs. Examples of these are CDL (Dietrich, 1979), NPN (Boebert et al, 1979), the notation of Parnas (Parnas, 1972) which also appears as SPECIAL (Robinson, 1976), and the notation used by Wymore (Wymore, 1967). In order to obtain the necessary formality in a cost-effective manner, all these languages took the form of text rather than diagrams; the CCITT Specification and Description Language (CCITT, 1980) has gone one stage further in having text and diagram forms which are equivalent and can be converted into each other automatically.

The main disadvantages of all these languages result from the use of the finite state machine model. This requires that every event and relevant state must be present explicitly in the specification and, for any large system, this involves a considerable number of states and events. Any attempt to introduce aggregation (criterion (c)), generalisation (criterion (b)) or monitors (criterion (e)) in order to reduce the size of the specification destroys the link to the underlying theory (Cohen, 1980), so that the resulting description no longer has a semantic model (criterion (m)). Without these features the specification has insufficient structure (criterion (a)).

2.11. Static Description Languages

The largest source of static description languages is the field of database systems, where the concern is in ensuring that a database accurately represents the state of the "real world" at some instant of time. In general, there is no attempt to describe the dynamic features which cause updates to the database, hence the use of the term "static". The Entity-Relationship model (Chen, 1976) was selected as a suitable representative of the database languages, but others which fall into this category are LEGOL (Stamper, 1977) which has been used in modelling statute law, SLICES (Steele & Sussman, 1979) which can represent ordered sets of constraints, and methods based upon invariants (e.g. Cun-

ningham & Kramer, 1977). By concentrating upon static aspects of a system, these languages provide a powerful form of monitor (criterion (e)); however, they fail to satisfy the criteria relating to dynamic behaviour (i.e. (d), (f), (h) and (i)).

2.12. Pre- and Post-condition Languages

In these languages each action is specified by stating the conditions which are necessary for it to commence (the "pre-conditions") and the conditions which will exist when it finishes (the "post-conditions"). For example, a square-root function could be specified as:

Pre-condition: A number, X, greater than or equal to zero, and some required tolerance on the answer, Y.

Post-condition: A result, R, such that $|X - R^2| \leq Y$.

This provides a good, non-algorithmic way of defining transformations upon data (criterion (g)), as exemplified in the work of Dijkstra (Dijkstra, 1976) which has been continued by Cunningham and Kramer (Cunningham and Kramer, 1977), the Vienna Development Language (Bjorner & Jones, 1978) and the related work by Jones (Jones(a), 1980), and the language Z (Abrial, 1980). However, all these languages provide no representation of time (criterion (h)) and cannot deal with concurrent systems (criterion (i)). Only the language Z has any form of text structure.

2.13. Event-triggered Languages

This type is differentiated from the others by its text form and the use of the concept of "events" without necessitating the use of system states (as is the case in finite state machine languages, Section 2.10). The Petri net languages (Section 2.16) are also based on events, but appear in a separate section because of their diagrammatic presentation.

There are a large variety of languages within this type, ranging from those not intended for computer processing (e.g. Jackson, 1981) to formal and complex ones with extensive computer support (e.g. Hewitt, 1977). There are examples which satisfy each one of the criteria, although no individual language satisfies all the thirteen. The most interesting of the group, because they provide features not found in other languages, are AP2 (Balzer & Goldman, 1979) which allows fuzzy values and historic references, and ACTORS (Hewitt, 1977) which is designed to permit incremental creation of specifications.

In general, the event-triggered view of systems offers a clear method for developing specifications by starting from the list of all possible events. However, it is not possible to specify all behaviour (e.g. maximum time delays between messages) solely in terms of external events; also this approach provides no obvious method of structuring large specification texts.

2.14. Specification Analysers

Although the languages of this type are not complete specification languages, as their purpose is only to analyse particular features of a specification, they have been included for completeness. As an example, SPECK (Quirk, 1978) deals only with the timing of messages, not their content, and checks to ensure that no messages can be missed due to time delays within the system.

2.15. Sequence Description Languages

The behaviour of a finite state automaton can be described fully by the sequences of messages which it will accept and send, this being an alternative to a finite state machine specification (Hopcroft & Ullman, 1969). By defining the sequences as regular expressions (Harrison, 1974) or path expressions (Campbell & Habermann, 1974), such a specification can be reduced to an acceptable size.

Milner's Calculus of Communicating Systems (Milner, 1980) and COSY (Lauer et al, 1979) take this form, and have been shown to produce useful theoretical results. However, the disadvantages of sequence descriptions are in their presentation; they have no text structure (criterion (a)) and require all actions to be described in sequences, not separately (criterion (d)).

2.16. Petri Nets

Petri nets (Petri, 1962) were devised as a visually simple representation of event-triggered systems, amenable to a variety of analyses (Peterson, 1981; Shaw, 1980). The original nets only had informal text labelling; however, the combination of visual simplicity and their ability to represent concurrency led to their use with formalised labels in GRAFCET (Bouteille, 1978), LOGOS (Rose et al, 1972), Pro-Nets (Noe, 1978) and SARA (Estrin, 1978). These languages all have limited facilities for aggregation but no facilities for generalisation or monitors; thus, specifications written in them tend to be large and lack structure. More recent work (e.g. Genrich et al, 1980) has introduced limited forms of generalisation (criterion (b)) and monitors (criterion (e)), but not a method of representing time (criterion (h)).

2.17. Languages using Axiomatics

The basic systems in mathematics, such as Euclidian geometry and the natural numbers, are defined axiomatically (Stewart, 1975), as this provides a concise and minimal definition. A number of specification languages have therefore used this approach in an attempt to obtain the same benefits. Examples are ADJ (Goguen et al, 1978), Affirm (Musser, 1979), CLEAR (Burstall & Goguen, 1977), iota (Nakajima et al, 1977), OBJ (Goguen, 1979) and the notation used by Schwartz and Melliar-Smith (Schwartz &

Melliar-Smith, 1981).

Axiomatic descriptions have the same disadvantages as sequence descriptions (see Section 2.15). Since actions are described in combinations rather than separately (criterion (d)), it is difficult to provide any structure to the specification (criterion (a)) without performing part of the design. Additionally, there are practical difficulties in constructing an adequate set of axioms which encapsulate exactly the required behaviour (Guttag, 1977); this casts doubt upon their suitability for use in a commercial environment.

2.18. Conclusions

From the comments about each category, which appear in sections 2.4 to 2.17, together with the detailed tables in Appendix A (see especially Table A.15, which gives a summary of Tables A.1 to A.14), the following conclusions can be drawn.

(a) No single language in the review satisfies all thirteen criteria. The event-triggered language, AP2, and English come closest to satisfying all thirteen, but English has no formal semantic model while AP2 has only operationally-defined semantics and lacks text structuring facilities.

(b) For the majority of languages with a well-defined semantic model the emphasis placed upon theoretical correctness appears to have resulted in a lack of features to aid comprehension.

(c) There is often no clear differentiation drawn between specification and design documentation. Almost all the languages require some design information (either algorithms or definitions of stored data) to be included in the specification.

(d) Direct use of, or the extension of, a language designed for other purposes (e.g. a programming language) appears to retain many of the disadvantages of that language as a result of the inclusion of features which were appropriate for the base language, but which are not necessary in specifications.

On this basis it was decided that a new language should be developed, which would attempt to combine the strengths of a theoretical semantic model with those features which had been noted as contributing to comprehensibility. In the next chapter, the fundamental decisions behind the design of this language are explained and then the language itself is introduced in Chapter 4.

CHAPTER 3

THE DESIGN OF A SPECIFICATION LANGUAGE

3.1. Introduction

The review of languages in the previous Chapter indicated the wide variety of views of systems which can be used in writing specifications. These views are similar to the scientific paradigms proposed by Kuhn (Kuhn, 1970; Floyd, 1979) in that, once a particular view has been selected, it is difficult to change. To design a specification language, it is necessary to select one view (or a compatible set of views) as a consistent framework (paradigm) for the language, whilst ensuring that this framework is sufficiently powerful to deal with a wide variety of types of system. This Chapter contains an explanation of the framework which was chosen, leaving the details of the language structure until Chapter 4.

3.2. General Approach

In Chapter 1.2 the role of specifications was discussed in the context of current practice in the British telecommunications industry. Any specification lan-

guage developed for use in this environment must recognise the practical need to minimise the degree of retraining of existing staff. This stresses the need for a language which:

(a) does not utilise new and complex symbols where existing English words would suffice,

(b) permits the use of terminology specific to each project, rather than enforcing some restricted set of terms,

(c) was designed with mathematical tractability being treated as secondary to the achievement of a language in which the necessary information can be easily expressed.

Despite the emphasis which this places upon the need for a language which appears acceptably familiar and readable to existing staff, it is not the aim to produce a specification which can be read by someone new to the project being specified. A specification is not intended to be suitable training material for staff entering a project, but forms the contractual definition of the work to be done (Mackie, 1979). Hence, the objective of a specification is the accurate definition of the required behaviour, not the provision of a structured introduction to the system.

3.3. Formality

In Chapter 2.2.12 a formal language was defined as one having well-defined syntactic and semantic models, so

that a specification written in the language avoids the problems of ambiguity found in natural languages. Formality in this sense is therefore an essential feature of the new language; but it does imply that, if the new language is to be small and simple, the resulting specifications will be less flowing than ones written in English. Once the complexity of a language approaches that of English, the fallible human ability to detect errors cannot be adequately supported by current computer-based techniques (James, 1981; also see Balzer et al, 1978 and Bobrow et al, 1977 for indications of the limitations of current techniques).

Hence the detailed design of the language must attempt to produce an acceptable compromise between flexibility (for the writer) and simplicity (for checking) in a manner which maintains the basic formality of the language. The remaining sections of this chapter discuss the main elements of the formal basis of the language in an informal manner, with the formal definitions being covered in Chapter 4.5.

3.4. A System and its Environment

3.4.1. The System

In Sections 1.2 and 1.3 a specification was shown to be primarily a means of communication between customer and supplier. This implies that much of the document will be written prior to both design and manufacture, so that

it forms a prediction of a future situation as it should exist after the product has been delivered to the customer (Lehman, 1981). In such circumstances, it is not possible to describe the proposed system by presenting details of its construction or internal operations, because these are not yet known. The descriptions may be couched in terms of an "abstract" design, not intended to prejudge the actual design process. However, this introduces the same type of problems as an "algorithmic" language (see Chapter 2.2.9), because the resulting specification cannot easily be separated into those details which are essential and those which are only a result of the choice of abstract design (Liskov & Zilles, 1978).

Such problems can be avoided by ensuring that the specification represents the way in which the customer will see the system, i.e. as a "black box" (Ashby, 1969; Weinberg & Weinberg, 1979); thus, all possible designs for the system are observationally equivalent (Milner, 1980) if they meet this external specification. Observational equivalence and the "black box" viewpoint both concentrate upon describing the customer's world (as proposed in e.g. Jackson, 1981), not upon details of the design, and are therefore likely to result in documents which are comprehensible to the customer.

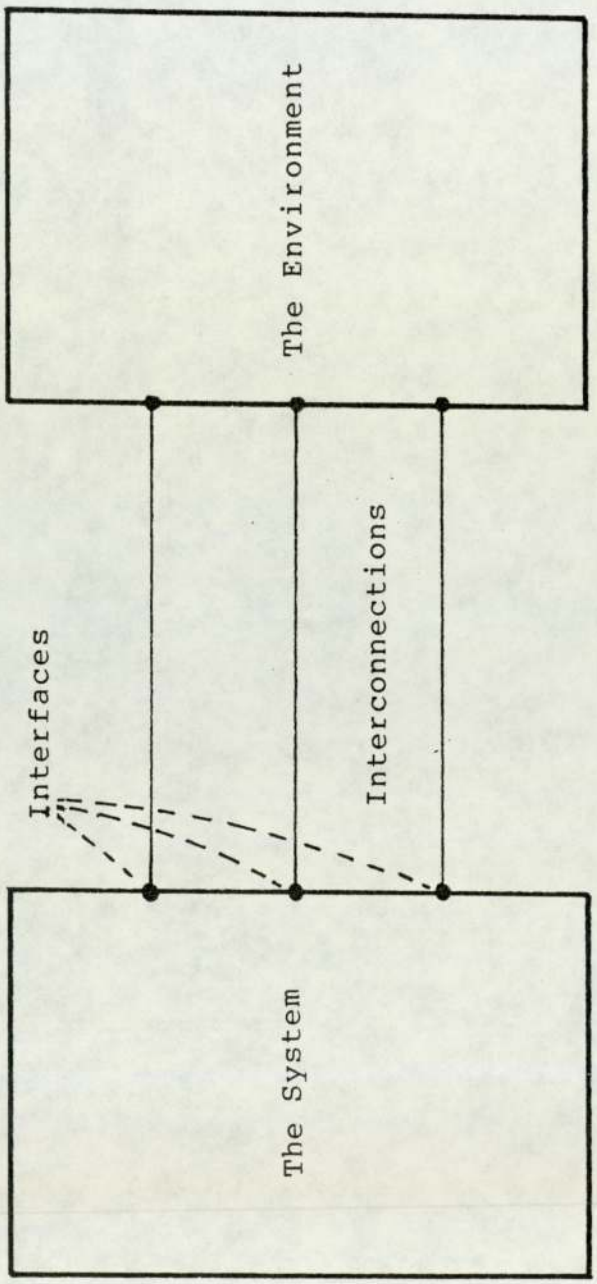
3.4.2. The Environment

The specification must however contain a clear definition of the boundary between the system and its environment, as this bounds the task to be performed (Lattanzi, 1980; Thatte, 1980). This can most easily be achieved by viewing the environment as a system also (Balzer & Goldman, 1979); the only difference between the system and its environment when viewed as systems is that the supplier (designer) does not have to design or manufacture a product which implements the environment.

Hence a specification takes the form of two (or more) descriptions of "black boxes" - at least one for the environment and one for the system being specified - linked together by a description of the connections between them, as depicted in Figure 3.1. Although it should always be possible to represent the system and the environment as one "black box" each, it is convenient to allow the use of more where physical separation (e.g. as of the subscribers of a telephone exchange) makes it difficult for a person reading the specification to view this as one entity.

This general structure for the specification has significant advantages for the semantic definition of the language, as is explained in Sections 3.5 to 3.7 below. It also captures the concepts of modularity (Stevens et al, 1974; Parnas, 1972) and observational equivalence, which provide independence from the particular technology used to implement the system. In this

FIGURE 3.1 THE GENERAL STRUCTURE OF A SPECIFICATION



way, the specification is able to act as a common reference for a number of implementations, to ensure that they are equivalent.

3.5. Communication by Message Passing

3.5.1. Messages

The main consequence of the "black box" model of systems is that the only way of obtaining information about a system is by sending messages to it and awaiting the replies. The "black box" representing the environment cannot have direct access to stored information in the system, as is the case in some other languages (e.g. SMSDL (Frankowski & Franta, 1980)); thus the specifier is forced to state all communications explicitly. This discipline can be strictly enforced as part of the checking facilities described in Chapter 5, and results in a method of specification which is highly "analogic" (as opposed to "Fregean" (Sloman, 1971)) in restricting the specifier to the same type of message-passing as will exist in the designed product. Sloman (op. cit.) suggests that "analogic" representations (such as message passing in the case of information-processing systems) are much more useful in problem solving situations than "Fregean" ones such as those covered in Sections 2.11, 2.12 and 2.17.

The sending and receiving of messages therefore become primitive concepts within the language, having the

following properties.

(a) A message is an instantaneous event involving the transmission of information. Hence a physical method of transmission which may represent a message as a sequence of voltage variations over some period of time is modelled in the language as a single, instantaneous event (usually at the final instant of the physical message). If the variation of a single, continuous waveform is significant, then it has to be modelled as a number of instantaneous messages.

(b) The transmission of a message is assumed to be instantaneous and error-free; thus, any delays or noise within the system are functions of the "black boxes", and not of the transmission medium. (The treatment of time is covered in more detail in 3.7 below.)

Any messages which are transmitted continuously for an indeterminate length of time (hereafter called "continuous messages") can be incorporated into this framework by considering only their extremities. Thus, the start and end of a continuous message are treated as instantaneous messages with exactly the properties noted above.

3.5.2. The Observer

The interconnections between the "black boxes" are not visible to any one of them, only to a hypothetical "observer" (Jensen et al, 1979), represented by the reader of the specification; it is this observer who at-

taches meanings to the names of the messages which pass through these interconnections. In this way the problems of ambiguity of names (Hayakawa, 1978) are resolved by forcing the sole definition of the messages to reside with the observer, and prohibiting each "black box" from maintaining its own, separate version. Within each "black box" the only "meaning" of a message is the response which it triggers.

3.5.3. Message Contents

As each message is treated as an instantaneous event, there is no need to introduce any detail of its physical structure into the specification. Where the behaviour of the system is dependent upon the content of a message, this can be modelled without the necessity for any description of how the content is encoded into the message. Reference to message components is achieved by naming each component, and these names may be organised into a hierarchy of any level of complexity to provide the required degree of discrimination between different messages.

One significant advantage of this hierarchical structuring of messages is that at any stage in the development process the specification need only contain as many levels of detail as are relevant to the current state of the specification. If extra detail is to be inserted at a later stage, this may be added as a further layer in the hierarchy.

3.6. The "Black Box" View

3.6.1. Models

Henceforth the word "model" will be used, instead of "black box", to represent a closed object which communicates by passing messages. A specification will therefore consist of a number of models, with at least one model for the system being specified and at least one for the environment. The term "model" was chosen to emphasise the distinction between the level of detail in the specification and the true complexity of the "real world" (as Hayakawa notes by freguent use of the phrase "the map is not the territory" (Hayakawa, 1978)). Thus, the specification can only be a limited analogue of the real world from some specific viewpoint (Kent, 1977).

3.6.2. Interfaces

The restrictions of message passing have been reinforced in the language by ensuring that models can only communicate with each other via well-defined interfaces, and that only the observer can see the interconnections between these interfaces. Thus, each model cannot know to which other models it is connected, and it must obtain any information about its environment by an exchange of messages. This ensures that the specification can contain no hidden assumptions (as can be the case in languages, such as SMSDL (Frankowski & Franta, 1980), which allow a

model direct access to information stored inside other models). Because all information within a model must have been obtained by an exchange of messages, the omission of such an exchange from the specification is easily detected.

A model can have properties which differentiate it from the other models within the specification (such as the unique telephone number of each subscriber on a telephone exchange), but these are not visible to other models, and cannot be directly updated or changed by other models.

3.7. Time

3.7.1. Requirements

A further consequence of the "black box" view of systems is that the definition of the required response times of the system must also treat each model as a closed object. Only the delay (or acceptable range of delay values) between any received message and the subsequent output message can be stated. This means that the model of time provided in a specification language can be extremely simple; it can be limited to consideration of "worst case" values and ignore the detailed timing problems which may arise during design.

3.7.2. Time Stamps

With the restriction of time delays to the models, transmission between models is assumed to be instantaneous. Thus, the use of a notional observer of the system (as discussed in 3.5.2) makes it possible to avoid the difficulties of introducing absolute time values into the specification (Lamport, 1978; Sernadas, 1979) as follows.

(a) Only the observer makes use of absolute values of time, in attaching a "time stamp" (Lamport, 1978) to each message transmitted between models.

(b) A model can introduce a delay between receipt of a message and any subsequent response. However, this delay is of a number of time intervals and does not require the model to recognise some instant on an absolute time scale.

(c) Messages are received by a model in absolute time sequence, but this takes the form of the value of the "time stamp" in the message, placed there by the observer.

Thus, models are only concerned with small time intervals and the ordering of sequences of messages by the values of their time stamps. As there is only a single observer (see 3.5.2), there are no problems due to different information transmission delays to different observation points. The only consequent deficiency in the language is that, for those cases where transmission delays are significant, extra models must be introduced

into the specification purely to represent this feature. However, this appears to be acceptable when compared with the advantages gained.

3.7.3. Time Viewpoint

Within the languages reviewed in Chapter 2 there are represented a number of different viewpoints of time, of which the following are examples.

(a) The static description languages (Chapter 2.11) ignore time by describing behaviour rules which must be true at all points in time.

(b) The pre- and post-condition language Z (Abrial, 1980; see also Chapter 2.12) is mainly used by its authors as if looking back on the system behaviour from "the end of time". Thus, the specification uses the equivalent of the passive past tense in English.

(c) Finite state machine languages (Chapter 2.10) describe actions at the time they are triggered, with reference to the previous behaviour of the system. This is analagous to the active current tense in English.

The viewpoint chosen for inclusion in the new language is the one which Sernadas calls "privileged initial time" (Sernadas, 1979), and which is used in Systematics (Grindley, 1975). It is identified by:

(i) specification of the behaviour as it appears at the instant at which it is triggered, making reference to past events,

(ii) all stored information within models being assumed to have been initialised before any actions take place, and thereafter only updated through message passing.

Apart from the treatment of stored information, this is equivalent to the "current time" viewpoint. This, together with a dynamic (active) rather than static (passive) description, seems to be easier to understand than other viewpoints (Hartley & Bunhill, 1977).

3.8. Memory

Having achieved an acceptable representation of time which suffices to order the events being specified, there is no need to resort to the explicit storage of information (e.g. in the "system state variables") to maintain the history of the system. By extending the idea of restricted access to past values (Nylin & Harvill, 1976) to unlimited access to all previous events (Balzer & Goldman, 1979; Schueler, 1977; Stamper, 1977), the need for algorithmic descriptions is much reduced. At the same time, the language moves closer to the natural mode of expression in English (Elton & Messel, 1978).

e.g. as in: "the last ...."

or: "the value at the time when....."

Although Sernadas (Sernadas, 1979) argues that it is permissible to represent information as being stored in memory and updated, this fails to recognise the problems caused by algorithmic descriptions (see Chapter 2.2.9),

as recognised by Balzer and Goldman (Balzer & Goldman, 1979), Grindley (Grindley, 1975) and Walters (Walters, 1979).

Direct reference to history is not normally a feature of practical designs, as it implies an extremely large amount of storage, together with consequentially large search times to extract the required information. However, as was pointed out in Chapter 1.4.1, the main purpose of a specification is to communicate information between people, not to demonstrate how the design could be made efficient; thus, the implied computational inefficiency is acceptable if it leads to improved comprehensibility. In order to make the use of direct reference to history easy for the specification writer, it is necessary to design into the language adequate modes of access to allow the extraction of individual messages, groups of messages and the total history, using terms such as "next", "last", etc., without direct reference to values of absolute time.

3.9. Structuring the Specification

The division of a specification into models for the system and its environment plus the interconnections provides a rudimentary structure to the document, but fails to provide any organisation to the contents of each model. A model contains the descriptions of the behaviour which it should exhibit on receiving messages through its

interfaces, and in the simplest possible form (as in a finite state machine model, see Chapter 2.10) this would appear as a complete list of the responses appropriate for each individual message which could be received.

In Chapter 2 the terms "aggregation" (Section 2.2.6), "generalisation" (also in Section 2.2.6) and "monitors" (Section 2.2.7) were introduced for types of structure which are appropriate to specifications. Theseprovide both briefer and more comprehensible descriptions by permitting statements which apply to classes of entities or events rather than just to individuals. Additionally there must be the capability to represent blocks of text which are repeated within a specification by some abbreviated references, as is done in computer programming languages with subroutines, macros, functions, procedures and similar devices. These must all be provided in a way which promotes their use, even at the cost of added complication in any supporting computer programs.

The surface form of a number of the languages reviewed in the previous chapter appears to have resulted from compromises in their design. These compromises were aimed at limiting their structuring power to match the capability of theorem proving systems or other manipulative methods, although this is normally not admitted to be one of the major parameters in their design (see for example (Boute, 1981)). The difficulty in using methods such as proofs of correctness, due to the NP-complete nature of the proof process (Lehman, 1981; Wirth(a),

1977) implies that this is likely to result in an unsatisfactory loss of comprehensibility whilst not providing any guarantee of mathematical tractability.

3.10. Incompleteness

A specification is only a model of part of the real world (see 3.6.1 above), and therefore cannot be assumed to be immutable; the real world will be changing continuously, and the specification must reflect these changes (Lehman, 1981; Liskov & Zilles, 1978). Additionally the specification is normally developed over a period of time by discussion between the customer and the supplier (Malhotra et al, 1980), in a manner which appears to parallel Popper's view of the development of scientific theories (Popper, 1974). Hence the specification document at any point in time only represents the latest available information, and may be incomplete or incorrect or both.

The language in which the specification is written must therefore permit incomplete information to be recorded (Hewitt et al, 1979), but in a way which indicates that it is incomplete. Hence it must be possible to differentiate between:

(a) information currently missing from the specification, but which is expected to be added as soon as it becomes available,

(b) values which are represented as ranges because the precise figures have not yet been decided,

(c) decisions where the input to the decision process may or may not be stated precisely, but the conditions under which the various outcomes are appropriate are not known precisely,

(d) situations where the particular outcome of a decision or the value of some piece of information is not important (like the "don't care" values in Boolean logic design of digital circuits),

(e) uncontrolled factors, such as the timing or content of messages from the environment, which must be modelled by statistical methods,

(f) precise information.

Extensive use of these facilities to represent imprecise information does however imply a high rate of change to the specification during the development of the system. This is one of the reasons why it is proposed that the specification writer should be supported by a comprehensive computerised facility as described in Chapter 5.

3.11. Form

Although a diagram can make obvious some aspects of the structure of information in a way which is difficult or impossible in text, there are considerable difficulties in designing good diagrammatic notations (Fitter & Green, 1979). In the case of specifications, one difficulty is the representation of the forms of structure (e.g. aggregation, generalisation and monitors) in a diagram. For example, the finite state languages which

use diagrams (e.g. SDL, see Chapter 2.10) do not provide sufficient structure and consequently produce large diffuse specifications.

Much of the information on a diagram must still appear as text labels upon the symbols, so it is still necessary to define a text form as a major part of any notation. This needs to be a formal, restricted language in order to avoid the problems which languages such as SADT (Ross, 1977) and PSL (Teichrow & Hershey, 1977) suffer in allowing unrestricted and unformatted labelling of their diagrams in English. It was therefore decided that the primary aim would be to derive a language consisting of text alone, leaving any diagrams to be produced manually as additions to the specification. This also reduces the complexity of any initial computer support software significantly, by avoiding the need for graphics input and output and permitting the use of readily-available syntax analysis techniques (see Chapter 5.2).

3.12. Summary

Sections 3.4 to 3.11 above have presented the reasoning which led to the adoption of the following fundamental features in the language being designed.

(a) Message passing as the only method of communication.

(b) Instantaneous, error-free message transmission.

(c) The treatment of models as closed entities, so that their information is only available by exchanges of

messages.

(d) Separate descriptions of the system being specified and its environment.

(e) Interconnections between the models only being visible to the single observer of the system.

(f) A simple model of time.

(g) Specification from a temporal reference of the "current time" with access to all the events which occured in the past.

(h) Direct access to past events, to avoid much of the description of data storage within the system.

(i) Structuring facilities which allow the description of the required behaviour in layers.

(j) Some shorthand reference for repeated behaviour.

(k) Facilities for recording imprecise information or behaviour specification in a way which indicates its nature.

(l) All information to be presented as text, with any diagrams being either derived from the text or produced manually.

These features taken together provide a framework which appears to be adequate for all information-processing systems, and which implies a strong discipline for ensuring consistency within a specification. In Chapter 4 the detailed design of the language is described, showing in 4.3 and 4.4 how an attempt has been made to capture the above features in a simple syntax, and then in 4.5 covering the formal definition of the language.

CHAPTER 4

THE DETAILED DESIGN OF A SPECIFICATION LANGUAGE

4.1. ASL

In order to permit unambiguous discussion of the relationship between the language being designed and other languages, it was decided to give it a name. The one chosen was 'ASL', this being an acronym for 'A Specification Language'. Chapter 3 contained discussion of the fundamental features of the language; the detailed design of ASL is now described in this chapter. The associated formal definitions of the language appear as Appendices, due to their length.

4.2. The Surface Appearance of ASL

The most important decision in the design of the language was that of its general appearance. The major factor affecting this decision was the size of the intended audience of the specifications written in the language. This involves hundreds of people of widely varying backgrounds at the sponsoring Company and, if the Company's customers are included, the numbers rise into the

thousands. Large-scale retraining of these people in the use of an abstract mathematical notation (e.g. CCS (Milner, 1980)) would be both difficult and time-consuming. It would also delay the use of specification languages, thereby losing some of the short-term benefits to the Company (see Chapter 1.4.3). There is no evidence to show that this loss is offset in the long-term as a result of using such an abstract notation.

It was therefore decided that ASL should use words from English wherever possible, and that the constructs of the language should have a simple reading which conveyed much of their meaning. In this way the amount of training required to read specifications written in ASL is minimised. Although there is not a corresponding reduction in the training required by specification writers, this still represents a significant overall reduction as readers are in the majority. Such simplicity in the form of the language was also seen as a factor in reducing any initial adverse reaction to the use of a formal language.

4.3. Consideration of Human Factors

4.3.1. Consequences of Earlier Decisions

The basic design of the language (reported in Chapter 3) results in the main organisation of a specification as two or more "black box" models (Chapter 3.6.1), communicating with each other by passing messages (Chapter

3.5.1) through interfaces (Chapter 3.6.2). This is directly reflected in the structure of the specification by requiring each model to be a separate identifiable block of text. There is also one additional block, containing details of the interconnections between the models and other information which is relevant to the observer (Chapter 3.5.2). As is explained in more detail in 4.5.4 below, this results in a simple relationship between the position of any name appearing in the specification text and its visibility to different parts of the system (often called the "scope" of the name).

The remaining portions of the language were relatively unconstrained by these factors; they are the result of an examination of a number of existing computer programming languages such as Pascal (Jensen & Wirth, 1975), Ada (Ichbiah et al, 1979) and PL/I (IBM, 1976) and of the few papers containing guidelines on language design (Fitter & Green, 1979; Gannon & Horning, 1975; Green et al, 1981; Hoare, 1973; Hobbs, 1977; Pratt, 1975; Tennent, 1977; Wirth, 1974). A number of the choices made during the design depart from the advice given in the above references, mainly in relation to those points where the design of programming languages appears to be compromised in order to achieve efficient compilation. The reasons for the particular choices which were made are discussed in Sections 4.3.2 to 4.3.5 below, whilst their detailed appearance is covered in 4.4.

4.3.2. Order within the Specification Text

Programming languages such as Pascal require the program text to appear in a particular sequence with, for example, the first appearance of any name having to be its definition. As has been noted (e.g. Peterson, 1980), this sequence conflicts with the top-down approach to the development of a system, where names are normally introduced before their definition. The purpose of such restrictions on sequence is to simplify the work of the compiler or interpreter, by making it possible to analyse the program fully in one pass over the text.

As ASL is not required to have a simple or efficient compiler, this type of restriction can be avoided. The non-algorithmic nature of ASL permits a further relaxation of restrictions, in that the order of the statements within any block (i.e. model) has no significance in terms of the semantics of the language. This allows a specification writer to present the information in whatever is the most comprehensible sequence.

4.3.3. Paragraph Numbers

One consistent difference between natural language descriptions and computer programs is that the former use paragraphs and paragraph numbers to organise the text (e.g. Mills & Walter, 1978), whilst the latter use words such as "BEGIN" and "END" to achieve the same effect. As the BEGIN-END form offers much less of a perceptual cue

to the reader, much reliance has been placed on the use of indentation (Rose & Welsh, 1981) and similar methods (Green, 1980) in the presentation of programs.

ASL uses a paragraph numbering scheme, with the decimal form of numbering (e.g. [1.3.15]). This has the following advantages:

(a) the structure is made visible without resorting to indentation,

(b) sub-paragraphs (and sub-sub-paragraphs) are easily identifiable from the number of levels in their paragraph number,

(c) no explicit indication of the end of a paragraph is needed; the next paragraph number is sufficient indication of the change of scope.

4.3.4. Comments

Examinations of the use of comments in computer programs (e.g. Weinberg, 1971) have shown that these are not always used appropriately. Too little emphasis appears to be given to general comments, which explain the overall structure and purpose of the program. It was therefore decided to restrict the use of comments in ASL to a few specific points in the language, in an attempt to foster their correct use. These three points are at the start of each block of text (i.e. model), in the definition of new names, and in paragraph headings (i.e. immediately after a paragraph number).

4.3.5. Alternatives in Behaviour

Where the response to a stimulus is dependent upon some conditions, it is more comprehensible if the normal behaviour is presented first and the less-frequent situations afterwards (Mills & Walter, 1978). If there are a number of optional responses, all equally likely, then it should not be necessary to use nested IF-THEN-ELSE statements to indicate the alternatives as this form can involve the "dangling ELSE" ambiguity (Aho & Ullman, 1977). ASL provides a different form for each of these cases.

(a) Where there is a normal response and one or more other options, then the normal response is given first followed by the word "unless" and the other options. Each option consists of a response together with the conditions under which it is appropriate.

(b) Where there is no obvious normal response, all the options are shown as sub-paragraphs after the word "select". Each sub-paragraph states the conditions which must be met for that option to be selected.

As these two offer all the necessary facilities, the IF-THEN-ELSE form which appears in most computer programming languages has not been provided.

4.4. The General Appearance of ASL

4.4.1. Introduction

The formal definition of a language consists of com-

prehensive syntactic and semantic rules, which usually cover many pages of text; even informal presentations of programming languages can take over 50 pages (e.g. Jensen & Wirth, 1975). Thus, ASL has been documented in an introductory report (Blackledge(b), 1981) and a language reference manual (Blackledge(a), 1982), which cover the language in much greater detail than is appropriate here. The remainder of section 4.4 therefore contains an outline of the surface appearance of ASL, and is supported by the formal definitions, which appear in Appendices B and C, and a small example specification in Appendix E.

4.4.2. Block Structure

A specification must consist of at least three blocks of text, as explained in section 4.3.1. More blocks may be used if this leads to a better representation of either the system being specified or its environment; an example would be the specification of a local telephone exchange, where the environment is more comprehensible if represented as a large number of copies of a subscriber model. Each block takes the form of a sequence of statements enclosed by a head and a tail, e.g.:

```
EXAMPLE_BLOCK is
  ....sequence of statements....
end of example_block
```

The reasons for the words "EXAMPLE_BLOCK" appearing in capitals in the block head, and lower case letters in the

FIGURE 4.1 THE STRUCTURE OF A SPECIFICATION IN ASL

Heading or Paragraph Number		Comments
AN_ASL_SPECIFICATION is		The system block.
[1]	Statement of models and	
[2]	their interconnections.	
[3]	Definition of common items,	
:	used in the models.	
[6]		
end of an_asl_specification		
THE_ENVIRONMENT is		The environment model.
[1]	Interfaces.	
[2]		
[3]	Properties of the	
[4]	environment, known only to the environment.	
[5]	Responses to situations	
[6]	rather than messages.	
[7]	Responses to particular	
[8]	messages.	
end of the_environment		
THE_SYSTEM is		The system model.
[1]	Interfaces.	
[2]		
[3]	Properties of the model,	
[4]	known only to the model.	
[5]	Responses to situations	
[6]	rather than messages.	
[7]	Responses to particular	
:	messages.	
etc.		
end of a_system		

block tail in the above example are explained in section 4.4.3 below. Figure 4.1 shows the structure of a specification in terms of such blocks of text.

4.4.3. Names

A unique name is given to each item (e.g. message, piece of stored information) defined by the specification writer; it takes the form of a sequence of characters or underscores, and must start with a letter.

e.g. aname, another_name, z123.

To achieve the required flexibility in the order of statements within a specification (see 4.3.2), it is essential to have a simple method of recognising those statements which define new names. Programming languages such as PL/I (IBM, 1976) use an identifying word (e.g. "DECLARE") at the start of each definition, but this is only necessary because they do not make use of the full character set available on most computers. ASL avoids the need for such a word by requiring all names to appear in lower case letters except in the statements where they are defined, where they are written in capital letters. This also has the advantage that definitions are consequently highlighted in the specification text, making them easier for the reader to detect.

4.4.4. The System Block

This block of the specification text contains all the

information which is external to the models and all the information which is common to the models. For example, it will include the definitions of all the valid message names, definitions of any common data types and details of the interconnections between the interfaces of the models. Its role is therefore purely supportive, and it contains no description of any part of the behaviour of the system.

4.4.5. The Models

Each model is represented by a block of text which contains:

(a) further definitions of names, but not of messages,

(b) a statement of the interfaces of the model, categorised into inputs, outputs and bothway (bidirectional) interfaces,

(c) statements defining the behaviour of the model, in the form described in 4.4.7 below,

(d) any operations used in describing the behaviour (see 4.4.9 below).

Names and operations defined inside a model are private to that model, in that they cannot be referenced from the system block or another model. This is a necessary constraint to achieve the correct form of "black box" specification, as described in Chapter 3.6.1.

4.4.6. Definition of Names and Messages

Although the uses of names (i.e. data types or stored values) and messages differ, the format of their definitions has been kept the same for simplicity. Hence the word "name" will be used throughout the remainder of this section, but the comments apply equally to messages. A name can be defined in one of two ways:

(a) as an instance of a defined data type, e.g.

```
COUNT : integer
```

which defines "count" to be of type "integer", or:

```
WEEKDAY : { monday, tuesday,
                wednesday, thursday, friday }
```

where the data type has been replaced by a list of permitted (constant) values,

(b) as a structure, consisting of a tree of elements; this form uses paragraph numbers to organise the structure, as in the following example:

```
NAME is
    [1] INITIALS is
        [1.1] INITIAL_1 : character
        [1.2] INITIAL_2 : character
    [2] SURNAME : string of character
    [3] TITLE : { mr, ms }
```

Note that, in the case of a message, the elements of the structure represent the information content of the message. It is also possible to give any name or element of a name any number of subscripts, so that it acts like a multi-dimensional array.

4.4.7. Behaviour and Rules

The basic description of the behaviour of a model consists of its responses to the stimuli which it can receive; this appears in ASL as a series of statements of the general form:

"on" STIMULUS "then" RESPONSE

where STIMULUS is a pattern for a received message (see Section 4.4.8) and a RESPONSE can be a call to an operation (see Section 4.4.9) or the sending of a message. The forms "start sending" MESSAGE and "stop sending" MESSAGE are provided for those cases where a message is to be sent continuously for a period. The above portion of the syntax of ASL is given in a form of BNF, which is explained in detail in Appendix B; for the examples in this chapter it is sufficient to note that symbols surrounded by quote marks (" ") are part of the language, whilst names in capitals represent parts where the specification writer substitutes details of the system concerned.

In addition to these simple stimulus-response statements, it is also possible to introduce rules which act as general constraints or monitor for exception conditions. These take the form:

"whenever" CONDITION "then" RESPONSE

where a CONDITION is some test on past and current messages and stored information; if the CONDITION becomes true, then the RESPONSE will occur. As with the simple stimulus-response statements, rules may contain alternatives. Used correctly, rules provide a powerful

means for expressing behaviour in a concise and comprehensible way.

4.4.8. Pattern-matching

On receiving a message, a model usually needs to examine its contents in order to determine the appropriate response. Hewitt (Hewitt, 1977) demonstrated how this could be achieved in an elegant manner by the use of pattern-matching, and made this one of the main features of his ACTORS language. ASL includes a simple variant of this idea, as demonstrated in the following example:

```
on ?x via input_line then .....
```

The question mark is used as a prefix to the variable name, 'x', to indicate that this is a pattern-matching variable, and whatever message is received will be associated with the name, 'x'. Thus, in the remainder of the statement, it is possible to refer back to the message as 'x' rather than as 'the message just received via input_line'. Pattern-matching can also be used in conjunction with another part of the language to produce extremely concise definitions of functions, as described in Section 4.4.9.

4.4.9. Definition of Common Operations

In ASL, an "operation" represents a general method of providing a shorthand for repeated behaviour. Unlike the concept of a function in mathematics, it does not have

any restriction on the number of arguments or on the number of results to be returned. Hence it is permissible to have an ASL operation with no arguments which returns no result. The general form of a operation definition is:

```
"operation" OPERATION_NAME
        "(" ARGUMENTS "-->" RESULTS ")"
        "is" SEQUENCE_OF_STATEMENTS.
```

For example:

```
[5] operation SQUARE_ROOT(x,t --> y) is
    [5.1] X,T,Y : decimal
    [5.2] y is ?z where abs(z*z - x) <= t
```

which also demonstrates the use of a pattern-matching variable (?z) to achieve a brief, non-algorithmic definition of the square-root function in terms of the inverse operation, squaring (represented as multiplication, z*z). Note that 't' is the required accuracy of the answer.

4.4.10. Incompleteness

ASL permits three kinds of incompleteness, covering information which is not yet available, information which will not become known, and also a "don't care" value. The word "undefined" is used to indicate that information is not at present available, but will become so later, while "unknown" is reserved for those cases where more detailed information is not expected to become available. Thus it is possible to create an outline version of a specification with some elements of the behaviour, contents of messages or operations left "undefined", and then to add

the missing information as it becomes available. In this way the specification writer is not forced to wait for the total information before starting to write a formal specification, but any areas which are incomplete are positively identified as such in the document.

4.5. The Formal Definition of ASL

4.5.1. Introduction

The preceding sections of this chapter have introduced ASL informally, but it is essential that the language is defined formally, as was pointed out in Chapter 2.2.12. This requires that the syntax (both context-free and context-sensitive) and semantics are themselves defined in some formal language. The following sections 4.5.2 to 4.5.8 provide an introduction to the methods which have been used to provide these definitions; the formal definitions themselves appear as Appendices B and C.

4.5.2. The Context-free Syntax

The context-free syntax of a language provides a method for identifying those sequences of characters which are well-formed statements in the language. Thus, it defines not only those statements which have a valid meaning in the language, but a much larger class of statements. Context-sensitive and semantic rules are

therefore required to identify from this class those which are meaningful.

Context-free syntax definitions are normally given as a set of productions which identify in a top-down fashion the permitted construction of statements from basic words and symbols. Although the syntax analyser which has been used in the trials of the language (see Chapter 5.6 and Chapter 6) takes in such syntax productions, the format of these makes them difficult to understand. The definition of ASL in Appendix B is therefore written in a variant of Backus-Naur Form (BNF) suggested by Wirth (Wirth(b), 1977), which results in a clearer, more concise definition.

4.5.3. Context-sensitive Rules

The BNF syntax definition of ASL can be used to detect incorrectly formed statements, but is not sufficient to identify any incorrect usage of names. This is because, although it is possible to identify that a particular word is a name without making any reference to more than one statement in a specification, the permitted usage of the name depends upon its definition and this is usually in another statement. There are two checks which must be applied to each occurence of a name in a specification:

(a) that the name has been defined in the appropriate place,

(b) that the name is being used in accordance with its

definition.

These two checks are usually known as "scope checking" and "type checking" respectively, and they are discussed in more detail in the next two sections. Due to the lack of restrictions on the order of statements in an ASL specification (see Section 4.3.2), these checks cannot be performed at the same time as the context-free syntax analysis as they require all the definitions to have previously been identified.

4.5.4. Scope of Names

The combination of block structure and paragraph numbering in ASL results in a simple definition of the scope of any variable name. The scope of a name is the part of the specification (i.e. blocks or paragraphs) in which it is valid to make reference to that name because it has been defined in that part of the specification. These rules are:

(a) names defined in the system block may be used anywhere in the specification,

(b) names defined inside a model cannot be used outside that model,

(c) a name defined in paragraph [x] is available throughout the model which contains the statement.

(d) a name defined in paragraph [x1.x2. ... xn] can be used in any paragraph or sub-paragraph commencing [x1.x2...x(n-1)...].

If a name is mentioned outside its valid scope, it is as-

sumed to be an occurence of a different name which has not been defined; this is treated as an error.

4.5.5. Type Checking

Type checking involves ensuring that, given an expression such as "a + b", both "a" and "b" represent values which can be subjected to the operation of addition; for example, it is assumed to be impossible to add a number directly to a string of characters. For ASL, it must also be ensured that messages are sent only via defined interfaces of the model which is doing the sending, and that conditions in rules (see Section 4.4.7) do evaluate to "true" or "false". The type checking rules are therefore of a similar format to the syntax rules, but for each position in the language which can be occupied by a name they must identify the appropriate data type. (To allow for statements in ASL where there are fewer constraints upon the data types, it is necessary to use the additional data types "void" and "any").

Unlike the use of BNF for context-free syntax, there does not appear to be any standardised method of defining type-checking rules. The rules for ASL, which appear in Appendix B.4, are therefore in the format used by Davie & Morrison (Davie & Morrison, 1981); this was chosen for its simplicity and clarity. The rule format is explained in Appendix B.3.

4.5.6. Semantic Definition

A semantic definition of a language provides the well-formed statements of that language with meaning, by relating those statements to some well-understood mechanism or model (in the mathematical sense of 'model'). Without such a definition, the language is merely sequences of words, open to any interpretation which a reader may wish to impose upon it. Even with such a model, formal proof procedures based upon it may still fall into the category of NP-complete problems. As with the type checking rules, there is no commonly-agreed form for semantic definition, but there are two distinct types:

(a) operational semantics, which define the language in terms of the results obtained when a program in the language is processed by some particular implementation of the language compiler,

(b) abstract semantics, which relate the language to some well-defined mathematical model, independent from any implementation of compilers or other tools.

As ASL is not a programming language, and is not expected to have a compiler, it will not be possible to define its semantics operationally. This probably is advantageous, as abstract definitions appear to be both simpler and more useful (Wirth(a), 1977). The semantic model used for the definition of ASL is explained in 4.5.7 below, with the treatment of timing information being covered in 4.5.8.

4.5.7. The Semantic Model

Marcotty and Ledgard (Marcotty & Ledgard, 1976) review a number of semantic models which have been used in the definition of programming languages, but these all have a strong algorithmic flavour which makes them unsuitable for use on ASL. It was therefore decided to use an alternative model, Petri nets, which has been used in the definition of the Epsilon simulation language (Jensen et al, 1979). Due to the expressive power of ASL, and thus the complexity of the resulting nets, it was found necessary to use a more expressive form of net, the Predicate/Transition net (Genrich et al, 1980), in place of that used for Epsilon. Rather than providing a re-expression of Predicate/Transistion net theory in a form which corresponds to the structure of ASL, the semantic definition takes the form of a set of rules for the conversion of ASL statements into a net. This translation is undertaken in stages:

(a) expansion of abbreviated forms (e.g. lists, arrays and structures) into individual items,

(b) unfolding of alternatives in behaviour, to produce an extended list of simple stimulus-response statements,

(c) translation of each stimulus-response statement into the equivalent net fragment,

(d) connection of the fragments into one single net, representing the whole system.

The detailed translation scheme is complex, and is therefore explained in Appendix C.

4.5.8. Time

Petri net models do not provide a direct representation for measured time, instead restricting themselves to the treatment of sequences of events (Peterson, 1981). This is not sufficient to represent the timing information in ASL; so it was necessary to devise an extension to the model to accomodate the additional information. As discussed in Chapter 3.7, the timing model necessary for adequate system specification in not as complex as that for the detailed design of hardware, for example. The Time Petri Net (TPN) model of Merlin (Merlin, 1974), which adds minimum and maximum firing times to the transitions in the Petri net, is therefore adequate for this purpose.

The arrival time of each message appears as an extra element in the tuples (sets of values; equivalent to the contents of a message in ASL) associated with tokens in the Predicate/Transition net model, and this time value is altered by the firing of a transition. One extra transition must also be added to the net to represent the observer (see Chapter 3.5.2), as this is the sole absolute time reference point. As a result of these additions, time within the model of an ASL specification is not continuous, but will always be represented by increasing values of the time attributes of tokens (messages).

4.6. Summary

In this chapter the detailed design of ASL has been described, showing how this was based upon the principles laid down in Chapter 3. This has shown how the features of the language are intended to satisfy the requirements listed in Chapter 2.2.13, and has indicated how the appearance of the language has been biased towards its intended audience. The formal definition of the language has been outlined, and in Chapter 5 it will be shown how this permits a wide range of computer-based support facilities. To provide further demonstration of the points made in this chapter, a complete example specification appears in Appendix E, together with an introductory explanation of the system in English.

CHAPTER 5

LANGUAGE SUPPORT FACILITIES

5.1. Introduction

In Chapter 2 mention was made of the advantages of restricting a specification language to the type of simple, context-free syntax found in computer programming languages. In this chapter those advantages are presented in more detail, in the form of a description of the type of computer-based facilities which can be provided to support the specification writer. These facilities should be provided as a single integrated system for the preparation of specifications, as this will allow them to be used in any combination and sequence; the alternative of enforcing a particular sequence would be in direct conflict with the aim of capturing the specification information as it becomes available (see Chapter 3.10). However, in order to provide some structure to this chapter, the facilities have been divided into four categories on the basis of their purpose; these are listed in the following table.

Section	Content
5.2	Checking, this being the application of self-consistency checks to the specification text.
5.3	Changes, and controlling them.
5.4	Validation, which is the process of ensuring as far as possible that the specification captures the intentions of the customer.
5.5	Verification, where the design is shown to fulfill the specification.

Some of the static checking facilities (those covered in Section 5.2.2) were implemented as explained in Section 5.6, so that limited support was available for trial uses of ASL. Provision of the remaining facilities is discussed in Chapter 8 as part of the proposals for further work.

5.2. Checking

5.2.1. The Types of Checking

The checking of a specification can be considered to consist of two parts:

(a) static checks, being those concerned with ensuring the self-consistency of the specification as a piece of text,

(b) dynamic checks, which attempt to detect inconsistencies in the behaviour described by the specification.

These are covered in Sections 5.2.2 and 5.2.3 respectively.

5.2.2. Static Checking

Many of the static checks which can be performed on ASL are identical to those applied in the compilation of programming languages such as Pascal (Jensen & Wirth, 1975). Appropriate techniques and tools for such a checking system have been widely published (e.g. Aho & Ullman, 1977; Johnson, 1979; Simpson, 1969). The main stages of static checking are as follows.

(a) Syntax analysis, to ensure that the text conforms to the syntax definition of the language. Although a syntactically correct specification may still be meaningless at the semantic level, this is a necessary first stage in the checking. Simple syntax errors may result in extensive and useless lists of "faults" being detected by the remaining stages of checking.

(b) Redundancy and completeness checks, which ensure that every name which has been defined (i.e. appears in block capitals) in the specification is also used (i.e. appears in lower case letters) within the appropriate scope (see Chapter 4.5.4), and that every name which has been used was also defined. The "completeness" which these checks ensure is not equivalent to a demonstration that the specification includes all the customer's requirements (see 5.4 below on validation). They will fail to detect the situation where all information relating to some required feature has been omitted from the

specification.

(c) Consistency checks, to ensure that every name is used in accordance with its definition and in the same fashion throughout the specification. As examples, every interface must be connected to another interface of an appropriate type (e.g. an output cannot be connected to another output), and arithmetic expressions must be constructed from conformable types (e.g. no attempts are made to add character data to integers).

The printed output from the static checker could include a formatted listing of the specification text, any appropriate error messages, a listing of any items which remain undefined, and a cross-reference listing which indicates all the places in the text where each name appears.

5.2.3. Dynamic Checking

ASL permits the specification of complex behaviour in a non-algorithmic manner (see Chapter 4.4), and the power of the language is such that it is possible to specify behaviour which is impossible to achieve. It is therefore essential that the checking facilities provide some analysis of the dynamic behaviour which is implied by the specification text, even if this can only detect the most severe errors or point to possible problems. Although there are a number of techniques for dynamic checking (see (DoI(a), 1981) for mention of some), none appear to

offer comprehensive analysis. Three techniques which conform to the model of systems described in Chapter 3 are as follows.

(a) Exhaustive simulation against test cases. This involves considerable human resources in the preparation of test cases and the evaluation of results as well as large amounts of computer time; also, it only demonstrates the absence of errors for the test cases. It is therefore not an acceptable method of checking, although it may be useful for other reasons (see section 5.4 on validation).

(b) Petri net analysis. If an ASL specification were converted into a Petri net, then there are known methods to check for deadlock, conflict and reachability. Unfortunately, these may not be satisfactory for analysing large specifications due to the computational resources and time required to produce the results. As with simulation, it may be necessary to identify a probabalistic approach which reduces the amount of computation required at the expense of introducing some risk of inaccuracy in the results.

However, standard Petri nets do not have a representation for the time duration of events; so they can indicate the existence of a problem when the time delays are actually sufficient to ensure that this does not occur. Merlin (Merlin, 1974) presented an

extension of Petri nets which include time delays, but the tools to analyse Time Petri Nets would have to be developed from this theory.

(c) Flow Algebra. The Calculus of Communicating Systems (Milner, 1980) is an algebraic approach to systems analysis which appears to be of similar power to Petri Nets. It has the additional advantage of having simple, algebraic rules for combining the behaviour descriptions of multiple systems or sub-systems. Its disadvantage is that it is a relatively new notation, so that there are no readily-available tools to perform the analysis. Also, like Petri Nets, it has no representation for time delay.

As a form of Petri net is being used as the semantic model for ASL (see Chapter 4.5.7), the adoption of the same model for dynamic checking is likely to minimise the amount of support software to be developed.

5.3. Changes

5.3.1. General

The need to make changes to a large specification is inevitable (Lehman, 1981); so it is essential that the specification writer receives sufficient support in:

(a) making the alterations correctly,

(b) retaining the history of changes to the document, including the reasons for them.

The alternative to this is the continuation of existing

practice, where documents are often not updated (or only updated infrequently) because of the difficulty of incorporating changes (Brooks, 1975).

5.3.2. Introducing Changes

Assuming that a specification is consistent (see 5.2.2) and has been validated (see 5.4), it is important to ensure as far as possible that the incorporation of an amendment does not introduce errors. The minimum requirement of the support system is therefore that it should make the author of the change aware of all the places in the specification which might be affected. This can be done by the provision of a cross-reference listing, as mentioned in section 5.2.2, leaving the author of the change to investigate which parts of the specification must be modified. This type of manual alterations has two major disadvantages:

(a) the whole specification must be re-submitted for checking, so that error messages may be produced for faults which existed before the amendment and are not due to it,

(b) checking takes place after the specification has been modified (i.e. a new issue has been created), making it more difficult to reverse the change if this becomes necessary due to any inconsistencies which it creates.

An improved system, providing interactive assistance of the type suggested by Sandewall (Sandewall, 1978) and

found in the Designer/Verifier's Assistant (Moriconi, 1979) and INTERLISP (Teitelman, 1978), would:

(a) maintain a list of all occurrences of names affected by the change, and prompt the user to make a positive statement as to the effect of the change on each one,

(b) recheck only those portions of the specification which have been changed, as they are changed,

(c) await the completion of the change (i.e. the exhaustion of the list of occurrences of affected names and the removal of any errors introduced with the change) before creating a new issue of the specification document, unless the user specifically requests that a new issue be created regardless of any outstanding errors.

Such a facility would greatly reduce the tedium of introducing amendments into large specification documents, and the consequent difficulties in ensuring that project staff are aware of the latest requirements. Provision of such facilities is discussed in Chapter 8.

5.3.3. The History of Change

Although many automated documentation support systems provide facilities for creating new issues (e.g. the PSL/PSA system (SDL, 1980)), few make any attempt to highlight the differences between adjacent issues or to record the reasons for the changes. The identification of statements which have been changed (perhaps by a vertical

line in the margins of the document) greatly assists the reader. A note of the reason for the change may be essential later when the resulting additional costs have to be apportioned between the customer and supplier. To provide these features in a support system, it is necessary to treat the specification not as a uniform sequential block of text but as a set of relations which can be held in a database. Changes to the specification are then viewed as updates to the database, with each update adding to, rather than overwriting, the earlier contents of the database.

The general structure of such a database will be a set of relations (Codd, 1970), each containing the date (or issue number) at which it was introduced, the date (or issue number) at which it was superseded by changed information, and either the reason for the change or a pointer to the reason. The original specification text for any particular date can then be recreated from the database by extracting all the relations which were valid at that date and reassembling these into text form. This method of handling changes to the specification over time is analagous to the handling of messages in ASL (see Chapter 3.8). It is also one of the major features of some recent proposals for support facilities for computer programmers, such as the Ada Programming Support Environment (DoI(b), 1981).

5.4. Validation

5.4.1. The Aims of Validation

Validation is the process of ensuring as far as possible that the specification correctly captures the customer's intentions. As there is no formal statement of the requirements other than the specification itself, it is not possible to use verification techniques (see section 5.5) which involve the rigorous comparison of two formal statements. Validation is a much weaker process than verification, and involves the presentation of the specification to the customer in an attempt to elucidate any discrepancies between it and the customer's mental model; this is an ill-defined process, with no guarantee that it will identify all the discrepancies. One of the main problems in validation is that of ambiguity, in that the customer and supplier may make different interpretations of the same statement. The use of a restricted, well-defined language such as ASL makes a large contribution to overcoming this problem.

A possible disadvantage of using ASL (or any other formal specification language) is that it may not be acceptable to the customer, perhaps because of the overhead of training large numbers of staff to read specifications written in it. In these circumstances, the specification would have to be converted into a format which is acceptable to the customer, as it is essential that the specification is approved before design commences (Cohen &

Burns, 1978). In the remainder of this section a number of possible conversions are discussed.

5.4.2. Manual Translation into English

One possibility is to use technical writers to translate the formal specification into English for presentation to the customer. It may be that this would lead to a clearer document in English, as it would be derived from the unambiguous, formal document. Such translation is however a labour-intensive, and therefore costly, process which could introduce errors. Thus some form of automatic translation would be preferable.

5.4.3. Automatic Translation

Given the restricted grammar of ASL, automatic translation into other similarly restricted notations is possible, although it may involve the development of some large computer programs. Translation into a stilted form of English is also feasible, if more difficult. The two most promising alternatives are discussed below.

(a) CCITT SDL (CCITT, 1980). The CCITT Specification and Description Language has been adopted as a standard by the telecommunications authorities of a significant number of countries. SDL is however a finite state machine model (see Chapter 2.10), and this makes the translation from ASL complex. In particular it may be impossible to generate satisfactory labels

for the system states automatically, as these are not represented at all an ASL specification.

(b) English. Systems such as MARGIE (Schank et al, 1973), GIST (Swartout, 1982) and SHRDLU (Winograd, 1972) have adequately demonstrated the generation of acceptable English sentences from a limited formal language, but large amounts of effort are required to develop such programs. Thus the feasibility of such translation has been noted, but there would have to be sufficient demand for the facility to justify the development costs.

5.4.4. Simulation

Where a system specification is very large, it may be unreasonable to expect the customer to identify all the nuances of behaviour implied by its contents, especially if the system is to include completely new features which are outside the customer's existing experience. One alternative to total reliance upon human interpretation of the text is the use of the specification as a simulation model, so that the customer can obtain insight by investigating the operation of the system on a number of test cases (Balzer & Goldman, 1979; Berild & Nachmens, 1978; Zurcher & Randell, 1968). As a specification is not intended to describe an efficient implementation (see Chapter 2.2.9), any simulation based upon it is likely to make inefficient use of computer time. However, the number of test cases should be small enough to make this ac-

ceptable when compared with the possible cost of an undetected error in the specification.

The development of a simple simulation system which would accept ASL specifications may not involve a large amount of programming (Lindstrom & Skansholm, 1981). There are however a number of features within ASL which may prevent effective simulation, as follows.

(a) Undefined items. The language was designed to permit the incremental creation of specifications; thus, at any point in time there may be large portions of a specification left "undefined" (see Chapter 3.10). The simulation system would either have to reject attempts to perform simulations on any specification with any undefined items, or be capable of identifying these and requesting the appropriate information from the human operator as required during the simulation run.

(b) Non-determinate behaviour (see Chapter 3.10). The "don't care" values and lists of possible alternative actions in ASL (see Chapter 4.3) represent non-deterministic choices. The simulation system would either have to interpret these as a request to some random selection mechanism, or interact with the human operator to obtain a decision.

(c) Specification of results, not methods. As a specification is intended to state the required results, ASL was designed to simplify the description of operations in terms of their input-output relationship alone (see Chapter 2.2.9). It is therefore likely

that a specification will contain one or more operations which are not described in terms of an algorithm, but merely as a statement of the conditions which apply to the output of the operation for any given input. Such an operation can be simulated by treating the conditions as a goal in an exhaustive search through all the values in the range of the function (the "British Museum Algorithm", (Balzer & Goldman, 1979)); this may be acceptable if the range of values to be searched is small. However, there are cases (such as numeric functions operating on the real numbers) where the range to be searched is so large that this method is unacceptable. The simulation system would have to be provided with some heuristic rules to detect such cases before starting a simulation run, so that the human operator can be warned that an "endless" search may be involved.

5.5. Verification

Verification is the process of ensuring, by formal reasoning, that a design or product does meet its specification (see e.g. Hantler & King, 1975). This is only possible in situations where both the specification and the design (or product) have been expressed in formal languages, and usually requires the assistance of computerised theorem-proving facilities (e.g. Boyer & Moore, 1979). However, verification has not been shown to be practicable for large systems (Lehman, 1981; Wirth(a),

1977); much recent work has therefore been directed at an alternative approach known as "transformational implementation" (e.g. Balzer, 1981). This attempts to ensure the correctness of the design by restricting the design process to a succession of small transformations of the original specification, each of which converts it into a slightly more algorithmic (and efficient) form. As transformational methods are relatively new, there have as yet been no demonstrations of the technique on large problems.

To convert an ASL specification into a suitable form for either of these methods would involve considerable manipulation of the specification. This is because ASL was designed to simplify the job of writing specifications; thus it allows such things as aggregation and generalisation (see Chapter 3.9) which are not directly expressible in the simple input languages used by existing theorem provers and transformational systems. The conversion would involve the dispersion of the higher-level constructs of ASL, so that equivalent conditions appeared in every individual item of behaviour. This is not excessively difficult, being similar to the translations involved in the semantic model of ASL (see Appendix C). However, the conversion program would itself require extensive validation as any errors in this would invalidate all subsequent verification using it.

5.6. The Demonstration Facilities

Given the range of computer-based facilities discussed in the previous sections of this chapter, one aim of the project was to provide some demonstration of the value of these. However, the amount of programming effort required to provide all of them was beyond the capacity of the project. An examination of results published by organisations using formal specification methods (e.g. Alford, 1977; Lattanzi, 1980) indicated that a large proportion of errors are likely to be trivial and may be detected by static cross-checking of the specification text. Thus, it was decided that the static checking facilities (see Section 5.2.2) would form a satisfactory part to demonstrate.

The methods used to provide these facilities are described in detail in Appendix D. In outline, a number of separate programs were developed to perform:

(a) syntax analysis,

(b) static cross-checking, and

(c) the production of a cross-reference list.

These operated as individual tools, rather than as an integrated set of facilities, as this maximised the number of tools which could be developed within the available time.

5.7. Summary

In this chapter a set of computer-based facilities have been described which would provide considerable support to specification writers. The intention is that these tools would make it practicable constantly to incorporate changes (as these become necessary) without creating the type of documentation control problems reported by Brooks (Brooks, 1975). A subset of these tools were implemented to provide a demonstration of their usefulness, whilst involving only a limited amount of programming; these have been used to support the trials of ASL, as reported in Chapter 6.

The use of a formal language, together with checking tools such as those described in 5.6 above, has been reported to have resulted in the detection before the start of design of over 50 percent of the errors in specifications for large software systems (Alford, 1977). The cost of correcting an error was also reported to increase by an order of magnitude if the error was not detected until the design was in progress. The possibility of obtaining a similar detection rate in telecommunications systems provides considerable justification for expenditure on checking tools.

CHAPTER 6

TRIALS AT GEC

6.1. Introduction

One of the terms of reference of the project (see Chapter 1.3.2, item (d)) was to introduce the chosen specification language into the Company. This activity was therefore combined with the need to obtain reactions to the design of ASL, giving a requirement for a number of initial trials of the language. These were each to involve the use of ASL to specify a relatively small system without requiring the involvement of the Company's customers; a total elapsed time per project of around one month was considered suitable. This limitation on the timescale was necessary because the effort expended on preparing the ASL specification formed an additional overhead on the projects concerned. As a consequence, only a small number of people were involved in the trials and recorded their opinions of ASL. These reactions do however form a basis upon which to modify the language and its computer-based support facilities before undertaking any large-scale implementation.

The main problem encountered in organising the trials

was that of identifying projects of a suitable size which were at an appropriate stage in their development. One of the projects chosen was subsequently delayed as a result of changes in marketing priorities, so that a replacement had to be found. Four trials have taken place, covering a range of types of system and a variety of levels of previous experience amongst the participants; these are reported in Sections 6.2 to 6.5 below. In each trial, the specification was reviewed by one or more people who had not been involved in its creation; both the writers and reviewers were then asked to complete a questionnaire to record their opinions. Their responses are discussed in Section 6.6.5.

The syntax analyser and the other static checking tools (see Chapter 5 and Appendix D) were used to aid the writers of the specifications. This acted as a check upon their comprehension of the language and highlighted any problems in this area. Difficulties encountered by the writers during the creation of the specifications were recorded as they occured, to provide further feedback. The difficulties and criticisms are discussed in Section 6.6, whilst more detail of these and the responses to the questionnaires appears in Appendix F.

6.2. Trial 1: The Data-rate Adaptor

The system used for this trial is part of a range of items being designed to support the Integrated Services Digital Network (ISDN), which will extend the use of

digitally-coded signalling from telephone exchanges right up to the subscribers' equipment. The particular item being specified in the trial is a data-rate adaptor, which takes in digitally-encoded data (e.g. from computer equipment), together with a digital carrier waveform at a higher pulse rate, and encodes the data onto the carrier. It also performs certain detailed modifications to the bit-stream which it outputs, such as inserting check digits. The intention is that this system, which has previously been implemented using standard integrated circuits, will be re-designed as a VLSI device.

No official specification for the system existed at the start of the trial; there were a number of unofficial documents which had been created for the purpose of explaining the system to interested parties, but these related to the larger entity of which the data-rate adaptor is only part. The personnel involved in the trial were all hardware engineers, and the individual who wrote the ASL specification had no previous experience of specification languages and limited experience of computer programming languages. This lack of appropriate background experience meant that training in the use of ASL took 2 weeks on a one student-one tutor basis. A simple guide to the construction of ASL specifications was created from the material used in this training period (Blackledge(b), 1982).

The use of ASL did not result in the identification of any errors which had previously been undetected; this is not unexpected as an implementation of the system

already existed. One of the reviewers did, however, comment that the structure of the specification had suggested an alternative design approach which had not previously been considered. Amongst a number of difficulties which arose during the production of this specification, only one reflected a significant failing in ASL, although even this one did not make it impossible to specify the required behaviour. This was in the handling of long sequences of related messages, which is common in the data-rate adaptor as it is handling sequences of bits representing characters. ASL requires that the specification mentions each bit individually, and provides no convenient method of referring to the sequence as a whole; this produces a large specification and makes this part of the behaviour harder to comprehend.

6.3. Trial 2: A Disk Checking System

The specification written in this trial was for a computer program to help in the commissioning and maintenance of CAT 5 and CAT 6 test equipment. CAT testers were developed by the Company to perform automatic diagnostic testing of electronic circuit boards. Because of the size of the test programs required for printed circuit boards containing up to 80 integrated circuits, the CAT testers use fixed-head disks to provide large amounts of magnetic storage. The first testers fitted with such disks suffered from a considerable number of problems,

apparently caused by unreliability of the disks. Further investigation, however, revealed that many of the problems were due to faults in the hardware which interfaced the disks to the testers. Some method of exercising and checking the disks was therefore required, to assist in locating faults. Three programs were written to aid in this checking, but these did not cover all the functions of the disk unit; also, they were unable to continue with the remaining tests after finding the first fault.

A new single utility to perform comprehensive checks on the disks was specified in ASL; this was a completely new specification as no other documentation had been written about this new program. The writer of the specification had not previously used a formal specification language, but had experience of writing specifications in English. No serious problems were encountered during the preparation of the specification. However, as this was the first of the trials to occur, many small faults were found in the definition of ASL; these are discussed in Section 6.6. As the specifier was filling the roles of "customer" and "supplier" (see Chapter 1.2.2), no major errors or omissions were detected when the specification was input to the static checking facilities or when it was reviewed. The specifier did however report that the use of ASL had forced the resolution of a number of minor inconsistencies and omissions during the construction of the specification.

6.4. Trial 3: R2 Signalling System

The R2 protocol for signalling between telephone exchanges is used in a number of countries (e.g. China, India) which are potential export markets for the System X family of exchanges. It uses pairs of audio tones, selected from four possible frequencies, to encode the digits 0 to 9 and certain control signals. Specifications already existed in English (Galvin, 1981), in message sequence charts (EODST, 1981) and in the finite state specification language, FSIS (BTS, 1981 ; Taylor, 1981), giving an opportunity to compare ASL with the type of specifications already used in the telecommunications industry.

Unfortunately, other target dates for the project were too pressing to permit the personnel to invest two weeks in learning ASL and then further time rewriting the specification. It was therefore decided that the specification would be produced by the designer of ASL (the author) and then subjected to review by a member of the project. The review would then provide a comparison of specifications written in four languages, undertaken by someone who had not written any one of them; criticisms arising would therefore relate to the general comprehensibility of the four forms.

During the preparation of the specification, it became apparent that a number of elements of the behaviour were not sufficiently well-defined. As an example, no information had been provided on how to recognise when

enough digits had been received to complete the telephone number being called. On checking with the project personnel it was found that these elements were ill-defined, and that this had already been recognised. However, both the FSIS and English-language specifications give no indication of this incompleteness. In contrast, the ASL specification demanded the use of the word "undefined" in each of the appropriate positions, making the incompleteness explicit.

6.5. Trial 4: Part of an Operating System

The Telecommunications Research Laboratory at the GEC Hirst Research Centre have been working for a number of years on the development of a distributed system using a number of microprocessors (Nissen & Geiger, 1979). The aim of this work has been to produce a flexible system in which both the processing power and the operating system are distributed over the variable number of microprocessors involved. One central feature of this system is therefore the organisation of the flow of messages between the various processors, as there is no fixed allocation of tasks to processors; this job is undertaken by software modules known as "route-handlers".

This route-handler module was selected as the system to be specified in ASL as it is not too large to be specified in a short period, but does include some reasonably complex behaviour. The main difference between this trial and the others is that the research engineer who

wrote the specification had previously made use of other formal specification languages. These languages had included ones based upon more mathematical notation than ASL (e.g. Jones(a), 1980), so comments from this trial would provide some evaluation of the comprehensibility of ASL relative to these.

One problem which arose during the preparation of the specification was a consequence of the writer's previous experience, which caused him to misunderstand the object-oriented view embodied in ASL. The initial version of the specification treated a model as a mathematical function, and attempted to call it recursively. Although this is a technique used in many other specification languages, it is meaningless in ASL where the only way of communicating with a model is by the transmission of messages. The occurrence of this difficulty does suggest that the restrictions imposed by the object-oriented view had not been explained sufficiently well in the language reference manual. The trial produced a number of suggestions for minor improvements to the language syntax, and also the identification of four points where the language definition was incomplete; these are discussed briefly in Section 6.6, and listed in Appendix F.

6.6. Criticisms and Comments

6.6.1. Sources of Comments

As mentioned in Section 6.1 above, problems encountered during the preparation of the specifications were recorded as they occurred. Then, after each specification had been completed, it was reviewed by one or more people and the immediate comments and criticisms again recorded. Finally, each participant was asked to complete a questionnaire so that any further thoughts and general opinions were captured. Sections 6.6.2 to 6.6.4 below cover the problems which arose during the trials, and then Section 6.6.5 covers the responses to the questionnaire. All the results which are discussed relate to the design of ASL. The documentation used to support the trials (Blackledge(a), 1982; Blackledge(b), 1982) was the subject of some criticism, but this is not directly relevant to the evaluation of the language.

6.6.2. Unintentional Inconsistencies

As was pointed out in Chapter 4.3.1, there is very little published material which gives constructive advice on the process of language design. ASL was therefore the result of a number of attempts at such a design; each attempt was subjected to criticism, which formed the basis for the next attempt. During the trials it became apparent that the syntax definition still failed to capture

the author's intentions in all cases, as in the following examples.

(a) The syntax production for PREFIX allowed references to the "first" and "last" messages matching a particular pattern, but not to the intermediate ones.

(b) A bothway interface might receive and send messages with the same name, but there was no way to select messages in one particular direction from the history of the model.

(c) Messages were forced to have at least one component in order to allow values to be assigned to them; hence a message with one component effectively had two names where one would have sufficed.

(d) Pattern-matching variables could not be used in place of an interface name, and anonymous pattern-matching variables (i.e. "?" with no name following it) could not be used in the place of a stimulus.

There were also a number of other similar items, all of which are listed in Appendix F.1.2. These points were all treated as errors in the syntax definition, and therefore corrected immmediately. The full syntax definitions in Appendix B show only the corrected forms.

6.6.3. Simple Extensions

The specification writers taking part in the trials made proposals for extensions to the language which they felt would assist them in their task. Those extensions which were simple and also consistent with the concepts

in ASL were added into the language as they arose. Some of these are described below, whilst those which were not incorporated are discussed in Section 6.6.4.

(a) The ability to use paragraph numbers with no paragraph body (except perhaps a comment) provides a way to organise the text within a single model. This use of paragraphs as a means of introducing headings for sections of the text is totally consistent with the intentions of the paragraph numbering scheme (see Chapter 4.3.3).

(b) Common operation definitions can be placed within the system block, rather than having to appear within every model which uses them.

(c) Sequences are necessary within the RESPONSE part of a single behaviour statement. It is not practicable to express all sequence constraints as general rules, and there are many cases where arbitrary sequencing is insufficient.

The complete list of these extensions appears in Appendix F.1.3, and they are all included in the syntax definitions in Appendix B.

6.6.4. Further Possible Extensions

A number of other points were raised during the trials, but were not seen as simple alterations to the language syntax and have therefore not been incorporated. These are listed in Appendix F.1.4 and F.1.5. The main reason why none were incorporated into the language was

that it still proved possible to complete the trial specifications without these features in the language, whilst the time taken to extend the semantic model (see Chapter 4.5.7) to include them would have delayed the completion of the trials.

6.6.5. Responses to the Questionnaire

With only one exception, all the participants in the trials completed the questionnaire, so providing a record of their comments and opinions after the completion of their role in the exercise. The one exception was the writer of the specification for the R2 signalling system; as this role was filled by the designer of ASL (the author), this would not have provided any additional information. The design of the questionnaire is covered in more detail in Appendix F, but its intention was to get the participants to record their views on as much of the language as possible. To this end, it was based upon a sequence of multiple-choice questions, but with room for free-form comments after each question. Additionally, a number of other questions were introduced which were intended to elicit more general comments.

Due to the small number of participants (eight in all) and the large number of uncontrolled variables in the trials, the volume of data from the questionnaires was insufficient to submit to normal parametric statistical analysis. Analysis was further complicated by the large proportion of the results which appeared as free-

form comments; however, this was a direct consequence of attempting to ensure that the participants gave the maximum amount of information in their replies. Despite this, there was a large degree of similarity in the content of many of these comments. The discussion of results which follows has therefore been based upon the number of participants who answered positively, negatively or neutrally to the questions about ASL (see Appendix F.2.2 for further details). This data was subjected to the Kolmogorov-Smirnov nonparametric test (Siegel, 1956), as this is suitable for small samples. The answers were then re-analysed on the basis of two factors which divided the participants into groups, to see if there was any significant correlation between the groupings and the opinions expressed. The factors used were the role played by the participant (writer or reader) and their previous experience of formal specification languages. Here the Fisher exact probability test (Siegel, 1956) was used, as this is a correlation test suitable for small samples.

Only one criticism of ASL was identified by these analyses, and this was only significant at the 90 percent level. This criticism was that, although the readers found the paragraph numbering helpful, the writers did not like the amount of repetitive writing which it involved. One writer had wished to insert an additional paragraph between two existing ones, and found that this necessitated a large amount of writing.

In no other case was there a sufficiently distinct pattern in the responses to form a sound basis for any crit-

icism of ASL. There were, however, seven items on which there was support for features of ASL (at the 90 percent level or better, taking all participants together), these being:

3.1(a), the block structure of an ASL specification,

3.1(b), the use of the system block for common information,

3.1(c), the "black box" view of models,

3.1(g), the form of definitions,

3.1(h), the method of describing behaviour,

3.1(i), the use of "unless" for alternatives,

3.1(q), "whenever" as a way of describing conditional actions.

Appendix F.2.2 also contains details of those responses which took the form of comments and therefore were not suitable for the above analysis. From these comments, two points are worthy of note. Firstly, the writers with no previous experience of formal specification languages found the specifications hard to write, but this seemed to be due to the need to be rigorous rather than to any features of ASL. Secondly, the use of "black box" models was seen to be an advantage of ASL, both in terms of the resulting style of the documents, and of the method of constructing specifications which it embodies.

6.7. Summary

One of the most important elements of the project was

seen to be the practical testing of the ideas which had been developed in Chapters 2, 3 and 4. Despite the problems in arranging for a number of such tests to be carried out in timescales to suit the project, four trial specifications were successfully produced. The participants displayed considerable interest and enthusiasm, and contributed a significant amount of constructive criticism of ASL. Many of the difficulties which arose during the trials can be traced back to deficiencies in the documentation used to train the participants, but some did relate to the design of the language. In the next chapter these results are used in the evaluation of the progress made by the project.

CHAPTER 7

EVALUATION

7.1. Introduction

The achievements of the project can be assessed on two bases, by making use of the information from Chapter 2 (the evaluation of other specification languages) and that from Chapter 6 (the trials within the Company). In Section 7.2, ASL is considered in relation to the selection criteria which were developed in Chapter 2; this produces a comparative evaluation of the language and identifies the degree of success in meeting the design criteria. Then, in Section 7.3, the reactions of the participants in the trials are discussed and some deficiencies of the language noted.

7.2. Comparative Evaluation

To provide a meaningful comparison it is necessary to evaluate ASL against the same criteria and in the same way as the languages reviewed in Chapter 2 and Appendix A. Thus, the comments which follow are shown against the same headings and identification letters as were used in

Chapter 2.2.13 and throughout the tables in Appendix A. It should also be noted that, under the categorisation made in Chapter 2.3, ASL is an event-triggered language.

Preliminaries:

Form: ASL has only a text form.

Computer/Manual: Some computer facilities exist.

Use: The language is not in regular use, but has been demonstrated on complete examples.

(a) Block or Paragraph Structuring.

An ASL specification is divided into blocks which represent models. Within each block, paragraph numbers may be used to provide appropriate structure, as explained in Chapter 4.3.3. This gives a greater structuring capability than in any of the other formal languages reviewed in Chapter 2.

(b) Generalisation

The language permits the definition of operations with any number of arguments and results, by treating these as a text "macro" (e.g. see Cole, 1980) rather than as strict mathematical functions. Properties which are common to a number of models, such as the format of messages, are "factored out" and only defined once in the system block. These provide comprehensive facilities for generalisation which are much more powerful than those in high-level programming languages such as Pascal, Ada or PL/I.

(c) Aggregation

Both messages and definitions can be hierarchical structures with any number of levels, and for definitions

this applies to both data types and instances of objects. It is not possible for one object to be an instance of more than one data type; however, no case has so far been encountered where this has been necessary.

(d) Separate Description of each Action

This is enforced by the language, in the shape of the:

"on" STIMULUS "then" RESPONSE.

form of statements which describe behaviour.

(e) Monitors

The ASL form of a monitor is:

"whenever" CONDITION "then" RESPONSE.

and monitor actions take priority over simple behaviour actions. Thus, a monitor can be used to represent behaviour under exceptional cicumstances, such as an overload.

(f) Historic and Descriptive Reference

ASL provides historic references by treating each interface of each model as an infinite buffer, and descriptive reference by allowing a message to be identified by the values of its contents. One important factor in making these facilities easy to use is pattern-matching:

```
e.g.    sum(all ?z)
            where (?y.size = z)
            and (y = coin via coinslot)
```

which provides temporary names for patterns and the selected values.

(g) Non-algorithmic

ASL does have a construct which can act as an assignment statement; however, without the other control state-

ments found in programming languages (e.g. loop statements and jumps), this is not sufficient to describe complex behaviour algorithmically. Thus, the design of the language makes it difficult to write algorithmic descriptions, whilst the use of pattern-matching and local definitions make non-algorithmic specification easy.

e.g. a simple definition of a square root might be

```
[5] operation SQUARE_ROOT(x, t --> y) is
        [5.1] X / THE INPUT : decimal
        [5.2] T / REQUIRED ACCURACY OF ANSWER
                : decimal
        [5.3] Y / THE ANSWER : decimal
        [5.4] y is ?z
                where abs(z*z - x) <= t
```

(h) Representation of Time Duration

Time delays can be represented in behaviour statements as:

"within" TIME_DELAY "of" STIMULUS "then" RESPONSE.

e.g.
```
within 1 second of lift_handset
            via subscriber_line(?x)
    then start sending dial_tone
            via subscriber_line(x)
```

and a timeout (i.e. a response to the non-arrival of a message within a fixed time period after an event) can be represented as a monitor:

"whenever" CONDITION "then" RESPONSE.

e.g. whenever 15 seconds after (start of dial_tone
via subscriber_line(?x)) then ....

so that the language provides all the timing facilities required for behavioural specifications.

(i) Recognition of Concurrency

Interfaces to models are treated as sequential channels, but any number of interfaces may be active at the same point in time. The language therefore allows concurrent activities to be described; however, apart from simply limiting the use of interfaces and local variables to be sequential, it does nothing to restrict the system behaviour to be safe. Thus, the onus of ensuring this is upon the specification writer.

(j) Acceptance of Fuzzy Values

ASL provides three features which address this area: the words "undefined", "unknown" and "dont care" can be used to indicate incomplete knowledge; actions can be specified non-deterministically, in terms of the desired result; some values (e.g. time delays) can be stated as ranges of acceptable values, rather than as single quantities. This does not cover the type of "fuzzy" values used in fuzzy logic (Gaines, 1976), but does provide facilities equal to those in any of the other specification languages reviewed in Chapter 2 (apart from English).

(k) Notational Redundancy

Due to the extensive use of words from the English language, as explained in Chapter 4.2, specifications written in ASL have a similar level of notational redun-

dancy to Pascal programs. However, the paragraph numbering in ASL provides a high level of perceptual recoding of the structure without producing verbose specifications.

(l) A Simple Syntax

The syntax of the language (which appears in Appendix B) has been shown to be suitable for recursive-descent syntax analysis, and is simpler than the syntax of many programming languages.

(m) A Semantic Model

The semantic model for ASL was described in Chapter 4.5.7, and is elaborated in Appendix C. Although this does not give a complete formal definition of the language from first principles (it relies upon the existing definitions of Predicate/Transition nets (Genrich et al, 1980)), it is sufficient to ensure an unambiguous definition of the language.

Table A.15 in Appendix A shows a comparison of ASL and the best language from each of the categories identified in Chapter 2.3. From this table it can be seen that the design of ASL has produced a language which does not have the failings noted in the other languages which were reviewed in Chapter 2.

7.3. Feedback from the Trials

7.3.1. The Significance of the Results

Although the comparative evaluation in Section 7.2

demonstrated that ASL had met its design criteria, it was also necessary to obtain confirmation of this from practical trials of the language (as reported in Chapter 6). This feedback provides a more detailed critique of the usability and comprehensibility of the language to its intended audience. As mentioned in Chapter 6, this information takes the form of both a record of problems which arose during the writing of the specifications, and the questionnaires which were completed by all the participants.

Statistical analysis of the results was restricted to simple nonparametric tests due to the composition of the sample; only a small number of people were involved, and it was not possible to select these to ensure a proper cross-section of the total audience. Analysis was further complicated by the impracticability of performing parallel trials with control groups using other specification methods. Such parallel trials are uncommon in experiments with languages (see e.g. Shiel, 1981; Weinberg, 1971) due to the number of additional participants and considerable extra cost involved. The small total number of participants also made it impracticable to test the questionnaire on a sample of the audience, as is normally suggested (e.g. Kornhauser & Sheatsley, 1965). However, the backgrounds of the people who did take part do between them cover many parts of the total audience:

(a) hardware engineers, with no previous experience of specification languages,

(b) systems engineers, with some experience of the use of finite state languages and message sequence charts for specifications,

(c) a research engineer who had previously used a number of formal specification methods, and

(d) other systems people, with previous experience in the use of programming languages.

There is sufficient general agreement in their responses to provide confidence in the acceptability of a formal language such as ASL to the wider audience. Although the trials were undertaken within a telecommunications company, only one (the R2 signalling system)was specific to that industry. The success in the other trials indicates the suitability of ASL for a wide range of information-processing systems.

The problems which did arise during the trials fall into three categories, those resulting from unintentional inconsistencies in the syntax definition of the language, those which required simple enhancements, and those which involved significant changes to it. These three were covered in Section 6.6, which also contained details of the questionnaire and the responses which it elicited. Section 7.3.2 below discusses the overall pattern of the results.

7.3.2. The Pattern of the Results

The results reported in Chapter 6.6 indicate that ASL was found to be satisfactory by the personnel participat-

ing in the trials. All of the specifications were completed to the level of detail which the participants deemed appropriate, and in every case the language was able to express the necessary behavioural description. The spread of responses to some questions in the questionnaire reflects the range of personal likes and dislikes of the participants; however, there was general support for almost all features of the language. There were a number of items where either the role played by a participant or any previous experience of specification languages seemed to be related to the opinions expressed, although only one was statistically significant (see Chapter 6.6.5). However, in no case was there general agreement that ASL was unsatisfactory. This provides practical evidence to support the major decisions taken in the design of the language, as described in Chapters 3 and 4.

Most of the minor difficulties which did arise during the preparation of the specifications were resolved immediately, in some cases by incorporating into the language improvements suggested by the specification writers. One feature which was reported to be extremely useful was the "black box" view, with the associated message-passing semantics. This was found to act as a strict discipline upon the writers, and formed a positive method for constructing the specification (as described in (Blackledge(b), 1982)).

CHAPTER 8

CONCLUSIONS

8.1. Achievements

The aims of the project were to identify a suitable formal specification language, and to introduce this into use within GEC Telecommunications Ltd.. This language must be a practical tool for the construction of specifications of large and complex systems; it must also be suitable for use by the existing staff of the Company after a limited amount of training.

Firstly, a comprehensive review of specification languages was undertaken. This involved the examination of the large number of existing languages. From the analysis of these and their use on small example problems, a set of criteria was developed by which they could be evaluated. The outcome of this review was that all the languages were found to be deficient, the most frequent failing being in the area of facilities for handling complexity. There was also a noticeable separation into two groups of languages, those using strict mathematical notation and those using more natural methods of expression; this was particularly relevant to the aim of

easy introduction into the Company.

As a consequence of the review, a new specification language was designed; this was called A Specification Language (ASL). Its main advantages over the other languages are as follows.

(a) Comprehensive block and paragraph structuring facilities, to provide perceptual cues to readers of the resulting specifications. The use of paragraph numbers to organise the text, although a common feature of documents written in English, was not a part of any of the specification languages reviewed.

(b) A simple view of systems as "black boxes" which communicate by passing messages. This provides both a suitable method for constructing specifications and a strong, although not total, check upon completeness.

(c) A simple model of time, sufficient to present the main performance requirements for the system.

(d) The recognition that incomplete information must be recorded, and that specifications are normally created incrementally.

(e) Features which strongly promote non-algorithmic specification, so that specifications do not contain unnecessary information about possible design decisions. These features include the ability to refer to information by its description, rather than by name, and unlimited access to the previous behaviour of the system.

(f) A simple syntax which is based upon the use of appro-

priate words and phrases rather than special symbols. A limited set of computer-based facilities was constructed to support the use of the language, in recognition of the practical problems involved in the creation, checking and maintenance of large specifications.

ASL was then used in the preparation of four specifications within the Company, in order to confirm its suitability and to identify the type of training material required for full implementation. These trials covered both hardware and software systems, and were not specific to the problems of the telecommunications industry. The participants in these trials represented a wide range of background experience. Thus, although the amount of data collected is small, it provides a high degree of confidence in the acceptability of the language within the Company.

A number of problems did arise during the trials, but none were sufficiently serious to prevent the specifications being written, and most were resolved immediately. As a result, a number of worthwhile improvements have been incorporated into the language. The participants in the trials also recorded their opinions of ASL in a questionnaire; their answers showed support for the main features of the language, and no significant criticisms.

The work reported here has therefore sucessfully achieved the aims of the project, although complete implementation of the new specification method in an organisation of the size concerned can be expected to spread over a number of years.

8.2. Outstanding Problems

Although there are no difficulties which prevent the use of ASL to gain the benefits outlined in Chapter 1.4.3, there are two problems worthy of note. These will make it difficult to obtain all the theoretically possible benefits of using a formal specification language, but do reflect areas which are still the subject of much current research.

(a) Verifiability.

In order to ensure that the language was suitable for practical use by existing personnel, its design was biased towards enhancing its expressive power. As a consequence, it is much more difficult to use ASL in the formal verification of the design of a system than to use one of the more restricted languages associated with current program-proving systems (e.g. Boyer & Moore, 1979). However, there are no reports of the routine use of formal verification methods on large systems, only on relatively small individual programs. Hence, ASL seems to reflect the only approach which is currently practicable for large systems.

(b) Time.

The simple model of time used in ASL specifications is adequate for conveying gross timing information and for "worst case" delay simulation. However, the language does not provide a convenient method for representing actions which occupy a small, finite amount of time. It can express the necessary constraints upon behaviour, but

only in a rather cumbersome and verbose manner. Thus, this is an area which requires an appropriate change to the syntax and to the semantic model. If this is not done, then writers will tend to omit these details from their specifications. Then it will not be possible to ensure that the system has no detailed timing problems by analysing the ASL specification.

As there is no practical solution to these problems at present, their existence is not sufficient reason to delay the introduction of ASL into the Company. Such delay would merely result in the loss of the financial benefits associated with formal methods, whilst not guaranteeing higher benefits in future.

8.3. Further Development

ASL has been demonstrated to be suitable for use within the Company, but the facilities and documentation developed to support the trials are not adequate for full-scale implementation.

(a) Enhancements to the Language

A number of useful extensions to the language were not incorporated merely to avoid delaying the trials. These items, which are listed in Appendix F.1.4, should therefore form the first step in the further development. The improvements to the documentation suggested below would then reflect these enhancements.

(b) Documentation.

One of the criticisms made by most of the participants in the trials was of the format and content of the Language Reference Manual (Blackledge(a), 1982). This document needs to be expanded and reorganised to take account of these criticisms.

(c) Training Material.

The existing introductory guide (Blackledge(b), 1982) only covers the initial information required by the writers of specifications; there is no equivalent document for people who will only read them. Both this and any guide developed for readers need to be supported by a progressive series of example specifications which cover more complex uses of the language. For the training of large numbers of people, it will also be necessary to produce a number of lecture courses for different levels of staff.

(d) Improved Support Facilities.

Support facilities, such as a simulation system which works from the ASL code, are required in order to detect as many specification errors as possible. The limited set of facilities which were produced to support the trials were only intended as prototypes, for demonstration purposes. A completely new support system is required, with an interface between the user and the tools, such that, as more facilities become available, they do not appear as something separate and different from the previous tools. One particular criticism which came from the writers of the specifications was that the paragraph

numbering scheme involved much repetitive writing and made the insertion of additional paragraphs very time-consuming. The provision of more sophisticated editing facilities may therefore be more important than was originally imagined.

Although it is possible that the use of ASL could continue without investment in the items listed above, it is unlikely that specification writers would find it a sufficiently rewarding tool. It was purposely designed to be suitable for processing by computer, but it also relies upon such support to provide many of the benefits to its users.

APPENDIX A

REVIEW OF SPECIFICATION LANGUAGES

1. General Layout

This appendix contains the detailed results of the evaluation of candidate specification languages. Because of the large number of languages to be presented, the information has been condensed into tabular form, using the coding explained in the following notes. There are fifteen tables, numbered A.1 to A.15, with the first fourteen corresponding to the fourteen language categories identified in Chapter 2.3, and the fifteenth being a summary table as explained in Section 4 of this appendix. Each table has the different languages displayed vertically in the left-hand column, and then sixteen further columns containing the results of the evaluation; the first three of these record general information (explained in Section A.2, below) and the remaining thirteen are for the criteria, (a) to (m), as listed in Chapter 2.2.13. The interpretation of the values placed in these thirteen columns appears in Section A.3, below.

2. Columns for General Information

Three columns have been included to give a general picture of the form and status of the language :

(a) Form (column headed "F")

Indicates the visible form of presentation:

T = text,

G = graphics,

X = tabular.

For the few languages which make use of multiple forms, more than one symbol appears in the table.

(b) Computer/Manual (column headed "C")

The reported status of computer support facilities, as these may not exist even though a language is suitable for computer processing:

M = only manual use; no computer facilities exist,

C = some computer facilities exist.

(c) Use (column headed "U")

Whether the language is in practical use anywhere:

P = proposal only,

E = has been used on complete examples,

A = in actual, regular use.

3. Values Used in the Assessment

In all cases except criterion (k), the failure of a language to provide any facility under a particular heading is denoted by a dash ("-"). Where numeric values are given, the magnitude of the number is not intended to in-

dicate an order of acceptability amongst the alternatives.

(a) Criterion (a), Block or Paragraph Structuring

1 = has at least the facility to divide a specification into blocks with headings or titles.

(b) Criterion (b), Generalisation

1 = a simple facility, allowing the writer to define common behaviour once (i.e. like "subroutines" in programming languages).

2 = comprehensive facilities for "factoring out" common properties of all kinds.

(c) Criterion (c), Aggregation

1 = permits the naming of collections of objects.

2 = permits one object to be a member of more than one named collection.

(d) Criterion (d), Separate Description of Each Action

1 = possible, although not enforced by the structure of the language.

2 = enforced.

(e) Criterion (e), Monitors

1 = exception conditions can be represented by monitors, separate from the description of individual actions.

(f) Criterion (f), Historic and Descriptive Reference

1 = historic reference only.

2 = descriptive reference only.

3 = historic and descriptive reference.

(g) Criterion (g), Non-algorithmic

1 = possible, although the language is mainly

algorithmic in design.

2 = the design of the language makes algorithmic description difficult, so non-algorithmic predominates.

(h) Criterion (h), Representation of Time Duration

1 = timeouts only.

2 = timeouts and time delays due to actions.

3 = clocked, synchronous action only.

(i) Criterion (i), Recognition of Concurrency

1 = yes.

(j) Criterion (j), Acceptance of Fuzzy Values

1 = allows values to be stated as ranges.

2 = allows fuzziness and incompleteness to be indicated as such.

(k) Criterion (k), Notational Redundancy

As this is not a yes/no choice, the languages have been compared to the level of redundancy represented by a programming language such as Pascal, and marked:

T = much terser than Pascal; this is taken to represent a very low level of redundancy, which is likely to prove difficult for readers,

A = average; approximately the same level of redundancy as Pascal, and likely to be readable and reasonably brief,

V = verbose; a much higher level of redundancy than in Pascal, likely to lead to large specifications.

(l) Criterion (l), a Simple Syntax

1 = known to be acceptable to standard programming language compiler techniques.

(m) Criterion (m), a Semantic Model

1 = operationally defined semantics (no theoretical model exists, but computer facilities have been developed to a stage which must effectively form a semantic model).

2 = a theoretical semantic model exists.

4. Summary Table

Table A.15 contains the language from each category which satisfies the largest number of the criteria (i.e. has the least number of "-"s); where more than one language satisfied the same number of criteria, an arbitrary choice was made. ASL, the new language, has also been included as the last item in the table. The purpose of this table is merely to indicate the degree to which the thirteen criteria, even though not a complete test, were successful in indicating deficiencies in the languages reviewed. This suggests that the list of criteria was sufficient for its purposes.

TABLE A.1 UNIVERSAL LANGUAGES

Language	F	C	U	Criteria a	b	c	d	e	f	g	h	i	j	k	l	m
APL (Jones & Kirk, 1980)	T	C	P	1	1	-	1	-	-	-	-	-	-	T	1	1
Decision Tables (Humby, 1973)	X	C	A	-	-	-	2	-	-	2	-	-	-	T	1	1
English	T	M	A	1	2	2	1	1	3	2	2	1	2	V	-	-
Pascal (Jensen & Wirth, 1975)	T	C	P	1	1	1	1	-	-	-	-	-	-	A	1	2
PDL (Caine & Gordon, 1975)	T	C	A	1	1	1	1	-	-	-	-	-	-	A	1	-
Prolog (Clocksin & Mellish, 1981)	T	C	P	-	1	-	-	-	2	2	-	-	-	T	1	2
SETL (Schwartz, 1973)	T	C	P	-	1	-	-	-	-	2	-	-	-	T	1	2

TABLE A.2 COMPUTER HARDWARE DESCRIPTION LANGUAGES

Language	F	C	U	Criteria a	b	c	d	e	f	g	h	i	j	k	l	m
AHPL (Hill & Peterson, 1973)	T	C	A	1	-	-	2	-	-	-	3	-	-	T	1	1
DDL (Duley & Dietmeyer, 1968)	T	C	A	1	-	-	2	-	-	-	3	-	-	T	1	1
HARTRAN (Bown, 1978)	T	C	A	1	-	-	2	-	-	-	3	-	-	T	1	1
ISPS (Bell & Newell, 1971)	T	C	A	-	-	-	-	-	-	-	-	-	-	T	1	1
TEGAS6 (Szygenda, 1980)	T	C	E	1	1	-	2	-	-	-	2	-	-	A	1	1

TABLE A.3 NEW PROGRAMMING LANGUAGES

Language	F	C	U	Criteria a	b	c	d	e	f	g	h	i	j	k	l	m
Ada (Ichbiah et al, 1979)	T	C	A	1	2	1	1	-	-	-	-	1	-	A	1	1
Ada Extension (ANNA) (Krieg-Bruckner & Luckham, 1980)	T	C	E	1	2	1	1	1	-	-	-	1	-	A	1	2
Alphard (Wulf et al, 1976)	T	C	A	1	2	1	1	-	-	-	-	-	-	A	1	1
Gamma (Falla, 1981)	T	C	A	1	2	1	1	-	-	-	-	-	-	A	1	1
Gypsy (Ambler & Good, 1977)	T	C	A	1	1	1	1	-	-	-	-	1	-	A	1	1

TABLE A.4 DERIVATIONS FROM PROGRAMMING LANGUAGES

Language	F	C	U	Criteria a	b	c	d	e	f	g	h	i	j	k	l	m
DDN (Riddle et al, 1979)	T	C	A	1	2	1	2	1	-	1	-	1	-	A	1	1
Delta (Holbeck-Hanssen et al, 1975)	T	C	A	1	1	1	2	1	-	1	1	1	-	A	1	1
Epsilon (Jensen et al, 1979)	T	C	E	1	1	1	2	1	-	1	1	1	-	A	1	2
RLP (Davis & Rauscher, 1979)	T	C	A	1	-	-	2	-	-	2	-	-	-	A	1	1
SMSDL (Frankowski & Franta, 1980)	T	M	E	1	2	1	2	1	-	1	2	1	2	A	-	-
SPECLE (Biggerstaff, 1979)	T	C	A	1	1	1	2	1	-	1	-	-	2	A	1	-
SREM (Alford, 1977)	T	C	A	1	1	1	2	-	-	1	2	1	-	A	1	1
Mascot (RSRE, 1978)	T	C	A G	1	1	-	1	-	-	-	-	1	-	A	1	1

TABLE A.5 FLOW CHARTS

Language	F	C	U	Criteria a	b	c	d	e	f	g	h	i	j	k	l	m
Flow Charts (Wayne, 1973)	G	M	A	1	1	-	1	-	-	-	-	-	-	A	-	-
Flowgrams (Karp, 1978)	G	C	A	-	-	-	1	-	-	-	-	-	-	A	1	-
Progression Charts (System X, 1979)	G	M	A	1	1	-	1	-	-	-	-	1	-	A	-	-
SX/1 (Corker & Coakley, 1976)	G	C	A	1	1	-	1	-	-	-	-	-	-	A	1	1

TABLE A.6 HIERARCHIC DESCRIPTION METHODS

Language	F	C	U	Criteria a	b	c	d	e	f	g	h	i	j	k	l	m
CADIS (Bubenko & Kallhamer, 1971)	T	C	A	-	2	2	-	-	2	2	-	-	-	T	1	-
CORE (Mullery, 1979)	G	M	A	1	1	1	2	-	-	2	-	-	2	T	-	-
HIPO (Stay, 1976)	G	M	A	1	1	-	-	-	-	2	-	-	-	T	-	-
HOS (Hamilton & Zeldin, 1976)	T	C	A	-	1	1	-	-	-	2	-	-	-	T	1	2
PSL (Teichrow & Hershey, 1977)	T	C	A	1	1	1	1	-	-	1	-	-	2	A	-	-
SADT (Ross, 1977)	G	M	A	1	1	1	2	-	-	2	-	-	2	T	-	-
SSA (Gane & Sarson, 1979)	G	M	A	1	1	1	2	-	-	2	-	-	2	T	-	-

TABLE A.7 FINITE STATE MACHINE LANGUAGES

Language	F	C	U	Criteria a	b	c	d	e	f	g	h	i	j	k	l	m
CDL (Dietrich, 1979)	T	C	A	1	-	-	2	-	-	2	-	-	-	A	1	2
FSIS (Taylor, 1981)	T	C	A	1	-	-	2	-	-	2	-	-	-	T	1	2
Function Flowchart (Hemdal, 1973)	G X	M	A	1	-	-	2	-	-	2	-	-	-	T	-	-
State Transitn Diagram (Kawashima et al, 1971)	G	M	A	-	-	-	2	-	-	2	-	-	-	A	-	2
NPN (Boebert et al, 1979)	T	C	A	1	-	-	2	-	-	2	-	-	-	A	1	2
- (Parnas, 1972)	T	M	P	1	-	-	2	-	-	2	-	-	-	A	1	2
SOM (Braek, 1979)	G	M	A	1	-	-	2	-	-	2	-	-	-	A	-	2
SPECIAL (Robinson, 1976)	T	C	A	1	-	-	2	-	-	2	-	-	-	A	1	2
SDL (CCITT, 1980)	G T	C	A	1	-	-	2	-	-	2	1	-	-	A	1	2
- (Wymore, 1967)	T	M	E	-	-	-	2	-	-	2	-	-	-	T	-	2

TABLE A.8 STATIC DESCRIPTION LANGUAGES

Language	F	C	U	Criteria a	b	c	d	e	f	g	h	i	j	k	l	m
Entity-Relation Model (Chen, 1976)	T	C	A	-	-	2	-	1	-	2	-	-	2	A	1	2
LEGOL (Stamper, 1977)	T	C	E	-	1	2	-	1	1	2	-	-	-	A	1	2
SLICES (Steele & Sussman, 1979)	T	C	E	-	2	2	-	1	-	2	-	-	-	T	1	1
Invariants (Cunningham & Kramer, 1977)	T	C	E	-	-	1	-	1	-	2	-	-	-	T	1	2

TABLE A.9 PRE- AND POST-CONDITION LANGUAGES

Language	F	C	U	Criteria a	b	c	d	e	f	g	h	i	j	k	l	m
- (Dijkstra, 1976)	T	M	E	-	1	-	2	-	-	2	-	-	-	T	1	2
VDL (Note 1) (Bjorner & Jones, 1978)	T	M	A	-	2	1	2	-	-	2	-	-	-	T	1	2
Z (Abrial, 1980)	T	M	A	1	2	1	2	-	-	2	-	-	-	T	1	2
Lambda Calculus (Cleaveland, 1980)	T	M	E	-	2	1	2	-	-	2	-	-	-	T	1	2

Note 1 Jones (Jones(a), 1980) uses a variant of this notation.

TABLE A.10 EVENT-TRIGGERED LANGUAGES

Language	F	C	U	Criteria a	b	c	d	e	f	g	h	i	j	k	l	m
ACTORS (Hewitt, 1977)	T	C	A	-	1	1	2	1	2	2	-	1	-	T	1	2
AP2 (Balzer & Goldman, 1979)	T	C	A	-	2	1	2	1	3	2	2	1	2	T	1	1
AUTOSATE (Gatto, 1974)	T	C	A	-	-	-	2	-	-	2	-	-	-	V	-	-
BDL (Hammer et al, 1977)	X G	C	A	-	-	1	2	-	-	1	-	-	-	T	1	1
CASCADE (Solvberg, 1973)	T G	C	A	-	-	1	2	-	-	2	-	-	-	T	1	1
DMTLT (Sernadas, 1979)	T	M	P	1	1	-	2	-	1	2	-	1	-	T	1	2
DATAFLOW (NCC, 1969)	T	C	A	-	-	1	2	-	-	-	-	-	-	T	1	-
EDDAP (Lindgreen, 1973)	T	C	A	1	-	1	2	-	-	1	-	-	-	A	1	-
FDL (Marconi Radar, 1980)	T	C	E	1	1	1	2	1	2	1	-	1	-	A	1	1
Information Algebra (CODASYL, 1962)	T	M	P	-	-	1	2	-	-	1	-	-	-	T	-	-
JSD (Jackson, 1981)	G	M	A	1	1	1	2	-	-	2	-	-	-	T	-	-
Metaprogramming (Lawson, 1977)	T	C	E	1	1	1	2	-	-	1	-	-	-	A	1	2
STREMA (Clark, 1978)	T	C	A	-	-	1	2	-	-	2	-	-	-	T	1	1
Systematics (Grindley, 1975)	T	M	A	1	1	1	2	-	1	2	-	-	-	A	1	-
Systematrix (Jaderlund, 1980)	X	C	A	-	-	1	2	-	-	2	-	-	-	T	1	1

TABLE A.11 SPECIFICATION ANALYSERS

Language	F	C	U	Criteria a	b	c	d	e	f	g	h	i	j	k	l	m
SPECK (Quirk, 1978)	T	C	A	-	-	-	2	-	1	2	2	1	-	T	1	1
- (Laventhal, 1979)	T	M	P	-	-	-	-	1	1	2	-	-	-	T	1	2

TABLE A.12 SEQUENCE DESCRIPTION LANGUAGES

Language	F	C	U	Criteria a	b	c	d	e	f	g	h	i	j	k	l	m
CCS (Milner, 1980)	T	M	E	-	1	1	-	-	-	2	-	1	-	T	1	2
COSY (Lauer et al, 1979)	T	C	A	1	-	1	-	1	-	2	-	1	-	T	1	2
Path Expressions (Campbell & Habermann, 1974)	T	M	P	-	-	-	-	-	-	2	-	1	-	T	1	2
Regular Expressions (Harrison, 1974)	T	M	P	-	-	1	-	-	-	2	-	-	-	T	1	2

TABLE A.13 PETRI NETS

Language	F	C	U	Criteria a	b	c	d	e	f	g	h	i	j	k	l	m
GRAFCET (Bouteille, 1978)	G	M	A	1	1	-	2	-	-	2	-	1	-	A	1	-
LOGOS (Rose et al, 1972)	G	C	A	-	-	-	2	-	-	1	-	1	-	T	1	2
Petri Nets (Petri, 1962)	G	C	A	-	-	-	2	-	-	2	-	1	-	T	1	2
Pro-Nets (Noe, 1978)	G	C	A	1	1	-	2	-	-	1	-	1	-	T	1	2
Pre-T Nets (Genrich et al, 1980)	G	M	E	-	-	1	2	1	-	2	-	1	-	T	1	2
SARA (Estrin, 1978)	G T	C	A	1	1	1	2	-	-	1	-	1	-	T	1	2

TABLE A.14 LANGUAGES USING AXIOMATICS

Language	F	C	U	Criteria a	b	c	d	e	f	g	h	i	j	k	l	m
ADJ (Goguen et al, 1978)	T	M	E	1	1	1	-	1	-	2	-	-	-	T	1	2
Affirm (Musser, 1979)	T	C	A	1	1	1	-	1	-	2	-	-	-	T	1	2
CLEAR (Burstall & Goguen, 1977)	T	M	E	1	2	1	-	1	-	2	-	-	-	T	1	2
iota (Nakajima et al, 1977)	T	C	A	1	1	1	-	1	-	2	-	-	-	T	1	2
OBJ (Goguen, 1979)	T	C	A	1	1	1	-	1	-	2	-	-	-	T	1	2
- (Hoare, 1969)	T	M	P	-	1	1	-	1	-	2	-	-	-	T	1	2
- (Schwartz & Melliar-Smith, 1981)	T	C	E	-	-	-	-	1	-	2	-	1	-	T	1	2

TABLE A.15 SUMMARY

Language	F	C	U	Criteria a	b	c	d	e	f	g	h	i	j	k	l	m
Universal English	T	M	A	1	2	2	1	1	3	2	2	1	2	V	-	-
CHDL TEGAS6	T	C	E	1	1	-	2	-	-	-	2	-	-	A	1	1
New Programming Language ANNA	T	C	E	1	2	1	1	1	-	-	-	1	-	A	1	2
Derivation from Programming Language Epsilon	T	C	E	1	1	1	2	1	-	1	1	1	-	A	1	2
Flow Charts SX/1	G	C	A	1	1	-	1	-	-	-	-	-	-	A	1	1
Hierarchic Descriptions PSL	T	C	A	1	1	1	1	-	-	1	-	-	2	A	-	-
Finite State SDL	G T	C	A	1	-	-	2	-	-	2	1	-	-	A	1	2
Static Description LEGOL	T	C	E	-	1	2	-	1	1	2	-	-	-	A	1	2
Pre- and Post-Condition Language Z	T	M	A	1	2	1	2	-	-	2	-	-	-	T	1	2
Event-Triggered AP2	T	C	A	-	2	1	2	1	3	2	2	1	2	T	1	1
Specification Analysers SPECK	T	C	A	-	-	-	2	-	1	2	2	1	-	T	1	1
Sequence Description COSY	T	C	A	1	-	1	-	1	-	2	-	1	-	T	1	2
Petri Nets SARA	G T	C	A	1	1	1	2	-	-	1	-	1	-	T	1	2
Languages using Axiomatics OBJ	T	C	A	1	1	1	-	1	-	2	-	-	-	T	1	2
The New Language ASL	T	C	E	1	2	1	2	1	3	2	2	1	2	A	1	2

APPENDIX B

THE SYNTAX DEFINITIONS FOR ASL

1. Backus-Naur Form

The modified form of Backus-Naur Form (BNF) used to define formally the syntax of ASL is taken from a proposal by Wirth, (Wirth(b), 1977); BNF was chosen as it is the most commonly-used form of syntax definition. The following paragraphs describe the main features of BNF, but a more detailed explanation is given in Backhouse (Backhouse, 1979).

(i) terminal symbols (i.e. reserved words and symbols which are part of the language) appear surrounded by quote marks,

e.g. "send" "connections"

(ii) non-terminals (i.e. words used to describe the structures or patterns of the language) appear in upper case letters, and their names cannot contain blanks. The underline character is used as a separator instead of a blank.

e.g. MESSAGE PARAGRAPH_NUMBER

(iii) a sequence in BNF indicates a sequence in ASL,

e.g. PARAGRAPH_NUMBER INTERFACE_NAME

indicates that there must be a paragraph number followed by an interface name, with one or more blanks between them.

(iv) curly brackets, { }, indicate repetition,

e.g. { PARAGRAPH_NUMBER INTERFACE_NAME }

indicates a repetition of zero or more occurrences of a paragraph number followed by an interface name.

(v) square brackets, [], indicate an optional item,

e.g. ["next"] MESSAGE

permits the word "next" to be present, or to be omitted.

(vi) the OR symbol, ¦, indicates alternatives,

e.g. "next" ¦ "(" "+" INTEGER ")"

states that "next" and "(+1)" are permissible alternatives.

(vii) parentheses, ( ), are used to group items to avoid ambiguity,

e.g. ("on" ¦ "within" TIME_DELAY "of")

ensures that the options are "on" and "within 1 second of", and that "on 1 second of" is not allowed.

(viii) a BNF statement is the name of the non-terminal followed by an equals sign ("="), followed by its definition in terms of terminals or other non-terminals, and ending in a full-stop.

e.g. PARAGRAPH_NUMBER = "[" INTEGER
{ "." INTEGER } "]".

which states that a PARAGRAPH_NUMBER is made up of:

(a) open square brackets, followed by

(b) an integer (a non-terminal, which would be

defined elsewhere in terms of digits), followed by

(c) a sequence of zero or more occurrences of a full-stop in front of an INTEGER, followed by

(d) close square brackets.

2. Syntax Definitions

These are given in alphabetic sequence; the top level in the set of productions is SPECIFICATION.

```
A_MESSAGE = ANY_NAME [ CONTENTS ] [ ROUTE ].

A_RESPONSE =
        ( "send" ¦ "start" "sending" )
             ( ANY_NAME [ CONTENTS ] ROUTE ¦
             REPLIES ¦
             LOOSE_END [ ROUTE ] ) ¦
        LOOSE_END ¦
        "stop" "sending"
             ( ANY_NAME [ CONTENTS ] ROUTE ¦
             REPLIES ¦
             LOOSE_END ¦
             "current" "messages" ROUTE ).
```

```
A_STIMULUS =
        ANY_NAME [ CONTENTS ] ROUTE ¦
        ( "start" ¦ "end" ) "of" ANY_NAME
            [ CONTENTS ] ROUTE ¦
        LOOSE_END [ ROUTE ].

ADD_OPERATOR = "+" ¦ "-".

ALTERNATIVES = "any" "one" "of"
            PARAGRAPH_NUMBER RESPONSE
            { PARAGRAPH_NUMBER RESPONSE }.

AN_INTERFACE = ANY_NAME "." ANY_NAME.

AND = "&" ¦ "and".

ANY_NAME =
        NAME_IN_LOWER_CASE ¦ "?" [ NAME_IN_LOWER_CASE ].

ANY_QUALIFIED_NAME = ANY_NAME { "." ANY_NAME }.

ASSIGNMENT = "is" CONDITION [ LOCAL_DEFINITION ].

BASIC_TYPE = "integer" ¦ "decimal" ¦ "character" ¦
        "boolean" ¦ "interface" ¦ "message".

BEHAVIOUR = ("on"¦ "within" TIME_DELAY "of")
        STIMULUS "then" RESPONSE [ UNLESS ].
```

```
BUILT_IN_OPERATION = "sum" "(" EXPRESSION_C ")" ¦
        "count" "(" EXPRESSION_C ")" ¦
        "min" "(" EXPRESSION_C "," EXPRESSION_C ")" ¦
        "max" "(" EXPRESSION_C "," EXPRESSION_C ")".

CAPITAL_LETTER = "A" ¦"B" ¦"C" ¦"D" ¦"E" ¦"F" ¦"G" ¦
                 "H" ¦"I" ¦"J" ¦"K" ¦"L" ¦"M" ¦"N" ¦
                 "O" ¦"P" ¦"Q" ¦"R" ¦"S" ¦"T" ¦"U" ¦
                 "V" ¦"W" ¦"X" ¦"Y" ¦"Z".

COMMENT = "/" NAME_IN_CAPITALS { COMMENT_WORD }.

COMMENT_WORD = NAME_IN_CAPITALS ¦ NUMBER ¦ ADD_OPERATOR ¦
        MULTIPLY_OPERATOR ¦ RELATIONAL_OPERATOR.

CONDITION = EXPRESSION_A { OR EXPRESSION_A }.

CONSTANT = NAME_IN_LOWER_CASE ¦ RANGE ¦ EXPRESSION_C.

CONTENTS =
        "=" EXPRESSION_C [ LOCAL_DEFINITION ] ¦
        "with" CONTENTS_CONDITION [ LOCAL_DEFINITION ].

CONTENTS_CONDITION = CONTENTS_EXPRESSION
            { OR CONTENTS_EXPRESSION }.

CONTENTS_EXPRESSION = CONTENTS_EXPRESSION_B
            { AND CONTENTS_EXPRESSION_B }.
```

```
CONTENTS_EXPRESSION_B = [ NOT ] CONTENTS_NAME
                RELATIONAL_OPERATOR CONTENTS_NAME.

CONTENTS_NAME = ANY_QUALIFIED_NAME.

DEFINITION = NAME_IN_CAPITALS [ COMMENT ]
                { "," NAME_IN_CAPITALS [ COMMENT ] }
           (":" (TYPE_NAME ¦ SET_DEFINITION) ¦
           "is" (PARAGRAPH_NUMBER DEFINITION
                { PARAGRAPH_NUMBER DEFINITION } ¦
                     DEFINITION)).

DIGIT = "0" ¦"1" ¦"2" ¦"3" ¦"4" ¦"5"
                ¦"6" ¦"7" ¦ "8" ¦"9".

END_OF_BLOCK = "end" "of" NAME_IN_LOWER_CASE.

EXCEPTION_CONDITION =
           "whenever" CONDITION "then" RESPONSE UNLESS.

EXPRESSION_A = EXPRESSION_B { AND EXPRESSION_B }.

EXPRESSION_B = [ NOT ] EXPRESSION_C
           [ RELATIONAL_OPERATOR EXPRESSION_C ].

EXPRESSION_C = EXPRESSION_D { ADD_OPERATOR EXPRESSION_D }.

EXPRESSION_D = EXPRESSION_D
           { MULTIPLY_OPERATOR EXPRESSION_E }.
```

```
EXPRESSION_E = ANY_NAME ¦ LITERAL ¦
          "(" CONDITION ")" ¦ REF_TO_PAST_MESSAGES.

FIXED_RELATIONSHIP = (NAME_IN_LOWER_CASE ¦ QUALIFIED_NAME)
               "is" CONDITION.

GOAL = "take" "any" "action" "to" "achieve"
               NAME_IN_CAPITALS [ LOCAL_DEFINITION ].

INTEGER = DIGIT { DIGIT }.

INTERCONNECTIONS = "connections"
          PARAGRAPH_NUMBER INTERFACE_LIST "to"
               INTERFACE_LIST [ LOCAL_DEFINITION ]
        { PARAGRAPH_NUMBER INTERFACE_LIST "to"
               INTERFACE_LIST [ LOCAL_DEFINITION ] }.

INTERFACE = ("input" ¦ "output" ¦ "bothway")
          (NAME_IN_CAPITALS ¦
          PARAGRAPH_NUMBER NAME_IN_CAPITALS
               { PARAGRAPH_NUMBER NAME_IN_CAPITALS } ).

INTERFACE_LIST =
          "(" AN_INTERFACE { "," AN_INTERFACE } ")" ¦
          QUALIFIED_NAME.

INTERFACE_NAME = NAME_IN_LOWER_CASE.
```

```
ITERATOR = "for" "all" (EXPRESSION_B ¦ STIMULUS)
              [ LOCAL_DEFINITION ].

LIMITER = "approximately" ¦ "<=" ¦ ">=" ¦ "^>" ¦
              "<>" ¦ "^=" ¦ "^<" ¦ ">" ¦ "<".

LITERAL = LOOSE_END ¦ NUMBER.

LOCAL_DEFINITION = "where"
         ( (DEFINITION ¦ CONDITION) ¦
         PARAGRAPH_NUMBER (DEFINITION ¦ CONDITION)
              { PARAGRAPH_NUMBER (DEFINITION ¦
                   CONDITION)}).

LOOSE_END = "unknown" ¦ "undefined" ¦ "dont" "care".

MESSAGE_DICTIONARY = "messages"
         ( (DEFINITION ¦ MESSAGE_EQUIVALENCE) ¦
         PARAGRAPH_NUMBER
              (DEFINITION ¦ MESSAGE_EQUIVALENCE)
              { PARAGRAPH_NUMBER
              (DEFINITION ¦ MESSAGE_EQUIVALENCE) } ).

MESSAGE_EQUIVALENCE = QUALIFIED_NAME "is" QUALIFIED_NAME
         [ "where" { PARAGRAPH_NUMBER
              MESSAGE_EQUIVALENCE } ].
```

```
MODEL = START_OF_BLOCK
        MODEL_ STATEMENT { MODEL_STATEMENT }
        END_OF_BLOCK.

MODEL_STATEMENT =
        PARAGRAPH_NUMBER (INTERFACE ¦ DEFINITION ¦
             RULE ¦ BEHAVIOUR ¦ OPERATION_DEFINITION ¦
                  MODEL_STATEMENT ).

MULTIPLY_OPERATOR = "**" ¦ "*" ¦ "/".

NAME_IN_CAPITALS =
        CAPITAL_LETTER ¦ CAPITAL_LETTER ¦ DIGIT ¦ "_" ¦
        [ SUBSCRIPTS ].

NAME_IN_LOWER_CASE =
        SMALL_LETTER ¦ SMALL_LETTER ¦ DIGIT ¦ "_" ¦
        [ SUBSCRIPTS ].

NUMBER = DIGIT { DIGIT } [ "." DIGIT { DIGIT } ].

NUMBER_OF_MODELS = "created" "from"
        PARAGRAPH_NUMBER QUANTITY NAME_IN_LOWER_CASE
        { PARAGRAPH_NUMBER QUANTITY NAME_IN_LOWER CASE }.
```

```
OPERATION_DEFINITION =
        "operation" NAME_IN_CAPITALS
        [ "(" [NAME_IN_LOWER_CASE
             { "," NAME_IN_LOWER_CASE } ]
        [ "-->" NAME_IN_LOWER_CASE {","
             NAME_IN_LOWER_CASE } ] ")" ] [ COMMENT ]
        "is"
           OPERATION_STATEMENT { OPERATION_STATEMENT }.

OPERATION_NAME = NAME_IN_LOWER_CASE.

OPERATION_STATEMENT = DEFINITION ¦ OTHER_BEHAVIOUR ¦
        RULE ¦ PARAGRAPH_NUMBER OPERATION_STATEMENT.

OR = "¦" ¦ "or".

OTHER_BEHAVIOUR = BEHAVIOUR ¦ RESPONSE ¦
        ITERATOR "then" RESPONSE.

PARAGRAPH_NUMBER =
        "[" INTEGER { "." INTEGER } [ COMMENT ] "]" ¦
        "(." INTEGER { "." INTEGER } [ COMMENT ] ".)".
```

```
PREFIX =
        ("time" ¦ "start" ¦ "end" ¦ "duration") "of"
            [ "sending" ¦ "receiving" ] )
            ( "last" ¦ "first" ¦ "#" INTEGER ) ¦
        "last" ¦ "first" ¦ "all" ¦ "#" INTEGER ¦
        "sending" ¦ "receiving".

QUALIFIED_NAME = NAME_IN_LOWER_CASE "." NAME_IN_LOWER_CASE
        { "." NAME_IN_LOWER_CASE }.

QUANTITY = LOOSE_END [ "number" ] ¦
        LIMITER EXPRESSION_E ¦ RANGE.

RANGE = EXPRESSION_E [ "to" EXPRESSION_E ].

REF_TO_PAST_MESSAGES = PREFIX
        ( ANY_NAME [ CONTENTS ] ROUTE ¦ "message" ).

RELATIONAL_OPERATOR = "<=" ¦ ">=" ¦ "^=" ¦
        "<>" ¦ "=" ¦ ">" ¦ "<" ¦ "^>" ¦ "^<" ¦
        "in" ¦ "while" ¦ "after" ¦
        "at" "same" "time" "as".

REPLIES = "(" ANY_NAME [ CONTENTS ] [ ROUTE ]
        { "," ANY_NAME [ CONTENTS ] [ ROUTE ] } ")"
            [ ROUTE ].
```

```
RESPONSE = SELECTION ¦ ALTERNATIVES ¦ GOAL ¦
         SIMPLE_REPLY ¦ LOOSE_END ¦ RESPONSE_SEQUENCE.

RESPONSE_SEQUENCE = "sequence"
         PARAGRAPH_NUMBER SIMPLE_REPLY
         { PARAGRAPH_NUMBER SIMPLE_REPLY }.

ROUTE = "via" ANY_NAME.

RULE = EXCEPTION_CONDITION ¦ FIXED_RELATIONSHIP ¦
         SEQUENCE.

SELECTION = "select"
         ( "(" CONDITION ")" { PARAGRAPH_NUMBER
              "(" VALUE ")" "when" RESPONSE } ¦
         { PARAGRAPH_NUMBER "(" CONDITION ")"
              "when" RESPONSE } )
         PARAGRAPH_NUMBER "otherwise" RESPONSE.

SEQUENCE = "sequence" [ ROUTE ]
         { PARAGRAPH_NUMBER [ "optional" ]
              [ "next" ¦ "(" "+" INTEGER ")" ]
              A_MESSAGE }.
```

```
SET_DEFINITION =
        "{" CONSTANT [ COMMENT ]
             { "," CONSTANT [ COMMENT ] } "}" ¦
        "(*" CONSTANT [ COMMENT ]
             { "," CONSTANT [ COMMENT ] } "*)".

SIMPLE_REPLY = BUILT_IN_OPERATION ¦
        "(" A_RESPONSE { "," A_RESPONSE } ")"
        NAME_IN_LOWER_CASE [ ASSIGNMENT ] ¦
        A_RESPONSE ¦ "do" "nothing".

SMALL_LETTER = "a" ¦"b" ¦"c" ¦"d" ¦"e" ¦"f" ¦"g" ¦
               "h" ¦"i" ¦"j" ¦"k" ¦"l" ¦"m" ¦"n" ¦
               "o" ¦"p" ¦"q" ¦"r" ¦"s" ¦"t" ¦"u" ¦
               "v" ¦"w" ¦"x" ¦"y" ¦"z".

SPECIFICATION =
        (SYSTEM_BLOCK MODEL MODEL ¦
        MODEL SYSTEM_BLOCK MODEL ¦
        MODEL MODEL SYSTEM_BLOCK)  { MODEL }.

STANDARDS = "refers" "to"
        (QUALIFIED_NAME ¦
        PARAGRAPH_NUMBER QUALIFIED_NAME
             { PARAGRAPH_NUMBER QUALIFIED_NAME } ).

START_OF_BLOCK = NAME_IN_CAPITALS [ COMMENT ] "is".
```

```
STIMULUS = A_STIMULUS ¦
         "(" A_MESSAGE { "," A_MESSAGE } ")" [ ROUTE ].

SUBSCRIPTS = "(" CONDITION { "," CONDITION } ")".

SYSTEM_BLOCK = START_OF_BLOCK
         SYSTEM_STATEMENT { SYSTEM_STATEMENT }
         END_OF_BLOCK.

SYSTEM_STATEMENT = PARAGRAPH_NUMBER
         ( NUMBER_OF_MODELS ¦ INTERCONNECTIONS ¦
         MESSAGE_DICTIONARY ¦ OPERATION_DEFINITION ¦
         DEFINITION ¦ STANDARDS ¦ SYSTEM_STATEMENT ).

TIME_DELAY = LOOSE_END ¦ RANGE [ UNITS ].

TYPE_NAME = ( "subset" [ "(" RANGE ")" ] ¦
         "string" ¦ "set" ) "of"
              ANY_NAME [ LOCAL_DEFINITION ] ¦
         LOOSE_END ¦ BASIC_TYPE ¦
         ANY_NAME [ LOCAL_DEFINITION ].

UNITS = NAME_IN_LOWER_CASE.
```

```
UNLESS = "unless"
         ( "(" CONDITION ")" "when" RESPONSE ¦
         PARAGRAPH_NUMBER "(" CONDITION ")"
              "when" RESPONSE
              { PARAGRAPH_NUMBER "(" CONDITION ")"
                   "when" RESPONSE } ).

VALUE = ANY_QUALIFIED_NAME ¦
         LOOSE_END ¦ NUMBER.
```

3. The Type-matching Rule Format

Each BNF production in the definition of ASL is further qualified by a type-matching rule; this indicates how the data types of the terminal and non-terminal symbols in the production must be related to each other. As there is no commonly-agreed standard presentation for type-matching rules, a version of that used by Davie and Morrison (Davie & Morrison, 1981) has been adopted. This has a simple form, and introduces only a small amount of extra notation as follows.

(a) Data type names are shown enclosed in angle braces ("<>"), with names in lower case letters indicating defined types and names in capital letters indicating type variables.

(b) Each type-matching rule produces a result (the "type" of the statement or expression), which is shown after the symbol "=>".

(c) The basic types in ASL are <boolean>, <character>,

<constant>, <decimal>, <integer>, <interface>, <message>, <model>.

(d) In addition to the basic types it is necessary to have <void> for expressions which do not produce a result of any particular type, and <any> for items such as "undefined" (see Chapter 4.4.10) which can be used in place of various types.

(e) The type-matching rules take the same pattern as the BNF productions, but with type names in the positions previously occupied by terminals or non-terminals which represent names, e.g.:

BNF production:-

CONDITION = EXPRESSION_A {OR EXPRESSION_A}.

Type-matching rule:-

<boolean> {OR <boolean>} => <boolean>.

which states that , if two EXPRESSION_A's are linked by an OR, they must both be of type <boolean> and will give a result of type <boolean>.

There are an infinite number of legal data types in ASL, constructed from the basic types by recursive application of the following rules.

(i) For any data type <T>, <*T> is the data type of a vector with elements of type <T>.

(ii) A user-defined structure (see Chapter 4.4.6) is represented as the data types of its elements, in the form of a list of lists which mirrors the tree-structure of the definition.

(iii) An operation with arguments of types <T1>, <T2>, ... <Tn> and results of types <TO1>, ... <TOm> has

the type (<TO1>, ... <TOm>).

(iv) Enumeration types (i.e. lists of constants in braces) have the type <set of constant>.

(v) A "subset" of a type <T> is treated as still being of type <T>, whilst a "set" of <T> has type <set of T>.

(vi) A "string" of elements of type <T> has type <string of T>.

4. The Type-matching Rules

In order to reduce the number of rules to be presented, the following have not been included as their elements all have type <void> and their result is also <void>:

(a) productions relating to the construction of names (e.g. CAPITAL_LETTER, SMALL_LETTER, DIGIT),

(b) terminals which are connectives in expressions (e.g. the arithmetic and relational operators),

(c) comments,

(d) productions which merely offer a list of alternatives which are themselves complete productions (e.g. MODEL_STATEMENT).

Also, some of the productions have been grouped together as their type rules are identical. Where the options in a BNF rule (i.e. the portions in square brackets ("[]") or braces ("{}")) may cause the result to be different, multiple type-matching rules have been included to cover the various cases. The rules are listed below in alphabetic

sequence.

A_MESSAGE, A_STIMULUS

(i) <message> => <message>.

(ii) <message> "=" <T> => <message>.

(iii) <message> "with" <boolean> => <message>.

(iv) <message> "via" <interface> => <void>.

A_RESPONSE

("send" ¦ "start" "sending" ¦
"stop" "sending") <T> => <T>.

ALTERNATIVES

"any" "one" "of" { <void> <void> }
=> <void>.

AN_INTERFACE

<model> "." <interface> => <void>.

ANY_NAME

(i) <T> => <T>.

(ii) "?" <void> => <any>.

(iii) "?" => <any>.

ANY_QUALIFIED_NAME

(i) <T1> "." <T2> => <T2>.

(ii) <T1> "." ... "." <Tn> => <Tn>.

```
BEHAVIOUR
     (i) "on" <void> "then" <void> => <void>.
     (ii) "within" <integer> "of" <void> "then"
               <void> => <void>.

CONDITION, CONTENTS_CONDITION
     (i) <boolean> { OR <boolean> } => <boolean>.
     (ii) <T> => <T>.

CONTENTS_EXPRESSION_A, EXPRESSION_A
     (i) <boolean> { AND <boolean> } => <boolean>.
     (ii) <T> => <T>.

CONTENTS_EXPRESSION_B
     (i) NOT <T> RELATIONAL_OPERATOR <T> => <boolean>.
     (ii) <T> RELATIONAL_OPERATOR  <T> => <boolean>.
     (iii) <T> => <T>.

DEFINITION
     (i) <void> ":" <T> => <void>.
     (ii) <void> ":" "{" <constant> ","
          ... "," <constant> "}" => <void>.
     (iii) <void> ":" "subset" <T> "to" <T> "of" <T>
               => <void>.
     (iv) <void> ":" "set" "of" <T> => <void>.
     (v) <void> ":" "string" "of" <T> =<void>.
     (vi) <void> "(" <integer> "to" <integer> ")"
               ":" <T> => <void>.
     (vii) <void> "is" <void> <Tl>
```

```
                    <void> <T2>
                       :       :
                    <void> <Tn> => <void>.

END_OF_BLOCK
    "end" "of" <model> => <void>.

EXCEPTION_CONDITION
    "whenever" <boolean> "then" <void> => <void>.

EXPRESSION_B
    (i) <T> RELATIONAL_OPERATOR <T> => <boolean>.
    (ii) NOT <boolean> => <boolean>.
    (iii) <T> => <T>.

EXPRESSION_C
    (i) <integer> { ADD_OPERATOR <integer> }
              => <integer>.
    (ii) <decimal> { ADD_OPERATOR <decimal> }
              => <decimal>.

EXPRESSION_D
    (i) <integer> { MULTIPLY_OPERATOR <integer> }
              => <integer>.
    (ii) <decimal> { MULTIPLY_OPERATOR <decimal> }
              => <decimal>.
```

```
FIXED_RELATIONSHIP
    <T> "is" <T> => <void>.

GOAL
    "take" "any" "action" "to" "achieve" <void>
              => <void>.

INTERCONNECTIONS
    "connections" <void> <void> "to" <void>
                   :        :          :
                  <void> <void> "to" <void>
                   => <void>.

INTERFACE
    ("input" ¦ "output" ¦ "bothway") <void> => <void>.

INTERFACE_LIST
    "(" <void> "," ... <void> ")" => <void>.

ITERATOR
    "for" "all" <boolean> => <boolean>.

LOCAL_DEFINITION
    (i) "where" <boolean> => <void>.
    (ii) "where" <void> => <void>.

LOOSE_END
    ("undefined" ¦ "unknown" ¦ "dont care") => <any>.
```

```
MESSAGE_DICTIONARY
    "messages" <void> => <void>.

MESSAGE_EQUIVALENCE
    (i) <message> "is" <message> => <void>.
    (ii) <message> "is" <message>
              "where" <T> "is" <T> => <void>.

NUMBER_OF_MODELS
    "created" "from"
              <void> <integer> <model>
              <void> <integer> <model>
                   :       :       :
              <void> <integer> <model> => <void>.

OPERATION_DEFINITION
    (i) "operation" <void> "is" <void> => <void>.
    (ii) "operation" <void> "(" <TI1> "," ... <TIn>
              "-->" <TO1> "," ... <TOn> ")"
               "is" <void> => <void>.

OTHER_BEHAVIOUR
    <boolean> "then" <void> => <void>.

PARAGRAPH_NUMBER
    "[" <integer> "." .... <integer> "]"
              => <void>.
```

```
QUALIFIED_NAME
    (i) <T1> "." <T2> => <T2>.
    (ii) <T1> "." ... <Tn> => <Tn>.

QUANTITY
    LIMITER <integer> => <integer>.

RANGE
    <T> "to" <T> => <T>.

REF_TO_PAST_MESSAGES
    (i) PREFIX <message> "via" <interface>
            => <message>.
    (ii) PREFIX <message> CONTENTS "via" <interface>
            => <message>.

REPLIES, STIMULUS
    (i) "(" <void> "," ... <void> ")" => <void>.
    (ii) "(" <message> "," ... <message> ")"
            "via" <interface> => <void>.

RESPONSE_SEQUENCE, SEQUENCE
    "sequence" <void> <void>
                :       :
               <void> <void> => <void>.
```

SELECTION

(i) "select" "(" <T> ")"

<void> "(" <T> ")" "when" <void>

: : : :

<void> "otherwise" <void>.

(ii) "select"

<void> "(" <boolean> ")" "when" <void>

: : : :

<void> "otherwise" <void>.

SIMPLE_REPLY

(i) "do" "nothing" => <void>.

(ii) Operation calls are treated as explained in Section B.3, item 4.

STANDARDS

"refers" "to" <void> => <void>.

START_OF_BLOCK

<void> "is" => <void>.

UNLESS

(i) "unless" "(" <boolean> ")" "when" <void> => <void>.

(ii) "unless" <void> "(" <boolean> ")" "when" <void>

: : :

<void> "(" <boolean> ")" "when" <void> => <void>.

APPENDIX C

THE SEMANTIC DEFINITION OF ASL

1. Introduction

The reasons for requiring a formal semantic definition of ASL were covered in Chapter 4.5.6; this appendix merely provides the details of the model which has been used. In order to simplify the task of producing the semantic definition, no attempt has been made to complete this down to the level of basic mathematical logic. The theory developed by the authors of Predicate/Transition nets (Genrich et al, 1980) and Time Petri Nets (Merlin, 1974) has been assumed as primitives and the necessary model constructed on top of these. Rather than presenting the theory of this net model, it was felt to be appropriate to describe the process of translating an ASL specification into an equivalent net. This is consistent with taking Predicate/Transition net theory to be already well-defined, but also provides the basis for the design of a computer program to perform this translation. A brief resume of the firing rules for Predicate/Transition nets and Time Petri nets is given in Section 5 of this appendix.

The aim of the model is to capture the intended properties of the language as described in Chapters 3 and 4. The following are a few examples of these properties.

(a) Each model is a closed entity; information may only be transfered between models by means of messages.

(b) Message transmission is treated as instantaneous and error-free.

(c) Models may introduce a time delay between the receipt of a message and the consequent response.

(d) All the information ever sent to a model is always available to that model for re-examination.

(e) The interfaces of a model act as simple sequential channels (except for "bothway" interfaces, which act as a pair of channels in opposite directions), and can therefore only receive or send one message at a time.

(f) Actions which do not use the same resource (e.g. the same interface) can take place concurrently.

(g) Absolute time information originates from the observer; models only measure small intervals of time from the receipt of messages.

(h) Monitors (i.e. statements of the form "whenever...") have priority over simple behaviour statements, so that it is possible to use a monitor to override the normal response in exceptional circumstances.

(i) "undefined" and "unknown", which are used when it is not (yet) possible to completely specify a system, act like additional elements in all defined data types. An operation given an "undefined" argument

will produce an "undefined" result.

These are only a small number of the properties which are represented in the semantic model, and are listed only to give an indication of the type of constraints which the model contains.

The expressive power of ASL, for example in the use of operations and pattern-matching, make it difficult to provide a direct mapping from the syntax definition into the modified Predicate/Transition nets. This has therefore been split into the three stages of transformation, translation and connection; these are described in the following three sections. From the complexity of each of the three stages it will be seen that this process is not suitable for manual operation. The development of a computer program to perform this task is part of the further work proposed in Chapter 8.

2. Transformation

The transformations described in this section operate on ASL specifications at the syntactic level, reducing the variety of statement types down to one basic form:

"if" CONDITION "then" "(" TIME_DELAY ")" RESPONSE.

where the RESPONSEs are constrained to be of a very simple form. In order to provide identification of event-triggered behaviour for the subsequent translation stage, the CONDITIONs for messages take the form "event(MESSAGE)". All the stimulus-response behaviour is shown with a TIME_DELAY; any behaviour which was in the

"on...then..." form will have a time delay of zero.

Figures C.1 to C.10 each describe one of these transformations by showing the syntax form it deals with and the result which it produces. These appear as simple examples only, not as the full BNF conversion rules; more complex forms (such as nested "select" statements) require recursive application of the transformations in order to obtain complete simplification. In all cases where repeated application of the rules is required this is done by starting with the most deeply nested part of the expression. Local definitions (of the form "where...") and pattern-matching variables (i.e. those prefixed by "?") are treated as a form of abstraction, as in bracket abstraction (Turner, 1979) or lambda abstraction (Stoy, 1977). They can therefore normally be removed by simple replacement of the appropriate names by the expressions to which they are equivalent; the comment above about the ordering of repeated applications of the transformations also applies in this case. One exception to this is where a defined operation is used recursively; if the recursion is local to an operation which defines some mathematical function (i.e. it does not send or receive messages) then the recursive definition appears unchanged as a recursively-defined predicate at the appropriate location in the net model. Otherwise the recursion is modelled as an iterative loop in the net, taking the same form as the treatment of iterators in Figure C.15.

FIGURE C.1 REPLACEMENT OF OPERATIONS

(a) An operation with no arguments:

```
....then signal
```

and its definition:

```
operation SIGNAL is
                a_response
```

becomes:

```
....then a_response
```

(b) With arguments but no result:

```
....then dispense(x)
```

and:

```
operation DISPENSE(d) is
     [1] D : contents
     [2] send drink with
          contents = recipe(d) via dispenser
```

becomes:

```
....then send drink with contents=recipe(d)
                    via dispenser
```

(c) A value-returning operation, such as:

```
....square_root(a, b)....
```

and:

```
operation SQUARE_ROOT(x, t --> r) is
     [1] X, T, R :decimal
     [2] r is ?y where abs(r*r-x) <= t
```

becomes:

```
....?z1 where (abs(z1*z1-a)<= b)....
```

where "z1" is a new unique name created for the purpose.

FIGURE C.2 REPLACEMENT OF FIXED RELATIONSHIPS

Given a fixed relationship definition:

```
a_name_in_lower_case is some_expression
```

and some mention of the same name:

```
.... a_name_in_lower_case....
```

then the mention of the name transforms to:

```
.... ( some_expression ) .....
```

Note

The name being replaced can be either a simple name or a qualified name.

FIGURE C.3 SEPARATION OF LISTS INTO INDIVIDUALS

(a) Lists of responses.

```
on stimulus_x then ( response_1 , .... )
```

becomes:

```
if event ( stimulus_x ) then response_1
if event ( stimulus_x ) then response_2
    etc..
```

(b) Lists of stimuli.

```
on ( a_message_1 , .... ) then response_x
```

becomes:

```
if event ( a_message_1 ) then response_x
if event ( a_message_2 ) then response_x
    etc..
```

FIGURE C.4 CONVERSION OF "SELECT" EXPRESSIONS

```
(a)  select ( a_condition )
         paragraph_1 ( value_1 ) when response_1
         paragraph_2 ( value_2 ) when response_2
            :              :        :        :
         paragraph_n-1 ( value_n-1 ) when response_n-1
         paragraph_n otherwise response_n

     becomes:

         if ( a_condition = value_1 ) then response_1
         if ( a_condition = value_2 ) then response_2
         :          :          :        :      :
         if ( a_condition = value_n-1 ) then
                                          response_n-1
         if ( a_condition <> value_1 )
            and ( a_condition <> value_2 )
             :          :                :
            and ( a_condition <> value_n-1 )
                                        then response_n


(b)  select
         paragraph_1 ( condition_1 ) when response_1
         paragraph_2 ( condition_2 ) when response_2
              :             :          :      :
         paragraph_n-1 ( condition_n-1 ) when
                                          response_n-1
         paragraph_n otherwise response_n

     becomes:

         if ( condition_1 ) then response_1
         if ( condition_2 ) then response_2
         :          :          :      :
         if ( condition_n-1 ) then response_n-1
         if ( not condition_1 )
            and ( not condition_2 )
             :              :
            and ( not condition_n-1 ) then response_n
```

FIGURE C.5 CONVERSION OF "UNLESS" EXPRESSIONS

A statement with an "unless" part:

```
on stimulus then normal_response
   unless
      paragraph_1 ( condition_1 ) when response_1
      paragraph_2 ( condition_2 ) when response_2
          :              :          :        :
      paragraph_n ( condition_n ) when response_n
```

becomes:

```
if event ( stimulus ) and condition_1 then response_1
if event ( stimulus ) and condition_2 then response_2
:    :        :       :     :         :     :
if event ( stimulus ) and condition_n then response_n
if event ( stimulus ) and ( not condition_1 )
     and ( not condition_2 )
      :     :        :
     and ( not condition_n ) then normal_response
```

FIGURE C.6 EXPANSION OF LOCAL DEFINITIONS

(a) The expression:

```
r is ?y
     where y*y = x
```

becomes: `r*r = x`

so removing all references to "y".

(b) "In-line" definitions, such as:

```
.... y .... where Y : a_type
```

are treated as if defined normally, so the "where" part is merely left out of the transformation.

(c) More complex uses of patterns, e.g.:

```
z is ?x where
     (x in b_signals) and (encode(y) = x)
```

are simplified as:

```
((z is encode(y)) and (z in b_signals))
```

(d) Uses of names given simple values by local definitions, e.g.:

```
....x.... where x = f(y)
```

become: `....f(y).....`

(e) Names given values by inverse operations, e.g.:

```
....x....      where y=x*x
```

become: `((....x....) and (x*x=y))`

FIGURE C.7 REPLACEMENT OF LOCAL VARIABLES

To reduce all references within a model to one consistent form, all the uses of local variables inside a model are replaced by the appropriate references to an imaginary interface.

Thus:

```
.... then local_name is value_expression
```

becomes:

```
.... then send local_name = value_expression
              via local_name_interface
```

and:

```
.... local_name ....
```

becomes:

```
.... ( last message sent via
                local_name_interface )
```

FIGURE C.8 SEQUENCES OF ACTIONS

This transformation only applies to the use of "sequence" in the RESPONSE part of an "on...then...", "within...then...", or "whenever...then.." statement. For the treatment of global sequence constraints see Figure C.9.

For this type of sequence:

```
....then sequence
          paragraph_1 action_1
          paragraph_2 action_2
               :          :
          paragraph_n action_n
```

becomes:

```
....then ( action_1 ; action_2 ;
                .... ; action_n )
```

FIGURE C.9 SEQUENCE CONSTRAINTS

These general constraints upon behaviour have to be replaced by a set of monitors which have the same effect. Thus:

```
paragraph_1 sequence
     paragraph_2 action_1
     paragraph_3 optional action_2
     paragraph_4 action_3
          :          :
     paragraph_n action_n
```

will be transformed into a set of statements of the following form:

```
if event (action_3) and not
   ((last message via action_2_interface = action_2)
     or (last message via action_1_interface =
          action_1)) then undefined
if event ( action_2 ) and not
     (last message via action_1_interface = action_1)
          then undefined
etc..
```

FIGURE C.10 CONVERSION OF MONITORS

Monitor expressions, which have the form "whenever....", are translated directly into the required form with the exception those which represent timeouts.

(a) Simple monitors, such as:

```
whenever a_condition then a_response
```

become:

```
if a_condition then a_response
```

(b) Timeouts, such as:

```
whenever 15 seconds after lift_handset
        via subs_line(x) then....
```

are represented as:

```
if timer ( 15 seconds after lift_handset
        via subs_line(x) ) then....
```

3. Translation

The translation of the behaviour statements into fragments of Predicate/Transition nets uses five patterns for net elements, as shown in Figures C.11 to C.15. These patterns encapsulate the following concepts.

(a) (Figure C.11) There is one place in the net for each interface of a model, except for bothway interfaces which are treated as two interfaces. This place holds a single token, to reflect the limitation that each interface can only receive a single message at any instant of time.

(b) (Figure C.12) As there may be a number of monitors associated with one message, it is necessary to await the decisions of all these monitors before proceeding with any direct response to the message.

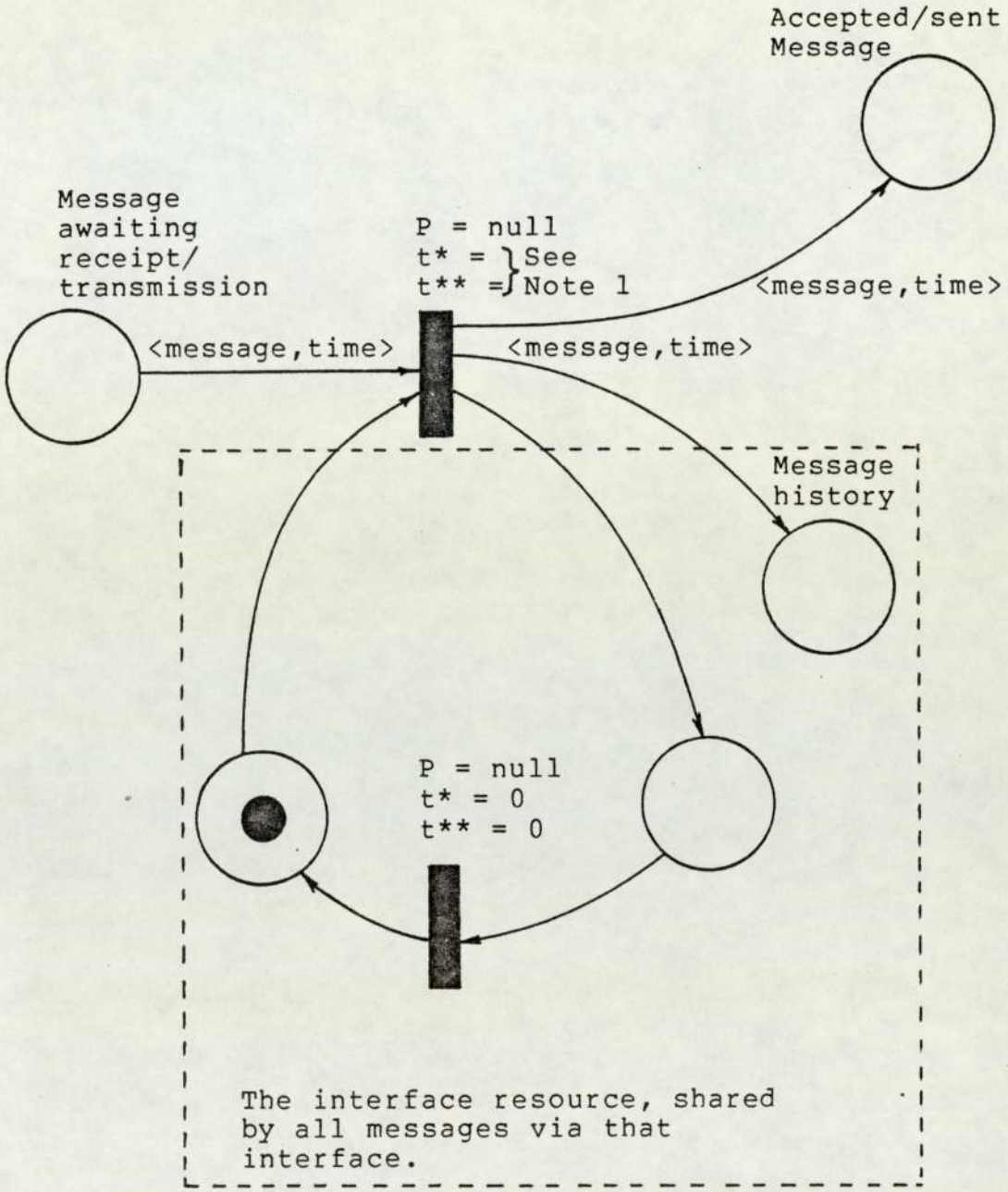
(c) (Figure C.13) The receipt of a message first causes any monitor associated with that message to be checked. After the monitor has decided whether to take action, the message then may cause some response.

(d) (Figure C.14) Timeouts in a monitor start the operation of a local clock, which may be terminated by the arrival of some message or by the end of the appropriate period of time.

(e) (Figure C.15) A "for all" statement causes an iterative loop to be entered, producing the set of responses in some arbitrary sequence.

The conventions used in Figures C.11 to C.15 are taken

FIGURE C.11 TRANSLATION OF RECEIPT OF MESSAGES



Note 1: For all incoming messages, t* = 0 and t** = infinity; for outgoing messages, t* = minimum delay time and t** = maximum delay time from the ASL statement concerned.

FIGURE C.12 TREATMENT OF MULTIPLE ARCS

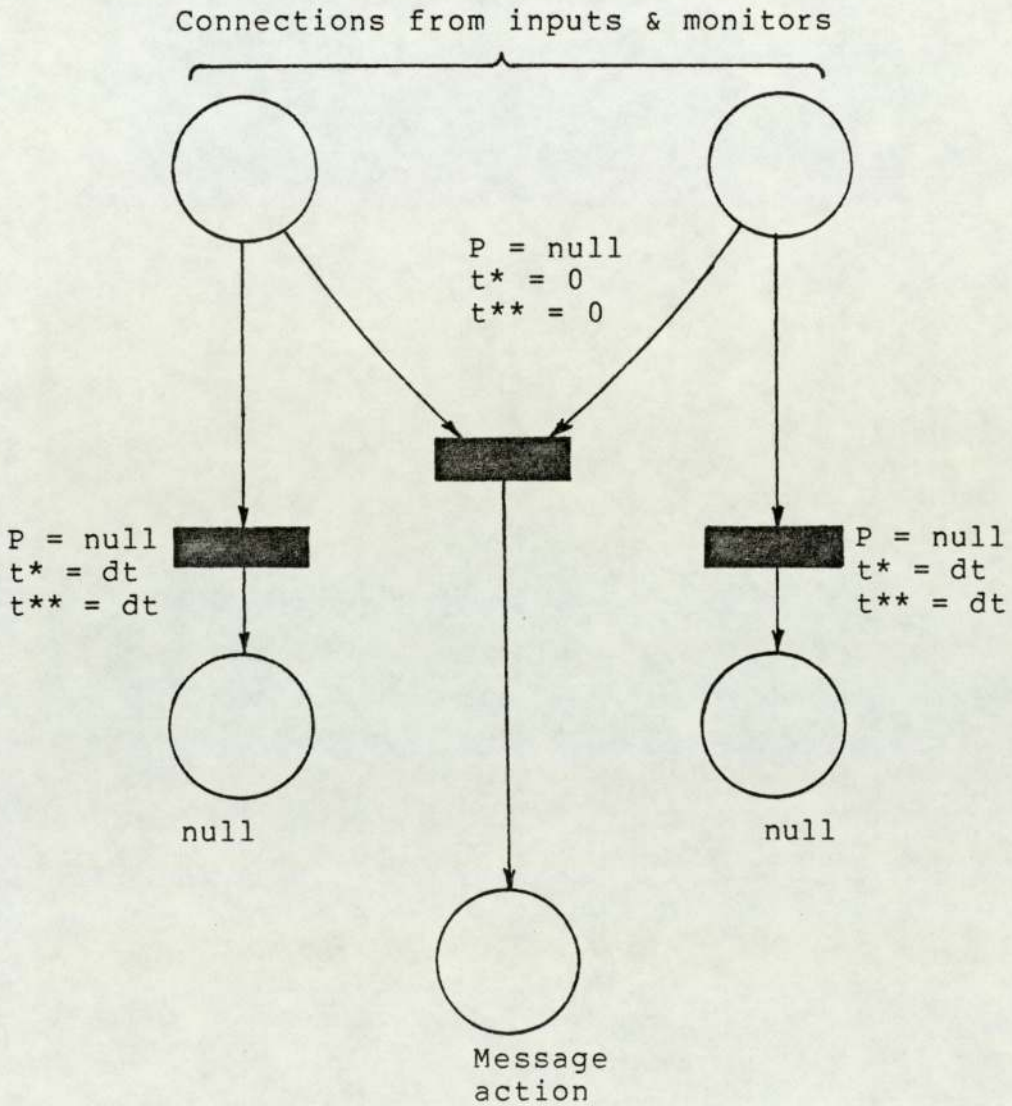


FIGURE C.13 TRANSLATION OF OTHER BEHAVIOUR

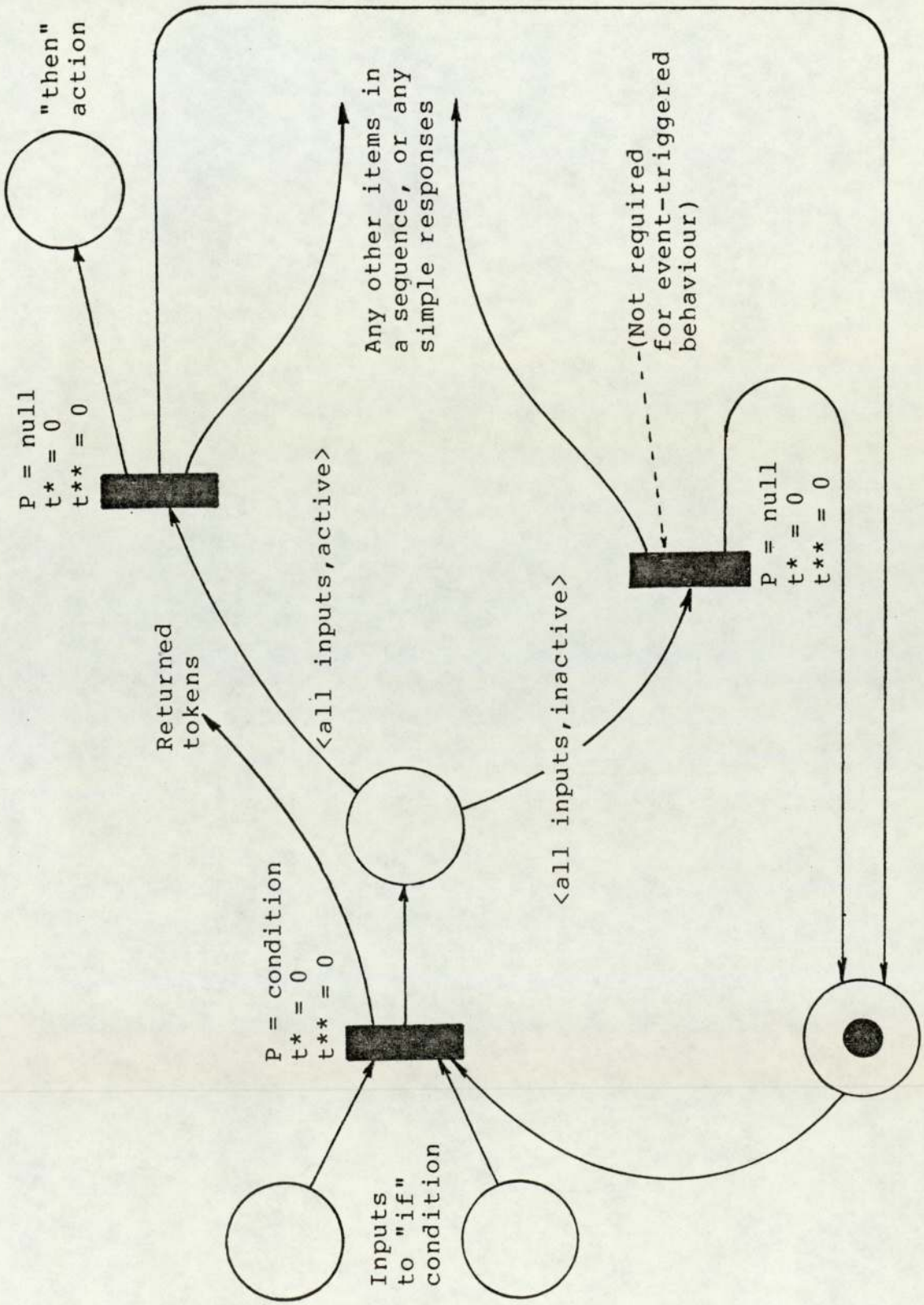


FIGURE C.14 TRANSLATION OF TIMEOUTS

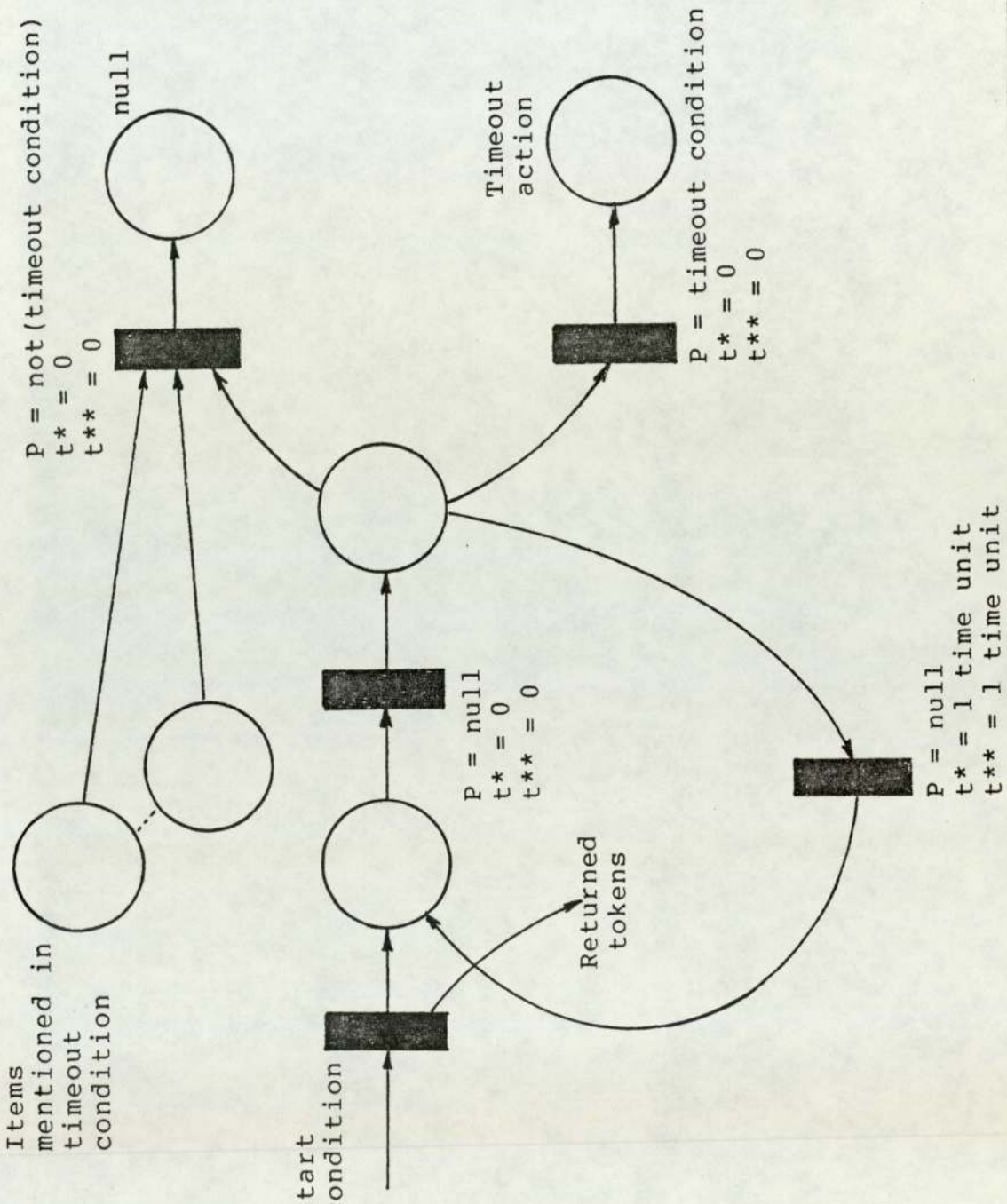
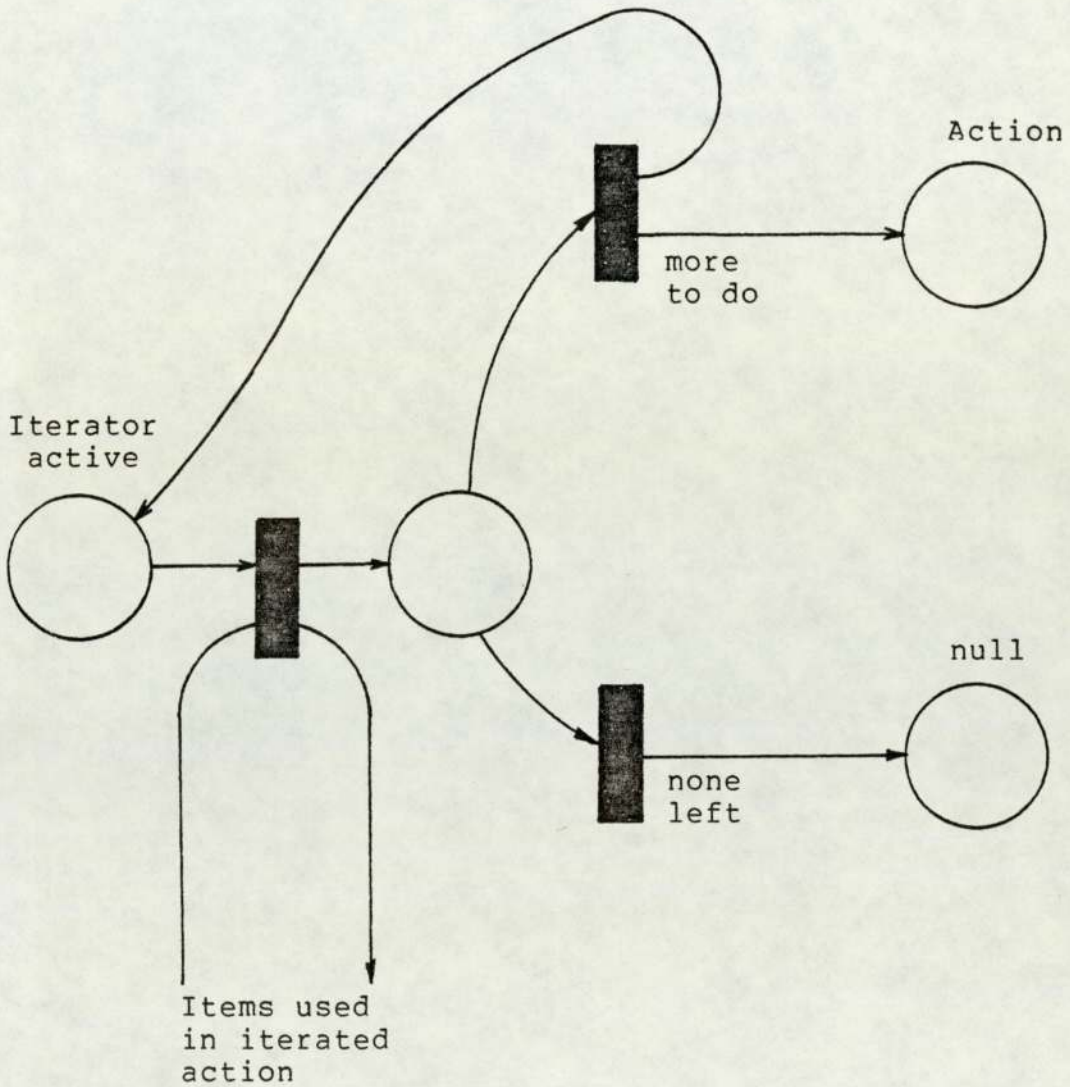


FIGURE C.15 TREATMENT OF ITERATORS



from the Predicate/Transition net and Time Petri net models unchanged. Each transition is labelled with a predicate which controls its firing; this is shown as "P = ...". Also against each transition are its minimum and maximum delays before firing, shown as "t*" and "t**" respectively. The tuples (lists of values) associated with the tokens are shown as lists of names inside angle brackets ("<>") adjacent to the arcs along which they pass.

Each statement is translated into one or more of these patterns, with identifying labels being associated with the places which will connect it to the remainder of the net (see Section C.4, below). The transitions in the net are all treated as timed and given a minimum and maximum firing time; these are initially set to zero and infinity respectively, giving the equivalent to an untimed Predicate/Transition net. For any behaviour statement with a positive time delay, this is placed on the transition which represents the associated response so that the action of any monitors and the storage of the message in the history buffer is treated as instantaneous in all cases.

The arcs in the net fragments are labelled with the format of the tuples which will flow along those arcs (i.e. the structure of the appropriate messages). In order to simplify the labelling process, the total content of a message is always represented in the appropriate tuples even if some of the elements of the message are never used. Similarly, once the time-stamp has been at-

tached by the "observer" this is treated as an extra element in the message and carried everywhere. The conditions in the statements representing the ASL specification become the predicates attached to the transitions of the net model; at this stage any of these conditions which refer to the order of messages in time (e.g. by using "after" or "last") are converted into the equivalent arithmetic conditions upon the time-stamps in the messages. This also involves the translation of continuous messages into instantaneous ones, by considering only their start and end points; any references elsewhere to these signals are then converted into equivalent expressions relating to the interval between the start and end times of the signal. Figure C.16 indicates how these temporal references are translated.

4. Connection

The collection of net fragments produced by the translation process are connected together to form a single net representing the whole system. This is achieved by collapsing all the places which carry identical labels (with two exceptions which are covered below) into a single place for each label. This procedure will only be successful if applied to a specification which has no context-free or context-sensitive errors in it. The exceptions mentioned above relate to "unknown" and "undefined" elements in the specification. If there are multiple uses of these within the specification, it is

FIGURE C.16 TREATMENT OF TEMPORAL OPERATORS

operator	instantaneous	continuous
x at same time as	x.time = y.time	x.time = y.time
y after x	y.time > x.time	n/a
y after start of x	n/a	y > x.start.time
duration of x	n/a	x.end.time - x.start.time
time of x	x.time	x.time
sending x via y	n/a	last message sent via y = start of x
receiving x via y	n/a	last message received via y = start of x

operator	instantaneous or continuous
last x	(x1 in x) and ((? in x) and (x1.time > ?.time))
first x	(x1 in x) and ((? in x) and (x1.time < ?.time))
#n x	(x1 in x) and (y1 in x) and ....(yn-1 in x) and (x1.time < y1.time) and .... (x1.time < yn-1.time) and not ((z in x) and (z.time >= x1.time))

not valid to collapse them down into a single place, as they represent different unknowns.

Interconnections between the different models in the specification act as a relabelling operation, so that each interconnection appears as a single place in the connected net; this place represents any message in transit between the models involved in the connection. Once the net has been fully connected, it is then possible to check that the information flowing out of each place in the net is available to that place (i.e. is contained in the tuples flowing along the arcs into that place).

5. The Firing Rules

This section contains a brief statement of the rules for the firing of transitions in the timed Predicate/Transition net model used here. Its purpose is merely to show how the timing element has been added, and not to provide a full mathematical treatment of this net model.

A Predicate/Transition net has the following constituents (Genrich et al, 1980).

(i) A directed net, (S,T;F), where S is the set of predicates (places), T is the set of transitions and F is the set of arcs (i.e. F is some subset of the union of SxT and TxS).

(ii) A set, U, of operators and predicates.

(iii) A labelling of arcs, assigning to all elements of F a formal sum of n-tuples of variables where n is the

'arity' of the predicate associated with the arc.

(iv) An inscription on transitions, assigning to some elements of T a logical formula built from equality and the operators and predicates given in U. Any variable occuring free in a transition must be present in one or more of the adjacent arcs.

(v) A marking of places with n-tuples (tokens).

(vi) A natural number, K, which is the upper bound for the number of copies of the same item which may occur at a single place.

and the transition rule states that a transition may fire when:

- all input places to the transition carry enough tokens to satisfy the necessary predicates,
- the resulting number of tokens on the output places of the transition will not exceed K after the firing.

In order to extend this untimed net to handle the required time delays, the following additional constituents have to be added (Merlin, 1974).

(vii) Associated with each transition, i, is a tuple, [t*i, t**i]. The value of t*i is the time which must elapse between the conditions of the untimed firing rule (above) becoming true and the firing of the transition, whilst t**i is the maximum time for which firing can be delayed. So, for all i:

- t*i and t**i are real numbers,
- $t^*i, t^{**}i \geq 0$,
- $t^*i < t^{**}i$.

(viii) Added to each tuple (token) is a time value, t, and for any transition, i, with tokens on its input places with time values t1, t2, ... tn, then the time value, t', in the tokens which it puts on its output places is given by:

$$t' = tx + dt$$

where $t^*i \leq dt \leq t^*i$

and $tx = \max(t1, t2, \ldots, tn)$.

The transition rule is also changed by the addition of a third condition, so that a transition will fire if:

- all input places to the transition carry enough tokens to satisfy the necessary predicates,
- those tokens have been present for a period of time equal to or greater than t*i,
- the firing of the transition will not cause the number of tokens on any of the output places to exceed K.

Under this model time does not operate as a continuous variable, but increases irregularly; this is because time is treated as an attribute of the tokens, and is only updated when an event takes place.

6. Semantic Checking of Specifications

Although the net model has been taken to be complete (see Section 1 of this Appendix), this does not necessarily mean that there are practicable methods of ensuring the "correctness" of a specification. Three main problems exist:

(a) even for specifications written in a subset of mathematical logic, the task of proving particular properties of the system may demand human guidance to avoid unbounded searches,

(b) such proofs of correctness are only undertaken for those properties which the specifiers consider important; there is no method for deciding which properties should be shown to be correct,

(c) ASL was purposely designed to be provide expressive power. It permits the specification of functions which cannot be realised (by injudicious use of the "where..." construct), and expects the specified system to contain concurrent activities. Handling these is beyond the capabilities of present program-proving techniques.

The complexity of the net models for most real system specifications may make it impracticable or impossible to analyse the nets for reachability, etc.. Simulation (see Chapter 5.4.4) would then be the only recourse. The problems listed above make it unrealistic to attempt to provide further assistance for formal semantic verification at this time.

APPENDIX D

THE STATIC CHECKING FACILITIES

1. Introduction

The static checking facilities developed to support the trials consist of a syntax analyser, a consistency checker and a cross-reference list generator, as outlined in Chapter 5.2.2. The particular computer programs used are not worth detailed examination, as they were only intended to be sufficient to demonstrate the value of computer-based support. They provide the minimum level of assistance for the practical trials, which are reported in Chapter 6. The following sections therefore discuss only the general structure of the programs, and the format of their inputs and outputs.

2. The Syntax Analyser

2.1. Recursive Descent Analysis

Given the requirement to produce a syntax analyser for a language (ASL) which was still being designed, the use of a compiler-generator (e.g. Johnson, 1979) or a

syntax-driven analyser (e.g. Simpson, 1969) was seen as a way to avoid significant re-programming whenever part of the language syntax was changed. No suitable parser-generator was readily available to the project, so it was decided to write a syntax-driven analyser specifically for ASL. Even though this involved some programming effort, it freed the language from the paradigms of existing high-level languages and, given that the syntax could be restricted to the simplest possible form, involved only a few weeks of computer programming. The analyser was therefore written to perform recursive-descent analysis (Davie & Morrison, 1981); this reduces the complexity of the analyser program at the expense of run-time overheads caused by the extensive use of recursive subroutines.

The basic principle of this method is to treat each production in the syntax as a call to a subroutine which either reads the next token from the input or generates a further call to the subroutine, depending upon the next item in that production. The syntax productions can be held as simple tables, with each row representing a production and the entry in each column being the index of another row (if that entry represents the name of another production) or a call to a primitive operation (such as reading in the next word from the specification being analysed). This permits rapid and easy changes to the syntax of the language by updating the table, whilst not seriously affecting the efficiency of the program.

2.2. The Syntax Rule Format

The only significant disadvantage of the recursive-descent method is that it does not permit repetition and optional items to be represented directly as in BNF (See Appendix B.1). It is instead necessary to use recursion in place of repetition and to introduce extra rules to represent options. For example the BNF production:

```
PARAGRAPH = "[" INTEGER ¦ "." INTEGER ¦ "]".
```

has to become two rules:

```
PARAGRAPH = "[" INTEGER PARAGRAPH_TAIL.
PARAGRAPH_TAIL = "." INTEGER PARAGRAPH_TAIL ¦ "]".
```

In the case of an optional item, such as:

```
A_MESSAGE = ANY_NAME [ CONTENTS ] [ ROUTE ].
```

this has to be translated as:

```
A_MESSAGE = ANY_NAME A_MESSAGE_TAIL.
A_MESSAGE_TAIL = CONTENTS A_MESSAGE_END ¦
                 A_MESSAGE_END.
A_MESSAGE_END = ROUTE ¦ EMPTY.
```

resulting in a much larger number of rules than in the BNF equivalent, and these rules are also much more difficult to understand. For this reason BNF was used in the definition of ASL in Appendix B.

The final form of the rules input to the syntax analyser is shown in Figure D.1. There are five sections to these, as follows.

(a) One or more lines of text, which are read by the program and then printed as a heading at the top of the output listing (see Section D.2.5).

FIGURE D.1 FORMAT OF THE SYNTAX RULES

```
ASL Version 6.                                                    (a)

#
SPECIFICATION                                                     (b)
achieve, action, after,                                           }
   :     :     :                                                  } (c)
where, while, within.                                             }

A_MESSAGE = ANY_NAME A_MESSAGE_TAIL.                              }
A_MESSAGE_TAIL = CONTENTS A_MESSAGE_END, ROUTE,                   }
                 EMPTY.                                           } (d)
A_MESSAGE_END = ROUTE, EMPTY.                                     }
    :    :      :                                                 }
VALUE = ANY_QUALIFIED_NAME, LOOSE_END, NUMBER.                    }

#

A_MESSAGE = SYSTEM_STATEMENT.                                     }
A_MESSAGE_TAIL = SYSTEM_STATEMENT                                 } (e)
    :         :                                                   }
VALUE = SYSTEM_STATEMENT                                          }

#
```

(b) The name of the syntax rule which represents the top level of the syntax definitions.

(c) A list of all the words which are part of the language, and so cannot be re-defined within a specification.

(d) The syntax rules.

(e) Alternative rules, to be used in attempts to recover from syntax errors.

Of these, only the additional rules for error recovery (item (e) above) are explained further, in Section D.2.3 below.

2.3. Error Recovery

Only the simplest form of error recovery has been provided in the analyser; this is of the type which is sometimes called "panic mode" (Aho & Ullman, 1977). Once an error has been detected in the input, the analyser program skips over the specification text until it finds the start of the next paragraph (i.e. a "["). Syntax analysis can then recommence at the start of a new statement, but this requires that the analyser be told where in the syntax to restart. Hence, the syntax rules contain an extra part, which gives for each production the name of a suitable point at which to attempt to restart. These points must be productions which have a paragraph number as their first item, to match the point at which the analyser will restart.

Figure D.2 contains an example of the listing pro-

duced by the syntax analyser, showing an error recovery action. Two sets of messages are inserted into the listing.

(a) The first group indicate the position of the error, by printing an asterisk ("*") beneath the first character which has not been accepted. On the line below this is printed the name of the syntax rule and the item within that rule at which the error was detected.

(b) The second group indicates, again by an asterisk, the point at which the syntax analysis was restarted.

Thus, all the characters from the first asterisk up to (but not including) the second asterisk have been ignored by the analyser. This can lead to the reporting of spurious errors in the remainder of the specification if the portion which was ignored did contain some important phrase (such as the end of one model and the start of another). For most cases it does result in an acceptable recovery from the error, and permits the analysis of the remainder of the specification. More complex error recovery techniques (e.g. James & Partridge, 1973) were considered; however, these involved considerable extra programming effort to produce only a limited improvement in the level of service to the user.

2.4. The Input to the Analyser

Specifications are prepared using the standard IBM text editor (IBM, 1978) provided as part of the IBM Time

```
A.S.L. Version 6.
-----------------------------------------
000010 VENDING_SYSTEM / TEA VENDING MACHINE  is
000020
000030   [1] created from
000040      [1.1] 1 tea_machine
000050      [1.2] 1 user
000060
000070   [2] connections
000080      [2.1] user.fingers to tea_machine.coinslot
000090      [2.2] user.fingers to tea_machine,selector
ERROR:                                        *
IN RULE A_CONNECTION     WHEN CHECKING FOR STRING .
000100      [2.3] user.fingers to tea_machine.refund_button
            *   RESTART POINT
000110      [2.4] tea_machine.refund_chute to user.hand
000120      [2.5] tea_machine.dispenser to user.hand
000130      [2.6] tea_machine.status_light(select_range) to user.eyes
000140
000150   [3] messages
000160      [3.1] REFUND:undefined
000170      [3.2] STATUS is
000180         [3.2.1] DRINK:select_range
000190         [3.2.2] VALUE:boolean
000200      [3.3] MONEY is
000210         [3.3.1] SIZE:coin_size
000220         [3.3.2] WEIGHT:grams
000230      [3.4] REJECT is
000240         [3.4.1] SIZE:coin_size
000250         [3.4.2] WEIGHT:grams
```

FIGURE D.2 THE LISTING PRODUCED BY THE ANALYSER

Sharing Option interactive computing service. This offers simple program editting facilities which are not specific to any particular language, but does not include formatting capabilities. This falls far short of the features suggested in Chapter 5.3, but was readily available and did not require the development of any computer programs.

2.5. The Output from the Analyser

The analyser produces two outputs, as follows.

(a) A listing of the specification

The analyser lists the specification text as it is read in, printing it in the format shown in Figure D.2. This displays each line of text exactly as typed, but with the addition of a line of asterisks as separators between the blocks of text. It also shows any error messages, as explained in Section D.2.3 above, and gives at the end of the print a count of the number of errors detected.

(b) Tables for input to the consistency checker

The second output from the analyser is not intended to be presented to the writer of the specification, as it is merely a set of table entries which are passed to the checker program (see Section D.3, below). This is done automatically, as the manual extraction of this information for the checking process would be likely to introduce errors which did not exist in the original specification.

3. The Consistency Checker

3.1. Method

The checking to be performed was explained in Chapter 5.2.2; it consists of such things as ensuring that every name used in the specification has been properly defined. The checker program must therefore represent a body of rules, each of which is nearly independent of the others. Initial attempts to write a Pascal program to perform this function showed that this was a significant programming task in relation to the amount of time available. It was therefore decided to use a higher-level language called Prolog (Clocksin & Mellish, 1981) instead of Pascal, as Prolog directly supports the programming of functions as sets of rules.

As a consequence of this decision, the checker program consists of less than 100 lines of Prolog code (see Section D.3.2 below) and only took a few days to develop. However, Prolog is an interpreted language and makes relatively inefficient use of computer time when compared with a Pascal program to do the same job. This has not caused any operating difficulties for specifications of the size created so far, but may make this particular implementation of the checking system unacceptable in the long-term.

3.2. The Rule Format

Individual rules are expressed as Prolog terms, using the standard Prolog syntax. To get the checker to produce helpful error messages, the rules define the conditions which are invalid, as in the following example.

The rule: If, in paragraph P of model M, there is an action which sends a message, X, via an interface, Y, then X must have been defined as a message.

is encoded as:

```
error('message not defined',M,P,X,Y) :-
        send(M,P,X,Y),
        not(messages(_,_,X)).
```

where "send" and "messages" are the names of tables, as described in Section D.3.3, below. Figure D.3 contains a complete listing of the rules in Prolog; these make reference to the following functions, whose definitions have not been included.

(a) basic_type(A), which is true if A is one of the defined basic types in ASL (boolean, character, decimal, integer, interface and message).

(b) in_scope_of(A,B), which determines whether the paragraph number B is within the scope of paragraph number A.

(c) interface(A,B,C), which checks to see if C is an input, output or bothway interface of model A.

(d) loose_end(A), which is true if A is equal to "undefined", "unknown" or "dont care".

(e) same_type(A,B), which determines whether the messages

FIGURE D.3 THE STATIC CHECKING RULES

```
serious_error('duplicate definition',A,B,[C]):-definitions(A,B,C),
    definitions(A,D,C),B¬==D,samescope(B,D).
serious_error('model not defined',A,B,[C]):-society(A,B,C),
    not(models(C)).
serious_error('no inputs to',A,[],[]):-models(A),not(inputs(A,_,_)),
    not(bothways(A,_,_)).
serious_error('no outputs to',A,[],[]):-models(A),not(outputs(A,_,_)),
    not(bothways(A,_,_)).
serious_error('interface not connected',A,B,[C]):-interface(A,B,C),
    not(connections(_,_,[A,C],_)),not(connections(_,_,_,[A,C])).
serious_error('loopback',A,B,[C,D,E,F]):-connections(A,B,[C|[D]],
    [E|[F]]),C==E.
serious_error('one end undefined',A,B,[C,D,E,F]):-
    connections(A,B,[C|[D]],[E|[F]]),(not(interface(C,_,D));
    not(interface(E,_,F))).
serious_error('invalid connection',A,B,[C,D,E,F]):-
    connections(A,B,[C|[D]],[E|[F]]),(inputs(C,_,D),inputs(E,_,F);
    outputs(C,_,D),outputs(E,_,F)).
serious_error('undefined message',A,B,[C,D,E]):-txmit(A,B,C),
    not(messages(_,_,C)).
serious_error('undefined message component',A,B,[C]):-
    msg_compt_ref(A,B,C),not(message_compt(_,_,C)),
    not(temp_var(A,B,C)).
serious_error('undefined interface',A,B,[C]):-txmit(A,B,_,C),
    not(interface(A,_,C)).
serious_error('function not defined',A,B,[C]):-opn_uses(A,B,C),
    not(operations(A,_,C)),not(operations(D,_,C),system(D)).
serious_error('illegal update',A,B,[C]):-updates(A,B,C),
    not(definitions(A,D,C),in_scope_of(D,B)).
```

contd.

FIGURE D.3 continued

```
serious_error('undefined name',A,B,[C]):-references(A,B,C),
    not(definitions(A,E,C),in_scope_of(E,B)),not(definitions(D,_,C),system(D)),
    not(temp_var(A,B,C)),not(loose_end(C)),
    not(constant(A,_,C)),not(constant(E,_,C),system(E)).
serious_error('message in wrong direction',A,B,[D,E]):-
    (send(A,B,[C|[D]],E),inputs(A,_,E);
     receive(A,B,[C|[D]],E),outputs(A,_,E)).
serious_error('message type conflict',A,B,[D,E,F,G]):-
    txmit(A,B,[C|[D]],E),(connections(_,_,[A|[E]],[F|[G]]);
     connections(_,_,[F|[G]],[A|[E]])),txmit(F,_,[H|[D]],G),
    not(same_type(H,C)).
serious_error('undefined type name',A,B,[C]):-type_ref(A,B,C),
    not(basic_type(C)),not(loose_end(C)),not(definitions(A,E,C),
    in_scope_of(E,B)),not(definitions(D,_,C),system(D)).
serious_error('func parm not defined',A,B,[C]):-(ip_parms(A,B,C);
    op_parms(A,B,C)),not(definitions(A,D,C),in_scope_of(D,B)).
serious_error('duplicate components',A,B,[C]):-component_def(A,B,C),
    component_def(A,D,C),B¬==D,samescope(B,D).
serious_error('dupl message component',A,B,[C]):-message_compt(A,B,C),
    message_ccmpt(A,D,C),B¬==D,samescope(B,D).

an_error('model not used',A,[],[]):-models(A),not(society(_,_,A)).
an_error('message never used',A,B,[C]):-messages(A,B,C),
    not(txmit(_,_,[_|[C]],_)).
an_error('function not used',A,B,[C]):-operations(A,B,C),
    (system(A),not(opn_uses(_,_,C)),not(references(_,_,C));
     not(opn_uses(A,_,C)),not(references(A,_,C))).
an_error('definition never used',A,B,[C]):-definitions(A,B,C),
    system(A),not(references(_,_,C)),
    not(type_ref(_,_,C)).
```

contd.

```
an_error('name not used',A,B,[C]):-definitions(A,B,C),not(system(A)),
     not((updates(A,D,C);references(A,D,C));type_ref(A,D,C)),
     in_scope_of(B,D)).

an_error('message not received',A,B,[D,E,F,G]):-send(A,B,[_|[D]],E),
     not(loose_end(D)),not(loose_end(E)),(connections(_,_,[A|[E]],
     [F|[G]]);connections(_,_,[F|[G]],[A|[E]])),
     not(receive(F,_,[_|[D]],G)).
an_error('message type inconsistent',A,B,[C,D]):-txmit(A,B,[C|[D]],E),
     txmit(A,_,[G|[D]],_),not(same_type(C,G)).
an_error('temporary not used',A,B,[C]):-temporaries(A,B,C),
     not(references(A,D,C),in_scope_of(D,E)).
an_error('func output not given a value',A,E,[C]):-op_parms(A,B,C),
     not(updates(A,D,C),in_scope_of(D,B)).

warning('component never mentioned',A,B,[C]):-component_def(A,B,C),
      system(A),not(component_ref(_,_,C)).
warning('message compt never mentioned',A,B,[C]):-message_compt(A,B,C),
     not(msg_compt_ref(_,_,C)),not(component_ref(_,_,C)).
warning('local component never mentioned',A,B,[C]):-
     component_def(A,B,C),not(system(A)),not(component_ref(A,_,C)).
warning('message never switched on',A,B,[C,D]):-send(A,B,[endcont|[C]],
     D),not(send(A,_,[startcont|[C]],D)).
warning('message never switched off',A,B,[C,D]):-
     send(A,B,[startcont|[C]],D),not(send(A,_,[endcont|[C]],D)).
warning('constant never mentioned',A,B,[C]):-constant(A,B,C),
     (system(A),not(references(_,_,C));not(system(A)),not(references(A,_,C))).
```

FIGURE D.3 continued

A and B are of the same message type (i.e. both are instantaneous or both are continuous).

(f) temp_var(A,B,C), which is true if C is a temporary variable (i.e. a pattern-matching variable; a name preceded by "?") which was defined in model A such that paragraph number B is within the scope of that definition.

(g) txmit(A,B,C,D), which checks if C is a message which was sent or received by model A via interface D, and that D is not a "loose_end" (see (d) above).

3.3. The Input Format

To simplify the checker program as far as possible, its input format was restricted to that of standard Prolog terms. The syntax analyser was therefore made to produce a list of such terms, containing details of the names found in the specification text and the mode in which they are used. The general format for these terms, expressed in BNF (see Appendix B.1), is:

TERM = TABLE_NAME "(" MODEL_NAME "," PARAGRAPH_NUMBER
{ "," OTHER_NAMES } ")".

although the full-stops (".") inside the paragraph numbers had to be replaced by commas to conform to Prolog syntax. As an example:

messages(vending_system, [1,1] , coin).

states that "coin" was defined as a "message" in paragraph [1.1] of "vending_system". There are 25 of these tables, and their formats are shown in Figure D.4. It

should be remembered that these tables are not produced by the specification writer, but are an automatic output from the syntax analyser and therefore do not need to be in a readable form.

3.4. The Output from the Checker

The checker program produces a simple list of the errors which have been detected, quoting the name of the text block and the number of the paragraph which contains the error. It also shows sufficient additional information to identify the error within the paragraph. Figure D.5 shows an annotated example of a few such messages. To aid the user of this program, the errors are listed in three categories.

(a) Serious errors, such as the use of names which have not been defined. These are classified as serious because they invalidate some portion of the specification.

(b) Errors, such as the definition of a name which is never used. These may not invalidate any part of the specification, but do indicate incompleteness.

(c) Warnings, such as the lack of references to a component of a message or a constant. As ASL permits structured messages to be assigned values at the message level, it may be valid for some components never to be mentioned individually. However, the checker program produces a warning message so that the specification writer is aware of this.

FIGURE D.4 THE TABLES USED IN STATIC CHECKING

(a) For the system; one record generated for a specification.

"system" "(" SYSTEM_NAME ")" "." .

(b) For monitors; one record per monitor statement in the specification.

"monitor" "(" BLOCK_NAME "," PARAGRAPH_NUMBER ")" "." .

(c) For messages sent and received; one record per send or receive action

("receive" ¦ "send")
"(" BLOCK_NAME "," PARAGRAPH_NUMBER ","
"[" MESSAGE_TYPE "," MESSAGE_NAME "]" ","
INTERFACE_NAME ")" "." .

where MESSAGE_TYPE indicates whether it is an instantaneous message or the start or end of a continuous message.

contd.

FIGURE D.4 continued

(d) For the definition and use of names.

```
TABLE_NAME "(" BLOCK_NAME "," PARAGRAPH_NUMBER ","
          A_NAME ")" "." .
```

where TABLE_NAME can take the following values:

Table Name	Record per
bothways	bothway interface definition
component_def	definition of a component of a name
component_ref	use of a component name
constant	name used as a constant in a definition
definitions	definition of a top-level name
inputs	input interface definition
ip_parms	argument in an operation definition
message_compt	definition of a component of a message
messages	definition of a top-level message name
models	definition of a model
msg_compt_ref	use of a message component name
op_parms	a result in an operation definition
operations	the definition of an operation
opn_uses	a use of an operation
outputs	output interface definition
references	indeterminate reference to a name
society	model name being used in system block
temporaries	use of a name after a "?"
type_ref	use of a name as a data type
updates	assignment of a value to a name

contd.

FIGURE D.4 continued

(e) For interconnections.

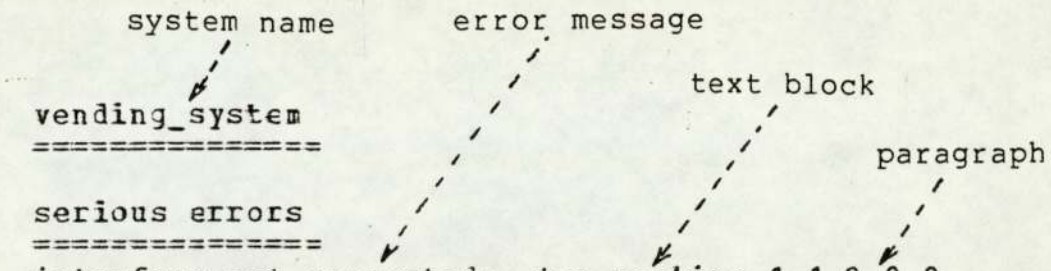
```
"connections" "(" BLOCK_NAME "," PARAGRAPH_NUMBER ","
    "[" MODEL_NAME "," INTERFACE_NAME "]" ","
    "[" MODEL_NAME "," INTERFACE_NAME "]" ")" "." .
```

Note:

In all cases, the format for paragraph numbers is:

```
PARAGRAPH_NUMBER = "[" INTEGER PARAGRAPH_BODY.
PARAGRAPH_BODY = "," INTEGER PARAGRAPH_BODY |
                 "]" .
```

FIGURE D.5 ERROR MESSAGES FROM STATIC CHECKING




```
vending_system
==============

serious errors
==============
interface not connected:  tea_machine 1.1.0.0.0.
  coin_slot

one end undefined:  vending_system 2.1.0.0.0.
  user  fingers  tea_machine  coinslot

undefined message component:  user 5.0.0.0.0.
  selected

message type conflict:  tea_machine 7.1.0.0.0.
  status  status_light  user  eyes

message type conflict:  tea_machine 7.1.0.0.0.
  status  status_light  user  eyes

message type conflict:  tea_machine 7.2.0.0.0.
  status  status_light  user  eyes



  errors
==============
message never used:  vending_system 3.6.0.0.0.
  fillup

message not received:  user 4.0.0.0.0.
  money  fingers  tea_machine  coinslot

message not received:  user 4.0.0.0.0.
  money  fingers  tea_machine  selector

message not received:  user 4.0.0.0.0.
  money  fingers  tea_machine  refund_button
```

4. Cross-reference Listing

The cross-reference listing is produced by a simple Pascal program; this extracts all the occurences of names from the table entries used by the consistency checker and sorts them into alphabetic sequence. Figure D.6 shows an example of the listing, which has columns for the name, the type of appearance being reported (see below), and the text block and paragraph number in which the name appears. The types of appearance (the second column on the listing) are:

(a) DEC. The declaration of a name which is not an operation or an interface.

(b) OPN. The declaration of an operation.

(c) REF. A reference to a name, other than to update its value.

(d) IFC. The declaration of an interface.

(e) UPD. A reference to a name which involves updating its value.

as this information may help in the task of making changes to the specification (see Chapter 5.3.2).

FIGURE D.6 THE CROSS-REFERENCE LISTING

CROSS-REFERENCE LISTING FOR vending_system

NAME	TYPE	BLOCK	PARAGRAPH
boolean	REF	vending_system	3. 2. 2.
c	REF	vending_system	5.
cm	REF	vending_system	5.
cms	REF	vending_system	5.
coffee	DEC	vending_system	4. 1.
coin_size	REF	vending_system	3. 3. 1.
		vending_system	3. 4. 1.
		tea_machine	5. 1.
coin_slot	IFC	tea_machine	1. 1.
	REF	tea_machine	8.
coinslot	REF	vending_system	2. 1.
cs	REF	vending_system	5.
cup	DEC	vending_system	4. 5.
d	REF	tea_machine	1.
	DEC	tea_machine	1.
	REF	tea_machine	1. 1.
		tea_machine	1. 2.
dispense	DEC	vending_system	3. 7.
	REF	user	7.
		tea_machine	1.
		tea_machine	1. 1.
		tea_machine	6.
		tea_machine	11.
dispenser	REF	vending_system	2. 5.
		tea_machine	1.
	IFC	tea_machine	2. 2.
drink	DEC	vending_system	3. 2. 1.
eyes	REF	vending_system	2. 6.
	IFC	user	1. 1.
	REF	user	3.
fillup	DEC	vending_system	3. 6.
	REF	tea_machine	6.
fingers	REF	vending_system	2. 1.
		vending_system	2. 2.
		vending_system	2. 3.
	IFC	user	2.
	REF	user	4.
		user	5.
		user	6.
grams	REF	vending_system	3. 3. 2.

APPENDIX E

AN EXAMPLE SPECIFICATION

1. Introduction

The purpose of this example is to display the general appearance and style of ASL specifications, so a simple system was chosen in order to minimise the need for introductory explanation in English. Additionally, to demonstrate the use of the existing syntax analyser and checker facilities (see Appendix D), the specification is shown with one syntax error and a number of other errors. This appendix therefore contains a brief introduction to the system (section E.2), the listing of the specification as produced by the syntax analyser (section E.3) and the error messages from the checker (section E.4). Both of the listings are supplemented by comments on some of the points demonstrated.

2. The System

The system to be specified is a tea-vending machine, treated at the level of the interface to the customer. Thus, the ASL specification consists of two models, one

for the vending machine (the system model) and one for the user (the environment model). In outline, the vending machine is intended to operate as follows.

(a) The machine supplies a range of drinks, and there is some method (e.g. buttons or dials) by which the user may select a particular drink.

(b) For each drink there is a signal (e.g. a lamp) which indicates if the machine contains sufficient ingredients to provide that drink. This is intended to be a positive indication, so that lack of ingredients and lamp failure should both be visible to the user.

(c) The user inserts coins into the machine to make up the price of the drink required, and then operates the selection mechanism to request that drink.

(d) The machine will only accept a defined range of coins, and will immediately reject any other denominations or any damaged coins.

(e) The user must be provided with a facility to terminate the transaction prematurely (e.g. a refund button), and should then receive back all the coins inserted since the start of the transaction.

(f) If the user requests a drink when the value of the coins inserted is not equal to the price of that drink, then all the coins should be refunded and no drink provided.

The ASL specification attempts to capture these concepts without introducing unnecessary constraints upon such things as the range of drinks or the number of acceptable denominations of coins.

3. The ASL Specification

As mentioned in Section E.2, there are two models in this specification; thus, with the addition of the system block, this produces three blocks of text. These are shown in outline in Figure E.1, which also identifies the purpose of the main paragraphs within each of these blocks. The specification appears in Figure E.2 in the form of the listing produced by the syntax analyser; the line numbers down the left-hand side of the listing are used in the following comments to identify the use of some features of the language.

(a) (Line 000010) "VENDING_SYSTEM" appears in capital letters for its introduction (i.e. its definition). The words in capitals after the oblique stroke ("/") are a comment, to give a brief description of the system.

(b) (Lines 000150 to 000280) The messages include both simple and structured definitions. Once again, the names being defined appear in capitals, whilst the data type names (which are defined elsewhere) appear in lower case letters.

(c) (Lines 000380 to 000450) "SELECT_RANGE" is used in the system model, the environment and in the definition of messages, so it is defined only once in the system block. Its data type is a list of all the possible values which it can take, enclosed in braces ("{}").

(d) (Line 000580) The word "unknown" is used to indicate

FIGURE E.1 THE STRUCTURE OF THE EXAMPLE SPECIFICATION

Heading or Paragraph Number		Comments
VENDING_SYSTEM is		The system block.
[1]	Statement of models and	
[2]	their interconnections.	
[3]	Definition of common items,	
:	used in both models.	
[5]		
end of vending_system		
USER is		The environment model.
[1]	Interfaces.	
[2]		
[3]	Responses to particular	
:	messages.	
[8]		
end of user		
TEA_MACHINE is		The system model.
[1]	Interfaces.	
[2]		
[3]	Properties of the model,	
:	known only to the model.	
[5]		
[6]	Responses to situations	
:	rather than messages.	
[7]		
[9]	Responses to particular	
:	messages.	
[10]		
[11]	The definition of an operation.	
end of tea_machine		

FIGURE E.2 THE EXAMPLE SPECIFICATION

```
A.S.L. Version 6.
-------------------------------------
000010 VENDING_SYSTEM / TEA VENDING MACHINE  is
000020
000030    [1] created from
000040        [1.1] 1 tea_machine
000050        [1.2] 1 user
000060
000070    [2] connections
000080        [2.1] user.fingers to tea_machine.coinslot
000090        [2.2] user.fingers to tea_machine.selector
000100        [2.3] user.fingers to tea_machine.refund_button
000110        [2.4] tea_machine.refund_chute to user.hand
000120        [2.5] tea_machine.dispenser to user.hand
000130        [2.6] tea_machine.status_light(select_range) to user.eyes
000140
000150    [3] messages
000160        [3.1] REFUND:undefined
000170        [3.2] STATUS is
000180            [3.2.1] DRINK:select_range
000190            [3.2.2] VALUE:boolean
000200        [3.3] MONEY is
000210            [3.3.1] SIZE:coin_size
000220            [3.3.2] WEIGHT:grams
000230        [3.4] REJECT is
000240            [3.4.1] SIZE:coin_size
000250            [3.4.2] WEIGHT:grams
000260        [3.5] REQUEST_DRINK:select_range
000270        [3.6] FILLUP:ingredients
000280        [3.7] DISPENSE:ingredients
000290
```

contd.

FIGURE E.2 continued

```
000300   [4] INGREDIENTS is
000310         [4.1] COFFEE:kilos
000320         [4.2] TEA:kilos
000330         [4.3] SUGAR:kilos
000340         [4.4] MILK:litres
000350         [4.5] CUP:integer
000360         [4.6] WATER:litres
000370
000380   [5] SELECT_RANGE : { cms / COFFEE WITH MILK AND SUGAR,
000390                        cm / COFFEE WITH MILK,
000400                        cs / COFFEE WITH SUGAR,
000410                         c / COFFEE,
000420                       tms/ TEA WITH MILK AND SUGAR,
000430                        tm / TEA WITH MILK,
000440                        ts / TEA WITH SUGAR,
000450                         t / TEA }
000460
000470 end of vending_system
       ***************************************************
000480
000490
000500 USER is
000510
000520   [1] input
000530         [1.1] EYES
000540         [1.2] HAND
000550
000560   [2] output FINGERS
```

contd.

```
000570
000580   [3] on status via eyes then unknown
000590   [4] on unknown then send
000600         money with (size=unknown) & (weight=unknown) via fingers
000610   [5] on unknown then send
000620         request_drink with selected in select_range via fingers
000630   [6] on unknown then send refund via fingers
000640   [7] on dispense via hand then unknown
000650   [8] on reject via hand then unknown
000660
000670 end of user
```

****

```
000680
000690
000700 TEA_MACHINE is
000710
000720   [1] input
000730      [1.1] COIN_SLOT
000740      [1.2] SELECTOR
000750      [1.3] REFUND_BUTTON
000760
000770   [2] output
000780      [2.1] REFUND_CHUTE
000790      [2.2] DISPENSER
000800      [2.3] STATUS_LIGHT(select_range)
000810
000820   [3] STOCK : ingredients
000830
```

FIGURE E.2 continued

contd.

```
000840   [4] SELECTIONS(select_range) is
000850      [4.1] RECIPE :ingredients
000860      [4.2] PRICE :pence
000870
000880   [5] VALID_COINS(undefined) is
000890      [5.1] SIZE :coin_size
000900      [5.2] VALUE :pence
000910
000920   [6] stock is sum(all fillup) - sum(all dispense)
000930
000940   [7 / MONITOR STATUS OF INGREDIENTS ]
000950      [7.1] whenever stock < selections(?x).recipe then (
000960                        stop sending status with value=true,
000970                        start sending status with value=false)
000980                              via status_light(x)
000990
001000      [7.2] whenever stock >=selections(?x).recipe then (
001010                        stop sending status with value=false,
001020                        start sending status with value=true)
001030                              via status_light(x)
001040
001050   [8 / CHECK COIN] on money with size=?x via coin_slot then
001060                              do nothing
001070         unless not (x in valid_coins.size) when
001080                     send reject with size=x via refund_chute
001090
001100   [9 / REFUND] cn refund via refund_button  refund_money
ERROR:                                           *
IN RULE BEHAVICUR        WHEN CHECKING FOR STRING then
001110
001120   [10] within 1 to 2 second of  request_drink=?d
         *   RESTART POINT
001130                                          via selector
```

FIGURE E.2 continued

contd.

```
001140                then send dispense=selections(d).recipe
001150                                          via dispenser

001160           unless [10.1 / WRONG MONEY] sum(?z) ¬=
001170                              selections(d).price
001180                         where
001190                          (z=money.value via coinslot after
001200                            max((time of last dispense via dispenser),
001210                                (time of last refund via refund_chute)))
001220                           & (money.size=?x) & (x in valid_coins.size)
001230                                          when refund_money
001240                    [10.2 / DRINK NOT AVAILABLE] sending status with
001250                               value=false via status_light(d)
001260                                     when refund_money
001270
001280   [11] operation REFUND_MONEY is
001290                  for all money with (size=?z) &
001300                      (z in valid_coins.size) via coin_slot
001310                       after max((time of last dispense via dispenser),
001320                                 (time of last refund via refund_chute))
001330                    then send reject with size=z via refund_chute
001340
001350 end of tea_machine

       ************************************************************

001360 ....
-------------------------------------------
         1 ERRORS.
       CHECKING COMPLETED SUCCESSFULLY
```

FIGURE E.2 continued

some information which will not become available. In this case it is because the behaviour of the user must be treated as random.

(e) (Line 000920) "stock" is a continuously updated value; it is defined as a relationship between the amount of ingredients loaded into the machine ("fillups") and the amount of ingredients dispensed. All three names in the equation ("stock", "fillup" and "dispense") are structures with six components (as defined in type "ingredients" in lines 000300 to 000360); they can be used in this way because they have identical structure and component names.

(f) (Lines 000940 to 000960) This expression uses "whenever" to continuously monitor for a particular condition (the stock of any ingredient falling below the amounts required in the recipe for any of the drinks). It uses a pattern-matching variable ("?x") to stand for "any drink", so producing a succinct specification of the required behaviour for all drinks and all ingredients in a single statement.

(g) (Lines 001050 to 001080) An example of the description of a direct response to a stimulus, written in the "on...then..." form. It also demonstrates the use of "unless" to deal with exceptions (in this case, the rejection of invalid coins).

(h) (Line 001100) A syntax error, due to the omission of the word "then" between the stimulus and the appropriate response. This cause the specification text to be skipped up to the restart point at the beginning

of line 001120.

(i) (Lines 001280 to 001330) Because the action of refunding all the money inserted thus far occurs in three situations (lines 001100, 001230 and 001260), this behaviour has been defined as an operation. Note that this operation is not a function, as it has has no arguments and returns no result.

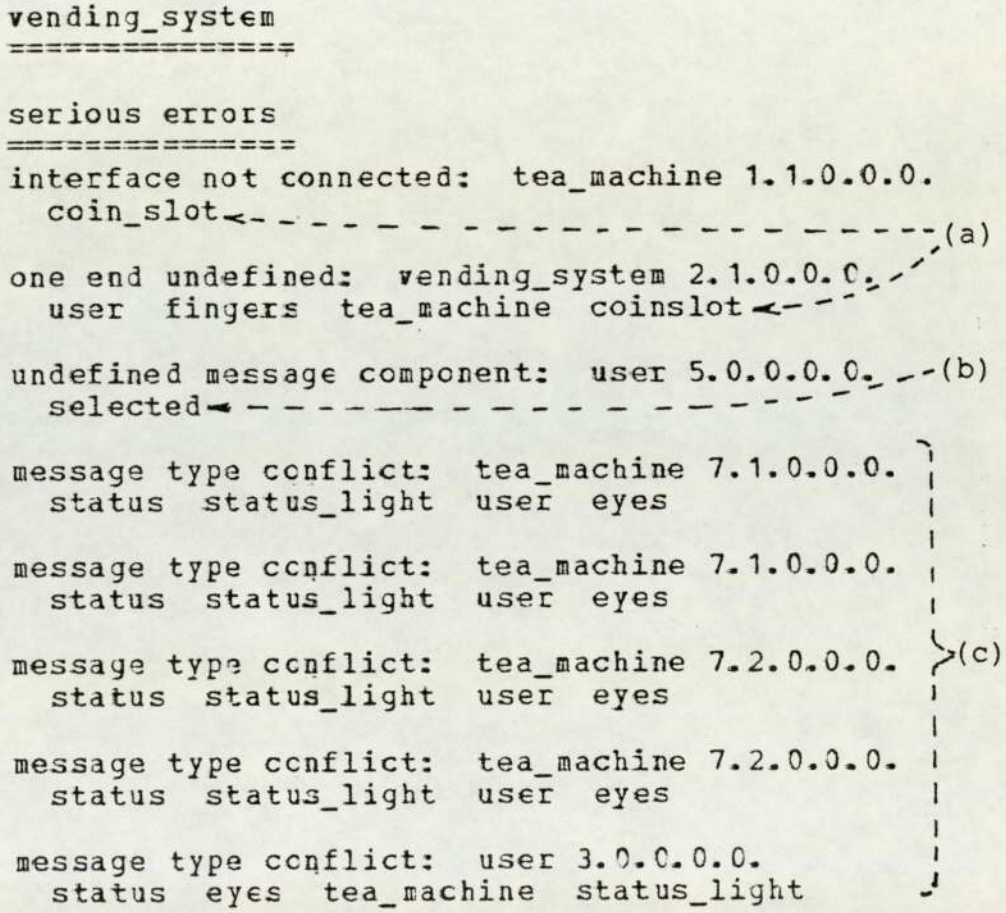
This example is not in any way representative of the specifications produced at GEC Telecommunications Ltd.; however, as the above comments show, it does demonstrate some of the power of the language. Any more realistic example would have been significantly larger, and would have required a considerable amount of introduction in English to provide the necessary background information.

4. Errors in the Specification

After the correction of the one syntax error, the specification was subjected to the static checks (described in Appendix D.3). The serious errors and errors which were identified appear in the listing in Figure E.3; the warnings have not been included in order to reduce the size of the figure. Some of the error messages have been annotated with letters which refer to the comments in the paragraphs below.

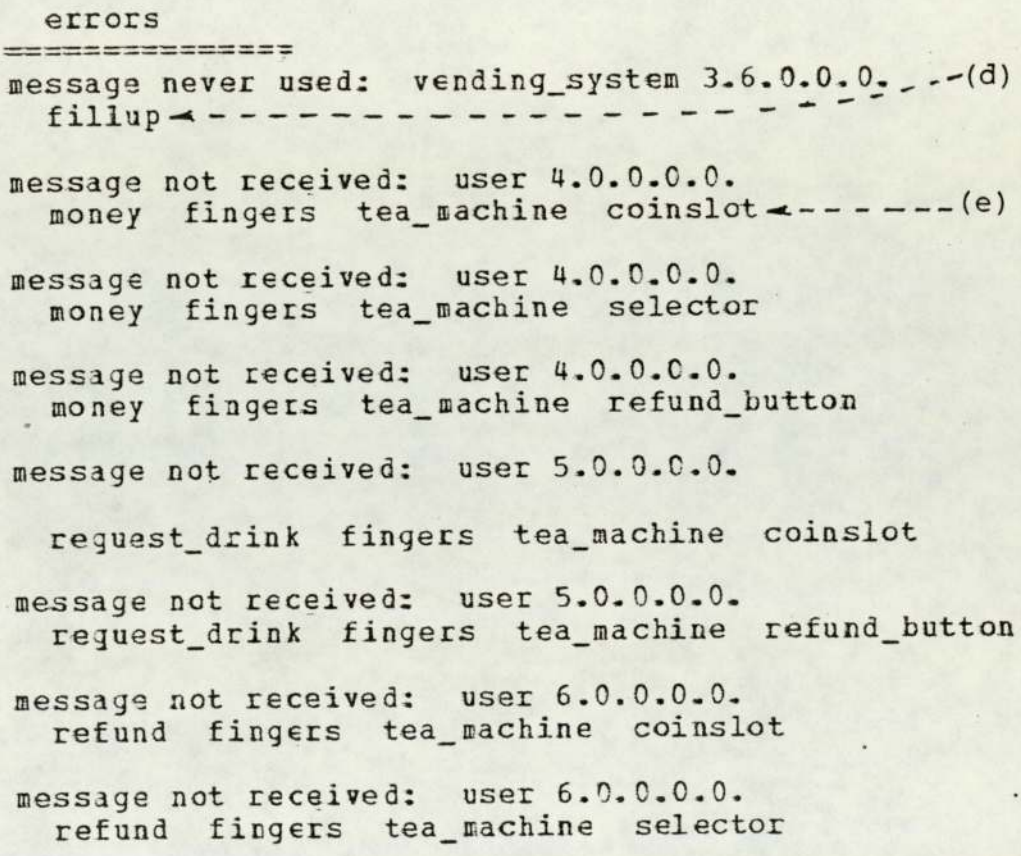
(a) These two error messages are related, in that the first refers to an interface named "coin_slot", whilst the second refers to "coinslot". The omission of the underscore character has resulted in there be-

FIGURE E.3 THE ERRORS DETECTED IN THE SPECIFICATION



contd.

FIGURE E.3 continued



ing two unique names where only one should exist.

(b) The reference to "request_drink with selected..." in line 000610 of Figure E.2 is not consistent with the fact that in line 000260 "request_drink" is defined as having no components.

(c) The "status" being sent via "status_light" is a continuous message (i.e. sent by "start sending...") when it leaves the "tea_machine", but the "user" is not trying to receive it as "on start of....". Hence there is a conflict between the behaviour descriptions in the two models, which must be resolved and corrected.

(d) No provision has been made in the specification for the "tea_machine" to be refilled with ingredients, and this is recognised as a message which has been defined but is never used.

(e) Another error message which is a consequence of the mis-spelling of "coin_slot", as mentioned in (a) above.

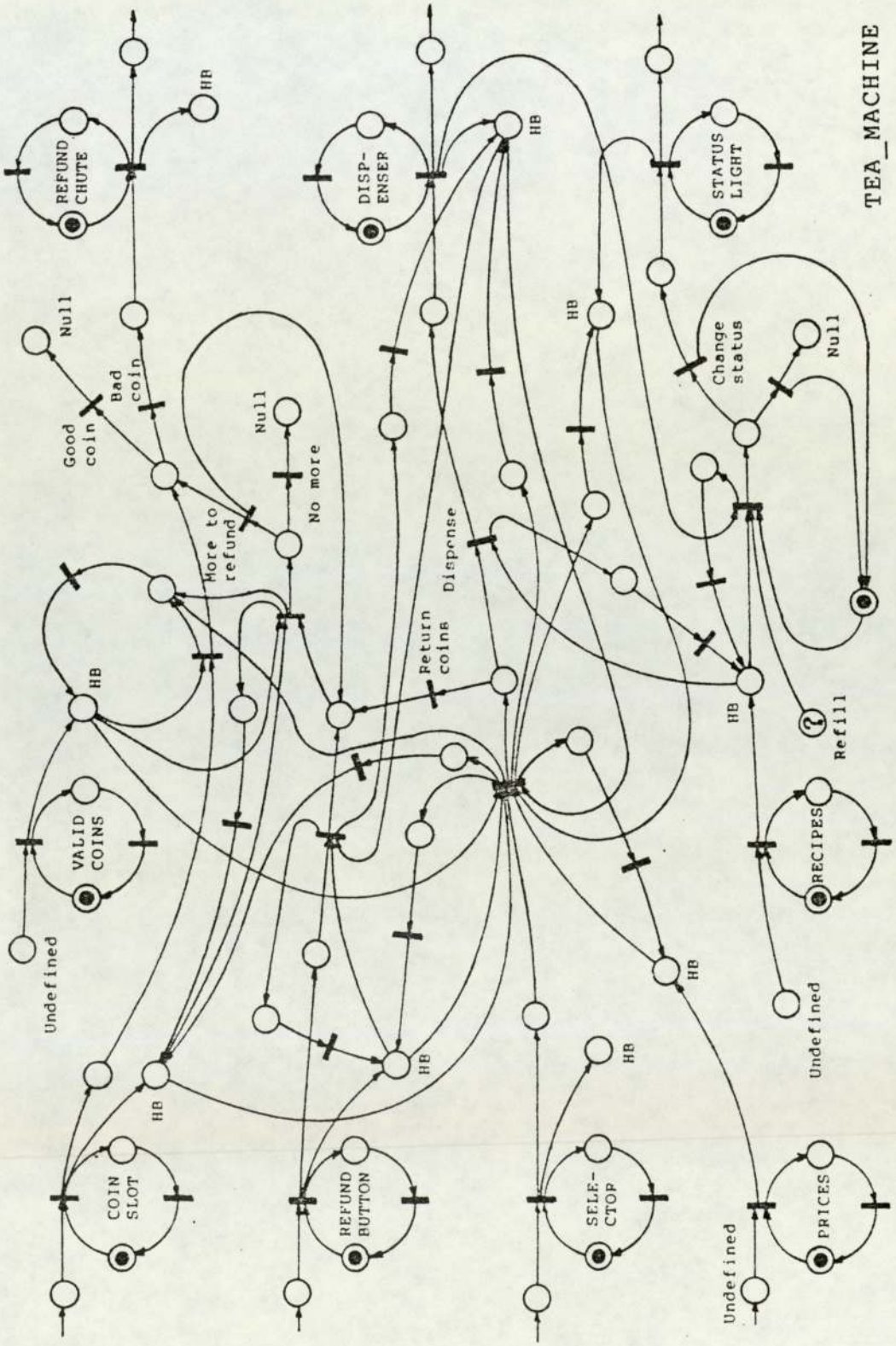
Finally, as no computer facilities were available for the semantic checking, a Predicate/Transition net model of the system was derived manually from the specification. Figure E.4 shows an incomplete version of this net; in order to simplify the diagram only the tea_machine has been shown, and almost all of the labels on places and transitions have been omitted. The manual translation which produced this net is likely to have introduced errors itself, but it is still possible to identify errors in the specification, as in the following examples.

(i) "refill" (shown in Figure E.4 as a place containing a question mark) has no source. This had been identified by the static checks (see (d) above), but had not been corrected.

(ii) Both a refund and the selection of a drink may take place at the same time. As the specification states no priorities, this will result in a drink and a refund.

(iii) Similarly, there is nothing to cover the insertion of a coin at the same time as a request for a refund.

FIGURE E.4 THE NET MODEL OF THE SPECIFICATION



APPENDIX F

RESULTS OF THE TRIALS AT GEC

1. Problems Arising During the Trials

1.1 The Categories

As a large proportion of the problems which arose concerned errors and omissions in the original version of the syntax definition, there was no particular pattern to them. The complete list of problems which follows has therefore been split into four categories based upon their effect upon the definition of ASL. These categories are:

(a) inconsistencies,

(b) simple alterations and extensions,

(c) missing constructs,

(d) other proposals for alterations,

and they appear as sections F.1.2 to F.1.5 respectively.

1.2. Inconsistencies

This category contains the largest number of items, but all are of a minor nature. They all represent points

where the initial syntax definition of ASL contained unnecessary restrictions or unintentionally awkward constructions.

(a) Any results being returned by a defined operation could only be given values in a fixed relationship statement. Thus it was not possible to utilise the "select" form to provide a more comprehensible definition.

(b) Although it was possible to use:

    x in z

where z is a defined data type, the form:

    x in { a, b, c, d }

was not allowed.

(c) "first" and "last" were provided to refer to past messages, but there was no similar method of referring to the intermediate messages.

(d) A bothway interface might be sending and receiving identically-named messages, but it was not possible to specify in a message-pattern that only one direction should be chosen.

(e) Additional brackets were required around message patterns, eg:

    send (x with y) via z

as the "via z" was being associated with the "y" rather than with the "x".

(f) Messages could not have any information content unless they had at least one component. Thus, a message with only one component had to have two names, one for the message and one for the component. This was

both unnecessary and confusing.

(g) An anonymous pattern-match (i.e. "?" without a following variable name) could not be used in the place of a received message.

(h) The name of an interface for either received or transmitted messages could not be a pattern-matching variable.

(i) "undefined" or "unknown" messages had to be sent via "undefined" and "unknown" interfaces respectively. It was not possible to have:

```
....send undefined via x
```

(j) Behaviour statements inside the definition of an operation could not make use of the "unless" form for representing alternatives.

(k) Names representing logical values (i.e. having the values "true" and "false") had to be compared with a logical constant to form a valid condition:

```
....select ( y=true ).....
```

rather than allowing the simpler form:

```
....select ( y )....
```

(l) The first word in a comment had to be alphabetic, and comments could not contain any special symbols.

All the items listed above were treated as errors in the syntax definitions, and were therefore corrected as soon as they were detected. Appendix B.2 shows only the corrected version of the definitions.

1.3. Simple Alterations and Extensions

The participants in the trials suggested a number of changes to ASL. Of these, some were simple to introduce into the language, whilst others implied significant alterations to the formal definitions. The items listed below fall into the category of simple changes, and have all been included in the syntax shown in Appendix B.

(a) An extension to the paragraph numbering scheme allowed paragraph numbers containing only a comment to be used as headings. This improved the facilities for structuring the text inside a model.

(b) Operations (see Chapter 4.4.9) were originally called "functions", but this was felt to be confusing as they are not restricted to being strict mathematical functions.

(c) The symbol ":=" was originally used in fixed relationships instead of "is". This was changed to avoid confusion with the use of the same symbol as an assignment operator in many programming languages (e.g. Pascal (Jensen & Wirth, 1975)).

(d) "interface" and "message" were added as primitive types in the language, so that arguments passed to operations could be of these types.

(e) Definitions were allowed to make use of local sub-definitions, of the form "where...." (see Chapter 4.4.9), e.g.:

```
X : interface where x in incoming_trunks
```

(f) Any defined operation which is common to a number of

models may be placed in the system block, rather than having to be repeated inside each model which uses it.

(g) Limits of ranges may be shown as simple expressions, rather than having to be written as constants, and ranges may be used as a shorthand inside enumerated data types.

(h) The response to a stimulus may be expressed as an ordered sequence, if necessary, by using the word "sequence" followed by the appropriate actions as a series of sub-paragraphs.

1.4. Missing Items

A number of the comments relate to features which are definitely missing from ASL. However, the incorporation of these would require extensions to the type-checking rules (see Appendix B.3 and B.4) or the semantic model (see Appendix C). They have therefore been left as part of the further development of the language, as discussed in Chapter 8.3.

(a) The use of number bases other than 10 (e.g. octal, hexadecimal).

(b) Operators to work on data types constructed using "string" (e.g. concatenation and sub-string operators).

(c) A method of defining a data type as the union or intersection of a number of other data types.

(d) The ability to give values to the whole of a data

structure in a single statement, without having to mention each component by name.

(e) Some specific facility for initialising the models. This would have to cover both the setting of initial values of names and also the equivalent of switching on the power supply.

(f) Extension of the domains represented by data types so that references to non-existent interfaces or array subscripts do not merely return "undefined". It may be necessary to have some identifiable indicator for each sort of error.

(g) Although a method of describing sequences of actions was added to the language (see Section 1.3(h) of this appendix), both this and references back to past messages became cumbersome when the sequences were long and uniform (e.g. a sequence of bits making up a character). This is mainly due to the amount of information which has to be repeated for each item in the sequence. It should be possible to devise an improved form of syntax which avoids this repetition.

1.5. Other Proposals for Alterations

The remaining items, which are listed below, were not as clearly defined as those covered in previous sections; most of them arose as tentative suggestions, which their proposers were unable to expand upon. They have therefore not been incorporated into the language or included in the proposals for further development, and some of them

are incompatible with the original design aims of ASL.

(a) One of the reviewers of the data-rate adaptor specification felt that this would have been easier to understand if the system block had contained a statement of the relationship between the interconnections and the messages. That is, he would have liked to see the definitions of the messages passing through an interconnection placed next to the statement of that interconnection. Although this would have been simple to arrange in the case which he was reviewing, other specifications involved the same message name passing over more than one interconnection. Enforcing this linking of interconnections and messages would therefore lead to duplication of information in many cases.

(b) Operations are permitted to return multiple results, as in the following example:

```
x, y, z is three_result(a, b)
```

but this syntax may not be particularly clear if the list of names spreads over more than one line in the specification text. Some form of bracketing may assist the reader, but ASL already makes use of all the normally available forms of bracket.

(c) The form "take any one of....", introduced to indicate a non-deterministic choice, can be achieved by using a "select" with the selection between the alternatives based upon "undefined". This therefore represents an unnecessary duplication of facilities in the language, so that one form could be removed. It

is not obvious, however, whether the removal of this type of redundancy would have any effect upon the comprehensibility of specifications.

(d) In the disk checking system (see Chapter 6.3) one field in a message had two different meanings, depending upon circumstances. The specification writer suggested that ASL should allow the field to be given more than one name, with these being treated as aliases. This would add complexity to the checking of specifications (see Chapter 5.2); it would be necessary to ensure that there was no conflict between the uses of the aliases, such as the concurrent assignment of different values to aliases for the same name.

(e) The existing implementation of the route-handler module (see Chapter 6.5) involves the dynamic creation of new instances of the route handler as a result of the module calling itself recursively. ASL does not have any method of dynamically creating new models, as this would violate two basic principles of the language:

(i) models are not aware of the interconnections or of the other models in the system, as they only know about their own interfaces,

(ii) the system block, which contains details of the interconnections, is not an active entity and cannot receive messages.

The only way to describe the required situation in ASL is to create the maximum number of route handlers

which can ever exist, but with these remaining dormant until sent a message. However, this seems a rather cumbersome method of achieving the desired result, and further investigation is required into possible alternatives.

2. The Questionnaire

2.1. Design of the Questionnaire

The small number of participants in the trials presented problems in the analysis of the results, as discussed in Chapter 7.3.1. One further consequence was that it was not possible to test the questionnaire on a small sample of the audience, as is normally suggested (e.g. Kornhauser & Sheatsley, 1965). It was therefore decided to attempt to maximise the opportunities for the participants to record their comments in any form which they felt appropriate. Thus, the core questions were presented in multiple choice form, to ensure that some answer would be given on all the features of the language, but with plenty of space left for free-form comments. Other questions were then introduced which asked for opinions and more general comments.

The questionnaire, which is shown in full in Figure F.1, consisted of the six sections listed in the table below. These progress from general questions about the background of the participants to more particular questions about their experiences with ASL. This is the

FIGURE F.1. THE QUESTIONNAIRE

QUESTIONNAIRE ON THE TRIAL USE OF ASL.

INTRODUCTION

You have taken part, either as a specification writer or reviewer, in one of the projects which have used ASL. I would now like to have a record of your comments and criticisms, in order to consider how ASL can be improved. This questionnaire is therefore intended to act as a guide, by listing a series of questions and possible answers, so that the responses from all the people who took part are in a consistent form. However, please note that:

(a) The list of questions (and the alternative answers) may not cover all the comments which you wish to make. Space has been left between questions for you to record any extra information, and if you require even more space please use the reverse side of the pages.

(b) If you do not feel that the answers provided cover your point of view, then add another answer or give additional explanation in the space following the question.

(c) If any question appears irrelevant, or if you have no particular views on it, then don't answer that one.

The questionnaire is in six sections, but Section 5 is only appropriate to those people who have used other specification languages aswell as ASL, and some of the questions in Sections 3 and 4 are particular to people who wrote (or in some cases, reviewed) specifications. Please ignore the questions which do not relate to the role in which you contributed to the trials.

Thank you very much for your co-operation.

P.Blackledge.

Extn. 3481

- 1 -

contd.

FIGURE F.1. continued

SECTION 1 GENERAL

1.1 Please give your name, in block capitals:

.....

1.2 Date completed: .....

1.3 Have you previously been involved in the use of any of the following specification languages ? (Please tick as appropriate.)

English ....

Progression Charts ....

Message Sequence Charts ....

CCITT SDL ....

FSIS / FCIS ....

Jones' Rigorous Method ....

CCS ....

Any Others (Please give names) .....

.....

- 2 -

contd.

FIGURE F.1. continued

1.4 Have you personally used a high-level programming language (e.g. Pascal, Coral, Fortran or some form of program design language) in the projects on which you have worked ? (Please tick.)

Yes ....

No ....

1.5 Which ASL specification(s) were you involved with?

.....

.....

1.6 In what capacity were you involved?

Reader .....

Writer .....

- 3 -

contd.

FIGURE F.1. continued

SECTION 2 SPECIFICATIONS

2.1 What do you feel will be the effect on project progress of insisting that a formal specification is written before design is commenced ? (Please tick.)

It will hinder progress ....

It will have no overall effect ....

It will save time in the end ....

2.2 Do you feel that the creation of a formal specification will help by detecting errors which would otherwise not have been noticed until much later ?

Yes, it will ....

No, it won't ....

Don't know ....

- 4 -

contd.

FIGURE F.1. continued

2.3 Are there any other advantages or disadvantages of formal specifications which you can think of?

2.4 (a) Is it advantageous to restrict all projects within the company to one particular specification language ?

Yes ....

No ....

Don't know ....

(b) If your answer to (a) was 'No', please indicate how many different languages you would allow.

contd.

FIGURE F.1. continued

SECTION 3 ASL.

3.1 What did you think of each of the following features of ASL? (Please tick & comment as appropriate)

	Awkward/ unclear	Neutral	Clear
(a) the block structure	....	....	....
(b) The use of the system block for common information	....	....	....
(c) the "black box" models	....	....	....
(d) the upper/lower case distinction in names	....	....	....
(e) paragraph numbers	....	....	....
(f) the siting of definitions	....	....	....
(g) the form of definitions	....	....	....

- 6 -

contd.

FIGURE F.1. continued

	Awkward/ unclear	Neutral	Clear
(h) the "on ..... then ....." way of describing behaviour	....	....	....
(i) the use of "unless" for alternatives in behaviour	....	....	....
(j) the use of "select"	....	....	....
(k) the "?x" way of matching against messages	....	....	....
(l) references to time delays	....	....	....
(m) the limitations on the siting of comments	....	....	....
(n) the method of describing sequences of messages	....	....	....
(o) local definitions, using "where"	....	....	....

- 7 -

contd.

FIGURE F.1. continued

	Awkward/ unclear	Neutral	Clear
(p) functions as a shorthand	....	....	....
(q) the use of "whenever" to deal with exception conditions	....	....	....
(r) the difference between instantaneous and continuous messages	....	....	....

(s) any other features on which you wish to comment..

contd.

FIGURE F.1. continued

3.2 (For reviewers only)

(a) How difficult did you find it to understand the ASL specification ?

Very easy ....

Easy ....

OK ....

Hard ....

Very hard ....

Other (Please explain) ...

(b) What did you find most difficult to understand?

contd.

FIGURE F.1. continued

3.3 (For writers only)

(a) How difficult did you find it to write a specification in ASL ?

Very easy ....

Easy ....

OK ....

Hard ....

Very hard ....

Other (Please explain) ..

(b) What did you find the most difficult feature of the language to understand ?

(c) What did you find the most difficult feature to use ?

- 10 -

contd.

FIGURE F.1. continued

(d) What did you find the most useful feature of ASL ?

(e) Did writing the ASL specification uncover any errors or problems which had not previously been detected ?

3.4 Any other comments which you would like to make on the use of ASL, or on the structure and layout of specifications written in the language.

contd.

FIGURE F.1. continued

SECTION 4 SUPPORT AND DOCUMENTATION

4.1 The Language Reference Manual. Please comment upon:

(a) general readability.

(b) ease of finding required information.

(c) does it contain the information which you require?

- 12 -

contd.

FIGURE F.1. continued

(d) are there any items which are not sufficiently well explained ?

4.2 The guide, "An Outline Method for Writing Specifications in ASL."

(a) general readibility.

(b) is the information presented in a useful sequence?

(c) does it contain the information which you require?

contd.

FIGURE F.1. continued

(d) are there any items which are not sufficiently well explained ?

4.3 (For writers only)

(a) Please comment upon the existing computer-based facilites (syntax analyser and checker).

(b) What additional facilities would you most like to see ?

- 14 -

contd.

FIGURE F.1. continued

SECTION 5 COMPARISONS

NOTE: This section is only appropriate to those people who have used another specification language, as it asks for comparisons between ASL and other languages.

If you have not used another specification language, then please go straight to Section 6.

5.1 Please identify the other specification language(s) with which you will be comparing ASL.

5.2 Please identify the merits/demerits of ASL when compared with the other language(s). Five main areas of comparison are listed below, but please add any others which you feel are appropriate.

(a) The structure and sequence of the specification.

- 15 -

contd.

FIGURE F.1. continued

(b) The method of describing behaviour.

(c) The language syntax.

(d) The underlying model of systems.

(e) The comprehensibility of the resulting specifications.

contd.

FIGURE F.1. continued

5.3 Any other points of comparison.

contd.

FIGURE F.1. continued

SECTION 6 ANY OTHER REMARKS

- 18 -

order suggested in Kornhauser and Sheatley (op cit).

Section	Content
1	Name, etc., and previous experience of formal languages.
2	Attitudes to specifications generally.
3	Detailed comments on ASL.
4	Comments on documentation supporting ASL.
5	Comparisons of ASL and other languages.
6	Any other remarks.

The covering note, giving instructions to the participants, attempted to induce them to make full use of the space for comments, opinions, etc..

2.2. The Responses

As a result of the emphasis placed upon the recording of opinions, the responses have to be viewed in two parts. Table F.1 covers those questions which had multiple-choice answers, where the responses have been analysed by counting the number of positive, negative and neutral answers. This allowed points of general agreement amongst the participants to be extracted; these were discussed in Chapter 6.6.5. Many of the remainder of the responses, which took the form of unstructured comments, corresponded to the problems which had been recorded dur-

ing the trials. As these are covered in Sections 1.2 to 1.5 of this appendix, they have not been repeated here. Table F.2 therefore contains a precis of the remaining comments.

TABLE F.1 RESPONSES TO MULTIPLE-CHOICE QUESTIONS

Note: Throughout the table "Y" indicates a yes, "N" a no, "+" indicates a positive response, "-" a negative one, and "0" a neutral one.

		Person							
	Trial	1 1	2 1	3 1	4 2	5 2	6 3	7 4	8 4
	Question								
1.3	Specn language experience	N	N	N	N	Y	Y	Y	Y
1.4	Programming experience	Y	Y	N	N	Y	N	Y	Y
1.6	Role (Reader or Writer)	W	R	R	W	R	R	W	R
2.1	Effect of specn on progress	+	0	0	+	+	+	+	+
2.2	Effect of specn on errors	0	-	+	+	0	0	+	+
2.4	Better to use one language	0	+	0	0	0	+	-	-
3.1	The parts of ASL								
	(a) the block structure	+	0	+	+	+	+	+	+
	(b) the system block	+	+	0	+	+	+	+	0
	(c) "black box" models	+	+	+	0	+	+	+	+
	(d) use of upper/lower case	0	-	0	+	0	0	+	0
	(e) paragraph numbers	-	+	0	0	+	+	-	+
	(f) siting of definitions	0	-	-	0	-	+	+	0
	(g) the form of definitions	0	+	+	+	0	+	+	+
	(h) "on....then...."	+	0	+	+	+	+	+	+
	(i) "unless"	+	+	0	+	+	+	+	+
	(j) the use of "select"	0	0	0	0	0	+	+	+
	(k) pattern-matching	-	-	0	+	0	+	+	0
	(l) time delays	-	-	0	-	+	-	0	+
	(m) siting of comments	-	0	+	0	0	0	-	-
	(n) description of sequences	-	-	-	0	+	0	0	0
	(o) local definitions	-	0	0	0	+	+	+	+
	(p) operations	+	0	+	+	+	0	+	0
	(q) "whenever"	+	+	+	-	+	+	+	+
	(r) instant/contin. messages	-	+	0	+	+	+	+	0
3.2	(a) comprehensibility		-	0		-	-		0
3.3	(a) ease of use (writers)	-			-			+	
4.1	The language reference manual								
	(a) readability	-	0	-	0	0	0	0	0
	(b) ease of reference	0	0	-	-	0	+	-	-
	(c) information content	-	0	0	0	0	-	-	0
	(d) explained well	0	0	0	-	0	-	-	0
4.2	The outline method guide								
	(a) readability	+	0	+	0	+	+	0	+
	(b) information sequence	+	-	+	0	+	+	+	+
	(c) information content	+	0	+	0	+	+	+	+
	(d) explained well	0	-	+	0	+	-	+	+

TABLE F.2 THE OTHER COMMENTS

Question	Person	Comment
2.3 (Advantages/disadvantages of formal languages)		
	1	Many people need to be fluent in the language before it is useful.
	4, 7 & 8	Permits validation and verification.
	6	A single, standard language is needed worldwide.
3.1(f) (Siting of definitions)		
	5	The freedom to site definitions anywhere can easily be misused.
3.1(m) (Siting of comments)		
	2	Comments difficult to identify because of different opening and closing "brackets".
3.1(s) (Other comments on features of ASL)		
	2	It would be easier to read if reserved words were highlighted.
3.2(b) (Most difficult feature for readers)		
	6	Recognition of reserved words.
	8	Reference to history instead of "state".
3.3(b) (Most difficult feature for writers)		
	1	Difference between instant and continuous messages.
	1	The method of expressing time delays is awkward.
3.3(d) (Most useful feature for writers)		
	4	The block structure.
	7	The "black-box" view.
3.4 (Any other comments on ASL)		
	8	The constructive methodology is very useful.
4.1(a) (Reference manual - readability)		
	4	A reader unfamiliar with BNF may find it very difficult to understand.

contd.

TABLE F.2 continued.

Question	Person	Comment
4.1(b)	(Ease of finding information)	
	7	Had to jump about between sections to find things.
4.1(c)	(Information content)	
	7	Some of the "limited" syntax in the text is misleading.
4.1(d)	(Items insufficiently explained)	
	7	The linking of more than two models.
4.3(a)	(The computer-based support facilities)	
	1	Error messages are too cryptic.
	4	Better error recovery needed.
4.3(b)	(Most urgent enhancements)	
	1	The ability to use the ASL code as a simulation model.
	4	A more sophisticated editor for ASL text.
5.2(a)	(Specification structure)	
	4	Little different from using English.
	6	Not as obvious as in progression charts or English, but better than in FSIS.
5.2(b)	(Method of describing behaviour)	
	1	Stilted.
	6	Does not seem to enforce complete description.
	7	Adequate and natural.
5.2(c)	(The syntax)	
	1	Possibility of misunderstandings due to use of English words.
	6	Discouraging when compared to progression charts.
	7	A more concise notation would be better.

contd.

TABLE F.2 continued

Question	Person	Comment
5.2(e) (Comprehensibility)		
	7	Readable and comprehensible, but verbose.
5.3 (Other points of comparison)		
	6	The flexibility of ASL leaves it to the writer to achieve comprehensibility.
	7	Unable to manipulate to perform proofs.
6 (Any other comments)		
	2	Does not cover optional and desirable features of a system.
	4	Separation of behaviour part of specification from design constraints is beneficial.
	6	Needs to be used on larger examples.
	7	No facilities for expressing performance requirements.

APPENDIX G

GLOSSARY OF TERMS AND ABBREVIATIONS

This glossary contains an alphabetic list of words which have been used with particular technical meanings, plus the few abbreviations which appeared in the text. In each definition, words which themselves appear in the glossary are shown underlined.

abstract. (Applied to a description or specification.) At a more general level; having much of the detail removed, in order to produce a simpler description.

aggregation. A named collection of information.

algorithmic. (Applied to a language.) Requiring operations to be described in terms of a step-by-step method (i.e. in the form in which that operation might be performed by a computer).

analyser. (As in e.g. "syntax analyser".) A computer program which performs some form of checking upon statements in some language.

applicative. (Applied to a programming language.) A type of programming language which avoids the use of variables and assignment statements, and instead follows the style of pure mathematics.

ASL. An acronym for "A Specification Language".

assertion. A statement of conditions which must be true at all points in time or at particular (named) points in time.

assignment. (In a programming language.) The operation of associating a new value with a name (known as a variable). Any previous value associated with that name is destroyed by an assignment operation.

axiomatic. (Of a specification language.) Describing the behaviour by means of statements which define the relationships between the various parts of that behaviour.

Backus-Naur Form. A language which is used to define the context-free syntax of a language.

behaviour. (Of a system.) The responses which the system will make when subjected to external stimuli. Both the stimuli and the responses take the form of messages.

black box. (Of a system.) A term used in systems engineering to signify that a system is being viewed only in terms of its externally-visible behaviour, and without considering any underlying mechanism which might be producing that behaviour.

BNF. See "Backus-Naur Form".

change control. An administrative procedure which attempts to ensure the compatibility of alterations to different parts of a product.

CHDL. See "computer hardware description language".

chunk. A term used by psychologists to represent a single

"unit" of information in human memory.

computer hardware description language. A language which describes digital computer operation in terms of the transfer of data between hardware registers.

concurrent. (Of a system.) Having a number of parts of its behaviour which may be taking place at the same time.

conflict. (Of the behaviour of a system, particularly in Petri net models of systems.) A situation where the system has a number of possible responses to a stimulus, and no way of identifying which of the alternatives should be chosen. Hence, the behaviour in this situation is non-deterministic.

context-free. (Of the syntax of a language.) A form of syntax definition where the rules do not refer to the context of a statement (i.e. other statements in the text) in order to determine whether it is syntactically correct. The requirement to make a language have a context-free syntax acts as a limitation on the complexity of that language.

cross-reference. The equivalent of an index, being a list of all the names used in a specification, indicating every place where each name is used.

database. An organised, computer-based information storage and retrieval system, which permits users to access the information without having any knowledge of the form in which it is physically stored.

data type. (In a programming language.) The name of a class of objects; it identifies both the domain of

values which can be taken by those objects, and the operations which may be performed upon them. Often abbreviated to "type".

deadlock. (Of a system.) A situation where a system fails to respond to stimuli due to some unresolved contention for limited resources.

declaration. (In a programming language.) The statement which introduces a new instance of a particular data type.

denotation. (As in "denotational semantics".) The attribution of meaning to statements in a language by refering to ("denoting") one or more mathematical expressions which define the equivalent operation.

descriptive reference. A reference to an object by means of a list of its attributes, and not by the use of its unique name.

editor. (In computer-based systems.) A program which allows a user to create and modify blocks of text through some kind of computer terminal.

environment. Everything which is not part of the system being specified. Usually, only that very small part of the total environment which is in direct communication with the system needs to be considered.

firmware. An integrated circuit device containing some information which is not destroyed when the device is switched off, but which can still be altered as necessary.

formal. (Of a language.) Having well-defined syntax and semantics. This requires that the syntax and seman-

tics are defined in terms of some mathematical model.

function. (In mathematics.) A relationship between the members of two sets such that every member of the first set has a relationship to one memeber of the second set. Can be thought of as a subroutine in a programming language which, when given some inputs, will always return a result.

functional. (Of behaviour.) Concerned only with the responses made to external stimuli, and not with the mechanisms which create those responses. (See also "black box".)

generalisation. (In the description of behaviour.) Description of behaviour which is appropriate to whole classes of events or objects rather than just to individuals.

graphic. (Of a language.) Having pictures or diagrams as its major form for presenting information.

hardware. The physical items (e.g. electrical components, nuts, bolts, printed circuit boards, metalwork) from which a product is constructed.

heuristic. (Of a method.) Consisting of guidelines or "rules of thumb", and so not guaranteed to always produce the desired result.

hierarchical. (Of the design or documentation of a system.) Organised as an ordered set of levels, with the top level being the most abstract, and with each subsequent level adding more detail.

high-level language. A term usually taken to mean programming languages such as FORTRAN, Pascal and Ada

(which are at a "high level" of abstraction when compared with machine code).

imperative. (Of a programming language.) Indicates a type of language which uses variables and assignment statements. Used as the opposite of applicative.

implicit. (Of the specification of behaviour.) Not directly describing a method by which the stimulus-response behaviour of the system could be achieved. Used as the opposite of algorithmic.

instance. (Refering to a data type.) An individual object which is a member of that data type.

interface. A point of connection between a system and its environment.

interpretation. (Of a model.) The method of providing readers with a link between the abstract symbols in the model and the real entities which they represent.

invariant. A statement defining a condition which must not be violated by the system being specified. The condition may be required to hold at particular points in time or at all times.

issue. (Of a document.) The release of a particular version of the document to its audience. Issues are usually uniquely identified by an issue number, so that readers are made aware that the content of the document has been changed.

language. A set of symbols together with a set of rules ("grammar") which defines the meaningful sequences of those symbols.

level. One of an ordered set of descriptions of a system

with different degrees of abstraction. (See also hierarchical.)

message. An instantaneous transfer of information between a system and its environment through one of the interfaces of that system.

methodology. Used in the American sense, meaning a method plus the appropriate organisational support.

minimality. (Of a specification.) Stating no more information than is necessary to define the required behaviour precisely.

model. An abstract description of a system written in some formal language.

modular. (Of a design.) Organised as a set of building blocks, each of which can be replaced or redesigned independently of the others.

natural. (Of a language.) Used in normal (i.e. written and spoken) communication, unlike computer programming languages which were designed for a particular purpose and do not have a spoken form.

notation. A set of symbols and rules for their use. The word is used to indicate a symbol system which is not a complete language.

object. A model of a physical entity or concept, represented by the name of the object together with a collection of properties which are relevant to the intended use of that model.

operation. (In ASL.) A defined sequence of behaviour, or a function. Used to avoid repeated writing of common expressions.

operational. (Of the semantics of a language.) Defined in terms of the operation of a particular implementation of the language, rather than in terms of an abstract mathematical model.

parallel. Taking place at the same time. (See concurrent.)

post-condition. One of a set of statements which will be true after the completion of the operation to which they refer, provided that the pre-conditions of that operation were true when it commenced.

pre-condition. One of a set of statements which must be true before a particular operation is invoked if that operation is to produce the required result. (See the related term, post-condition.)

primitive. (Of some term in a specification or programming language.) Assumed as basic, and therefore not defined in terms of its construction from simpler operations.

production. (In the definition of the syntax of a language.) One of the rules which define the permitted sequences of symbols taken from the alphabet of the language.

program proving. The procedure of constructing a mathematical proof which demonstrates that a computer program performs precisely the operations required by its specification.

proof. A constructive demonstration, in mathematical logic, that one or more statements are the consequences of a set of premisses.

property. An attribute of an object, which can be represented at any point in time by some value.

recursive-descent. A simple method of implementing the syntax analysis of a language, using a set of subroutines which call each other recursively during the analysis of statements.

requirement. One part of the behaviour demanded of a system by its specification.

rigorous. (Of a specification.) Expressed in a formal language, but relying upon informal reasoning, rather than a complete proof, for any demonstration of correctness.

semantics. (Of a language.) The rules which identify those statements conforming to the syntax of the language which also have valid meaning.

software. Computer programs.

specification. A description of the required behaviour of a system in terms of the responses which that system will make to any stimuli which it receives.

statement. A sequence of symbols in some language which form a logical unit of meaning. Analagous to a sentence in a natural language.

sub-system. Some portion of a system which has been identified as an element in the hierarchical description of that system. At the next lower level in the hierarchy that sub-system is treated as a system itself.

syntax. (Of a language.) The set of rules which define the valid sequences of symbols taken from the alphabet of the language. These rules are known as

productions.

system. Some object which has been identified as separate from its environment so that it can be treated as the intended product from the design process.

textual. (Of a language.) Presented as strings of characters, rather than as pictures or diagrams. The opposite of graphic.

theorem proving. The mathematical methods used in program proving.

top-down. (Of a design method.) Strictly following the development of a hierarchical set of descriptions of the system from the most abstract down to the most detailed.

tractability. (Of a language.) Ease of manipulation, in the way that algebraic equations may be manipulated without affecting their meaning.

transformation.

(i) On data. Some manipulation performed by a system upon the data.

(ii) As a method of implementing a system. The derivation of a product from a specification by a sequence of small improvements, each of which will not affect the correctness of the system.

type. (In a programming language.) "type" is an abbreviation of data type.

validation. Checking which involves the comparison of a model with a set of mental concepts, and which can therefore never demonstrate total correctness. (See verification, which covers formal checking.)

variable. (In a programming language.) A name which is associated with a single storage location. At any point in time, a variable has only one value, and when this value is changed by an assignment statement any previous value is destroyed.

verification. Checking by the comparison of two descriptions, both written in formal languages. Thus, verification can show that one of the descriptions is a correct representation of the other in a way that validation cannot.

REFERENCES

(Abrial, 1980) J.R.Abrial, "The Specification Language Z - Syntax and Semantics", Oxford University Programming Research Group, April 1980.

(Aho & Ullman, 1977) A.V.Aho and J.D.Ullman, "Principles of Compiler Design", Addison-Wesley, Reading, Ma., 1977.

(Alberts, 1976) D.S.Alberts, "The Economics of Software Quality Assurance", pp 433-432 in AFIPS National Computer Conference, 1976.

(Alford, 1977) M.W.Alford, "A Requirements Engineering Methodology for Real-time Processing Requirements", IEEE Transactions on Software Engineering, Vol.SE-3, No.1, pp 60-69, January 1977.

(Alford, 1979) M.W.Alford, Presentation on SREM at the Symposium on Formal Design Methodology, Cambridge, England, April 1979.

(Ambler & Good, 1977) A.L.Ambler and D.I.Good, "Gypsy - A Language for Specification and Implementation of Verifiable Programs", SIGPLAN Notices, Vol.12, No.3, pp 1-10, March 1977.

(Aron, 1976) J.D.Aron, "Systems Development", Joint IBM & University of Newcastle Seminar on Computing Systems Design, 1976.

(Ashby, 1969) W.R.Ashby, "An Introduction to Cybernetics", University Paperbacks, England, 1969.

(ASTG, 1981) "Report of the Advanced Software Techniques Group", British Telecom, July 1981.

(Backhouse, 1979) R.C.Backhouse, "Syntax of Programming Languages", Prentice-Hall International, London, 1979.

(Baker, 1972) F.T.Baker, "Chief Programmer Team", IBM Systems Journal, Vol.11, No.1, pp 56-73, 1972.

(Balzer, 1981) R.Balzer, "Transformational Implementation: An Example", IEEE Transactions on Software Engineering, Vol.SE-7, No.1, pp 3-14, January 1981.

(Balzer et al, 1978) R.Balzer, N.Goldman and D.Wile, "Informality in Program Specifications", IEEE Transactions on Software Engineering, Vol.SE-4, No.2, pp 94-103, March 1978.

(Balzer & Goldman, 1979) R.Balzer and N.Goldman, "Principles of Good Software Specification and Implications for Specification Language", pp 58-67 in Proc. IEEE Conf. Specification of Reliable Software, 1979.

(Bell et al, 1973) C.G.Bell, J.Grason and A.Newell, "Designing Computer and Digital Systems" , Digital Press, Maynard, Ma., 1973.

(Bell & Newell, 1971) G.Bell and A.Newell, "Computer Structures: Readings and Examples", McGraw-Hill, New York, 1971.

(Berild & Nachmens, 1978) S.Berild and S.Nachmens, "CS4 - A Tool for Database Design by Infological Simulation", in "Tutorial: Software Methodology", eds. C.V.Ramamoorthy and R.T.Yeh, IEEE Computer Society, 1978.

(Biggerstaff, 1979) E.J.Biggerstaff, "The Unified Design Specification System (UDS^2)", pp 104-118 in Proc. IEEE Conf. on Specifications of Reliable Software, 1979.

(Bjorner & Jones, 1978) D.Bjorner and C.B.Jones (eds), "The Vienna Development Method: The Meta-Language", Lecture Notes in Computer Science, No.61, Springer-Verlag, Berlin, 1978.

(Blackledge(a), 1981) P.Blackledge, "The Selection of a Specification Language", pp 25-30 in Proc. 4th. IEE Int. Conf. on Software Engineering for Telecommunication Switching Systems, Warwick, England, July 1981.

(Blackledge(b), 1981) P.Blackledge, "An Introduction to Specifications, Specification Languages and ASL", GEC Internal Report, October 1981.

(Blackledge(a), 1982) P.Blackledge, "A Specification Language (ASL), Reference Manual", GEC Internal Report, January 1982.

(Blackledge(b), 1982) P.Blackledge, "An Outline Method for Writing Specifications in ASL", GEC Internal Report, March 1982.

(Bobrow et al, 1977) D.G.Bobrow, R.M.Kaplan, M.Kay, D.A.Norman, H.Thompson and T.Winograd, "GUS: A frame-driven dialogue system", Artificial Intelligence, Vol.8, pp 155-173, 1977.

(Boebert et al, 1979) W.E.Boebert, W.R.Franta and H.Berg, "NPN: A Finite-State Specification Technique for Distributed Software", pp 139-149 in Proc. IEEE Conf. on Specifications of Reliable Software, 1979.

(Boute, 1981) R.T.Boute, "Towards a Theory of System Semantics", Bell Telephone Mfg. Cy., Belgium, 1981.

(Bouteille, 1978) D.Bouteille, "Un Diagramme Fonctionnel au Service des Automatismes Pnuematiques", Energie Fluide, No.104, pp 29-34, June 1978.

(Bown, 1979) G.C.S.Bown, "HARTRAN: A Hardware Description Language for Digital System Design", Hirst Research Centre Report No. 16447A, November 1978.

(Boyer & Moore, 1979) R.S.Boyer and J.S.Moore, "A Computational Logic", Academic Press, London, 1979.

(Braek, 1979) R.Braek, "Functional Specification and Description Languages as a Practical Tool for Improved System Quality", pp 1.3.1.1-9 in Proc. Telecom 79, Geneva, September 1979.

(Brooks, 1975) F.P.Brooks Jr., "The Mythical Man-month: Essays on Software Engineering", Addison-Wesley Inc., Reading, Ma., 1975.

(BTS, 1981) "Functional Signalling and Interface Specification for R2 MFC as Used in China, Columbia, India and Portugal", Document TF.20.03.06, British Telecommunications Systems Ltd., August 1981.

(Bubenko & Kallhammer, 1971) J.Bubenko and O.Kallhammer, "CADIS Computer Aided Design of Information Systems", Proc. 1st. Scandinavian Workshop on Computer-aided Information Systems Analysis and Design, Denmark, April 1971.

(Burstall & Goguen, 1977) R.M.Burstall and J.A.Goguen, "Putting Theories Together to Make Specifications", pp 1045-1058 in Proc 5th Int. Jnt. Conf. on Artificial Intelligence, 1977.

(Caine & Gordon, 1975) S.H.Caine and E.K.Gordon, "PDL - A tool for software design", pp 271-276 in AFIPS Conference Proceedings, Vol.44, National Computer Conference, 1975.

(Campbell & Habermann, 1974) R.H.Campbell and A.H.Habermann, "The Specification of Process Synchronisation by Path Expressions",pp 89-102 in Lecture Notes in Computer Science, No.16, Springer-Verlag, Berlin.

(CCITT, 1980) CCITT Plenary Assembly Document No.20 (Study Group XI, Contribution No.395), Draft Recom-

mendations Z101-Z104, "Functional Specification and Description Language (SDL)", June 1980.

(Chen, 1976) P.P.S.Chen, "The Entity-Relationship Model - Towards a Unified View of Data", ACM Transactions on Database Systems, Vol.1, No.1, pp 9-36, March 1976.

(Clark, 1978) I.A.Clark, "STREMA: Specifying Application Processes Using Streams", Computer Journal, Vol.21, No.1, pp 25-30, February 1978.

(Cleaveland, 1980) J.C.Cleaveland, "Mathematical Specifications", SIGPLAN Notices, Vol.15, No.12, pp31-42, December 1980.

(Clocksin & Mellish, 1981) W.F.Clocksin and C.S.Mellish, "Programming in Prolog", Springer-Verlag, Berlin, 1981.

(CODASYL, 1962) CODASYL Language Structure Group, "An Information Algebra, Phase I report", Communications of the ACM, Vol.5, No.4, pp 190-204, April 1962.

(Codd, 1970) E.F.Codd, "A Relational Model of Data for Large Shared Data Banks", Communications of the ACM, Vol.13, No.6, pp 377-387, June 1970.

(Cohen, 1980) B.Cohen, "System Specification - Hardware and Software - as Practised in the Telecommunications Industry", CREST Course, Brunel University, July 1980.

(Cohen, 1981) B.Cohen, "Further Thoughts on the Contractual Model of Product Development", pp 61-68 in Proc. System Design Seminar "Emerging Formalisms", S.T.L. Ltd., Harlow, England, February 1981.

(Cohen & Burns, 1978) B.Cohen and G.Burns, "The Contractual Methodology", S.T.L. Internal Report, December 1978.

(Cole, 1980) A.J.Cole, "Macroprocessors", 2nd. Edition, Cambridge University Press, England, 1980.

(Corker & Coakley, 1976) M.Corker and F.P.Coakley, "Automatic Code Generation for SPC Call Processing", pp 27-30 in Second Internat. IEE Conf. on Software Engineering for Telecommunication Switching Systems, 1976.

(Cunningham & Kramer, 1977) R.J.Cunningham and J.Kramer, "An Approach to the Design of Distributed Computer Control System Software Using a Processor Module Concept", pp 79-85 in Proc. IEE Int. Conf. on Distributed Control Systems, September 1977.

(Dahl & Nygaard, 1966) O-J.Dahl and K.Nygaard, "SIMULA An Algol-based Simulation Language", Communications of the ACM, Vol.9, No.9, pp 671-678, 1966.

(Davie & Morrison, 1981) A.J.T.Davie and R.Morrison, "Recursive Descent Compiling", Ellis Horwood Ltd., Chichester, 1981.

(Davis & Rauscher, 1979) A.M.Davis and T.G.Rauscher, "Formal Techniques to Ensure Correctness in Requirements Specifications", pp 15-35 in Proc. IEEE Conf. on Specifications of Reliable Software, 1979.

(Davis & Vick, 1977) C.G.Davis & C.R.Vick, "The Software Development System", IEEE Transactions on Software Engineering, Vol SE-3, No.1, pp 69-84, January 1977.

(Dawkins, 1982) P.H.Dawkins, "Lead Time Reduction for New Products", PhD dissertation, University of Aston in Birmingham, 1982.

(Deen, 1977) S.M.Deen, "Fundamentals of Database Systems", Macmillan Press, London, 1977.

(Demuynck & Meyer, 1979) M.Demuynck and B.Meyer, "Les Langages de Specification", E.D.F.- Bulletin de la Direction des Etudes et Recherches Serie C - Mathematiques, Informatique, No.1, pp 39-60, 1979.

(Dietrich, 1979) R.Dietrich, "On a Compilable Call Processing Specification", pp 1173-1179 in Proc. Internat. Switching Symposium, Paris, 1979.

(Dijkstra, 1972) E.W.Dijkstra, in "Structured Programming", ed. O-J.Dahl et al, Academic Press, New York, 1972.

(Dijkstra, 1976) E.W.Dijkstra, "A Discipline of Programming", Prentice-Hall, Englewood Cliffs, NJ, 1976.

(DoI(a), 1981) "Ada-based System Development Methodology Study Report", Department of Industry, September 1981.

(DoI(b), 1981) "United Kingdom Ada Study Final Technical Report", Department of Industry, July 1981.

(Duley & Dietmeyer, 1968) J.R.Duley and D.L.Dietmeyer, "A Digital System Design Language", IEEE Transactions on Computers, Vol.C-17, pp 850-861, 1968.

(Elton & Messel, 1978) L.R.B.Elton and H.Messel, "Time and Man", Pergammon Press, England, 1978.

(EODST, 1981) "National R2 Signalling System (National) as Used in Bahrain", EODST CP(81)20, British Telecom-

munications Systems Ltd., 1981.

(Estrin, 1978) G.Estrin, "A Methodology for Design of Digital Systems - Supported by SARA at the Age of One", pp 313-326 in AFIPS Conference Proceedings, Vol.47, 1978.

(Falla, 1981) M.Falla, "The Gamma Software Engineering System", Computer Journal, Vol.24, No.3, pp 235-242, August 1981.

(Fitter & Green, 1979) M.Fitter and T.R.G.Green, "When Do Diagrams Make Good Computer Languages?", Int. J. Man-Machine Studies, Vol.11, No.2, pp 235-261, 1979.

(Floyd, 1979) R.W.Floyd, "The Paradigms of Programming", Communications of the ACM, Vol.22, No.8, pp 455-460, August 1979.

(Frankowski & Franta, 1980) E.N.Frankowski and W.R.Franta, "A Process Oriented Simulation Model Specification and Documentation Language", Software - Practice & Experience, Vol.10, pp 721-742, 1980.

(Gaines, 1976) B.R.Gaines, "Foundations of Fuzzy Reasoning", Int. J. Man-Machine Studies, Vol.8, pp 623-668, 1976.

(Galvin, 1981) J.L.Galvin, "Proposals for an All-purpose R2 Signalling Specification to Permit a Wide Range of R2 Variant Signalling Systems", Report input to EODST, British Telecommunications Systems Ltd., January 1981.

(Gane & Sarson, 1979) C.P.Gane and T.Sarson, "Structured Systems Analysis: tools and techniques", Prentice-Hall Inc., Englewood Cliffs, NJ., 1979.

(Gannon & Horning, 1975) J.D.Gannon and J.J.Horning, "Language Design for Programming Reliability", IEEE Transactions on Software Engineering, Vol.SE-1. No.2, pp 179-191, June 1975.

(Gatto, 1974) O.T.Gatto, "AUTOSATE", Communications of the ACM, Vol.7, No.7, pp 425-432, July 1964.

(Genrich et al, 1980) H.J.Genrich, K.Lautenbach and P.S.Thiagarajan, "Elements of General Net Theory", pp 21-163 in Lecture Notes in Computer Science, No.84, Springer-Verlag, Berlin, 1980.

(Gerhart & Yelowitz, 1976) S.Gerhart and L.Yelowitz, "Observations of Fallibility in Applications of Modern Programming Methodology", IEEE Transactions on Software Engineering, Vol.SE-2, No.3, pp 195-207, September 1976.

(Goguen, 1979) J.A.Goguen, "An Introduction to OBJ: A Language for Writing and Testing Formal Algebraic Program Specifications", pp 170-189 in Proc. IEEE Conf. on Specifications of Reliable Software, 1979.

(Goguen et al, 1978) J.A.Goguen, J.W.Thatcher and E.G.Wagner, "An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types", pp 80-149 in "Current Trends in Programming Methodology , Vol.IV", ed. R.T.Yeh, Prentice-Hall Inc, Englewood Cliffs, NJ., 1978.

(Green, 1977) T.R.G.Green, in Panel Discussion, p. 179 in "Software Engineering", ed. R.H.Perrott, Academic Press, London, 1977.

(Green, 1980) T.R.G.Green. "Programming as a Cognitive Activity", pp 271-320 in "Human Interaction with Computers". eds. H.T.Smith & T.R.G.Green, Academic Press, New York, 1980.

(Green et al, 1981) T.R.G.Green, M.E.Sime and M.J.Fitter, "The Art of Notation", pp 221-251 in "Computing Skills and the User Interface", eds. M.J.Coombs and J.L.Alty, Academic Press, London, 1981.

(Gries, 1971) D.Gries, "Compiler Construction for Digital Computers", Wiley, New York, 1971.

(Grindley, 1975) K.Grindley, "Systematics - A New Approach to Systems Analysis", McGraw-Hill, London, 1975.

(Guttag, 1977) J.Guttag, "Abstract Data Types and the Development of Data Structures", Communications of the ACM, Vol.20, No.6, pp 396-404, June 1977.

(Hamilton & Zeldin, 1976) M.Hamilton and S.Zeldin, "High Order Software: A Methodology for Defining Software", IEEE Transactions on Software Engineering, Vol.SE-2, No.1, pp 9-32, March 1976.

(Hammer et al, 1977) M.Hammer, W.G.Howe, V.J.Kruskal andI.Wladawsky, "A Very High Level Programming Language for Data Processing Applications", Communications of the ACM, Vol.20, No.11, pp 832-840, November 1977.

(Hantler & King, 1975) S.L.Hantler and J.C.King, "An Introduction to Proving the Correctness of Programs", Computing Surveys, Vol.8, No.3, pp 331-335, September 1976.

(Harrison, 1974) M.A.Harrison, "Some Linguistic Issues in Design", pp 405-415 in "Basic Questions of Design Theory", ed. W.R.Spillers, North-Holland, 1974.

(Hartley & Burnhill, 1977) J.Hartley and P.Burnhill, "Fifty Guidelines for Improving Instructional Text", Programmed Learning and Educational Technology, Vol.14, pp 65-73, 1977.

(Hayakawa, 1978) S.I.Hayakawa, "Language in Thought and Action", 4th. edition, Harcourt Brace Jovanovich, New York, 1978.

(Hemdal, 1973) G.Hemdal, "The Function Flowchart: A Specification and Design Tool for SPC Exchanges", pp 262-270 in Proc. IEE Internat. Conf. on Software Engineering for Telecommunication Switching Systems, 1973.

(Henninger, 1979) K.L.Henninger, "Specifying Software Requirements for Complex Systems : New Techniques and their Application", pp 1-14 in Proc IEEE Conf. on Specifications of Reliable Software, 1979.

(Hewitt, 1977) C.Hewitt, "Viewing Control Structures as Patterns of Passing Messages", Artificial Intelligence, Vol.8, pp 323-364, 1977.

(Hewitt et al, 1979) C.Hewitt, G.Attardi and H.Lieberman, "Specifying and Proving Properties of Guardians for Distributed Systems", MIT Artificial Intelligence Laboratory, A.I. Memo 505, June 1979.

(Hill, 1972) I.D.Hill, "Wouldn't it be nice if we could write computer programs in ordinary English - or would it?", Computer Bulletin, Vol.16, No.6, pp 306-312, June 1972.

(Hill & Peterson, 1973) F.J.Hill and G.R.Peterson, "Digital Systems: Hardware Organisation and Design", Wiley, New York, 1973.

(Hoare, 1969) C.A.R.Hoare, "An Axiomatic Basis for Computer Programming", Communications of the ACM, Vol.12, No.10, October 1969.

(Hoare, 1973) C.A.R.Hoare, "Hints on Programming Language Design", Stanford University Technical Report No CS-73-403, 1973.

(Hobbs, 1977) J.R.Hobbs, "What the Nature of Natural Language Tells Us About How to Make Natural-language-like Programming Languages More Natural", SIGPLAN Notices, Vol.12, No.8, pp 85-93, August 1977.

(Holbeck-Hanssen et al, 1975) E.Holbeck-Hanssen, P.Handlykken and K.Nygaard, "System Description and the Delta Language", Delta Project Report No.4, Norwegian Computer Centre Pub. No.523, Oslo, September 1975.

(Hopcroft & Ullman) J.E.Hopcroft and J.D.Ullman, "Formal Languages and Their Relation to Automata", Addison-Wesley, Reading, Ma., 1969.

(Humby, 1973) E.Humby, "Programs from Decision Tables", Macdonald Ltd., London, 1973.

(IBM, 1976) "OS PL/I Checkout and Optimising Compilers: Language Reference Manual", Fifth Edition, IBM Corporation, October 1976.

(IBM, 1978) "OS/VS2 TSO Terminal Users Guide", Fifth Edition, IBM Corporation, June 1978.

(Ichbiah et al, 1979) J.D.Ichbiah, B.Krieg-Bruckner, B.Wichmann, H.F.Ledgard, J-C.Heliard, J-R.Abrial, G.P.Barnes and O.Roubine, "Preliminary Reference Manual for the Ada Programming Language", SIGPLAN Notices, Vol.14, No.6, Part A, June 1979.

(Jackson, 1981) M.A.Jackson, "System Development Method: JSD", IEE Colloquium on Formal Design Techniques for Microprocessor Systems, May 1981.

(Jaderlund, 1980) C.Jaderlund, "Systematrix - Concepts", Systematic AB, Stockholm, Sweden, 1980.

(James, 1981) E.B.James, "The User Interface: How We May Compute", pp 337-371 in "Computing Skills and the User Interface", eds. M.J.Coombs and J.L.Alty, Academic Press, London, 1981.

(James & Partridge, 1973) E.B.James and D.P.Partridge, "Adaptive Correction of Program Statements", Communications of the ACM, Vol.16, pp 27-37, 1973.

(Jensen et al, 1979) K.Jensen, M.Kyng and O.L.Madsen, "A Petri Net Definition of a System Description Language", pp 348-368 in Lecture Notes in Computer Science, No.70, Springer-Verlag, Berlin, 1979.

(Jensen & Wirth, 1975) K.Jensen and N.Wirth, "Pascal User Manual and Report", 2nd. edition, Springer-Verlag, Berlin, 1975.

(Johnson, 1979) S.C.Johnson, "YACC - Yet Another Compiler Compiler", UNIX Programmer's Manual, Volume 2, Section 19, Digital Equipment Corp., 1979.

(Jones, 1979) T.C.Jones, "A Survey of Programming Design and Specification Techniques", pp 91-103 in Proc. IEEE Conf. on Specifications of Reliable Software, 1979

(Jones(a), 1980) C.B.Jones, "Software Development: A Rigorous Approach", Prentice-Hall, London, 1980.

(Jones(b), 1980) T.C.Jones, "Programming Quality and Productivity: An Overview of the State of the Art", I.T.T. Programming Technology Centre, June 1980.

(Jones & Kirk, 1979) W.T.Jones and S.A.Kirk, "APL as a Software Design Specification Language", Computer Journal, Vol.23, No.3, pp 230-232, June 1979.

(Karp, 1978) A.Karp, "Develop Software with Flowgrams", Electronic Design, Vol.26, No.16, pp 110-114,September 1978.

(Kawashima et al, 1971) H.Kawashima, K.Futami and S.Kano, "Functional Specification of Call Processing by State Transition Diagram", IEEE Transactions on Communication Technology, Vol.COM-19, No.5, pp 581-587, October 1971.

(Kent, 1977) W.Kent, "Entities and Relationships in Information", in "Architecture and Models in Data Base Management Systems", ed. G.M.Nijssen, North Holland Pub. Co., 1977.

(Kornfeld & Hewitt, 1981) W.A.Kornfeld and C.Hewitt, "The Scientific Community Metaphor", IEEE Transactions on Systems, Man and Cybernetics, Vol.SMC-11, No.1, pp 24-33, January 1981.

(Kornhauser & Sheatley, 1965) A.Kornhauser and P.B.Sheatley, "Questionnaire Construction and Interview Procedure", pp 546-587 in "Research Methods in Social Relations", eds. C.Selltiz, M.Jahoda, M.Deutsch and S.W.Cook, Methuen & Co. Ltd., London, 1965.

(Krieg-Bruckner & Luckham, 1980) B.Krieg-Bruckner and D.C.Luckham, "ANNA: Towards a Language for Annotating Ada Programs", SIGPLAN Notices, Vol.15, No.11, pp 128-138, November 1980.

(Kuhn, 1970) T.S.Kuhn, "The Structure of Scientific Revolutions", 2nd. edition, Univ. of Chicago Press, 1970.

(Lamport, 1978) L.Lamport, "Time, Clocks and the Ordering of Events in a Distributed System", Communications of the ACM, Vol.21, No.7, pp 558-565, July 1978.

(Lattanzi, 1980) L.D.Lattanzi, "An Analysis of the Performance of a Software Development Methodology", GTE Automatic Electric Journal, pp 41-46, March 1980.

(Lauer et al, 1979) P.E.Lauer, P.R.Torrigiani and M.W.Shields, "COSY - A System Specification Language Based on Paths and Processes", Acta Informatica, Vol.12, pp 109-158, 1979.

(Lauther, 1979) U.Lauther, "A Min-cut Placement Algorithm for General Cell Assemblies Based on a Graph Representation", pp 1-10 in Proc. 16th Design Automation Conference, June 1979.

(Laventhal, 1979) M.S.Laventhal, "Synchronisation Specifications for Data Abstractions", pp 119-125 in Proc. IEEE Conf. on Specifications of Reliable Software, 1979.

(Lawson, 1977) H.W.Lawson Jr., "Programming, Architecture and Complexity", Report LITH-MAT-R-1977-28, Linkoping University, Sweden, 1977.

(Lehman, 1979) M.M.Lehman, "The Environment of Design Methodology", keynote address to the Symposium on Formal Design Methodology, Cambridge, England, April 1979.

(Lehman, 1981) M.M.Lehman, "The Environment of Program Development and Maintenance - Programs, Programming and Programming Support", Dept. of Computing Report 81/2, Imperial College, London, January 1981.

(Lewin, 1977) D.Lewin, "Computer-Aided Design of Digital Systems", Edward Arnold, London, 1977.

(Lindgreen, 1973) P.Lindgreen, "The Development of a Computerised Tool for Systems Design based on the Qualitative Information Theory", pp 63-83 in "Approaches to System Design", NCC, Manchester, 1973.

(Lindstrom & Skansholm, 1981) H.Lindstrom and J.Skansholm, "How to Make Your Own Simulation System", Software-Practice & Experience, Vol.11, No.6, pp 629-637, June 1981.

(Liskov & Zilles, 1978) B.Liskov and S.Zilles, "An Introduction to Formal Specifications of Data Abstractions", pp 1-32 in "Current Trends in Programming Methodology, Vol.I", ed. R.T.Yeh, Prentice-Hall Inc., Englewood Cliffs, NJ., 1978.

(Losleben, 1980) P.Losleben, "Computer Aided Design for VLSI", in "Very Large Scale Integration (VLSI) : Fundamentals and Applications", ed. D.F.Barbe, Springer-Verlag, Berlin, 1980.

(Mackie, 1979) L.Mackie, "Software Reliability : Understanding and Improving It", pp 31-1 to 31-10 in Proc. AGARD Conf. Avionics Reliability, its Techniques and Related Disciplines, 1979.

(Mackie, 1981) L.Mackie, Presentation to the GEC Software Engineering Group, February 1981.

(Malhotra et al, 1980) A.Malhotra, J.C.Thomas,

J.M.Carroll and L.A.Miller, "Cognitive Processes in Design", Int. J. Man-Machine Studies, Vol.12, pp 119-140, 1980.

(Marconi Radar, 1980) "FDL - Draft Issue", Marconi Radar Systems Ltd., Chelmsford, September 1980.

(Marcotty & Ledgard, 1976) M.Marcotty and H.F.Ledgard, "A Sampler of Formal Definitions", ACM Computing Surveys, Vol.8, No.2, pp 191-276, June 1976.

(Merlin, 1974) P.M.Merlin, "A Study of the Recoverability of Computing Systems", Ph.D. Thesis, The University of California, Irvine, 1974.

(Miller, 1967) G.A.Miller, "The Psychology of Communication: Seven Essays", Penguin Books Ltd., Harmondsworth, 1967.

(Mills, 1975) H.D.Mills, "How to Write Correct Programs and Know it", Proc. IEEE Conf. on Reliable Software, 1975; appeared in SIGPLAN Notices, Vol.10, No.6, pp 363-370, June 1975.

(Mills & Walter, 1978) G.H.Mills and J.A.Walter, "Technical Writing, 4th Edition", Holt Rinehart and Winston, New York, 1978.

(Milner, 1980) R.Milner, "A Calculus of Communicating Systems", Lecture Notes in Computer Science, No.92, Springer-Verlag, Berlin, 1980.

(Moore, 1956) E.F.Moore, "Gedanken Experiments on Sequential Machines", in "Automata Studies", Princeton University Press, Princeton, NJ, 1956.

(Moriconi, 1979) M.S.Moriconi, "A Designer/Verifier's Assistant", IEEE Transactions on Software Engineering, Vol.SE-5, No.4, pp 387-401, July 1979.

(Mullery, 1979) G.P.Mullery, "CORE - A Method for Controlled Requirement Specification", pp 126-136 in Proc. 4th. Int. Conf. on Software Engineering, September 1979.

(Musser, 1979) D.R.Musser, "Abstract Data Type Specification in the Affirm System", pp 47-57 in Proc. IEEE Conf. on Specifications of Reliable Software, 1979.

(Nakajima et al, 1977) R.Nakajima, M.Honda and H.Nakahara, "Describing and Verifying Programs with Abstract Data Types", pp 527-555 in "Formal Descriptions of Programming Concepts", ed. E.J.Neuhold, North-Holland, 1977.

(Naur, 1960) P.Naur (ed), "Report on the Algorithmic Language ALGOL60", Communications of the ACM, Vol.3,

pp 299-314, 1960.

(Naur & Randell, 1969) P.Naur and B.Randell (eds), "Software Engineering", NATO Science Committee, January 1969.

(NCC, 1969) "DATAFLOW - Project Evaluation Report", NCC, Manchester, September 1969.

(Neumann et al, 1980) P.G.Neumann, R.S.Boyer, R.J.Feiertag, K.N.Levitt and L.Robinson, "A Provably Secure Operating System: The System, its Applications and Proofs", Report CSL-116, SRI International Inc., Menlo Park, CA, May 1980.

(Nissen & Geiger, 1979) J.C.D.Nissen and G.V.Geiger, "A Fault-tolerant Multimicroprocessor for Telecommunications and General Applications", GEC Journal of Science and Technology, Vol.45, No.3, pp 116-122, 1979.

(Noe, 1978) J.D.Noe, "Hierarchical Modelling with Pro-Nets", pp 155-160 in Proc. National Electronics Conference, 1978.

(Nylin & Harvill, 1976) W.C.Nylin Jr. and J.B.Harvill, "Multiple Tense Computer Programming", SIGPLAN Notices, Vol.11, No.12, pp 74-93, December 1976.

(Parnas, 1972) D.L.Parnas, "A Technique for Software Module Specification with Examples", Communications of the ACM, Vol.15, No.5, pp 330-336, May 1972.

(Peterson, 1980) J.L.Peterson, "Design for a Spelling Program: An Experiment in Program Design", Lecture Notes in Computer Science No. 96, Springer-Verlag, Berlin, 1980.

(Peterson, 1981) J.L.Peterson, "Petri Net Theory and the Modeling of Systems", Prentice-Hall Inc., Englewood Cliffs, NJ, 1981.

(Petri, 1962) C.A.Petri, "Communication with Automata", Ph.D. dissertation, University of Bonn, 1962.

(Petri, 1979) C.A.Petri, "Concurrency", pp 251-260 in Lecture Notes in Computer Science, No. 84, Springer-Verlag, Berlin, 1980.

(Popper, 1974) K.R.Popper, "Conjectures and Refutations", 5th. edition, Routledge and Kegan Paul, London, 1974.

(POR 3231, 1976) Post Office Requirements for Telecommunications No. 3231, "Digital Main Network Switching Centre", Issue 4, British Post Office Telecommunications Headquarters, August 1976.

(Posner & Strike, 1976) G.J.Posner and K.A.Strike, "A Categorisation Scheme for Principles of Sequencing Content", Review of Educational Research, Vol.46, pp 685-690, 1976.

(Pratt, 1975) T.W.Pratt, "Programming Languages: Design and Implementation", Prentice-Hall, Englewood Cliffs, NJ., 1975.

(Quirk, 1978) W.J.Quirk, "The Automatic Analysis of Formal Real-time System Specifications", Report AERE-R9046, U.K.A.E.A., Harwell (H.M.S.O.), 1978.

(Ramamoorthy & So, 1978) C.V.Ramamoorthy and H.H.So, "Software Requirements and Specifications: Status and Perspectives", pp 43-164 in "Tutorial: Software Methodology", eds. C.V.Ramamoorthy and R.T.Yeh, IEEE Computer Society, 1978.

(Redwine et al, 1981) S.T.Redwine, E.D.Siegel and G.R.Berglass, "Candidate Thrusts for the Software Technology Initiative", Report AD-A102180, United States Department of Defence, May 1981.

(Riddle et al, 1979) W.E.Riddle, J.H.Sayer, A.R.Segal, A.M.Stavely and J.C.Wileden, "Abstract Monitor Types", pp 126-138 in Proc. IEEE Conf. on Specifications of Reliable Software, 1979.

(Robinson, 1976) L.Robinson, "Specification Techniques", pp 470-478 in Proc. 13th. Annual Design Automation Conference, 1976.

(Rose & Welsh, 1981) G.A.Rose and J.Welsh, "Formatted Programming Languages", Software - Practice and Experience, Vol. 11, pp 651-669, 1981.

(Rose et al, 1972) C.W.Rose, F.T.Bradshaw and S.W.Katzke, "The LOGOS Representation System", pp 187-190 in IEEE Computer Conference Digest, September 1972.

(Ross, 1977) D.T.Ross, "Structured Analysis (SA): A Language for Communicating Ideas", IEEE Transactions on Software Engineering, Vol.SE-3, No.1, pp 16-34, January 1977.

(RSRE, 1978) "The Official definition of MASCOT", R.S.R.E. L-303(S), Malvern, March 1978.

(Sandewall, 1978) E.Sandewall, "Programming in an Interactive Environment", Computing Surveys, Vol.10, No.1, pp 35-71, 1978.

(Schank et al, 1973) R.C.Schank, N.Goldman, C.J.Rieger III and C.Riesbeck, "MARGIE: Memory, Analysis, Response Generation and Inference on English", pp

255-261 in Proc. 3rd. Joint Int. Conf. on Artificial Intelligence, August 1973.

(Schueler, 1977) B.M.Schueler, "Update Reconsidered", pp 149-164 in "Architecture and Models in Database Systems", ed. G.M.Nijssen, North-Holland, 1977.

(Schwartz, 1973) J.T.Schwartz, "On Programming: An Interim Report on the SETL - Installment 1: Generalities", Computer Science Dept., Courant Institue of Mathematical Sciences, New York University, 1973.

(Schwartz & Melliar-Smith, 1980) R.L.Schwartz and P.M.Melliar-Smith, "Temporal Logic Specification of Distributed Systems", pp 446-454 in Proc. 2nd. Int. Conf. on Distributed Systems, Paris, April 1981.

(SDL, 1980) "A Technical Overview of the PSL/PSA Software System", Systems Designers Ltd., Camberley, 1980.

(Sernadas, 1979) A.Sernadas, "Temporal Aspects of Logical Procedure Definition", London School of Economics, December 1979.

(Shaw, 1980) P.D.Shaw, "Modelling of Telephone Call Processing Using Petri Nets", Ph.D. dissertation, University of Essex, July 1980.

(Shiel, 1981) B.A. Shiel, "The Psychological Study of Programming", Computing Surveys, Vol.13, No.1, pp 101-120, March 1981.

(Siegel, 1956) S.Siegel, "Nonparametric Statistics", McGraw-Hill, New York, 1956.

(Simpson, 1969) H.R.Simpson, "SAG: A Syntax Analyser Generator", Technical Note 739, Royal Radar Establishment, October 1969.

(Sleight & Kossiakoff, 1974) T.B.Sleight and A.Kossiakoff, "Use of Graphics in Software Design, Development and Documentation", Report No. APL-TG-1242, John Hopkins University, Silver Spring, Md., April 1974.

(Sloman, 1971) A.Sloman, "Interaction Between Philosophy and Artificial Intelligence: The Role of Intuition and Non-logical Reasoning in Intelligence", Artificial Intelligence, Vol.2, Nos.3&4, pp 209-225, 1971.

(Smith & Smith, 1977) J.M.Smith and D.C.P.Smith, "Database Abstractions - Aggregation and Generalisation", ACM Transactions on Database Systems, Vol.2, No.2, pp 105-133, June 1977.

(Solvberg, 1973) A.Solvberg, "Formal Systems Descrip-

tions in Information Systems Design", pp 85-93 in "Approaches to System Design", NCC, Manchester, 1973.

(Stamper, 1977) R.K.Stamper, "The LEGOL 1 Prototype System and Language", Computer Journal, Vol.20, No.2, pp 102-108, 1977.

(Stay, 1976) J.F.Stay, "HIPO and Integrated Program Design", IBM Systems Journal, Vol.15, No.2, pp 143-154, 1976.

(Steele & Sussman, 1979) G.L.Steele Jr. and G.J.Sussman, "Constraints", APL Quote Quad, Vol.9, No.1, Part 1, pp 208-225, June 1979.

(Stevens et al, 1974) W.P.Stevens, G.J.Myers and L.L.Constantine, "Structured Design", IBM Systems Journal, 1974.

(Stewart, 1975) I.Stewart, "Concepts of Modern Mathematics", Penguin Books Ltd., Harmondsworth, 1975.

(Stoy, 1977) J.Stoy, "Denotational Semantics", MIT Press, Cambridge, Ma., 1977.

(Swartout, 1982) W.Swartout, "GIST English Generator", USC/Information Sciences Institute, Marina del Rey, CA., April 1982.

(System X, 1979) System X Standards Document, "Progression/Flow Chart Codes of Practice", Post Office Telecommunications, 1979.

(System X, 1981) "System X Engineering Handbook", British Telecom, 1981.

(Szygenda, 1980) S.A.Szygenda, "Design Language/Register Transfer Level Simulator (DL/RTL) Feasibility Study Report", CCSS Inc., Austin, Texas, February 1980.

(Taylor, 1981) P.Taylor, "The Semantics of Signalling", pp 69-82 in Proc. of System Design Seminar "Emerging Formalisms", STL, Harlow, February 1981.

(Teichrow & Hershey, 1977) D.Teichrow and E.A.Hershey III, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems", IEEE Transactions on Software Engineering, Vol.SE-3, No.1, pp 41-48, January 1977.

(Teitelman, 1978) W.Teitelman, "A Display Oriented Programmers Assistant", Int.J. Man-Machine Studies, Vol.11, No.2. pp 157-187, 1978.

(Tennent, 1977) R.D.Tennent "Language Design Methods Based on Semantic Principles", Acta Informatica,

Vol.8, pp 97-112, 1977.

(Thatte, 1980) P.P.Thatte, "Interface Agreement Processor", GTE Automatic Electric Journal, pp 54-60, March 1980.

(Thomas & Carroll, 1981) J.C.Thomas and J.M.Carroll, "Human Factors in Communication", IBM Systems Journal, Vol.20, No.2, pp 237-263, 1981.

(Turner, 1979) D.A.Turner, "Another Algorithm for Bracket Abstraction", Journal of Symbolic Logic, Vol.44, No.2, pp 267-270, June 1979.

(Walters, 1979) S.J.Walters, "Systems Specifications", NCC Publications, Manchester, 1979.

(Wasserman & Stinson, 1979) A.I.Wasserman and S.K.Stinson, "A Specification Method for Interactive Information Systems", pp 68-79 in Proc. IEEE Conf. on Specifications of Reliable Software, 1979.

(Wayne, 1973) M.N.Wayne, "Flowcharting Concept and Data Processing Techniques", Canfield Press, 1973.

(Weinberg, 1971) G.M.Weinberg, "The Psychology of Computer Programming", Von Nostrand Reinhold, New York, 1971.

(Winograd, 1972) T.Winograd, "Understanding Natural Language", Academic Press, London, 1972.

(Winograd, 1979) T.Winograd, "Beyond Programming Languages", Communications of the ACM, Vol.22, No.7, pp 391-401, July 1979.

(Winston, 1976) P.H.Winston, "Artificial Intelligence", Addison-Wesley, Reading, Ma., 1976.

(Wirth, 1974) N.Wirth, "On the Design of Programming Languages", pp 386-393 in Proc. Information Processing 74, North Holland Pub.Co., 1974.

(Wirth(a), 1977) N.Wirth, in Panel Discussion, pp 179-180 in "Software Engineering", ed. R.H.Perrott, Academic Press, London, 1977.

(Wirth(b), 1977) N.Wirth, "What Can We Do About the Unnecessary Diversity of Notation for Syntactic Definitions?", Communications of the ACM, Vol.20, No.11, pp 822-823, November 1977.

(Wood, 1980) R.J.Wood, "A Program Model and Knowledge Base for Computer Aided Program Synthesis", pp 77-78 in Proc. 1st. Annual National Conf. on Artificial Intelligence, Stanford, CA, August 1980.

(Wulf et al, 1976) W.A.Wulf, R.L.London and M.Shaw, "An Introduction to the Construction and Verification of Alphard Programs", IEEE Transactions on Software Engineering, Vol.SE-2, No.4, pp 253-265, December 1976.

(Wymore, 1967) A.W.Wymore, "A Mathematical Theory of Systems Engineering", John Wiley & Sons, New York, 1967.

(Yourdon & Constantine, 1979) E.Yourdon and L.Constantine, "Structured Design", Prentice-Hall, Englewood Cliffs, NJ, 1979.

(Zurcher & Randell, 1969) F.Zurcher and B.Randell, "Iterative Multi-level Modelling - A Methodology for Computer System Design", pp 138-142 in Proc. IFIP Congress, 1968.

THE SELECTION OF A SPECIFICATION LANGUAGE

P. Blackledge

GEC Telecommunications Ltd., UK

INTRODUCTION

The increasing complexity of telecommunications systems, together with the cost and time required to develop the necessary hardware and software, highlight the waste of effort which may result from attempts to design systems before customer requirements have been adequately specified. However, despite the variety of specification methods which have been proposed, none have been widely accepted and used. The investigation reported in this paper examined a wide range of existing methods in order to select the most suitable one or failing that, to propose a basis for the development of a new one. Firstly, a rough specification for a specification language is presented, and then this is used as a set of selection criteria in a critical review of existing languages.

THE ROLE OF A SPECIFICATION

What is a Specification?

For the purpose of the investigation, "specification" was taken to mean the rigorous statement of the required input-output response (functional behaviour) of a system. This excludes all physical constraints (e.g. maximum size, heat dissipation) and may exclude many performance factors (e.g. degradation under overload), which would have to appear as additional documents, in natural language, attached to the behaviour specification. Although this is quite a restrictive definition, it does concentrate the investigation upon the main purpose of a specification.

What is a System?

The word "system" is used throughout this paper to mean any artifact intended to fulfill some purpose; the interface between a system and its environment is the only place at which the correct fulfillment of purpose can be monitored. Under this view of a "system", any level of hardware, software or combination of both can be treated as a complete system, and specified in terms of its interfaces with its environment.

The Role of a Specification

A specification forms a contract between the specifier (customer) and the designer (supplier), although it may not be a formal, legal contract. The purpose of the specification is to convey concepts from the mind of the specifier to the designer, so that the resulting product will adequately meet the specifier's needs. It can also play an important role in both acceptance testing and maintenance, as a clear statement of the intended behaviour of the system with which the actual behaviour can be compared.

THE INADEQUACY OF ENGLISH

The simplest proposal to improve specifications is to raise the standard of the natural language (e.g. English) documents, avoiding the introduction of a new specification method with the associated retraining of staff.

However, the British legal system provides a very good example of the likely difficulties over disputed interpretations of wording; Hill (30) and Henderson & Snowden (28) give examples relating to software which only re-emphasise that natural language is:

(i) too flexible in its use of context

(ii) ambigious

(iii) subject to changes in meaning over time.

Attempts to define rigorously the exact interpretation of each word as it is introduced run into problems of the size and verbosity of the document (as an example, see the Delta Project report, Holbeck-Hanssen et al (32)), and also of conflict with the reader's normal interpretation of the words. (It is interesting to note that the successor to Delta, the Epsilon language (Jensen et al (40)), has adopted a formal approach instead).

Hence, the use of natural language involves limitations which cannot be overcome, and an alternative is required which is more precise, and offers the advantages of a formal notation (see Iverson (37)).

GENERAL CHARACTERISTICS OF A SUITABLE LANGUAGE

In this section, the major, general characteristics required in a notation for specifications (a "specification language") are listed: the use of the term "language" does not exclude graphic notations from consideration. At this stage, there is purposely no discussion of how these characteristics might be included in a language, as that would be part of the design of the language, not its specification.

The major characteristics are:

Formality. The existence of a sound mathematicl basis is necessary for semantic non-ambiguity, and also aids in both computer processing of specifications and their use in proofs of correctness.

Comprehensibility. The notation of the language must not itself be obscure or misleading, so that the concepts embodied in a specification can be clearly expressed. This does not mean that the specification will be understandable by either a person untrained in the notation, or by someone trained in the notation but unfamiliar with the concepts being represented. As Mackie (49) points out, a specification should never be expected to act

4th Int.Conf. on Software Engineering for Telecommunication Switching Systems, University of Warwick, United Kingdom. 20-24 July 1981.

as training material for the induction of new project members.

Minimality. A specification should be minimal in two ways: it should describe exactly the required behaviour and no more - in particular it should say little or nothing about how the behaviour may be achieved; it should also avoid the verbosity introduced by attempts to make the notation resemble natural language.

Ability to Handle Complexity. Although the requirement for formality implies restrictions, the language must be able to handle large, varied and complex systems. In particular, there must be methods of structuring large specifications to improve comprehensibility, by allowing the abstraction of detailed information.

Ease of Change. The content of the specification will be subject to changes, either to improve or correct the specification, and it must be possible to incorporate these into the specification easily. It is to be hoped that a small change in the concepts would only result in a small change to the specification.

Tolerance of Incompleteness. If the language can only be used when the system can be specified in complete detail, this relegates it to a very late stage in the timespan of a project; also this would lead to other forms of documentation being created to fulfil the role of draft issues of the specification. Hence the language must expect, and tolerate, incompleteness and aid the later incorporation of the details as they become available.

SPECIFIC FEATURES

Although the general characteristics listed above are requirements of the language, they are too broad and vague to be considered a specification for a specification language, and can only be assessed subjectively.

This section brings out a number of more specific features which contribute to achieving the general characteristics; the result is still not a complete, rigorous specification of a specification language, but has sufficient detail to indicate deficiencies in many existing languages, as will be seen later.

Implicit Specification of Functions. Where there is an input-output transformation upon data, this should not be specified by describing an algorithm for producing the transformation; instead, the specification should state the required relationship between the input and output values. There may of course be situations when it is necessary to specify that a particular algorithm must be used but this must not result in a language which always demands an algorithm, as this is often a poor way to communicate a concept and may unintentionally introduce extra constraints upon the subsequent design. An example of this type of implicit specification is the use of pre-conditions and post-conditions, as in Jones (41).

Strong Data Typing. The association of a permitted range of values with each variable name as it is declared (as in Pascal and Ada) avoids the repetition of checks each time the value of a variable is changed; this therefore helps keep the specification small.

Abstract Specification of Complex Data Types. In the same way that implicit specification of functions describes the results of the transformation without detailing the means, complex organisations of data should be specified without describing a particular physical data structure. The algebraic specification of data types used by Zilles (72) is an example of this approach.

Localisation of References. The aims of minimisation of the size of the specification and the ease of subsequent alterations both suggest that all information about an entity, function, data type or relationship should be held as a highly localised group, with controlled references from other parts of the specification. The features of strong data typing and abstract specification of data types both contribute to this, but it can also be applied on a wider scale.

Representation of Time. The language must handle time, and particularly time sequence, in an adequate manner, allowing the specification of both sequential and parallel activities; this may not require the explicit inclusion of time in the language, given satisfactory means of expressing both sequencing and concurrency.

Computer Assistance. For a large specification, the job of manually checking that language rules have not been violated is extremely tedious and difficult; Goguen (20) has pointed out that many small example specifications in published papers are incorrect for want of computer-based checking facilities. A specification language should permit computer assistance in the form of syntax checking, the detection of simple redundancy (repeated information), inconsistency (different responses to the same input in the same state) and ambiguity (use of undefined terms, or definition of terms which are not used). Formal specifications in a "mathematical" notation often include assertions of properties; in such cases it is reasonable to demand computer-assistance in checking proofs of the assertions. It may also be desirable to use the specification as input to simulation software, to allow checking by simulation.

A REVIEW OF EXISTING LANGUAGES

The intention of this section is to provide a review of a very wide range of alternatives; this means that it is impossible to discuss the details of each language, and they are therefore grouped into categories based upon their primary features. Discussion of the mertis and demerits is therefore in terms of these categories:

(i) documentation aids
(ii) algorithmic languages
(iii) applicative languages
(iv) analytical tools
(v) state transition specifications
(vi) input-output relationships
(vii) axiomatic specifications.

Each is discussed below, and compared with the criteria; Table 1 then summarises the findings.

Documentation Aids

The main aim of documentation aids is to provide a good structure for a large specification; they normally centre round some graphic display of data flow or system structure, but do not provide a formal language for the specification of functions, leaving this to be done in natural language. Best known of the type are HIPO (Stay (68)) and SADT (Ross (61)), but there have been a number of others, such as AUTOSATE (Gatto (18)) and Sleight and Kossiakoff (65)
Although unsuitable as specification languages, some of the ideas on organising large specifications may still be relevant to documents written in a more suitable language.

Algorithmic Languages

The languages in this category are mostly attempts to extend the use of normal, high level programming languages to the construction of a "skeleton" of the system, although some use graphic representation rather than program text. Amongst the simplest are pseudo-code (IBM (34)), flowcharts (Wayne (71)), and flowgrams (Karp (43)), but there are many more sophisticated ones: LOGOS (Rose and Albarran (60)), Pro-Nets (Noe (55)) and SARA (Estrin (17)) which are based on Petri Nets; the Delta project (32) and its successor, Epsilon (40), based upon SIMULA67 (Dahl & Nygaard (13)), and similar methods such as Actors (Hewitt & Bishop (29)) and SREM (Alford (2)); RLP (Davis et al (14)); methods such as Gypsy (Ambler et al (3)) and MASCOT (RSRE (62)). One different approach is the SAFE project (Balzer et al (5)), which takes in a natural language program specification and attempts to resolve all ambiguities by man-machine dialogue.

The formality of these languages can be adequate, and some of the later ones provide sophisticated abstraction and proof facilities, but all are geared towards description by an algorithm rather than result specification.

Applicative Languages

APL (Iverson (36)) and LISP (McCarthy et al (50)) provide facilities for combining functions as in pure mathematics; these ideas have led to proposals for higher level languages (e.g. Backus (4) and Schwartz (63)) and the use of such notations for specifications (e.g. Jones & Kirk (42)).

All the applicative languages allow functions to be stated concisely, but do not inherently provide features such as strong data typing or abstract data types, which would form a "higher level" language defined on the applicative language.

Analytical Tools

There are a large number of languages which enable specifications written in them to be statically analysed, including flow algebra (Milner (51)), path expressions (Campbell & Haberman (8)), COSY (Lauer et al (45)), the lambda calculus (Stoy (69)), Petri Nets (Holt & Commoner (33)), regular expressions (Pulford (57)) and SPECK (Quirk (58)), but all are incomplete when viewed as specification languages. They all concentrate on some portion of the specification (e.g. resource allocation or message sequencing), and do not attempt to formalise the rest of the information.

State Transition Specifications

The mathematical theory of finite state machines provides a method of specifying the responses to input stimuli without resorting to algorithms; the theory also provides a method of checking the completeness of the specification.

Early methods of this type used the state transition diagram as their basic notation (see Kawashima et al (44) and Hemdal (27)), and this has been carried forward into later notations such as SDL (CCITT (9)); however, the lack of formality in the text associated with the diagrams was a serious limiting factor in their use. A number of finite state methods avoid the use of diagrams for this reason, e.g. CDL (Dietrich (15)), and the notation due to Parnas (Parnas (56)).

All the above finite state methods have the same disadvantage when applied to large systems, especially those involving concurrent activities - the number of possible system states rises extremely rapidly with the size of the system to be specified. This "state explosion" (Cohen (11)) means that specifications for large systems can become incomprehensible.

The hierarchical design method and SPECIAL language of Robinson (59) can overcome this problem in cases where it is reasonable to restructure the specification as a hierarchy of abstract machines, each building upon the next lower level machine.

Input-Output Relationships

This category includes the largest number of languages, of a wide range of styles, but all based upon specifying the relationship between input stimuli and output responses without describing an algorithm; in this they are similar to the state transition methods, but they do not demand unique identification of each system state. There are three main subdivisions within the category: graphic, relational and pre- and post conditions.

Graphic. The most complete example of a graphic input-output specification is the Predicate/Transition-Nets of Genrich and Lautenbach (19); these are an extension of the Petri Net which formalises the data transformation in mathematical notation.

Also of this type is the Jackson design technique (Jackson (38)), where the graphic display of inter-process communication is purposely stressed more than the internal data transformations performed by the processes.

Relational. BDL (Hammer et al (26)), CADIS (Bubenko & Kallhammer (6)), CASCADE (Solvberg (66)), DATAFLOW (NCC (54)), DMTLT (Sernadas (64)), HOS (Hamilton & Zeldin (25)), Information Algebra (CODASYL (10)), LEGOL (Stamper (67)), PSL (Teichrow (70)), Systematics (Grindley (21)) and Systematrix (Jaderlund (39)) all treat the specification information as relations in a text presentation.

Pre- and Post Conditions. The specification of functions by the necessary pre-conditions and post-conditions can assist in program verification (Dijkstra (16)); Jones (41) and Cunningham & Kramer (12) give examples of the use of this method on reasonably large functions. A similar type of specification, but using rewriting rules rather than logic nota-

tion, is Metaprogramming (Lawson (46)).

Not all the input-output relationship languages are sufficiently formal, despite having rigorous definitions, and only the graphic ones and DMTLT (63) and LEGOL (66) represent time sequence adequately. Also, they are all much better at the specification of data transformations (functions) than of complex data structures.

Axiomatic Specifications

Axiomatic specifications have been introduced mainly as a way of providing abstraction for data types, but as this is done by defining the permitted operations on the data it can be used for "systems".

Two main forms of axiomatic specification language have appeared: one uses first order logic, as in the work by Hoare (31), the iota language of Nakajima et al (53) and the Z language of Abrial (1); the other form, which has proved more popular, is based upon the theory of many-sorted algebras (Lawvere (47)).

Much of the development of the algebraic form is due to Guttag (Guttag (22), Guttag & Horowitz (23), Guttag (24)), but with similar languages being proposed by Burstall & Goguen (7), Liskov & Zilles (48) and Musser (52). Goguen in particular reports upon the implementation of computer assistance for his language, OBJ (Goguen (20)).

The first order logic and algebraic forms are equivalent in capability, and are good for specifying complex data types; both suffer from the difficulty of selecting a complete and consistent set of axioms - there are only heuristic rules to aid in this selection, with no guarantee of success.

SUMMARY

Table 1 draws together the appropriate points from the review; in each case the comment relates to the best language in each category.

TABLE 1 How the Language Measure up to the Specification

Characteristic or Feature	Category of Language: Doc. Aids	Algorithmic	Applicative	Analyt Tools	State Transition	I/O Specn.	Axiomatic Specn.
Formality	Bad	Good	Good	Good	Good	Good	Good
Comprehensibility	Good	Poor	Fair	Poor	Good	Good	Good
Minimality	Poor	Poor	Fair	*	Good	Good	Good
Ease of change	Poor	Poor	Poor	*	Poor	Good	Good
Tolerance of incompleteness	Good	Poor	Poor	*	Fair	Poor	Poor
Strong data typing	No	Yes	Yes	*	No	Yes	Yes
Abstract spec. of data types	No	No	No	*	No	No	Yes
Localisation of References	Poor	Poor	Poor	*	Poor	Good	Good
Representation of time	Poor	Poor	Poor	*	Poor	Good	Poor
Computer Assistance	Good	Fair	Fair	Good	Good	Good	Good
Handling Complexity	Good	Poor	Fair	Good	Bad	Fair	Poor

Note * - these are not applicable to the analytical tools.

CONCLUSIONS

As Table 1 shows, none of the languages fulfil all the requirements; those which appear to come closest (e.g. Predicate/Transition nets, Z, axiomatic specifications) have not yet been demonstrated on any large systems, so there is no evidence of the relevance of their deficiencies in relation to practical telecommunications problems.

The most practical course of action therefore appears to be to provide a "toolkit" for the systems analyst, who can then choose a method appropriate to the problem, or try several until an acceptable specification results. However, in order to minimise training in the use of notation, some kind of common, consistent framework is needed for all the tools, as was developed for APL by Iverson (37).

ACKNOWLEDGEMENTS

This work, which is taking place under the Interdisciplinary Higher Degrees scheme at the University of Aston in Birmingham, is supported by GEC Telecommunications and the Science Research Council.

Particular thanks go to my supervisors (John Flood, Rex Ford, Nigel Horne and Alan Montgomerie) and Bernie Cohen for constructive criticism of the manuscript.

REFERENCES

1. Abrial, J.R., 1980, "The Specification Language Z - Syntax and Semantics", Programming Research Group, Univ. of Oxford.

2. Alford, M.W., 1977, IEEE Trans. on Software Eng.,SE-3, 60-69.

3. Ambler, A.L., Good, D.I., et al, 1977, "GYPSY : A Language for Specification and Implementation of Verifiable Programs". Proc. ACM Conf. on Language Design for Reliable Software.

4. Backus, J., 1978, Comm. ACM, 21, 613-641.

5. Balzer, R., Goldman, N.,and Wile, D.,1978, IEEE Trans. on Software Eng., SE-4, 94-103.

6. Bubenko, J., and Kallhammer, O., 1971, "CADIS - Computer Aided Design of Information Systems". Proc. first Scandinavian Workshop on Computer aided Info.Sys. Analysis and Design.

7. Burstall, R.M., and Goguen, J.A., 1977, "Putting Theories Together to Make Specifications", Proc. Internat. Jnt. Conf. on Artif. Intell.

8. Campbell, R.H., and Habermann, A.N., 1974, "The Specification of Process Synchronization by Path Expressions" in Lecture Notes in Computer Science 16, Springer-Verlag, Berlin.

9. CCITT Working Party X-I/3-1, 1976 "Functional Specification and Description Language, SDL", Temporary Document No.35E.

10. CODASYL Language Structure Group, 1962, Comm. ACM, 5.

11. Cohen, B., 1980, "System Specification Hardware and Software - as Practiced in the Telecommunications Industry", CREST Course, Brunel Univ., London, England.

12. Cunnigham, R.J., and Kramer, J., 1977, "An approach to the Design of Distributed Control System Software", Proc. IEEE Internat. Conf. on Distrib. Control Systems.

13. Dahl, O.J., and Nygaard, K., 1966, Comm. ACM, 9, 671-678.

14. Davis, A.M., Miller, T.J., Rhode, E. Taylor, B.J., 1979, "RLP-An Automated Tool for the Processing of Requirements", IEEE COMPSAC 79.

15. Dietrich, R., 1979, "On a Compilable Call Processing Specification", Proc. Internat. Switching Symposium, Paris.

16. Dijkstra, E.W., 1976, "A Discipline of Programming", Prentice-Hall, Englewood Cliffs, N.J.

17. Estrin, G., 1978, "A Methodology for the Design of Digital Systems - Supported by SARA", AFIPS Conf. Proc., Vol. 47.

18. Gatto, O.T.. 1964, Comm. ACM, 7, 425-432.

19. Genrich, H.J., and Lautenbach, K., 1979, "The Analysis of Distributed Systems by Means of Predicate/Transition-Nets", in Lecture Notes in Computer Science 70, Springer-Verlag, Berlin.

20. Goguen, J.A., 1979, "An Introduction to OBJ; A language for writing and Testing Formal Algebraic Program Specifications", Proc. IEEE Conf. on Specifications of Reliable Software.

21. Grindley, C.B.B., 1975, "Systematics - A New Approach to Systems Analysis", McGraw-Hill Ltd, London, England.

22. Guttag, J., 1977, Comm. ACM, 20, 396-404.

23. Guttag, J. and Horowitz, E., 1978, Comm. ACM, 21, 1048-1064.

24. Guttag, J., 1979, "Notes on Type Abstraction" Proc. IEEE Conf. on Specifications of Reliable Software.

25. Hamilton, M., and Zeldin, S., 1976, IEEE Trans. on Software Eng., SE-2, 9-32.

26. Hammer, M.M., Howe, W.G. and Wledawsky, I., 1974, SIGPLAN Notices, 9 (4), 25-33.

27. Hemdal, G., 1973, "The Function Flowchart-A Specification and Design Tool for S.P.C. Exchanges", Proc IEE SETSS Conf.

28. Henderson, P., and Snowden, R.A., 1972, BIT, 12, 38-53.

29. Hewitt, C., and Bishop, P., 1973,"A Universal Modular Actor Formalism for Artificial Intelligence", Proc. 3rd Internat. Jnt. Conf. on Artif. Intell.

30. Hill, I.D., 1972, BCS Computer Bulletin, 16, 306-312.

31. Hoare, C.A.R., 1972, Acta Information, 1, 271-281.

32. Holbeck-Hanssen, E., Handlykken, P. and Nygaard, K., 1975, "Delta Project Report No.4", Norwegian Computer Centre Publication No. 523, Oslo.

33. Holt, A.W. and Commoner, F., 1970,"Events and Conditions", Applied Data Research, New York.

34. IBM, "Improved Programming Technologies: An Overview", IBM GHC20-1850.

35. Ichbiah, J.D., et al, 1979, SIGPLAN Notices, 14 (6).

36. Iverson, K.E., 1962, "A programming Language", Wiley & Sons, New York.

37. Iverson, K.E., 1980, Comm. ACM, 23(8), 444-465.

38. Jackson, M.A., 1978,"Information Systems : Modelling, Sequencing and Transformations", Proc. 3rd. IEE SETSS Conf..

39. Jaderlund, C., 1980, "Systematrix Concepts", Systematik AB, Stockholm, Sweden.

40. Jensen, K., Kyng, M. and Nielsen, M.,1979, "A Petri Net Definition of a System Description Language", in lecture notes in Computer Science 70, Springer-Verlag, Berlin.

41. Jones, C.B., 1980, "Software Development: A Rigorous Approach", Prentice-Hall International, Englewood Cliffs, N.J..

42. Jones, W.T. and Kirk, S.A., 1980, Computer Journal, 23, 230-232.

43. Karp, A., 1978, Electronic Design, 26, 84-88.

44. Kawashima, H., Futami, K. and Kano, S., 1971, IEEE Trans. Comms. Technology,COM-19, 581-587.

45. Lauer, P.E., Torrigiani, P.R. and Shields, M.W., 1979, Acta Informatica, 12, 109-158.

46. Lawson Jr., H.W., 1977, "Programming, Atchitecture and Complexity", Report LITH-MAT-R-1977-28, Linkoping University, Sweden.

47. Lawvere, F.W., 1963, Proc National Academy of Science, 50 869-872.

48. Liskov, B. and Zilles, S. 1977, "An Introduction to Formal Specifications of Data Abstractions", in Current Trends in Programming Methodology, Vol 1, ed.R.T. Yeh, Prentice-Hall Inc, Englewood Cliff, N.J.

49. Mackie, L., 1977, "Software Reliability-Understanding and Improving It", Proc AGARD Conf. on Avionics Reliability.

50. McCarthy, J. et al, 1965, "LISP 1.5 Programmers Manual", MIT Press.

51. Milner, R., 1978, "Algebras for Communicating Systems", Report CSR-25-78, Dept. of Computer Science, Univ. of Edinburgh.

52. Musser, D., 1979, "Abstract Data Type Specification in the Affirm System", Proc. IEEE Conf. on Specifications of Reliable Software.

53. Nakajima, R., Honda, M. and Nakahara, H., 1978, "Describing and Verifying Programs with Abstract Data Types", in "Formal Descriptions of Programming Concepts", ed. E.J. Neuhold, North-Holland Pub. Co.

54. NCC Ltd., 1969, "DATAFLOW - Project Evaluation Report", Manchester, England.

55. Noe, J.D., 1975, "Pro-Nets, for Modelling Processes and Processors", Conf. on Petri Nets and Related Topics, M.I.T.

56. Parnas, D.L., 1972, Comm. ACM, 14, 330-336.

57. Pulford, R.J., 1979, "The Use of Graph and Regular Expression Models in System Modelling", Report 317/SE/SARSA/WP1, Marconi Avionics Ltd, Borehamwood, England.

58. Quirk, W.J., 1978, "The Automatic Analysis of formal Real-time System Specifications". Report AERE-R 9046, H.M.S.O., London.

59. Robinson, L., 1976, "Specification Techniques", Proc. 13th Annual Design Automation Conference.

60. Rose, C.W. and Albarran, M., 1975, "Modelling and Design Description of Hierarchical Hardware/Software Systems", Proc.12th Annual Design Automation Conference.

61. Ross, D.T., 1977, IEEE Trans. on Software Eng. SE-3, 6-15.

62. RSRE, 1978, "The Official Definition of MASCOT", RSRE, Malvern, England.

63. Schwartz, J.T., 1973, "On Programming : An Interim Report on the SETL Project", Courant Institute of Mathematical Sciences, New York University.

64. Sernadas, A., 1979, "Temporal Aspects of Logical Procedure Definition", report from London School of Economics, England.

65. Sleight, T.P. and Kossiakoff, A., 1974, "Use of Graphics in Software Design, Development and Documentation", Report APL/JHU TG 1242, John Hopkins Univ., Maryland, USA.

66. Solvberg, A., 1973, "Formal Systems Description in Information System Design", in "Approaches to System Design", NCC Ltd, Manchester, England.

67. Stamper, R.K., 1977, Computer Journal, 20 102-108.

68. Stay, J.F., 1976, IBM System Journal, 15, 143-154.

69. Stoy, J.E., 1977, "Denotational Semantics-The Scott-Strachey Approach to Programming Language Theory", MIT Press.

70. Teichrow, D. and Herschey, E.A., 1977, IEEE Trans. on Software Eng., SE-3, 41-48.

71. Wayne, M.N., 1973, "Flowcharting Concepts and Data Processing Techniques", Canfield Press.

72. Zilles, S., 1976, "Data Algebra : A specification Technique for Data Structures", PhD thesis, MIT, Cambridge, Mass.