

QI QUÆSTIONES INFORMATICÆ

Volume 5 • Number 1

April 1987

G.R. Finnie	On Learning Styles and Novice Computer Use	1
P.S. Kritzinger	Local Area Networks in Perspective	11
S. Berman	Semantic Information Management	19
P.C. Pirow	Research Computeracy	23
C.H. Hoogendoorn	Experience with Teaching Software Engineering	36
C. Leveux	Education Rather than Training	41
D. Podevyn	Decision Tables as a Knowledge Representation Formalism	46
J. Roos	The Protocol Specification Language ESTELLE	51
L.J. van der Vegte	The Development of a Syntax Checker for LOTOS	63
	<i>BOOK REVIEWS</i>	71

An official publication of the Computer Society of South Africa and of the South African
Institute of Computer Scientists

'n Amptelike tydskrif van die Rekenaarvereniging van Suid-Afrika en van die Suid-
Afrikaanse Instituut van Rekenaarwetenskaplikes

QUÆSTIONES INFORMATICÆ

An official publication of the Computer Society of South Africa and of the
South African Institute of Computer Scientists

'n Amptelike tydskrif van die Rekenaarvereniging van Suid-Africa en van die
Suid-Afrikaanse Instituut van Rekenaarwetenskaplikes

Editor

Professor G. Wiechers
INFOPLAN
Private Bag 3002
Monument Park 0105

Dr P.C. Pirow

Graduate School of Business Admin.
University of the Witwatersrand
P.O. Box 31170, Braamfontein, 2017

Professor S.H. von Solms
Department of Computer Science
Rand Afrikaans University
Auckland Park
Johannesburg, 2001

Professor M.H. Williams
Department of Computer Science
Herriot-Watt University, Edinburgh
Scotland

Editorial Advisory Board

Professor D.W. Barron
Department of Mathematics
The University
Southampton SO9 5NH, UK

Professor J.M. Bishop
Department of Computer Science
University of the Witwatersrand
1 Jans Smuts Avenue
2050 WITS

Professor K. MacGregor
Department of Computer Science
University of Cape Town
Private Bag
Rondebosch, 7700

Prof H. Messerschmidt
University of the Orange Free State
Bloemfontein, 9301

Circulation and Production

Mr C.S.M. Mueller
Department of Computer Science
University of the Witwatersrand
2050 WITS

Subscriptions

Annual subscription are as follows:

	SA	US	UK
Individuals	R10	\$ 7	£ 5
Institutions	R15	\$14	£10

Quæstiones Informaticæ is prepared by the Computer Science Department of the
University of the Witwatersrand and printed by Printed Matter, for the Computer
Society of South Africa and the South African Institute of Computer Scientists.

THE DEVELOPMENT OF A SYNTAX CHECKER FOR LOTOS

L J van der Vegte
*Department of Computer Science,
University of Pretoria*

INTRODUCTION

In this article a brief overview is given of specification techniques in general and in particular, specification techniques used for the specification of communication protocols. This discussion forms the background against which the development of a syntax checker for LOTOS, LOTOSCHECK, was done. LOTOS is a specification technique which was developed during 1981-1984 for the specification of communication protocols. Development tools such as LOTOSCHECK are required in order to eventually semi-automate the lengthy and complex task of implementing and verifying a protocol.

In the first section of the article the concept of a specification is discussed in general. In Section 2 the increasing interest in specification techniques in the area of data communications is discussed. In this section, reference is made to some of the formal description techniques for protocol specification which are currently being developed. LOTOS is introduced as one of these techniques which is currently receiving a lot of attention.

An overview of LOTOS will be given in Section 3. This section will give a brief introduction to LOTOS and describe some of the more interesting characteristics of LOTOS.

In Section 4 the actual development of LOTOSCHECK is discussed. The parsing technique which was used will be discussed in particular.

The article concludes with references to other work already done in the area of semi-automated implementations of communication protocols.

1. SPECIFICATION AND SPECIFICATION TECHNIQUES

In recent years, the question of specification has given rise to much research within the Computer Science community. The interest in formal specification as an important phase during software development stems from experiences with unreliable, inefficient and incorrect implementations resulting from inaccurate designs.

The use of formal specification techniques has not been readily accepted by all people involved with software development. However, many advantages gained from the use of formal specification techniques, namely unambiguous, clear and concise specifications on which implementations can be based and tested against for conformance to the original requirements, are resulting in more support for these techniques in practice. This is specifically the case in the field of data communications [4].

A variety of specification techniques are available. These techniques can be broadly classified as informal, semi-formal or formal [14]. Informal methods have no mathematical basis, for example a natural language such as English. Semi-formal methods do have some mathematical basis, but this aspect of the method is not emphasised in its use. Formal techniques are entirely based on mathematics and consist of symbols plus rules for manipulating these symbols.

Specification techniques used for the specification of communication protocols can be categorised as formal and form the basis of the following discussion.

2. SPECIFICATION AND DATA COMMUNICATION PROTOCOLS

The large amount of software development currently being done in the area of data communications, has resulted in much interest in the development of specification techniques particular to this field of application. The following section will give a brief overview of work done in this area.

2.1 The interconnectibility of heterogeneous computer systems

Computer systems located all over the world should be able to communicate with one another. Each of these computer systems consists of hardware from various manufacturers, a variety of accompanying software, peripherals, human operators, real-time processes, etc. which together support a large variety of applications. A flexible method is required to facilitate the exchange of information between computer systems in a standardised manner.

Committees concerned with standardisation had, as early as 1960, discussed ideas on the standardisation of the development of data communication applications. These discussions led to the International Organisation for Standardisation (ISO) Technical Committee 97 establishing a new Subcommittee (SC 16) whose function was to launch a master plan to deal with the definition, development and verification of data communication applications. The Subcommittee (SC 16), known as the Open Systems Interconnection Subcommittee, developed a standard basic structure which could be used to define the functionality needed for communication between remotely separated application processes in heterogeneous network systems. This basic structure resulted in the OSI (Open Systems Interconnection) Reference Model which was approved as an international standard at the beginning of 1983 [6].

The OSI Reference Model serves as the major reason why there is currently so much interest in formal description techniques for the specification of protocols. Effective, open communication between interconnected computer systems demands strict adherence to this proposed standard data communications procedure.

The OSI Reference Model is written in English and is supported by many informal diagrams illustrating various aspects referred to in the text. The specification of any system in a natural language such as English can lead to subtle differences in interpretation and incompatible implementations in terms of interconnectibility. Thus, for the OSI Reference Model to be effectively used, the model needs to be specified in a more formal and unambiguous way. There are currently a number of formal description techniques which are receiving attention from international standards committees with the aim of developing a standard technique which can be used internationally to specify the protocol requirements of the OSI Reference Model. These specification techniques are referred to as Formal Description Techniques (FDTs). Two of these techniques are ESTELLE and LOTOS.

2.2 Formal description techniques for communication protocols

ESTELLE

ESTELLE is a formal description technique based on the extended finite state machine model and is currently being studied by the ISO/Technical Committee 97/Subcommittee 21/Working Group 1/Subgroup B. (It should be noted that as a result of the re-organisation of ISO's Subcommittees, Subcommittee 16 now functions as Subcommittee 21). Although based on the finite state machine model, the notation used in ESTELLE is not graphical, but procedural, very much like Pascal. The procedural nature of ESTELLE makes it quite easy to use and the complexities of a protocol can thus adequately be modelled by ESTELLE.

In the initial formal specification of ESTELLE published by ISO in April 1985, no formal definition of the semantics of ESTELLE was given, thus making analyzability of an ESTELLE specification very difficult or almost impossible. The most recent draft proposal of ESTELLE published by ISO describes the syntax of each allowable ESTELLE construction and a short narrative describing the semantics of the construction is provided. Each short narrative is by no means formal or mathematically based and this could result in difficulties as far as the formal verification of an ESTELLE specification is concerned [9].

ESTELLE is thus certainly a most useful and powerful formal description technique which can be used for the specification of data communication protocols, but there are people with specifically verification in mind, who would prefer a more mathematically based technique. LOTOS, described in the following paragraph, is such a technique.

LOTOS

LOTOS is currently being studied by the ISO/Technical Committee 97/Subcommittee

21/Working Group 1/Subgroup C. This specification technique is based on Milner's Calculus of Communicating Systems (CCS) and makes use of an algebraic notation [13].

The modelling power of LOTOS is excellent and is possibly more concise than ESTELLE. Furthermore, the ISO specification of LOTOS makes provision for a formal definition of the semantics of LOTOS, making analysability possible, although much research is still to be done in this area [10]. LOTOS has already been successfully used for the specification of some of the standard protocols as proposed by the OSI Reference Model. Thus, with LOTOS as a development tool of the future, supportive tools are required. LOTOSCHECK is a small step in this area of research.

Before discussing the development of LOTOSCHECK, an overview of some of the characteristics of LOTOS will be given. The reader requiring more detail is referred to the draft proposal of LOTOS issued by ISO, as well as a LOTOS tutorial which was drawn up by one of the original designers of LOTOS [10], [11].

3. LOTOS AND SOME OF ITS CHARACTERISTICS

LOTOS (Language of Temporal Ordering Specification) was developed by the ISO/TC97/SC21/WG1/FDT/Subgroup C during the years 1981-1984. The mathematical model of LOTOS is based on a modification of the Calculus of Communicating Systems (CCS) described by Milner, called CCS* [13]. The CCS* model represents a system as a set of processes and sub-processes which interact with each other and with their environment. These interactions take place at shared interaction points called gates, and are themselves called *events*. The allowable events associated with a process are defined as part of the specification of a process.

A LOTOS specification can be divided into a static and a dynamic section. The first section, or static section, is used for the specification of user defined types. The second section of a LOTOS specification describes the processes which together make up the system which is being specified. This is the dynamic section of the specification.

Section 3.1 will briefly discuss the static part of a LOTOS specification and in Section 3.2 the dynamic part will be described.

3.1 Data typing in a LOTOS specification

The definition of the sets of data values and the allowable operations on these data values which will be used in a LOTOS specification, is given in the static part of the specification. These definitions are based on a self-contained language called ACT ONE and are given in terms of a type construction [1]. ACT ONE is a specification method which makes use of algebraic equations in order to define data structures and permitted operations for these data structures.

3.2 Processes and behaviour expressions

The dynamic part of a LOTOS specification is defined in terms of processes. The notation used to define a process is $p(x_1, \dots, x_n) := B$ where p represents the name of the process, or behaviour identifier, x_1, \dots, x_n represents the variables which are used in the process and B represents the total behaviour of the process. This form is also termed a *process abstraction*. A process can make use of other processes in describing its behaviour. These references precede the actual definitions of the processes and are called *process instantiations*. A process instantiation is represented as $p(E_1, \dots, E_n)$, where E_1, \dots, E_n represent the actual values of x_1, \dots, x_n . The use of processes is thus much like the use of subroutines, where the process abstraction defines the logic of the process and its expected parameters, and the instantiation represents the activation of the process with the use of arguments in the place of the parameters.

The body of a process describes the observable behaviour of the process in terms of behaviour expressions. A *behaviour expression* represents the behaviour of a process in terms of sequences of possible events.

The most basic behaviour expression is *stop*. This expression indicates that the process which contains this behaviour expression will terminate its execution on reaching this point.

Complex behaviour expressions which describe the entire behaviour of a process, are made

up of simple behaviour expressions joined together by certain allowable operations. The most basic of these operations are the *action prefix* operator (indicated by ';'), and the *choice* operator (indicated by '[]'). The action prefix operator may be used to create a sequence of behaviour expressions, such as B1;B2;B3, etc. This sequence is interpreted as a temporal ordering of the behaviour expressions B1, B2 and B3. The choice operation introduces a measure of non-determinism to the behaviour of a process. The behaviour expression B1 [] B2 in a process indicates that the process will behave either as B1 or B2, depending on what event is offered by the process' environment.

The behaviour of any process could be described by using the action prefix and choice operators as the only allowable operators. This does however lead to exhaustively listing each possible behaviour pattern of a process. This would naturally lead to lengthy specifications. Other operations are thus available in LOTOS to simplify matters and will now be discussed.

3.3 Some CCS* operators

In addition to the action prefix and choice operators, LOTOS also has, amongst others, operators for the parallel composition of processes, the enabling and disabling of processes, restriction and guarding.

The parallel composition operators make provision for the concurrent interleaving of processes. The '|||' symbol is used to denote parallel execution of two or more processes, where no communication between the processes occurs. The events of each process take place in a specified order, but are interleaved in a manner which cannot be pre-determined. The parallel composition of two communicating processes is indicated by '||'. Two processes B1 and B2 which can communicate are thus represented as B1 || B2, and synchronise with respect to common events.

The environment can however have an effect on the communication between two processes. Sometimes this interference should not be possible. The restriction operator ('\') makes provision for this. The above two processes can communicate via interaction point 'a' without any interference from the environment at point 'a' if represented as follows: (B1|| B2)\[a].

The enable ('>>') and disable ('>') operators can be used to allow for the sequencing of processes depending on what events occur. B1 >> B2 implies that process B1 will execute first; should this process terminate successfully, that is, execute the basic *exit* process, then process B2 will begin executing, that is B2 will be 'enabled'. The disabling operator is interpreted as follows: B1 [> B2 implies that B1 will execute, but as soon as an event occurs which matches the first allowable event of B2, B1 will terminate and B2 will execute.

The guarding operator ('[E] ->') is similar to the conditional execution of an operation. In the example [E] -> B1, E represents a Boolean expression and B1 will only be executed if the value of E is true. If E is false, the process stops.

3.4 Action denotations and action-prefix expressions

An *action denotation* is used to indicate the potential occurrence of an event at an interaction point or gate. The notation used consists of the name of the interaction point followed by an exclamation mark followed by the event, for example a!E1 implies that an event E1 could possibly occur at gate a. The action denotation is associated with output being generated by a process at an interaction point.

An *action-prefix expression* is used to denote the fact that a process will accept input from its environment at a certain interaction point. The notation which is used to represent this activity is the name of the interaction point, followed by a '?', followed by a variable of a certain sort. Thus, 'b?x:wxy' implies that a variable x of type wxy will be input at interaction point b. Once this variable has been accepted at interaction point b, the associated process will continue with the following behaviour expression.

This concludes the brief overview of LOTOS and the actual development of LOTOSCHECK will now be discussed.

4. THE DEVELOPMENT OF LOTOSCHECK

A precompiler is a compiler in which the source code of a high level language is translated into another high level language, for example, a FORTRAN IV version of a program could be translated into a FORTRAN 77 version of that same program. LOTOSCHECK forms the first part of a precompiler for LOTOS. At this point in time, it seems possible that a complete precompiler for LOTOS could be developed along the lines of the techniques used by Hoare in his work on communicating sequential processes, with LISP or PROLOG being the eventual 'object code' [5].

The development of a compiler or precompiler can be done in phases or modules. The main phases of this development process are, lexical analysis, parsing, intermediate code generation, the optimisation of the intermediate code which is an optional phase, and finally, code generation. LOTOSCHECK is an implementation of the first two of these phases.

4.1 Lexical analysis

The input to the lexical analyser is the original source code. The lexical analyser's function is to extract strings of characters, called tokens, from the source code. Tokens are the atomic building blocks of a language, for example, punctuation symbols, identifiers, keywords, operators, etc. There are no specific techniques which dictate the way in which lexical analysis is done. The programmer must know what constitutes a correct token and the extraction of these tokens from the source code entails straightforward character string manipulation.

The strings of characters which constitute tokens in LOTOS are the normal identifiers, constants, keywords, punctuation symbols and a few special symbols as mentioned in Section 3. Many of these special symbols make use of the square brackets, '[' and ']'. Most standard keyboards do not make provision for these square brackets, so the combination '(-' has been used in LOTOSCHECK to indicate the use of '[' and '-') replaces the use of ']'. The disable-symbol in LOTOSCHECK is, for example, represented as '(->' instead of ']>'. The other symbols containing square brackets are similarly adjusted. For the remainder, the set of symbols as defined in the original specification of LOTOS, is used [10].

The lexical analyser for LOTOSCHECK consists basically of four modules. The first of these modules is responsible for reading the source code. A large buffer is used to store small pieces of the input. This module continually checks to see that the buffer contains some unprocessed input, and if not, buffers in the next couple of input lines. The second module is responsible for the generation of a listing file of the original source code. Each line of source code which is printed in the listing file is numbered. These numbers are referred to in error messages which are generated during parsing. The actual extraction of the tokens is done by the third module and the fourth module makes entries in a symbol table. The lexical analyser uses the symbol table to set up a cross reference map for all the identifiers which are used in the specification. This cross reference map is printed just after the listing of the source file and specifies the type and line usage of each identifier. The type of many of the identifiers can be extracted from the explicit definition of the identifier in a type construction. This construction is used to specify user-defined types. The remainder of the identifiers are not explicitly defined and their types are derived from the context in which they are used. These identifiers include the name of the specification, process names, gate identifiers and value identifiers.

A degree of error handling is provided by the lexical analyser. Any character which is not a member of LOTOS' alphabet of allowable characters will be ignored and an error message to this effect will be printed. Lexical analysis will however continue with a blank replacing the incorrect character. Errors found while setting up the cross reference map include the multiple declaration of identifiers and the illegal use of certain types in various positions. Identifiers are initially assigned a type based on a definition of the identifier or on the context in which the identifier is first used. Any re-definition of an identifier will be interpreted as a multiple declaration and will be flagged by an appropriate error message. The contextual usage of identifiers is also checked and error messages are printed in the case of any contradictory usage of identifiers.

4.2 Parsing

The function of the parser is to determine whether the given source code is syntactically

correct. In contrast to the lexical analysis phase, there are many definite parsing techniques which could be used. These techniques are either top-down or bottom-up, that is, an attempt is made to derive a sentence from the start symbol of the grammar or, to reduce a sentence to the start symbol of the grammar.

Many of the better known parsing techniques were considered before a final choice for LOTOSCHECK was made. Well known examples of bottom-up parsers include operator precedence parsers and the LR-parsers. The operator precedence parser is table-driven and although it is easy to implement, it is best used for the parsing of expressions and not for the more complex programming language constructions. The LR-parsing techniques can be efficiently used for almost any programming language construction, but were not considered for a number of reasons. The parsing tables used by LR-parsers for large grammars do become excessive. A grammar with 100 terminal symbols and 100 productions could result in a table of 20000 entries or more. Initially the precompiler was to be developed for a microcomputer environment and tables of this size make excessive demands on the available memory on such a microcomputer. Furthermore, the amount of work involved in correctly setting up a table of this size by hand is enormous. Parser generators such as YACC (Yet Another Compiler Compiler) can be used to automatically generate the required tables [2]. The absence of such an automatic parser generator and not wishing to attempt the daunting task of drawing up the table by hand, were thus the major reasons for not using the LR-parsing techniques.

The parsing technique which was used for LOTOSCHECK was taken directly from the work of Nicklaus Wirth [15]. The technique is a top-down technique and traces out a derivation of the given sentence from the start symbol. The technique is not table-driven, but makes use of a large linked list data structure. Other top-down parsing techniques such as recursive descent parsing and predictive parsing were also considered but each had its own disadvantages. Recursive descent parsing makes use of recursive procedures for each of the non-terminal symbols in the grammar. A grammar having 100 non-terminal symbols or more thus gives rise to 100 or more procedures which repeatedly call each other. The co-ordination of such a large number of small modules in contrast to the six required to set up the linked list data structure and a single module to do the actual parse, would only complicate the implementation. Predictive parsing on the other hand is a table-driven technique and drawing up the required parsing table manually once again seemed to demand an excessive amount of time.

The technique used for LOTOSCHECK thus represents the productions of the grammar in a linked list structure and a trace through this structure is made to ascertain whether a sentence is syntactically correct or not.

To set up this linked list structure, the BNF (Backus Nauer Form) of the grammar is represented in terms of a meta-language as follows:

```
<production> ::= <symbol> = <expression>.
<expression> ::= <term> {, <term>}.
<term> ::= <factor> {<factor>}.
<factor> ::= <symbol> | [<term>].
```

where { } represents zero or more occurrences, and
 , represents alternate productions in the original productions

and [] represents zero or more occurrences, and
 | represents alternate productions in the meta-language

The linked list structure which is created from this representation of the productions consists basically of two types of linked lists. The first is a linked list which contains all the non-terminal symbols of the grammar. The header of this list is the start symbol. The second type of linked list represents the right hand side of a production. Each of these linked lists has the non-terminal which appears on the left hand side of the production as its header. The nodes of these linked lists are of two types, namely those for terminal symbols and those for non-terminal symbols. 'Successor' links are used to link adjacent symbols in the list. The 'alternate' link of each of the nodes for the terminal and non-terminal symbols is used to link in any alternate productions which are derivable from the same non-terminal symbol.

The 'zero or more occurrences' of a certain symbol or string of symbols is also provided for. This is done by linking the last symbol in such a string to the first and providing the first symbol in the string with an 'empty' alternate.

An example of two productions represented in terms of the linked list data structure is given

in Figure 1.

Productions:

$A = Bd$, if.

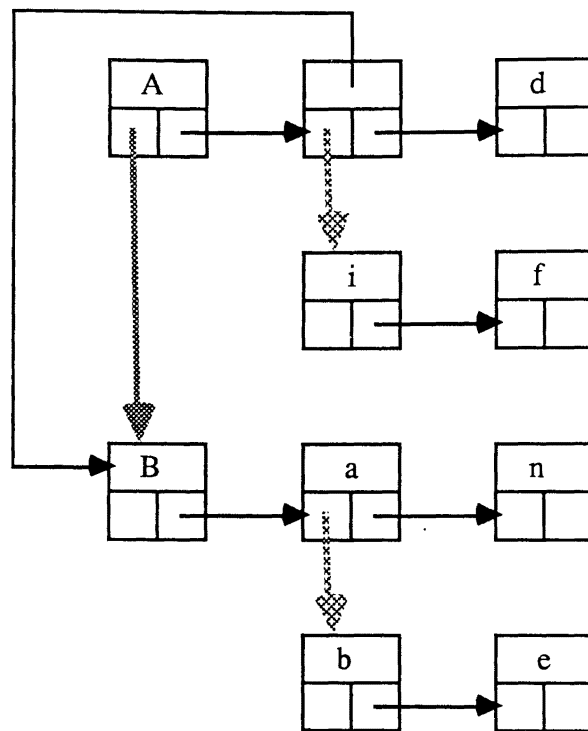
$B = an$, be.

A is the start symbol.

A, B are non-terminal symbols.

a, b, d, e, f, i, n are terminal symbols.

Linked list structure:




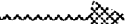

Key : 'successor' links 
 'alternate' links 
 links in the lists of non-terminal symbols 

figure 1

An Example of the Data Structure Used in the Parsing Phase of LOTOSCHECK

This technique for parsing actually results in the parser being somewhat of a 'black box'. In other words, except for error conditions, the parser could be used for any language, given that the grammar for that language is LL(1) and is correctly represented in terms of the given meta-language – another advantage in using the technique as this in itself will certainly lessen the amount of work required in similar projects, for example a syntax checker for ESTELLE.

The grammar which was used as the basis of LOTOSCHECK is taken from a paper summarising further research done in the area of LOTOS and not from the original LOTOS specification [7]. This grammar differs from the grammar given in the original specification of LOTOS [10]. The difference between the two BNF-forms is that the one resulting from the analysis of the original LOTOS specification makes provision for arithmetic expressions in terms of both prefix and infix notation, whereas the original specification only makes provision for

prefix notation. The use of infix notation for arithmetic expressions is far more common than prefix and thus it was decided to make provision for both in LOTOSCHECK.

Although the phases of the compilation process imply that lexical analysis be done before parsing, this is not the case. The parser actually is the main routine and the lexical analyser is activated by the parser each time the next token is required. This is also the case with LOTOSCHECK. The main program of LOTOSCHECK consists of modules which are concerned with various initialisation functions, such as setting up the linked list data structure for the parser, repeated 'calls' to the lexical analyser, the actual parse of the given source code, and finally, various housekeeping routines which print summary information of the number of errors, etc.

The error handling done during the parsing phase is based strictly on what, according to the linked list structure, is the next allowable token which may follow the current one. This method of recovering from an error could be likened to a type of panic mode, that is, any incorrect symbols following the current one are removed until a recognisable token is found, that is, until an allowable symbol is found. This may not be the most stylish way of recovering from errors, but similar methods are used in most top-down techniques which are not table-driven. This is the only major disadvantage of this type of top-down parsing technique.

All error messages are printed following the listing of the original source code and the cross reference map. These error messages refer to the relevant line numbers in which the errors occurred. The format of the error messages is very standard. An indication as to which token was incorrectly placed is given, followed by a list of allowable tokens for that specific position.

4.3 Development and operating environment

LOTOSCHECK is to be used as a development tool in a mainframe environment. Much of the initial development of the syntax checker was however done on an Olivetti M24 microcomputer. The implementation language which was used on the M24 was Digital Research's version of PL/1. This version of PL/1 corresponds almost exactly with the version of PL/1 available in the mainframe environment. The modules developed on the microcomputer were later downline loaded onto the mainframe and very few changes to the original source code were required before the system was running in the mainframe environment.

Testing of LOTOSCHECK was done using some smaller examples taken from the LOTOS tutorial [11], as well as the following more complete examples, namely, a system used to connect Remote Job Entry (RJE) devices across an X.25 network [12] and the trial conformance tests given in [8]. Further tests using complete LOTOS specifications of the Transport and Session layers of the OSI Reference Model are still to be done.

As mentioned at the beginning of this paragraph, LOTOS is to be used in a mainframe environment. It was thus not necessary to develop any other supporting tools for the use of LOTOSCHECK as the full screen editing facilities and other utilities already available in the mainframe environment can be used to create LOTOS specifications. A special run file is used to activate LOTOSCHECK. The user need only type in the word 'LOTOS' and then the name of the file in which the LOTOS specification is stored, and the syntax check will proceed.

5. CONCLUSIONS

A lot of research is being done in the area of semi-automated protocol implementation. Bochman, for example, considers the semi-automation of the protocol implementation activity based on a protocol formally specified in ESTELLE [3]. A Pascal implementation of the specification can be automatically generated. This Pascal code can be included in the eventual implementation of the protocol.

The development of LOTOSCHECK is a similar step in the direction of providing tools which will semi-automate the implementation of communication protocols which are specified in terms of LOTOS. The current use of LOTOSCHECK is limited to the syntactically correct specification of a communication protocol in LOTOS. The next phase of the project is to develop the complete precompiler. As mentioned earlier in this article, it seems possible that this next phase will rely heavily on the work done by Hoare in the area of communicating sequential processes. This will result in a very challenging and interesting area for further research.

REFERENCES

1. H Ehrig, W Fey and H Hansen., [1983], *ACT ONE: An algebraic specification language with two levels of semantics*. Bericht-Nr 83-03, Technical University of Berlin. February.
2. A V Aho and J D Ullman. , [1977], *Principles of compiler design.*, Addison-Wesley Publishing Co., London.
3. G v Bochman, G Gerber and J M Serre., [1984], *Semi-automatic implementation of communicating protocols*. University of Montreal.
4. G J Dickson and P E de Chazal., [1983], Status of CCITT description techniques and application to protocol specification., *Proceedings of the IEEE*, 71, 12, pp. 1346 -1355.
5. C A R Hoare., [1985], *Communicating sequential processes.*, Prentice-Hall Inc., Englewood-Cliffs, New Jersey.
6. ISO/TC97/SC16/WG1/IS7498, [1983] *Information processing systems. Open Systems Interconnection - Basic Reference Model..*
7. ISO/TC97/SC21/WG1/N51, [1985], *A critique of LOTOS.*, June.
8. ISO/TC97/SC21/WG1/N99, [1985], *Trial conformance tests in LOTOS.*, 1985.
9. ISO/TC97/SC21/WG1/N422, [1985], *Information processing systems. Open Systems Interconnection - ESTELLE - A Formal Description Technique based on an extended finite state machine model.*, September.
10. ISO/TC97/SC21/WG1/N423, [1985], *Information processing systems. Open Systems Interconnection - LOTOS - A Formal Description Technique based on the temporal ordering of observational behaviour.* Feb..
11. ISO/TC97/SC21/WG1/N938, [1985], *Provisional LOTOS tutorial.*, November.
12. D G Kourie., [1985], *A partial RJE PAD specification to illustrate LOTOS.* University of Pretoria, May.
13. R Milner., [1980], *A Calculus of Communicating Systems*. Lecture Notes in Computer Science. Springer-Verlag.,
14. K J Turner., [1984], *Towards better specifications*. ICL Technical Journal, May.
15. N Wirth., [1976], *Algorithms + Data Structures = Programs.*, Prentice-Hall Inc., Englewood-Cliffs, New Jersey.

BOOKREVIEW: *Understanding Expert Systems*, Mike Van Horn, Bantam, 1986.
ISBN 0-553-34168-5. 233 pages.

review by: Philip Machanick, *University of the Witwatersrand, Johannesburg 2001*

INTRODUCTION

Everyone is talking about expert systems, but no one knows what they are—give or take a few gurus in research establishments. From this point of view, there should be a ready market for a book which explains everything at a non-technical level. Taking into account the likely readership, the level at which topics are covered is important. Even if the intention is not to write a text book, a certain level of accuracy should be expected as well. Evaluated against these criteria, this book turns out to be a disappointment. Although the choice of topics, depth of coverage and the style of writing are appropriate, many details are wrong or questionable.

These points are examined in more depth.

NEED FOR SUCH A BOOK

Many managers, computer scientists and engineers received their education before expert systems became fashionable. They have a need for understandable introductions; some will only want to be in a position to understand the jargon, while others may desire a starting-point to acquiring deeper knowledge. The former group could probably live with a book with a good conceptual coverage and minimal detail, whereas the latter will want a good idea of the technical issues before they delve into the more specialized literature.

In order to appeal to both groups, a book needs to be very clearly written, with the more technical aspects presented either in a painless way, or in such a manner that they can easily be omitted.

LEVEL OF COVERAGE

Van Horn has taken the first option—painless coverage—which is the more challenging. It requires clear communication, without talking down to the reader. Other than a tendency to become repetitious (and the occasional use of folksy language), he has succeeded. The book is readable, and largely self-contained (although a good selection of references is also given).

His choice of topics and material is good, and will leave the reader with a good general impression of what AI is about, issues in deciding whether or not to build an expert system—technical and commercial, the limitations of the technology, what tools are available and an historical perspective of what has been and what is likely for the future.

In particular, his selection of systems to cover—given that a book of this nature cannot be expected to give a complete survey—is good. The section detailing the development of tools is based on contrasting MYCIN-based (backward-chaining diagnostic systems), R1-based (forward-chaining systems which fit parts together to form a whole) and HEARSAY-II-based systems (blackboard systems which make sense of large amounts of noisy data). Combined with a clear description of backward and forward chaining and a worked example of building a decision tree by induction using EXPERT-EASE, Van Horn gives a good overall impression of the available options.

FLAWS

Despite these positive points, the book is severely flawed. The author has obviously gone to great lengths to make sense of the huge amount of information available. However, his lack of experience in the field shows in many ways.

For instance, there is some evidence that he has confused the concepts of knowledge base and database. He consistently refers to the rules in the inference engine—most expert systems' inference engines are hard-coded in a language such as LISP. He also does not make a clear distinction between rule-based programming and a language such as LISP. Although rule-based programming is often implemented in LISP, LISP does not directly implement features such as data-driven programming.

The assertion that AI languages are based on predicate logic, rather than on the propositional logic used by conventional languages and the underlying machine was a bit of a surprise. Aside from the fact that this statement is not true, it is contradicted by other information in the book—that LISP is implemented in other languages, such as C. If LISP had a more powerful theoretical basis than other languages or the computer itself, it would be impossible to fully implement it on a conventional machine using a conventional programming language. The issue is not one of theoretical power, but of convenience of notation—how closely the language corresponds to the way you need to program.

To compound the error, he refers to several standard list-manipulation functions as “predicate functions”—a term reserved quite rightly in LISP for functions returning a logical result.

Some other points are not as far off the mark, but still reveal an element of misunderstanding. For instance, backward chaining is claimed to be impractical if all the possible solutions could not be held in memory. The real issue is the problem of being deluged with questions if there is not a relatively small set of solutions, with some clearly more likely than others. Also, the equivalence of data and program in LISP is given as the reason LISP programs are easy to change. This may be a factor in that it makes it easy to write debugging tools, but the real issue is the fact that LISP fits relatively easily into an interactive programming environment, which allows small changes to be made without having to recompile the whole program.

OVERALL

The beginner wanting an introduction leading to a more technical view of the subject would be disappointed with this book. Someone needing a non-technical overview who would skip the details may not be put off too much by the inaccuracies. Van Horn several times stresses a valuable piece of advice—to build an expert system, you need quality expertise. Perhaps a book about writing books about expert systems could contain similar advice: to write books about expert systems, you need quality knowledge about expert systems.

NOTES FOR CONTRIBUTORS

The purpose of the journal will be to publish original papers in any field of computing. Papers submitted may be research articles, review articles and exploratory articles of general interest to readers of the journal. The preferred languages of the journal will be the congress languages of IFIP although papers in other languages will not be precluded.

Manuscripts should be submitted in triplicate to:

Prof. G. Wiechers
INFOPLAN
Private Bag 3002
Monument Park 0105
South Africa

Form of manuscript

Manuscripts should be in double-space typing on one side only of sheets of A4 size with wide margins. Manuscripts produced using the Apple Macintosh will be welcomed. Authors should write concisely.

The first page should include the article title (which should be brief), the author's name and affiliation and address. Each paper must be accompanied by an abstract less than 200 words which will be printed at the beginning of the paper, together with an appropriate key word list and a list of relevant Computing Review categories.

Tables and figures

Tables and figures should not be included in the text, although tables and figures should be referred to in the printed text. Tables should be typed on separate sheets and should be numbered consecutively and titled.

Figures should also be supplied on separate sheets, and each should be clearly identified on the back in pencil and the authors name and figure number. Original line drawings (not photocopies) should be submitted and should include all the relevant details. Drawings etc., should be submitted and should include all relevant details. Photographs as illustrations should be avoided if possible. If this cannot be

avoided, glossy bromide prints are required.

Symbols

Mathematical and other symbols may be either handwritten or typewritten. Greek letters and unusual symbols should be identified in the margin. Distinction should be made between capital and lower case letters; between the letter O and zero; between the letter I, the number one and prime; between K and kappa.

References

References should be listed at the end of the manuscript in alphabetic order of the author's name, and cited in the text in square brackets. Journal references should be arranged thus:

1. Ashcroft E. and Manna Z., The Translation of 'GOTO' Programs to 'WHILE' programs., *Proceedings of IFIP Congress 71*, North-Holland, Amsterdam, 250-255, 1972.
2. Bohm C. and Jacopini G., Flow Diagrams, Turing Machines and Languages with only Two Formation Rules., *Comm. ACM*, 9, 366-371, 1966.
3. Ginsburg S., *Mathematical Theory of Context-free Languages*, McGraw Hill, New York, 1966.

Proofs

Proofs will be sent to the author to ensure that the papers have been correctly typeset and *not* for the addition of new material or major amendment to the texts. Excessive alterations may be disallowed. Corrected proofs must be returned to the production manager within three days to minimize the risk of the author's contribution having to be held over to a later issue.

Only original papers will be accepted, and copyright in published papers will be vested in the publisher.

Letters

A section of "Letters to the Editor" (each limited to about 500 words) will provide a forum for discussion of recent problems.

