ICL Technical Journal

Volume 3 Issue 4 November 1983

ICL Technical Journal

Contents

Volume 3 Issue 4

ICL Technical Journal

The ICL Technical Journal is published twice a year by Peter Peregrinus Limited on behalf of International Computers Limited

All correspondence and papers to be considered for publication should be addressed to the Editor

1983 subscription rates: annual subscription £13.00 UK, £15.50 ($35.00) overseas, airmail supplement £6.00 ($13.00), single copy£7.50 ($16.00). Cheques should be made out to 'Peter Peregrinus Ltd.', and sent to Peter Peregrinus Ltd., Station House, Nightingale Road, Hitchin, Herts. SG5 1SA, England, Telephone: Hitchin 53331 (s.t.d. 0462 53331).

The views expressed in the papers are those of the authors and do not necessarily represent ICL policy

Publisher
Peter Peregrinus Limited
PO Box 8, Southgate House, Stevenage, Herts SG1 1HQ, England

Printed by A.McLay & Co. Ltd., London and Cardiff ISSN 0142–1557

Expert system in heavy industry: an application of ICLX in a British Steel Corporation works

Bruce Hakami
ICL System Strategy Centre, Stevenage, Herts
John Newborn
British Steel Corporation, Scunthorpe, South Humberside

Abstract

The paper deals with a collaborative undertaking between ICL and the British Steel Corporation, the practical application of the expert system concept to the operation of a steel rolling mill. It is in two parts. The first, by Bruce Hakami (ICL) outlines the ICLX expert system 'package', using the rolling-mill application to illustrate the interaction of the user with the system. The second, by John Newborn (BSC) gives the arguments that led to the adoption of the expert system approach as an aid to the diagnosis of faults, and summarises the experience gained.

Part I The ICLX system

1 Introduction

ICLX is a system whose behaviour is governed by the information it holds on a particular subject. The information is initially obtained from human experts and therefore ICLX is referred to as an expert system. The information is obtained and held as a collection of facts and their relationships and is referred to as knowledge. The purpose of the knowledge is to help users solve problems in the particular subject area by a process known as problem solving. The knowledge is obtained from one or more experts by a process known as knowledge refining, so called because the system influences the expert to improve the knowledge. Both problem solving and knowledge refining processes are interactive, allowing the user to observe directly the consequence of his actions.

ICLX began as an experimental project to determine the necessary features of an expert system usable by people who are not necessarily familiar with programming or computers. Our initial ideas were influenced by ICL's own experience in diagnostic aids for computers, CRIB[1] and by major expert systems such as MYCIN[2] and PROSPECTOR[3]; but we wanted to ensure a practical foundation for the project and so we chose the problem of fault diagnosis in a steel rolling mill as a test bed for the development and evaluation of our ideas.

The next Section describes how the knowledge is formulated. The main features of problem solving and knowledge refining processes are described in the two following Sections and Section 5 relates some of the problems we encountered in our work.

2 Knowledge structure

The behaviour of the system is governed by the knowledge it holds. The main components of the knowledge are described below.

2.1 Tests

When a system is undergoing a diagnostic investigation, its current operating state is assessed on the basis of observations. All allowable observations are defined by the expert and are known as tests. A test is a description of the state of the system with a multiple choice part, e.g.:

```
TEST 36:

the state of scanner LED when the FAST/SLOW scanner switch is put to SLOW

   1 : flashing
   2 : permanently on
   3 : permanently off

DETAIL:

1.  select slow scan address test (TS13 down)
2.  rotate SELECT ADDRESS switch and observe LED1
```

The expert assigns an identifier to each test which will be used to refer to the test. In the example above the number 36 is the identifier. Sometimes it is useful to incorporate further notes of explanation or cautions to be taken. Such notes are stored optionally as detail and displayed only if the user chooses to do so.

2.2 Faults

Observations are normally expected to lead to one or more conclusions. The conclusions are known as faults. The expert defines all the faults that the system is expected to recognise. Each fault is defined as a statement with an identifier, e.g.

```
FAULT 4:

   reference amplifier module faulty resulting in incorrect speed of the main
   drive.
```

where the number 4 is the identifier.

2.3 Rules

Each rule specifies the conclusion to be drawn from a number of concurrent observations. The following is an example:

```
RULE ramf

    IF    the speed log printout indicates incorrect speed for the main drive
    AND   the main drive display does not indicate overspeeding
    AND   increase in mill load with its consequent reduction of speed does not
          cause a cobble

    AND   . . . . . .

    AND   . . . . . .

    THEN CONCLUDE

          reference amplifier module faulty resulting in incorrect speed of main
          drive.
```

Each rule is given an identifier (the letters ramf in the example).

2.4 Test costs and fault frequencies

These are numeric values that may be specified by the experts and are both factors considered by the system when it chooses a test to be performed. The cost of a test is a measure of the difficulty to perform that test.

The system computes the effectiveness of a test and weighs it against its cost and on the basis of this combined measure chooses a test to suggest.

Frequency is a measure of the likelihood that a fault is present and is used by the system to compute the effectiveness of various tests.

3 The problem solving process

This is the process that makes the stored knowledge available to the user, in a helpful way, to solve particular problems, e.g. to determine the fault in a diagnostic situation. The main characteristics of the process are described below with samples of dialogue for illustration, where the parts typed by the user are underlined:

3.1

The system is about to function passively, receiving whatever information the user wishes to supply, e.g.:

```
?:TEST 4

    speed indicator on the main drive:

        1 : normal
        2 : overspeed
        3 : underspeed
        4 : RESET
        5 : DELAY
        6 : DETAIL                    ?:1
```

In this example the user wishes to state the 'speed indicator of the main drive is in the normal state'. After typing 'TEST 4' the multiple choice question is displayed and the first choice is taken by the user.

3.2

The system is able to function actively, suggesting to the user the most suitable test to be performed, taking into account all that is known at the time about the particular problem situation, e.g.:

```
?:BEST
test 36:
        the state of scanner LED when the FAST/SLOW scanner switch is put to
        SLOW

            1 : flashing
            2 : permanently on
            3 : permanently off
            4 : RESET
            5 : DELAY
            6 : DETAIL                    ?:2
```

In the above example the user has invoked 'BEST' and the system has suggested 'TEST 36'. The effect is exactly the same as if the user himself invoked the test. The user in this example has responded with the second choice from the menu.

3.3

The user has the option not to perform the test suggested by the system. In the previous example the user could have responded with the fifth choice, i.e. 'DELAY'. The effect of that would have been for the system to continue as far as possible without asking this question again and the investigation might have reached a successful conclusion without this test.

3.4

The system is able to summarise the information gained in a particular investigation and the conclusions reached. For example the following dialogue lists all answered tests:

```
?:TESTS ANSWERED

        the following tests have been answered:
        4, 36
```

It is possible to find out what reply has been recorded for a given test by invoking the test command in the usual way:

```
?:TEST 36

        the state of scanner LED when the FAST/SLOW scanner switch is put to
        SLOW
```

```
1 : flashing
2 : permanently on
3 : permanently off
4 : RESET
5 : DELAY
6 : DETAIL                    : permanently on ?:
```

The last line of the example indicates the recorded reply of 'permanently on'.

It is also possible to display the tests which have been 'DELAYED', faults which are 'DELETED', i.e. rejected by reason of information recorded and faults which are possibly present by invoking:

'TESTS DELAYED', 'FAULTS DELETED' or 'FAULTS POSSIBLE'

respectively.

The user is able to determine whether a particular fault is rejected and why, or if that fault is a possibility what further evidence is needed to confirm it.

```
?:FAULT 7

    digital mill cubicle card 4214 faulty.

    The following further evidence is required to show that this fault is present:

        test 1:2        test 82:2        test 83:1        test 99:2
```

```
?:FAULT 21

    slow scanner card failure.

    This fault is rejected by the following evidence:

                                    test 35:2
```

3.5

The user is able to change or delete any information already recorded. The system will modify its conclusion accordingly. In the following example the user changes the response for 'TEST 4' from 'normal' to 'overspeed'.

```
?:TEST 4

    speed indicator on the main drive:

        1 : normal
        2 : overspeed
        3 : underspeed
        4 : RESET
        5 : DELAY
        6 : DETAIL                  :normal ?:2
```

The response could have been cleared to starting condition by choosing the 'RESET' option on the menu.

3.6

The dialogue with the system is very simple and free from statements of probability. The conclusions drawn by the system are equally clear. The following examples illustrate the types of conclusions drawn by the system:

```
I know of no faults to match the pattern of symptoms recorded. It may be that
this fault situation is one that I don't know of or that some symptoms are
recorded incorrectly.
```

```
I know of only one fault that matches the recorded pattern of symptoms. It is:

    FAULT 4:
        reference amplifier module faulty resulting in incorrect speed of the main
        drive.

Any further tests that I suggest are necessary to confirm this fault.
```

```
I know of only one fault that matches the recorded pattern of symptoms. It is:

    FAULT 4:
        reference amplifier module fault resulting in incorrect speed of the main
        drive.

I suggest, however, that you invoke BEST to confirm that this is indeed the fault.
```

3.7

All investigations are recorded. This enables a partially completed investigation to be restarted and continued at a later date. The investigation record is also used by the expert to monitor the effectiveness and correctness of the system.

3.8

The user may record comments relating to the investigation. The comment facility is used for communicating to the expert.

3.9

The reasoning used by the system is very simple. The system tries to prove the presence of every fault. Any fault whose presence cannot be proved is rejected. Finally, only the faults that are proved are listed.

3.10

Almost all the dialogue is user definable.

4 The knowledge refining process

This is the process that enables the expert to construct and improve the stored knowledge. The dialogue for communicating with this process is, in most cases, similar to the problem-solving dialogue and will therefore not be discussed. The main features are described below:

4.1

The knowledge consists of a collection of facts defined by the expert. The expert does not have to devise procedures or flowcharts.

4.2

The expert can present and test the facts in small increments.

4.3

The expert is assisted to observe the consequence of any modification to the knowledge. This is done, partly, by making available to the expert all the problem solving facilities.

4.4

Some of the facts compiled by the expert may be superfluous. The refining process exposes these as much as possible.

4.5

The compiled knowledge may be insufficient for complete resolution of problems. This situation is always exposed.

4.6

The system prevents the expert from compiling conflicting facts.

5 The problems encountered

To design an expert system, as indeed any system, involves a number of conflicting requirements. One requirement is that the system should be powerful and flexible, applicable to a wide range of problems. This can be achieved if a high-level abstract knowledge formulation scheme is adopted. The disadvantage of that is that it requires the expert to be considerably skilled at creative thinking and the system interface becomes more like a programming language, barring users who are not programmers. We decided to avoid this problem by adopting a more concrete scheme of knowledge formulation which is more directly suited to diagnostics.

Another, more specific, problem is the critical importance of the exact form of

words and sentences used by the system. Often a system message, designed to be helpful and clear can be misleading, or worse, offensive. To illustrate this point consider the following specific case, which is related to the command TEST and the message output by the system if the command is entered without the necessary parameter. It was thought that the user who did not know or remember the specification of the TEST command would invoke HELP TEST which would display the following information:

```
"TEST" is a command that must be followed by one of the following parameters:

    [test identity] : To obtain the description of the test and the recorded
                      outcome if any.

    [answered]      : To obtain a list of all TESTS for which an outcome is recorded.

    ----------------------------------------------------------

    ----------------------------------------------------------
```

Therefore any user who entered TEST without a parameter was likely to have done so inadvertently and the system could help by indicating that the word TEST was indeed expected and spelt correctly, but that something further was missing. We therefore designed the system to output the message:

TEST what?!

In the event some users found this offensive and replaced it by a more polite message.

Please enter test number after TEST

which in fact misleads the user to think that 'test number' is the only permissible parameter.

6 Conclusions

Our experience of ICLX highlighted a number of significant differences between expert systems and traditional computing. Expert systems are intended to be used by a very wide spectrum of people to solve a great variety of problems in very different environments. Given these variations it seems inappropriate to attempt to design a 'general purpose' expert system, even though the basic techniques themselves may be generally applicable. Therefore, when designing a system the following points should be borne in mind:

- the type of people to use the system i.e. scientists/technical/management etc.
- the type of reasoning needed for the problem, i.e. precise or approximate, forward or backward or both
- the quantity and depth of knowledge required
- the required response time and other performance-related factors
- the environmental conditions, i.e. dirty, noisy and hostile or clean, quiet and relaxed.

These factors represent challenging technical and man-machine problems that have to be solved if we are to employ and exploit expert systems.

References

1 ADDIS, T.R.: 'Towards an 'expert' diagnostic system', *ICL Tech. J.*, 1980, 2, (1), 79-105.
2 SHORTLIFFE, E.H.: *'Computer based medical consultation: MYCIN'*, Elsevier Computer Science Library.
3 DUDA, R.O. *et al.*: 'Model design in the PROSPECTOR consultant system for mineral exploration', in *Expert systems in the micro electronic age*, MICHIE, D. (Ed.), Edinburgh University Press.

Part II A case study in British Steel

1 Introduction

This collaborative exercise between BSC and ICL on the application of a knowledge-based system was initiated early in 1981, with the primary objective of demonstrating that fault data relating to a particular plant could be processed into a practical fault-finding strategy for that plant. BSC as operators of large-scale process plant have a continuing commitment to the development of fault-finding aids which will minimise plant down time when faults occur. In the 1970s great emphasis was placed on the functional design of equipment and on the presentation of documentation in a functional form. Although this approach resulted in significant improvements in the maintainability of equipment with regard to logical grouping and monitoring facilities, the resulting documentation was cumbersome and incapable of dealing with complex interactive control systems. It was in this area of plant engineering fault diagnosis that the 'expert system' was to be applied. As with many new concepts, a widely accepted definition is yet to emerge, but for a BSC application the following will suffice:

> 'An expert system is a computer-based method of making available, via a man-machine dialogue, skilled diagnostic knowledge relating to a particular plant, to less experienced staff,'

It was agreed to implement a trial on a chosen plant, with the following objectives:

(i) to establish an expert system for the Scunthorpe rod mill and establish its feasibility as an engineering tool
(ii) to maximise on ICL's previous developments in the form of 'RAFFLES' and 'CRIB', which were essentially a historical retrieval approach
(iii) to assess the future use of expert systems in BSC and to produce some general rules and guidelines on the application of such systems.

2 Background: the Scunthorpe rod mill

To give an understanding of the extent of the fault diagnosis problem it is necessary

to give a brief introduction to the production process and the type of equipment which is encountered in the mill.

The installation and commissioning of the Scunthorpe rod mill was completed in 1976. It is a four-strand mill and has a production capacity of 600 000 tonnes per year. The feedstock is 20 m long, 115 mm^2 billets, and rods are produced in the range of 5·5 to 13 mm diameter. The finishing speed of the rod is 60 m/s, and when installed the mill was among the fastest in the world.

The rolling stands are driven by 25 main drive motors ranging from 300 kW to 2500 kW, with high-speed closed-loop speed-regulator control; a digital set up and sequence-control system gives fine resolution coupled with accurate repeatability of the speed setting. The digital equipment also controls the eight shears and associated pinch rolls, horizontal loopers and water cooling, and provides cobble detection*. The pulpit* controls are accordingly complex.

The mill is equipped with five computers in a hierarchical arrangement providing production co-ordination, furnace control, mill-speed set-up scheduling and an alarms monitoring facility. With the very high speed of operation even at the intermediate stage the rod is travelling at speeds of up to 8 m/s and a single failure in the control equipment is likely to result in a mill cobble. If this occurs in the roughing mill or intermediate areas it is likely to affect the operation of adjacent strands. Cobbles are dangerous and expensive: a four-strand cobble can take up to 2 h to clear and during that time fault location is very limited, as the equipment must be isolated to allow removal of the cobble. The control systems are all highly interactive to maintain correct inter-strand speed ratios.

The alarms computer system is designed to provide a quick means of identifying faults as they occur on the mill. A total of 1500 alarm contacts are scanned at 1 s intervals and groups of alarms, where finer discrimination is required between first and subsequent faults, are dealt with by interrupts which give group scanning within 6 ms. It was in the alarms facility that we made our first attempts at a knowledge-based system way back in 1976. An engineers' message facility was provided whereby a message was given, suggesting the likely fault when particular alarms or combinations of alarms occurred. These messages were added to as experience was built up and on a historical basis a 'top ten' approach was taken: that is, a number of possible faults were identified, listed in rank order. In addition to the above, the mill-speed computer provides also a cobble log whereby all strand speeds, currents and control settings are logged at 0·1 s intervals for the 5 s preceding a cobble. The mill is thus very well covered with alarm and monitoring facilities; but a high level of expertise and familiarity is needed to make the correct deductions from the information provided.

3 The evaluation programme

There were several reasons why the rod mill was chosen for the trial:

* A 'cobble' describes the result of a sudden stoppage of some part of the machinery, when the material still in process is thrown into loops and tangles – cobbled up, in fact. The 'pulpits' are the control stations from which the operators look down on the process lines.

(i) This plant contains complex equipment which presents fault location problems.
(ii) It was decided to concentrate on those faults which would be likely to result in cobbles, as this would provide a clear cost/benefit assessment.
(iii) Considerable amounts of information relating to faults are available for use in answering questions.
(iv) An expert was available, with intimate knowledge of the faults which have arisen since the mill was commissioned.

The start of the exercise was to collect actual fault data in the detail required, in the form of a fault/symptom matrix. This was a slow and painstaking task, and after several months the database was still hardly adequate. The decision was then taken to expand the database to accommodate faults which had not yet occurred, but which might.

The second stage was to develop and prove the knowledge-refining program. With the expanded database of about 40 faults the program, after the expected debugging, succeeded in producing a decision tree that was logical but in some way impractical.

The third stage involved collecting data and experience from the 'expert diagnostician', as opposed to taking simply the plain facts recorded in the fault/symptom matrix. This was a joint exercise between ICL and BSC, but was heavily dependent on BSC's expertise and experience with the plant. New fault criteria were conceived, involving 'testing penalties' and 'frequencies of fault occurrence'. It might seem that an obvious approach would be to carry out the simpler tests which eliminate the majority of faults before tackling a more difficult test which may be time-consuming and require special test equipment, but this is not always the case.

The new database, derived from a fault matrix of 61 faults and 98 symptoms or tests, was available early in 1982.

At this stage it was agreed that the knowledge refining had reached a stage whereby it could be implemented in a user or advice-generator form on the rod mill. The program had been written in 1900 CORAL to match the development facility then available to the ICL engineers. The advice-generator facility had obviously to be captive to the confines of a particular plant, and from the point of view of plant geography it required several intelligent outstations. The equipment was required to be suitable for pulpit, motor room and substation environments. It was proposed by ICL that the advice generator should be written in Pascal for an ICL Personal Computer, equipped with Winchester-disc drive to provide a floppy-disc link between the two systems. The system was also to have three VDUs and keyboards. This system was installed on the rod mill in August 1982 and quickly grew into one containing 73 faults and 113 tests.

4 Some features of the system

The terminal is located alongside the mill's digital equipment and is the one

normally used by the technician when fault finding. The 'expert system' is not, in itself, a solution to the problem of fault finding, and full reliance on traditional tools and measuring instruments continues – oscilloscopes, UV recorders, AVOs and Microsecoms are used, in addition to the test and monitoring facilities in the equipment.

The program allows users to interact with the knowledge base in a natural manner, the dialogue between the system and the user allowing a flexibility of use similar to that offered by a human expert during a consultation. This is illustrated by the examples given in Part I of this paper. The system allows a course of reasoning, dependent on the user's replies to previous questions; and therefore only questions which relate to the problem under discussion are asked.

The user can input information into the system by describing symptoms before asking for the best action to take, or can simply seek advice from the system. The former leads to a solution in a shorter time but requires more knowledge from the user.

Commands The system uses the minimum number of commands, in the interests of simplicity of operation.

A menu is displayed at the start of a fault investigation. The HELP command provides comprehensive messages for the new user who requires details of the commands available to him.

The BEST command provides the question which will eliminate the maximum number of possible faults; a menu of possible answers to this question is displayed, requiring a numerical input as answer. Where more detailed instructions are required to answer this (or any) question asked by the system the user selects the part of the menu called DETAIL.

Other features Frequently during a fault investigation it is impossible or undesirable for the diagnostician to answer a question at that particular stage. Instead he can *delay* answering, and this allows the system to calculate the best question to ask next. He can also list details of the faults that are *possible* and can request the status of the evidence collected so far. In practice, some investigations cannot be concluded until some later date – for example, when there is an intermittent fault – so the system allows the user to *reconnect* later to continue an investigation.

All investigations can be logged into the system memory for post mortem analysis by the experts who are responsible for the management of the system. In addition, users can input *comments* to give a feedback or to provide information on problems encountered or on faults that the system does not currently include. Additional detail can be requested, and a valuable aid is the ability to define test procedures and test points, avoiding extracting key information from extensive documentation.

A display of the number of faults still not eliminated can be requested. This gives a positive indication of how well the investigation is proceeding. Additional tests can be requested to verify the diagnosis.

5 Experience with the system

The system has now been in use at the rod mill for approximately 12 months; it is now established and is considered a worthwhile engineering tool. It is considered cost effective: with strand downtime costing £9 per minute and cobbles costing approximately £1000 a time this can be easily demonstrated.

A frequent occurrence before the installation of the system was that the fault that caused a cobble could not be located and the technicians would advise 'try another billet' – which could be a very expensive way of proving that a fault still existed.

The exercise with ICL has been mutually beneficial and we at BSC certainly feel we have played a part in the development of ICLX. From our experience we can make the following general observations:

(i) When configuring an expert system database it is not sufficient to know how a plant works: the thought process must be, how does a plant fail?
(ii) It is vital to avoid ambiguity: questions must be carefully phrased to ensure that the desired response is achieved.
(iii) The tests must begin at the lowest level of skill likely to use the system: often the expert makes subconscious assumptions regarding the 'obvious' which are not so obvious to others.
(iv) It is necessary for the users to relate the system to the accepted expert for them to have confidence in the system.
(v) Once the system is off the ground it can be incrementally developed on the basis of feedback, at a fast rate; it then becomes even more acceptable to the users because they recognise their own contributions.
(vi) The logical approach to fault diagnosis can highlight deficiencies in the monitoring and alarms facilities: it has been found necessary to install additional hard-wired features to aid fault-finding.
(vii) The system can be very powerful as an aid in training; it has been used extensively in this way at Scunthorpe.

6 Future possibilities

We are currently considering the possibility of using an ICL PERQ computer to enable the knowledge refining for other systems to be carried out at Scunthorpe.

The rod mill is a modern compact arrangement, in contrast to many other areas of potential use of the expert system approach; consequently we feel that the advice-generator facility should be developed with the idea of a hand-held terminal in mind, to give the user the mobility necessary for operating on distributed plants.

In parallel with the exercise reported here BSC have developed their own expert system which is basically a simple virtual tree approach; this can be applied to less complex problems. However, we are convinced that the ICLX approach is what is needed for fault diagnosis in areas of complex interactive control, and it is in such areas that that expertise is most scarce.

Dragon: the development of an expert sizing system

M.J.R. Keen
Knowledge Engineering Group, Mainframe Systems Development Division, Bracknell, Berks.

Abstract

Considerable interest is now being shown in the development of knowledge-based systems – so called 'expert systems' – for use in a wide variety of different situations. Being dissatisfied with the capabilities of computer performance modelling and prediction ('sizing') systems that have been produced using conventional techniques, the author embarked on a project to build a pilot sizing system (called 'Dragon') using the rule-based approach of expert systems. This paper explains the background to this work and describes the tools and techniques that have been used. Details are given of the state of development of the Dragon system some eight months after serious work began. The paper is illustrated with examples taken from the Dragon knowledge base and with a typical consultation with the system.

1 Sizing in a computer sales environment

Like all computer manufacturers, ICL is in business to sell computer hardware and software. A key part of this selling process is the 'sizing' of the customer's requirements: agreeing with the customer his perception of his future needs, and matching this with sufficient computer resources. This involves:

(*a*) establishing the nature of the customer's future needs
(*b*) estimating the computer resource requirements for each part of this future need
(*c*) Combining these estimates to produce total/peak resource demands
(*d*) Choosing a suitable computer configuration, based on:
  – providing sufficient processing power
  – maximum practical device utilisations
  – the avoidance of system bottlenecks
  – storage and memory requirements
  – contingencies, etc.
(*e*) Ensuring response time targets/turnaround deadlines are met.

Frequently, several iterations are required, to balance cost, functionality and performance, before a satisfactory result is achieved.

Over the years, ICL has accumulated a great deal of experience at sizing up customers' workload requirements – particularly in the mainframe area. However,

the real expertise is held by just a few individuals and, as always, there are too few experts to meet the demand. This expertise covers four main aspects of the sizing process:-

(i) textbook knowledge of the theory and techniques of performance modelling and prediction – e.g. queuing theory
(ii) an understanding of the functionality and characteristics of the computer products – both hardware and software – that are likely to be proposed
(iii) heuristic knowledge about likely system bottlenecks, and areas of performance interaction
(iv) knowing what information is required to describe any particular type of workload and how to fill in the inevitable gaps in the information that is available.

It is in the fourth of these areas that much of the real expertise of sizing lies in that:

- customers often find it difficult to describe their future needs in detail
- this description is frequently 'patchy', with some aspects of the workload being described precisely while other, equally important, parts of the workload are virtually undefined
- sales timescales often preclude detailed workload analysis and sizing.

Further practical difficulties arise from the great many numbers that need to be manipulated during the course of a sizing – particularly where workload resource requirements have to be evaluated in detail. This produces both straightforward arithmetic errors and, more seriously as they are more difficult to trap, errors owing to omissions from the calculations.

As in any other area where expertise is in limited supply, short cuts have to be taken – increasing the risk to both ICL and its customers. Quite obviously, what is required is a computer system that can both automate the numeric side of the sizing process and take on the role of the sizing expert – for all straightforward sizing situations. This would then free experts to concentrate on the really complex and novel problems, developing the knowledge for an even more expert system in the future.

Such a system would start by establishing the customer's future workload by asking a series of questions, adapting the questioning to the type of workload and the level of information available. Like the human expert, the system would need to change its line of questioning if one course was found to be unproductive. Where the user is uncertain of the answer to a question, the system would need to be able to give advice and make helpful suggestions based on what is already known – again like the human expert. The system would need to insist on certain minimum levels of information being provided and should then fill in the gaps where the user has been unable to answer less significant questions by taking knowledgeable defaults. Once the customer's workload has been defined, its computer resource requirements would be 'costed' and this mapped onto a suitable hardware configuration. This would be modelled to ensure performance constraints – such as response time

targets – will be met. The system would need to produce a summary report for inclusion in sales proposals.

Such a system, as described above, would need to be highly flexible with a well engineered user interface that enabled it to be responsive to different levels of user skills. The ability of the system to explain its advice and lines of questioning would be crucial to its acceptance by those who have to use its advice as the basis for significant commercial decisions.

2 Conventional performance modelling

The type of system just described goes far beyond the scope of conventional performance-modelling systems. Most of the development work on such systems has been directed towards the later stages of the sizing process – the analysis of bottlenecks and queues – and these can only be used once the resource requirements of the workload have been defined. There are good reasons why this has been the case:

(i) The analysis of queuing situations can largely be carried out independently of any particular computer architecture – hardware or software. Although such models may have to be 'adjusted' to reflect the details of particular manufacturers' designs, they are widely applicable. Thus a queuing model of a moving-head disc system can be applied to a whole family of such devices, simply inputting the relevant hardware speeds as required.

(ii) The production of a system to perform resource costing requires both extensive knowledge of the architecture of the particular software being modelled and full details of path lengths, store use, etc. This is not normally available outside the organisation producing the software.

(iii) The use of conventional third generation programming languages (Fortran, Basic, COBOL, Pascal, etc.) requires considerable design effort to create the data structures to hold the knowledge base in an easily maintained form, and to manage the user dialogues. Failure to do this, for instance by embedding product knowledge within program code, produces a system that is both difficult and expensive to maintain.

Thus, while one can generalise and say that there is a substantial body of experience in modelling computer dynamics – analysing the queues and bottlenecks – such techniques are of little use in predicting future needs unless one can assemble all the necessary resource demands. Such information must be complete, correct, sensible etc., if the results of the queuing analysis are to have any value. Moreover, the analysis tools themselves generally require human experts to use them.

So, the building of a sizing system as described earlier would break much new ground and might well demand the use of 'novel' techniques. A possibility was the use of the expert system rule-based approach.

3 Knowledge-based systems

More popularly referred to as 'expert systems', knowledge-based systems embody organised knowledge about particular areas of human expertise which enable them to perform as skillful and cost-effective consultants. Having been developed out of research into artificial intelligence, such systems are an attempt to emulate the workings of a human expert. Some notable examples of practical expert systems that have been developed during the past few years are:-

MYCIN	which is concerned with the diagnosis and treatment of bacterial infections
PROSPECTOR	which evaluates geological survey data for potential mineral deposits
DENDRAL	which analyses mass spectrometry data.

The distinguishing feature of any true expert system is the separation of the knowledge side of the system from the mechanisms that make use of the knowledge. These two parts of an expert system are referred to as:

- the knowledge base, and
- the inference engine.

The power of any expert system is in its knowledge base. It is produced by expressing the human expert's knowledge as a set of 'rules'. This is done using a declarative programming style in which rules take the form:

IF 'condition' THEN deduce 'result'.

The complete set of expert's rules are processed to form the data structures that make up the knowledge base. Quite obviously a knowledge base is particular to a single area of knowledge (or domain) and for each new domain a new knowledge base has to be developed.

The inference engine provides the deductive mechanisms that make use of the information stored in the knowledge base. It also provides the user interface, including pseudonatural language processing for a smooth human interface and an ability to explain its own 'thinking' by displaying appropriate text stored within the knowledge base – for instance, to explain why a particular question is being asked. In theory, at least, the inference engine could be used for many different expert systems.

To build an expert system, one must make the choice between using a specialised programming language such as LISP or PROLOG, or basing the system on a package or expert system 'shell'. Of the languages, LISP is long established and much favoured in the USA where most expert systems have been developed using the language. PROLOG, on the other hand, is a much more recent development with a growing following around Europe (and in Japan). [At this point it should be noted that an expert system can be produced using conventional programming languages, though this would involve much more coding.] An expert system 'shell' is a

package for building and running expert systems and, as such, contains the software both for processing the rules to form the knowledge base and for the inference engine itself. Two shells which are currently available are:

EMYCIN	from the USA and written in LISP, and
SAGE	produced in the UK and written in Pascal.

Using a 'shell', one can generally expect to make rapid progress, though the design of the shell system may constrain the type of expert system that it can be used to build. Use of a language offers the promise of greater flexibility – at the cost of more programming effort.

Wishing to make rapid progress, it was first decided to investigate the use of an expert system shell for building the sizing system. As LISP was not available on 2900, EMYCIN was largely ruled out, so SAGE was considered in detail.

4 SAGE

SAGE is a package for developing and running consultative expert systems. It was developed by SPL at Abingdon (near Oxford) and was first released in the middle of 1982. It is running both on ICL's 2900 series mainframes and ICL PERQ. The SAGE package is in two parts:

(i) The SAGE 'compiler' which produces the knowledge base from the expert rules written in the SAGE programming language.
(ii) The SAGE 'executive' or inference engine.

The very high-level SAGE programming language which is used to encode the expert's knowledge is more like a stylised form of English than a conventional programming language. This allows rules to be written using the terminology of the expert while questions can be phrased in language that the end user will understand. The SAGE language is declarative (in that statements can be expressed in any order) rather than procedural and is based on the use of a small number of keywords. This permits stylistic flexibility on the part of the rule writer.

The main statement types used in a SAGE program are

ACTIONs	which control the consultation and govern the behaviour and overall course of action of the consultation
ASSERTIONs	entities with associated probability; they can also be used as simple YES/NO switches
OBJECTs	entities with associated numeric values, bounded by defined ranges and with inbuilt default clauses
RULEs	which enable the value of an ASSERTION or OBJECT to be established in a particular situation
QUESTIONs	which ask the user to input the value of an ASSERTION or OBJECT.

Text can be associated with all elements of the knowledge base, providing both questioning in end users' terms and the ability of the SAGE system to explain its deductive reasoning. Where the SAGE language is inappropriate – for instance for algorithmic procedures such as certain queuing theory calculations – these can be written in an appropriate language (e.g. Pascal) and integrated into the SAGE system.

The SAGE executive provides the user interface to the expert system. It manages the consultation on a goal directed basis and provides a range of command and diagnostic facilities, including:

HELP	
TRACE	to display the logical inference process of the expert system, and
LOG	to produce a hard copy of the consultation.

Part completed consultations can be SAVEd for subsequent RESTORing for iteration purposes.

During the course of a consultation, the end user may ask for:

- a question to be explained
- the reason why a question has been asked
- details of the current state of the consultation

using the information and text contained in the Knowledge Base.

Experience has shown that, despite the advanced thinking that is built into the SAGE package, it is easy to use by nonprogrammers. Moreover, the end product – the expert system – will be robust and can easily be made highly usable.

5 Dragon

In the late summer of 1982, consideration was being given to finding a solution to the problem described earlier – producing a sizing system for use by ICL sales staff. This coincided with the wish to find suitable domains of expertise to evaluate the newly announced SAGE package. Such domains would have to be able to justify in their own right the development of expert systems.

It was decided to combine those two activities and in October 1982 work started on an evaluation prototype – subsequently named Dragon. To provide a fairly complete picture of the problems to be handled in an expert sizing system, it was decided that Dragon would need to take a vertical slice through all the main steps in the sizing process, while at the same time restricting the range of products to be covered, to speed development.

Coding of the rules began in November, based on an existing set of detailed sizing knowledge, and before the middle of December the first part of the knowledge base was up, running and usable. This enabled the resource requirement of a TPMS/IDMS

system to be calculated – database size, processor and I/O costs per message, and main store quotas (TPMS is the VME 2900 TP monitor and IDMS is the CODASYL database management system.)

By the end of March 1983 a fairly complete version of Dragon was running and demonstrable. This performed the following functions:

- detailed evaluation of the resource requirements of a TPMS/IDMS system
- gross evaluation of MAC (multi-access) and batch resource requirements
- combining TP, MAC and batch workloads
- selection of a suitable 2900 series configuration to run the combined workloads
- calculation of response times (using a priority network queuing model written in Pascal)
- report production.

The network queuing model is of the class of such models often associated with the name of Dr. J. Buzen in the US and developed independently by ICL in the UK. Although it would be inappropriate to describe the model in detail in this paper, it is worth noting that the model takes advantage of the fact that the VME operating system assigns the same priorities to both the processor and peripheral systems, and that this effectively operates on a pre-emptive basis. Thus throughput of high-priority work is unaffected by work at lower priority. This enables the throughput of high-priority work (e.g. TP) to be calculated by itself and then lower-priority throughput to be calculated by subtracting TP throughput from that of the combined workload as a whole. The model calculates the service rate of the central system at varying levels of concurrency, with transaction queue lengths being used to calculate mean response times. 95th percentile response times, which depend on the higher variants of the service time distribution, are calculated by appropriate approximation of the central system characteristics.

Since March, the Dragon system has undergone extensive testing and refinement, particularly with a view to adding to the responsiveness of the system. The system now uses some 600 SAGE rules (or rule equivalents) and has taken between 16 and 20 man weeks of effort to produce so far – excluding the writing of the network queuing model – this short time being a direct result of the extensive human sizing knowledge base already existing inside ICL.

Where a coherent knowledge base does not already exist, a very great deal of effort can be involved in the knowledge elicitation process required to gather together the rules for an expert system. This is likely to be greater than the actual time spent coding up the rules. Even where a human knowledge base exists, the operation of encoding this knowledge into expert system rules is likely to uncover omissions and inconsistencies in the knowledge base. Certainly this occurred during the building of the Dragon system.

Validation of Dragon has been carried out at two levels:

(i) The performance information contained in the SAGE rules has been vali-

dated by extensive comparison of 'hand' calculations against measurements.

(ii) The Dragon logic has been compared with validated hand sizings, principally using the SAGE TRACE facility.

In the second of these, and the only specific to the development of SAGE-based expert systems, very few errors have been encountered in encoding the rules and these have rapidly been revealed using the TRACE facility.

A short annotated extract from the log of a typical Dragon consultation is given in Appendix 1. This shows precisely what is displayed on the screen and illustrates both the general style of the conversation between the user and the system, and the range of commands and responses. The form of the SAGE rules used in Dragon is illustrated by the extract given in Appendix 2; this is from the part that calculated the size of the database.

Dragon can be accommodated on an ICL PERQ; the example from which the Appendices are taken was run on a 2900 system under VME, which was supporting about 80 interactive users.

6 Conclusions

At the start of the work described in this paper, it was not certain whether it was possible/practical to build the type of all-embracing sizing system required by ICL sales staff. As a result of the development of Dragon, we now know that not only is such a system possible, but also that the use of expert system techniques enables a totally practical, robust, usable system to be created. Such a system can reflect many of the attributes of the human expert:

- a flexible approach to questioning
- the ability to explain questions
- the offering of alternatives
- the filling in of gaps in the client's information by making knowledgeable assumptions, etc.

For the beginner to knowledge engineering, the use of the SAGE package enables knowledge to be encoded rapidly without the need to worry about data structures, user interfaces, etc. The very high level declarative language used to specify knowledge to SAGE enables rapid development with few coding errors. SAGE's ready acceptance by nonprogrammers (such as the author) makes it possible for the expert system to be created directly by the person with the knowledge rather than working through an intermediary.

Acknowledgment

This paper was presented at the 11th European Conference on Computer Measurement, Copenhagen, October 4th-7th, 1983 and is published with the permission of the organisers.

Appendix 1 Typical consultation

The following are extracts from a consultation in which the system is used to specify the resources needed to handle a stated workload. The computer messages are given first, followed by the user's response.

Type BEGIN to begin

Please type a command : BEGIN

This is the fifth version of the ICL sizing model.

Profiles of three workload types – TP, MAC, batch – can be entered into the model, which will evaluate and report on the resource cost of running each one.

The three sets of resource costs are then brought together and various sizing techniques are used to select a configuration suitable for running the combined workload, with interactive response times being calculated. A final summary is produced.

* * Are you familiar with the use of this model?
(You may answer yes or no) : N (= NO)

The model has been designed so that if the answer to a detailed question is unknown you do not need to guess but may simply respond "U" to indicate that the answer is unknown. The model will then attempt to obtain the information it requires by asking other questions, or by taking default values.

However, a limited number of questions must be answered if the evaluation is to proceed; such questions are clearly identified and you will not be allowed to proceed without giving a definite answer.

Should you need further guidance in order to answer a question, or WANT TO KNOW THE DEFAULT ACTION ON A QUESTION, this can be obtained by replying "?"

* * The model can carry out sizings at various levels of detail. For this sizing, which of the following do you want the model to carry out?

1 a simple sizing
2 a detailed sizing
3 an expert sizing

Make a choice : 1 (other options simply take the model into more detail)

** Do you wish the model to evaluate a TP/IDMS workload?
(You may answer yes or no) : Y (= YES)

* First consider the size of the database.

What is the total number of records (in thousands) to be stored in the database?
(range is 0→100000) : 202

What is the average amount of data held in each individual database record, in bytes?
(Range 0→1000) : 35

What is the number of different types of record to be held in the database?
(Range is 1→1000) : 8

What is the number of different types of sets to be held in the database?
(Range is 1→1000) : 8

The resilience and security of the TP system can be improved by "duplexing" the online database.

Do you require all or part of your database to be duplexed?
(You may answer yes or no) : ?

Duplexing involves maintaining two up-to-date copies of the database online. If during normal operation there is a failure that makes one copy unavailable, the system is able to continue using the single remaining copy without interrupting the service. However, this does require extra online disc storage to hold two copies and extra database writes to keep both copies up-to-date.

The default is not to use the duplex facility.
Please reply : N (default taken)

* * * * * * * * *

With a packing density of 0.70, the total disc storage requirement for the IDMS database is 18.71 Mb. The recommended database page size is 2332.00 bytes. (first result – size of database)

[The question-and-answer session continues, dealing with OCP, I/O, mainstore and disc store requirements and throughput. The last is as follows.]

* **Now consider the throughput required.**

What is the peak message rate to be supported by the TP/IDMS system, in messages per second?
(Range 0→100) : 10

What is the maximum number of terminals to be connected to the system at any one time?
(Range 1→ 2000) : 600

Do you specifically want a dual OCP system to be proposed?
(You may answer yes or no) : N

* * * * * * * * *

The specified work load will require a total OCP capacity of 0.71 MIPS. Thus it will require a 2988 OCP based configuration. For this workload the chosen OCP system will deliver a total of 1.56 MIPS, giving an OCP utilisation of 45.46%

[Extracts from the summary produced by the system are:]

1 WORKLOAD

The specified workload consists of:-

TP/IDMS
A TP/IDMS workload with the following profile:-

peak message rate	: 10.00 per sec
max. number of active terminals	: 600.00
number of message types	: 13.00
av. number of transaction phases	: 1.45
Percentage of update messages	: 30.00%
average message profile:-	
input message length	: 39.00 chs.
output message length	: 293.00 chs.
number of input format effectors	: 3.90
number of output format effectors	: 29.30
message analyser path	: 1050.00 pli
application path	: 379.47 pli
number of DML calls	: 7.80
number of physical DB accesses	: 4.16

2 CONFIGURATION

To run the combined workload within the given constraints –
i.e. a maximum OCP utilisation of 70.00% – requires the
following configuration:-

2988 OCP system
3.34 Mb. main store
1.00 string(s) of discs with:-
3.00 EDS 80 drives
7.00 FDS 160 drives

With the above configuration, the average mainframe response time will be:
for TP 0.52 seconds.

Appendix 2 Some of the rules used by the system

The following short extract shows the style of the rules built into the system; it is taken from the part that calculates the size and main features of the database. The conventions are as follows:

1 SAGE keywords are in capitals
2 SAGE entities AREA, ACTION, OBJECT, RULE, QUESTION, ASSERTION are all followed by the name of the entity. This name identifies the entity to the model
3 Text strings are in quotes

```
AREA     idms_db_main:                                  (the area of the model
  "Calculates size of IDMS database"                    concerned with
                                                        calculating DB size)
ACTION  consider_db_size:
  "Consider the total database size"
  CONSIDER idms_db_size                                 (sets the goal of this
                                                        area as entity
                                                         idms_db_size)
OBJECT  idms_db_size:                                   (descriptive text)
  "the total database storage requirements in Mb"       (valid range)
  (0, 500 000)

RULE     db_size_calc_1:
  "This calculates the size of the database"            (alternative rules:
  idms_db_size IS (total_nbr_db_pages * db_page_size     1 – used when no.
  *(1 + percent_db_duplex/100))/1 000 000               of pages, and record
  PROVIDED nbr_db_records ≠ 0 AND                       length, are given or
    av_record_data_length ≠ 0                           can be found;
```

```
RULE    db_size_calc_2:                                  2 – when only gross
  "This calculates the size of the database"             vol. is known
  idms_size_database IS gross_data_vol *                 "nbr_db_records = 0",
  (1 + percent_db_duplex/100)/db_packing_density         etc. is catch-all clause
  PROVIDED nbr_db_records = 0  OR
    av_rec_data_length = 0

OBJECT  total_nbr_db_pages:                    (next level entity, required by
  "the total number of pages held in the IDMS database"   Rule 1 to calculate db size)
  (2, 250 000 000)

RULE    calc_tot_db_pages
  "This calculates the total number of database pages"
  total_nbr_db_pages IS nbr_space_man_pages + nbr_data_pages

OBJECT  nbr_space_man_pages
  "the number of database pages required for space management"
  (1, 250 000)

RULE    calc_space_man_pages:
  "This calculates the number of space management pages requires"
  nbr_space_man_pages IS
  round ((2 * nbr_data_pages / (db_page_size - 40)) + 0.5)

OBJECT  nbr_data_pages
  "the total number of data pages in the database"
  (1, 250 000 000)

RULE    calc_nbr_data_pages
  "This calculates the number of data pages required"
  nbr_data_pages IS
  round (( nbr_db_records * 1000 * av_db_record_length
        / (db_packing_density * (db_page_size - 40 ))) + 0.5)
```

These are followed by rules for setting the packing density, which give this as 0.60, 0.65 or 0.70 according as the average record length is greater than 800 characters, between 600 and 800 or less than 600, respectively; and then by rules that use the page size already calculated to give the optimum page size for MDSS discs. And so on.

The logic language PROLOG-M in database technology and intelligent knowledge-based systems

E. Babb
ICL Systems Strategy Centre, Stevenage, Hertfordshire

Abstract

The paper is concerned with the problem of accessing and updating a large shared body of data stored in a computer, in a way that will avoid the pitfalls to which many existing systems are liable: that is, of allowing insertion of data that can never be retrieved, deletions that cause previously inserted data to become unavailable, and insertion and deletion rules that contradict each other. The system proposed uses the implication network datamodel and is described here in terms of a logic language developed by the author for the purpose, PROLOG-M, meaning 'PROLOG plus model'. As well as forming a single unified system for controlling access to the stored data and maintaining its selfconsistency, the system also provides a compact version of standard logic, called model variable logic, which is described. Examples of the use of the model are given, including a 'real-life' system running on an ICL content addressable file store (CAFS) installation.

1 Introduction: The implication network model

The most widely accepted practice in modelling data is to describe the real world to which the data applies by a collection of relationships between variables. The variables can be, for example pressure and time in the case of a physical system or part-number and supplier-name in the case of a commercial system. Tests are sometimes included for checking the validity of data, but a weakness of many systems is that there is nothing to prevent any of the relationships being deleted inadvertently, and hence large chunks of the model lost, unless some specific test has been included to guard against this.

During the 1970s the author was involved in the design of the ICL content addressable file store, CAFS. One of the principal ideas in the CAFS database query language FIDL[1,2] was the implication network datamodel. This prescribed exactly how the data was viewed, in the sense that it reflected the way it was connected in the real world, and was also able to check the consistency of insertions and deletions; when a proposed update would lead to an inconsistency the user was required to either remove his updating information or to change the structure of his model.

The model has been developed beyond the form used in FIDL. A main purpose is to allow a large number of computer users to access a shared body of data in safety. Because of the model, the system behaves like a human in that each proposed alteration is checked against all previous entries; thus for a system concerned with engineering parts and their suppliers we can imagine the following dialogue with the computer:

(user) 1 suppliers of particular parts can only exist, if parts and suppliers already exist separately.
(computer) OK
2 GKN is a supplier
OK
3 supplier GKN supplies a MOTOR
sorry, contradicts statement 1, there is no separate part called MOTOR
4 MOTOR is a part
OK
5 supplier GKN supplies a MOTOR
OK
6 remove supplier GKN
sorry, contradicts statement 1 or 5

In practice these statements would be written in the formal logic language PROLOG-M, but this example gives an impression of what would be possible if a computerised natural language translator was used.

An important feature of PROLOG-M is the facility called model variable logic. Model variable logic is a generalisation of the current CAFS query language FIDL. In this language no explicit reference to relation names is required and so to obtain the suppliers of parts weighing more than 2 kg, the user can type:

?supplier : part-weight>2.00

The user need only have a general understanding of the meaning behind the special model variables 'supplier' and 'part-weight' for him to pose this query. This shorthand in FIDL and in PROLOG-M has been made possible because they both use the implication network model. PROLOG-M extends the shorthand in FIDL to allow queries with quantifiers 'all' and 'some' to be compactly expressed. For example, the part numbers which have not been used on any order can be found using the query:

?part-number : not some order-number

Paraphrased into English this becomes 'list the part-numbers not used on some (or any) order-number'.

PROLOG-M also extends the implication network model. In addition to the purely physical predicates (relations) allowed in FIDL, predicates can also be defined in terms of general rules.

The implication network model (Section 2) provides the model builder with a single unified method of restricting access to the model and of maintaining its internal consistency. The present paper describes it in terms of the logic language PROLOG-M, a development of PROLOG meaning 'PROLOG plus model'. The explanation is given mainly in terms of a model of a part supplier and orderer system for which a sample dialogue is provided (Section 3). Models are also illustrated by applications to modelling family relationships and a library system (Section 4). Finally, the valuable concept of model variable logic (Section 5) is defined and explained.

1.1 Glossary of logic programming

1.1.1 PROLOG-M. This is an enhanced totally logical form of standard PROLOG[3,4] implemented using a technique called the finite computation principle[5]. Finite computation principle is a transformation technique that takes a potentially infinite program specified in logic, and transforms it into an executable program. PROLOG-M currently uses a LISP-like notation for predicates. However, the more familiar conventional mathematical notation is used in this paper.

1.1.2 Variables and constants. Variables are written in lower case and constants in upper case.

1.1.3 Free variables. Such a variable currently has no special value attached to it.

1.1.4 Bound variables. Such a variable contains a definite value which can be printed out. A variable can be bound to integers, lists, sets, relations, and even function or predicate definitions.

1.1.5 Predicates. These express relations among variables and have the form p(x1, x2, ... xn) where p is a predicate name and x1, x2 ... are variables. In a logic language we define the meaning of a predicate by means of assertions: (p← means 'p defined as true')

```
(user)      ?owns(JOHN,2,BICYCLE,80)←
(computer)  OK
            ?owns(JOHN,3,BOAT,1000)←
            OK
```

which defines JOHN as the owner of two bicycles each worth £80 and three boats each worth £1000. This predicate can then be questioned:

```
(user)      ?owns(JOHN,qty,item,value)
(computer)  qty = 2, item = BICYCLE, value = 80
            qty = 3, item = BOAT,    value = 1000
```

Notice that unlike a normal function or procedure, such a call causes any variables that are 'free' to print out all the values that the earlier assertions have defined.

1.1.6 System predicates. These are predicates that have been built into the machine. For example the predicate 'plus (x, y, z)' would normally correspond to the equation x = y + z in mathematics. Such a predicate can be used just like the 'owns' predicate above as the following examples show:

```
(user)      ?plus(x,3,4)
(computer)    x = 7
            ?plus(7,3,w)
              w = 4
            ?plus(x,y,11)
            sorry, infinite or uncomputable
```

Notice how even arithmetic predicates are reversible. This is very useful in data-models because it allows us to include arithmetic processes as if they were stored relations. Notice also that if insufficient information is provided the system is able to let the user know; PROLOG-M does not attempt infinite computations. Generally, in this paper built-in predicates are referenced using conventional mathematical notation.

1.1.7 Definition of new predicates. Logic languages allow new predicates to be defined in terms of existing predicates. For example: (*a*←*b* below means *a* is defined as *b*)

```
(user)      ?owner-total(owner,item,tot)←
                        owner(owner,item,qty,value)&tot=qty*value
```

gives a new predicate where the total value 'tot' of all the items is given by multiplying the number of items 'qty' by the value of an individual item. Thus when this predicate is queried we obtain:

```
(user)      ?owner-total(owner,item,tot)
            owner = JOHN, item = BICYCLE, tot = 160
            owner = JOHN, item = BOAT     tot = 3000
```

However, just to emphasise reversibility the following query can also be asked:

```
(user)      ?owner-total(owner,item,3000)
            owner = JOHN, item = BOAT
```

with the third argument position bound to 3000.

1.1.8 Deletion. Definitions are deleted by typing an identical statement to the original assertion, but preceeded by a 'not'. For example:

```
(user)      ?not owns (JOHN,2,BICYCLE,80)←
(computer)  OK
```

will remove the earlier entry from the predicate 'owns'.

1.1.9 Logical implies. This uses the symbol '⇒'. If p(x)⇒q(x) then this means that if p(x) is true then q(x) must be true. However, if for example q(x) is true it does not mean that p(x) is true. If we regard p and q as sets then the set q must contain the set p.

1.1.10 Formulas. Formulas in this paper are restricted to conjunctions of predicates. This implies no loss of generality as the predicates themselves can be defined in terms of disjunctions.

1.1.11 Universal and existential quantifiers. These are meta predicates that test if an expression is true for all or some value(s) of a variable(s), respectively.

1.1.12 Implication network. The implication network p→q is defined in Section 2 as having the dual meaning:

(*a*) p⇒q
(*b*) representing the two formulas p & q, q.

In general q can itself be a network in which case it represents further logical implies and formulas.

2 The implication network model

The implication network provides the model builder with a single unified method of specifying constraints on changes and access to the model so that the consistency of the model is always maintained.

PROLOG currently contains no concept of a model and so all operations are unconstrained (see the sample dialogue in the glossary explaining predicate). The purpose of including the implication network model in PROLOG is to provide the same kind of constraints and compact language facility that ICL is now building into some of its CAFS database products. The insertion or deletion of data (and rules) can now be prevented, if they prove to be inconsistent with the model.

The implication network described in the following provides a constraint on inserts and deletions to the model. This is done by the machine calling an update predicate which by using the network determines if the new model state is still consistent with the network. The compact model variable logic is provided by a master predicate which again uses the network to give the relationship between any combination of model variables.

2.1 Constructing an implication network

2.1.1 Stage 1 – identifying constraints. The model builder first identifies the elementary predicates. When an argument position in two different predicates refers to the same class of object then the same variable name must be used. For example he might identify three predicates in a part supplier system:

ps(part,supplier)
p(part,partweight)
s(supplier,suppliercity)

He then observes that the existence of some entities implies the existence of other entities in the system. In terms of the model this means that data in one predicate logically implies that certain data must exist in other predicates. For example, he might decide that the existence of a supplier of a part logically implies that suppliers and parts must also exist separately:

ps(part,supplier) ⇒ p(part,partweight) &
ps(part,supplier) ⇒ s(supplier, suppliercity)

The user can assert:

(user) ?ps(NUT,GKN) ←

but he will only get the response OK if the data NUT exists in p and GKN exists in s. It also follows that an attempt to remove NUT from p using:

(user) ?not p(NUT,77) ←

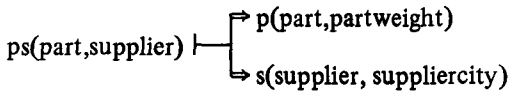
will be unsuccessful unless NUT,GKN is missing from ps.

2.1.2 Stage 2 – identifying the connection between entities. The model builder also wishes to be able to retrieve information from all the predicates just as if he were observing reality. For example, he observes that three formulas are sufficient:

ps(part,supplier) & p(part,partweight) & s(supplier,suppliercity)
p(part,partweight)
s(supplier,suppliercity)

The first formula simply joins all predicates together on the assumption that they are all connected by being part of the same model. The other two formulas are necessary because the 'logical implies' constraint above allows a part pl to exist in p even though pl does not exist in ps. No separate formula is needed to represent the ps predicate because this can only contain parts and suppliers that exist. Notice that the formula p&s is not included, because such a join has no physical meaning.

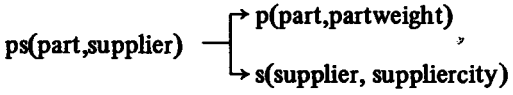
2.1.3 Stage 3 – drawing the implication network. It is possible to write the 'logical implies' identified in Stage 1 in the form of a network rather than the two expressions used earlier:



The three formulas identified in Stage 2 appear from this network by constructing each formula from the conjunction of each predicate with all predicates that this

predicate logically implies. Such an interpretation seems to serve well as it specifically excludes unnatural connections such as p&s while unifying the idea of constraint and connection.

When the network has this dual interpretation, the arrow '→' represents both logical implies '⇒' and a set of formulas connecting the predicates together. Thus the implication network of the part supplier system is:



The systems analyst/scientist when using this network model must always visualise both these formal interpretations when constructing a model.

2.2 *Using the implication network*

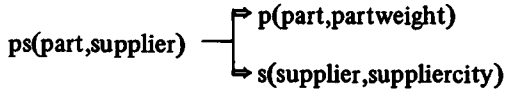
The network is stored in a computer as a predicate called 'network'. Thus the network above would be input to the computer using the two assertions:

?network(ps(part,supplier), p(part,partweight)) ←
?network(ps(part,supplier), s(supplier,suppliercity)) ←

Every update automatically invokes the 'logical implies' interpretation of this network using the update predicate defined below. However, every query automatically invokes the 'formula' interpretation of this network using the master predicate below.

2.2.1 Update predicate. The update predicate checks that all the logical implications between the predicates in the network are true after any alteration to the predicates composing the model. Thus in terms of the part supplier example, we replace every → by ⇒ and check the resulting formula:

for all part, supplier, partweight, suppliercity



is true for all states of the model variables.

2.2.2 Master predicate. The master predicate m(x1,x2,...) uses the model variables x1,x2,... and selects the formulas in the model which include all these varables. Surplus variables are ignored after these formulas are interpreted and the data from all the formulas united into a single set. For example, if we ask the model how the variable's partweight and suppliercity are related the system will use the formula ps&p&s. However, if the user asks for a list of suppliers and their cities the system uses formulas ps&p&s and s, although in this case because ps&p&s⇒s the larger formula is redundant and only s is used.

2.3 Formal definition of the model

A formal definition of the implication network model is given in the Appendix.

3 Dialogue with a simplified part-supplier-orderer system

This session illustrates setting up a part-supplier-orderer model, changing the data and rules in the model and querying the model. The orders model contains two predicates ps and pso with typical data as shown below. The first predicate ps gives the unit price of a part from a particular supplier. The second predicate pso gives the quantity of parts that an orderer wants from a supplier.

Each query or assertion is preceded by a question mark generated by the machine. The output messages below are the absolute minimum; a commercial version of PROLOG-M would explain itself if asked.

```
ps (part,      supplier,price)
    NUT        GKN    2
    BOLT       GKN    3
    WASHER     GKN    1
    BOLT       LUCAS  1000

pso(part,      supplier,orderer, qty)
    NUT        GKN    JIM     200
    BOLT       LUCAS  HENRY   2
```

3.1 Creating the model

An extra model variable 'cost' is required which gives the total cost of each order by multiplying the unit price by the quantity. This is represented by a predicate ord by typing the definition:

```
(user)      ?ord(part,supplier,orderer,qty,price,cost) ←
                pso(part,supplier,orderer,qty)sp & cost = qty * price
(computer)  OK
```

In addition, the user does not want orders placed unless there is a price quoted in the part-supplier catalogue ps. Thus he wants this new ord predicate to imply the ps predicate. The implication network is therefore:

ord(part,supplier,orderer,qty,price,cost) →ps(part,supplier,price)

To create the data above we assert that ord implies ps using the system predicate called network:

```
(user)      ?network(ord(part,supplier,orderer,qty,price,cost),
              ps(part,supplier,price)
(computer)  OK
```

3.2 *Insertion of data*

The predicates pso and ps are now updated using a series of PROLOG-M assertions:

```
?ps(BOLT,LUCAS,1000) ←
   OK
```

After this and every change to a predicate the update predicate is called to check that ord still implies ps. In this case, the update predicate is still true after this assertion, and so a further assertion is made:

```
?pso(BOLT,LUCAS,HENRY,2) ←
   OK
```

The pair BOLT LUCAS is present in the implied predicate ps and so this insert is allowed. The user now types:

```
?pso(NUT,GKN,JIM,200) ←
   INFINITE OR UNCOMPUTABLE TASK
```

The pair NUT,GKN does not exist in the implied predicate ps and so the insert is flagged uncomputable awaiting a suitable update to the predicate ps. Pairs of inserts can be made:

```
?pso(NUT,GKN,JIM,200) ← & ps(NUT,GKN,2) ←
   OK
```

The pso update is attempted first. However, this is delayed because of the absence of NUT,GKN in ps. The insert ps is therefore attempted followed now by a successful insertion to pso.

The dialogue above continues until all the tabulated data above has been inserted into the predicates composing the model.

3.3 *Deletion of data*

To illustrate deletion here is a short session to remove some data:

```
?not ps(NUT,GKN,2) ←
   INFINITE OR UNCOMPUTABLE TASK
```

Breaks logical implication ord to ps and is therefore uncomputable. To delete this item we must first remove 'NUT,GKN,JIM,200' from pso by adding this in the above command:

```
?not pso(NUT,GKN,JIM,200) ← & not ps(NUT,GKN,2) ←
   OK
```

This is just two assertions written on one line. The '&' allows the machine to choose

the execution order so that the overall deletion is successful.

3.4 Querying the model

In this session the data is assumed to be as in the box above. To list parts using formula ps the user types:

```
(user)       ?m(part)
(computer)    part=NUT
              part=BOLT
              part=WASHER
```

This uses formula ps to retrieve this information. To list the suppliers of washers the system again uses formula ps:

```
?m(part,supplier) & part=WASHER
  supplier=GKN part= WASHER
```

To list the parts, except BOLT the user types:

```
?m(part) & not part=BOLT
  part=NUT
  part=WASHER
```

To list part, supplier pairs even when the query involves other variables a 'colon' is used. To obtain the suppliers of parts with a cost equal to 400 the user types:

```
?part,supplier : m(part,supplier,price) & cost=400
  part=NUT supplier=GKN
```

However, because the 'cost' predicate is reversible we can also obtain the cost of all orders for part NUT and supplier GKN:

```
?cost:m(part,supplier,price) & part=NUT & supplier=GKN
  cost = 400
```

To obtain the parts which have been ordered from some or any person, the user types:

```
?m(part) & some orderer(m(part,orderer))
    part=NUT
    part=BOLT
```

To obtain the parts which have not been ordered from some person, the user types:

```
?m(part) & not some orderer(m(part,orderer))
    part=WASHER
```

To list parts and suppliers:

```
?m(part,supplier)
  part=NUT      supplier=GKN
  part=BOLT     supplier=GKN
  part=WASHER   supplier=GKN
  part=BOLT     supplier=LUCAS
```

From this listing it can be seen that the supplier of all parts is GKN. To list this using the computer the user types:

```
?m(supplier) & all part (m(part)⇒m(part,supplier))
  supplier=GKN
```

The suppliers of *not* all parts:

```
?m(supplier) & not all part (m(part)⇒m(part,supplier))
  supplier=LUCAS
```

The other parts made by the supplier of NUTs: (the '_x' allows two model variables, ranging over the set of parts, to exist at once)

```
?part:m(part_x,supplier) & part_x=NUT &
                m(part,supplier) & not part=part_x part=BOLT
  part=WASHER
```

4 Examples of networks

The networks that follow should really only contain predicates but sometimes the definitions of predicates are included instead, in which case they are in italics.

4.1 Family network

The following model could be used by a local authority, government or educational authority to model the members of a family. The model variables are male, female, father, mother and child. A practical model would be much larger and would include many additional facts such as name of doctor, age, record of illnesses, qualifications etc.

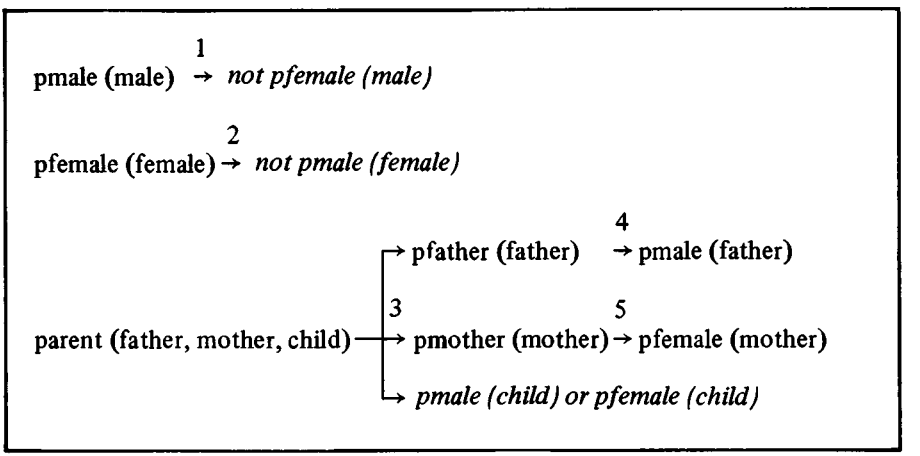
We would store facts as quite neutral assertions such as:

```
?pmale(JOHN-SMITH) ←
```

However, by including constraint rules we can reduce the chance of errors. Thus using the model below, the statement:

```
?pfemale(JOHN-SMITH) ←
```

will be flagged as uncomputable because we have included a rule that males and females cannot have the same name. This is not to say we could not include additional predicates to represent people whose names are ambiguous, such as Leslie.



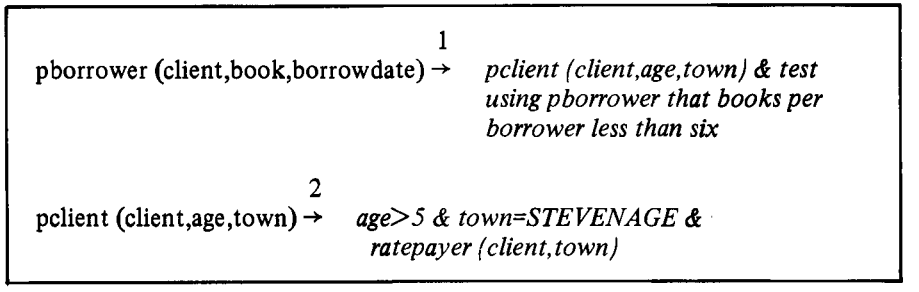
where the numbered rules mean:

1 A male is not a female. We cannot include a name into the pmale predicate if that name already exists in the pfemale predicate.
2 A female is not a male.
3 A father and mother are only parents of a child, if there exist a father and mother and their child is male or female.
4 A father is male.
5 A mother is female.

4.2 Library network

A town library contains the following rules.

1 A borrower is someone who is a client of the library and who has fewer than six books on loan.
2 A client of the library is someone who is older than 5 years, lives in Stevenage and is a ratepayer.



The main purpose of this model is to check updates to the model. Thus each time a book is borrowed we assert:

?pborrow(JACK-SMITH,PRIDE-AND-PREJUDICE,5OCT1983)←

Two checks are then made by the model. First that JACK-SMITH is a client of the library using pclient. Secondly, that JACK-SMITH has borrowed fewer than six books. In this system the user must remove the original assertion when a book is returned.

In addition to this crucial update role, the model can also be used to answer some quite useful questions such as 'what books have seven year olds borrowed':

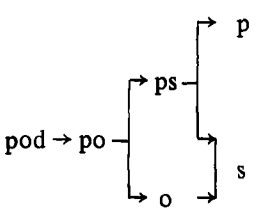
?m(book,age) & age=7.

4.3 An actual orders model

The following orders model has run on the ICL CAFS and consists of six basic predicates. Each predicate describes some semi-independent aspect of an order. Thus the supplier predicate s describes supplier together with a large number of descriptive argument places such as location, name etc.

s(supplier-number, ...),	available suppliers
p(part-number, . . .),	available parts
ps(part-number,supplier-number, . . .),	catalogue of parts and supplier
o(order-number,supplier-number, . . .),	complete order to a supplier
po(part-number,order-number, . . .),	entry in order for a part
pod(part-number,order-number,delivery date, . . .)	dates when parts in an order will be delivered

These were combined into the following network model:



However, this is only part of the network. In fact, each predicate has a primary key which uniquely identifies each entry in the predicate. For example, the primary key of the ps predicate is the 'part-number, supplier-number' pair, which can be checked by a predicate key (ps). Thus to represent this there is an additional link in the network:

ps(part-number,supplier-number, . . .)→key (ps)

Even larger networks result from including job-number in the model so that the originator of an order is also included. Some of these predicates can contain 10 to 20 arguments. To answer a query by actually writing this out would be impractically complicated – hence, the need for the master predicate.

5 Model variable logic

One of the big advantages of the implication network model is that model variables can even allow logic queries to be posed with no mention of the master predicate. The resulting logic language, although completely formal, has a compactness comparable with English and introduces the notion of 'typing' into logic. Although the translation is carried out by the machine, it is nevertheless important that the rule be simple enough for the machine translation to be easily explainable.

5.1 Conjunction rule

Consider the query 'suppliers of NUTs ordered by JIM':

```
?supplier:part=NUT & orderer=JIM
```

The machine accumulates all the variables mentioned before the colon and in the conjunction. It then includes a master predicate with all these variables as its argument:

```
?supplier:m(supplier,part,orderer) & part=NUT & orderer=JIM.
```

This set of accumulated variables is called the context. A colon such as the one above creates a new local context starting with just the variables before the colon.

5.2 Disjunction rule

The suppliers of the part NUT or to the orderer JIM is written as:

```
?supplier:part=NUT or orderer=JIM
```

Such a query is actually implemented as two separate processes:

```
?supplier:part=NUT
?supplier:orderer=JIM
```

but with their outputs being combined. Thus using our conjunction rule we obtain:

```
?supplier:m(supplier,part) & part=NUT
?supplier:m(supplier,orderer) & orderer=JIM
```

which translates back to the single query:

?supplier:m(supplier,part)& part=NUT or
m(supplier,orderer)&orderer=JIM

This shows that the context for each component of a disjunction is dealt with separately.

5.3 Some quantifiers

Queries involving quantifiers are particularly complicated in standard predicate calculus. One big benefit of model variables is in simplifying queries involving the quantifiers. The parts with no orderers can be obtained using the query:

?part: not some orderer

Explicitly, this means: consider each part in the set of parts m(part) and retrieve a part if no relationship m(part,orderer) can be found. Thus we would expect this to translate to:

?part:m(part) & not some orderer(m(part,orderer))

In detail, the original query is first transformed to normal logic notation:

?part: not some orderer(true)

The only free variable in this expression is 'part' and so we add m(part) using our conjunction rule:

?part:m(part) & not some orderer(true)

However, quantifiers create local variables and therefore local contexts. Thus the context where true has been written is part and orderer and so 'true' is replaced by m(part,orderer) to give:

?part:m(part) & not some orderer(m(part,orderer))

The answer to this query is WASHER using the data provided in Section 3.

5.4 Universal quantifiers

The suppliers of all the parts, in the catalogue of parts except NUT, can be obtained using:

?supplier : all part: part <>NUT

The colon after a quantified variable is a way of restricting the set of parts considered by a quantifier, in this case excluding NUT.

Translation of this occurs as follows. The only free variable in the whole expression

is 'supplier' and so from the conjunction rule the above translates to:

```
?supplier : m(supplier) & all part:part <>NUT(true)
```

As mentioned earlier, the colon resets the context and so 'part:part < >NUT' translates to 'part:m(part)&part <>NUT' using the conjunction rule. However, the context where 'true' has been written is 'supplier' and the locally created 'part' so that "true" is replaced by m(supplier,part) to give the explicit query:

```
?supplier : m(supplier) &
            all part:m(part)&part <>NUT(m(supplier,part))
```

5.5 *Dialogue with model variable logic*

The earlier session using explicit reference to the master predicate is repeated here using model variable logic:

```
?part:
?supplier,part:part=WASHER
?part:not part=BOLT
?part,supplier : price <20
?part : some orderer
?part : not some orderer
?part,supplier:
?supplier: all part
?supplier: not all part
```

List the other parts supplied by the supplier of a different part _x equal to NUT:

```
?part: some supplier(part_x=NUT & not part=part_x)
```

In each case the query is greatly simplified and a large amount of unnecessary detail removed.

6 Discussion

This paper has suggested the incorporation of the implication network model into a logic language. The combination of the two ideas is the model building language PROLOG-M.

A restricted form of the implication network model is already used with some success by certain customers of the ICL content addressable file store (CAFS).[1,2,6] In current database systems, the model has helped users to avoid inserting invalid data and removing data on which other data depends. It has also made it possible for many queries which, in the past, could only be posed by skilled programmers to be posed by users directly.

An orders model,[2] using an implication network, has been efficiently implemented

using CAFS. The model contained 15 to 20 predicates with 60 Mbytes of physical data in the form of relations. The nonphysical predicates were restricted to certain built-in operations such as the checking of primary keys. However, instead of checking the consistency of the whole model each time a change is made, only the areas of the model actually affected are checked for consistency.

Logic may not be considered efficient enough for the construction of all algorithms. The model variable logic might be the final point where a language meets the user. Underneath is the model and the predicates used in the model. However, underneath this is the existing library for performing specific computational tasks.

Most of the examples in this paper have been in the database and intelligent knowledge base area. However, PROLOG-M can be used to model scientific systems. In this case the model relates variables such as time, distance, voltage, flux, atomic-weight, etc.

Here is a simple database example which compares model variable logic with standard logic. We want the supplier who supplies all the parts in the part catalogue. In standard logic this must be written:

$$s : S(s,\_,\_)\&\text{all } p\ (P(p,\_,\_)\Rightarrow SP(s,p,\_))$$

Using tuple variables[7] this would be written as:

S.s : all P some SP (SP.s=S.s & SP.p=P.p)

Using model variables:

s : all p

Apart from the brevity of model variable logic, it also paraphrases easily into English – in this case "supplier of all parts".

7 Conclusion

This paper has suggested that a unified mathematical model should be included as an integral part of a logic language. The benefits of this approach are the compact model variable logic and an improved consistency in data or rules in the model.

Acknowledgments

The late Roy Mitchell, Vic Maller, Norman Truman, Len Crockford, Martin Stears, Owen Evans and the other members of the ICL Systems Strategy Centre, Stevenage, helped to formulate and develop the ideas in this paper. Their considerable help is gratefully appreciated by the author.

References

1 ADDIS, T.F.: 'A relation based language interpreter for a content addressable file store', *ACM Trans. Database Syst.*,, March 1982.
2 BABB, E.: 'Joined normal form: a storage encoding for relational databases', *ibid.* December 1982.
3 CLOCKSIN, W. and MELLISH, C.: *Programming in PROLOG,* Springer-Verlag, 1981.
4 KOWALSKI, R.A.: *Logic for problem solving,* North Holland, 1979.
5 BABB, E.: 'Finite computation principle – An alternative method of adapting resolution for logic programming', Presented at Logic programming 83, Portugal.
6 BABB, E.: 'Implementing a relational database by means of specialised hardware', *ACM Trans. Database Syst.*, 4, (1), 1979, 1-29.
7 CODD, E.F.: 'A relational model of data for a large shared data bank', *Commun.ACM,* 13, (6), June 1970.

Appendix

Formal definition of the implication network model

The model can be defined formally in predicate calculus as a predicate called model which relates a set of model variables.

Formal definition of the model

(1) model (w1,w1, . . .) ← u & m(w1,w2, . . .)

If the update predicate is true then use the master predicate to generate instances of w1,w2,w2, . . .

(2) m(w1,w2, . . .) ← formula(h) &
(w1,w2, . . .) is a subset of (z1,z2, . . .)&
interpret h

Consider each formula h. (*formula (h) defined below)
If the variables (w1,w2, . . .) are a subset of the variables (z1,z2, . . .) used in formula h then interpret formula h and thereby bind the variables in (w1,w2, . . .)

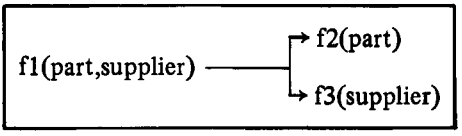
(3) u ← for all model variables, for all f(. .),g(. . .)
f(x1,x2, . . .) → g(y1,y2, . . .) ⇒
f(x1,x2, . . .)⇒g(y1,y2, . . .)

Check that every link (f → g) obeys the logical implication f⇒g for all model variables

* The formulas are obtained from the network f → g by creating one formula per node; nodes can be empty. Thus at node x we create a formula which is the conjunction of the predicate at x with all the predicates directly or indirectly pointed to by x.

Using the formal model

These formal definitions are explained using a part-supplier model:



Master predicate

The master predicate, rule 2, is used to obtain the instances of supplier:

(user) ?m(supplier)

Using the definition of the master predicate above we first generate all the formulas in the model in turn:

(m/c)	?formula(h)	
	h=f1(part,supplier)&f2(supplier)	therefore z1=part, z2=supplier
	h=f2(part)	therefore z1=part
	h=f3(supplier)	therefore z1=supplier

The model variable x1=supplier is a subset of z1=part, z2=supplier and so the instances of h=f1(part,supplier)&f2(supplier) can be generated:

(m/c) ?h
supplier = GKN
supplier = LUCAS

The variable x1=supplier is also a subset of z1=supplier and so the instances of h=f3(supplier) can also be generated:

(m/c) ?h
supplier = GKN
supplier = LUCAS
supplier = GEC

Update predicate

Updates are checked against the update predicate given in rule 3 above: In English this line checks that every arrow:

$$f \rightarrow g$$

obeys the logical implication $f \Rightarrow g$ for all model variables. In this example there are two links obtained using the query:

```
(m/c)       ?network(f,g)
              f=(f1,part,supplier)g=(f2,part)
              f=(f1,part,supplier)g=(f2,supplier)
```

Thus there are two tests to check that f1⇒f2 and f1⇒f3 for all instances of supplier and part:

```
(m/c)       ?for all (part,supplier) ( f1(part,supplier) ⇒ f2(part) &
                                       f1 (part,supplier) ⇒ f3(supplier) )
```

QPROC: a natural language database enquiry system implemented in PROLOG

M.G. Wallace and V. West
ICL Application Systems Division, Bracknell, Berkshire

Abstract

QPROC is an interactive natural language enquiry system providing access to a relational database. Both the system and its database are implemented in the PROLOG language. The paper aims to demonstrate the power and practicality of PROLOG for this type of application. After an introduction to QPROCs facilities and its architecture, the main body of the paper is concerned with the most interesting aspects of the implementation. These include the representation of natural language grammar and the generation of formal database queries.

1 Introduction

In 1979 ICL, the BBC and the Science & Engineering Research Council set up a bursary at the University of Southampton to investigate the use of natural language for database enquiries. The work was undertaken by one of us (MGW)[1] and a pilot implementation (QPROC) was completed.

QPROC is an interactive natural language enquiry system providing access to a relational database. Both the system and its database are implemented in the PROLOG language. This paper does not call for any great familiarity with PROLOG, but the reader may find it helpful to consult the glossary of terms given by Babb[2] in his paper in the same issue of this journal; a fuller treatment is given in the book by Clocksin and Mellish.[3] PROLOG provides two features which make it particularly suitable for parsing natural language: sophisticated pattern matching and automatic backtracking. It has therefore often been chosen for natural language applications.[4,5,6]

The main aim of this paper is to demonstrate the power and practicality of PROLOG for this type of application. After an introduction to QPROCs facilities (Section 2) and its architecture (Section 3), the main body of the paper is concerned with the most interesting aspects of the implementation (Section 4). The original version of QPROC ran under UNIX[7] on the Southampton PDP 11/34 and was converted to run under VME on the ICL 2900 using the PROLOG system developed at the University of York.[8] Some remarks on the portability of PROLOG are given in Section 5.

2 Facilities

The following session on an election database illustrates some of the QPROC facilities. Each of the user's queries or commands is followed by the system's response. A query must end with a question mark, full stop or exclamation mark. The queries have been numbered for reference in the notes that follow. The election database records (fictitious) details of candidates, constituencies and elections.

```
How old is Mrs Smith?                                        (1)
  37

List the Tory candidates.                                    (2)
  2 were found:
    Smith Disraeli

Which seats are Smith and Bolton the members for?            (3)
  Smith:   Worthing
  Bolton:  Southampton

Who stood at Worthing for the Labour party in 1974?          (4)
  Jones

Smith won what, when, where?                                 (5)
  elect1
  elect1: 1974
  1974 and elect1: Worthing

Where were the elections in 1974 and 1975?                   (6)
  elect1: Worthing
  elect2: Newham

Name a seat whose member is a Liberal.                       (7)
  Southampton

Who stood in an election at Southampton?                     (8)
  3 were found:
    Freeston Robson Bolton

Who stood in the election at Southampton?                    (9)
  The computer cannot understand what
  "the election at Southampton" refers to.

Who is a professor around here?                              (10)
                  ?????? ????
  Bolton

Bye                                                          (11)
  Goodbye
```

Notes

(4) The prepositional phrase 'at Worthing' qualifies the verb 'stood'.

(6) The prepositional phrase 'in 1974 and 1975' qualifies the noun 'elections'.

(7) Relative clauses may be freely used.

(8, 9) As there was more than one election at Southampton, the singular definite noun phrase 'the election at Southampton' causes an error message. This condition is known as presupposition failure.

(10) Unrecognised words are ignored, the offending word(s) being indicated by question marks.

3 Architecture

Fig. 1 illustrates the run-time architecture, which consists of application-independent and application-dependent components. The application-independent components are the initialiser, controller, parser, lexical analyser and formal query language interpreter.

The *INITIALISER* consults the other PROLOG files and enters the *CONTROLLER* which accepts the next natural language query from the user and passes it to the *PARSER* for grammatical analysis. The parser is generated by a PROLOG program from the natural language grammar rules (see Section 4.1). To dissect the query into its individual words the parser uses the *LEXICAL ANALYSER*. The result of parsing is a formal database query which the controller passes to the *FORMAL QUERY LANGUAGE INTERPRETER*. The query is expressed in a language known as *'DESCRIPTIONS AND QUALIFIERS'* (see Section 4.3.1). The parser may also generate formal (sub) queries, primarily to find the references of definite noun phrases like 'the election at Southampton'.

The application-dependent components are the live dictionary, data dictionary, data values and data. The *live dictionary* contains entries for all the natural language words relevant to the application (its generation is described later in this Section). The *data dictionary* provides a definition for the application relational database, in terms of its relations, attributes and domains. The *data values* give the values which may exist for each domain, which may be more extensive than those currently in the data. The *data* provides the database contents.

Fig. 2 illustrates the generation of the application live dictionary. This involves two further application-independent components, the core dictionary and the dictionary generator, and one further application-dependent component, the application dictionary. The *core dictionary* contains entries for the natural language words common to all applications (the current English version contains words like 'a', 'and', 'is', 'list', 'which' and 'who'). The *application dictionary* contains entries for the natural language words specific to an application but not included in the

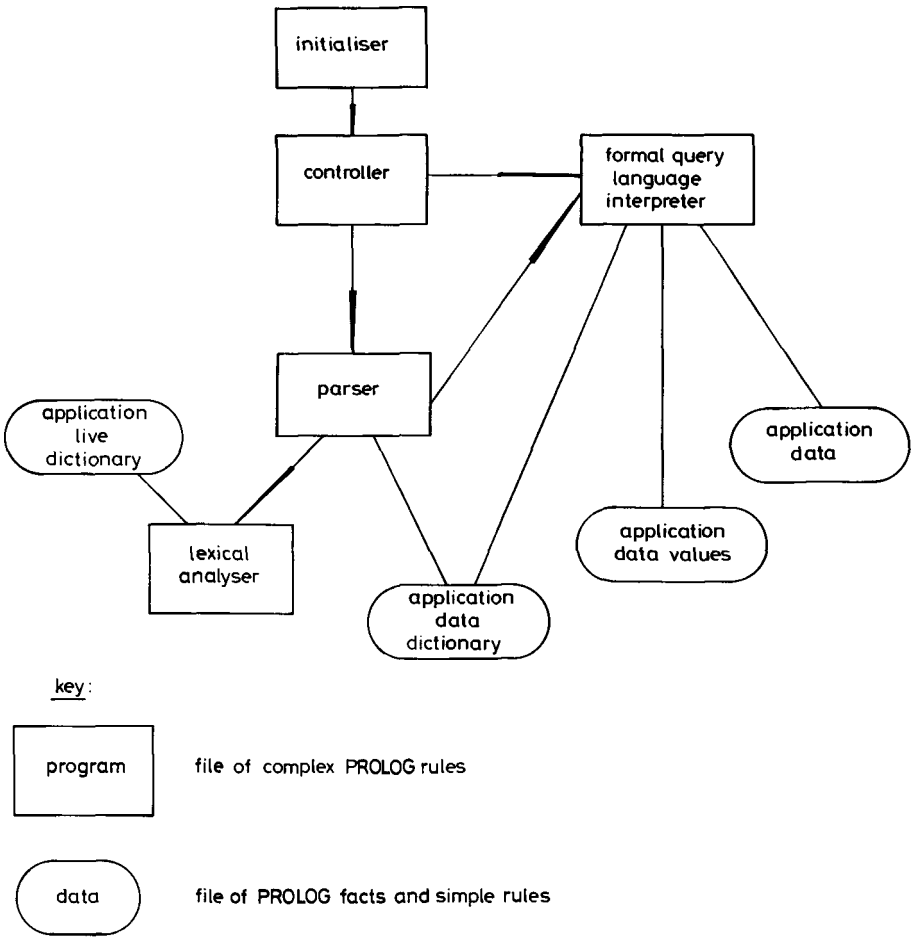


Fig. 1 Run-time architecture

data dictionary or data values (e.g. synonyms like 'Tory' for 'Conservative' and 'seat' for 'constituency').

4 Implementation

The aspects of the implementation discussed here are:

- the representation of natural language grammar
- the natural language dictionaries
- the generation of formal database queries
- an additional component, the convertor: its place in the architecture is described below.

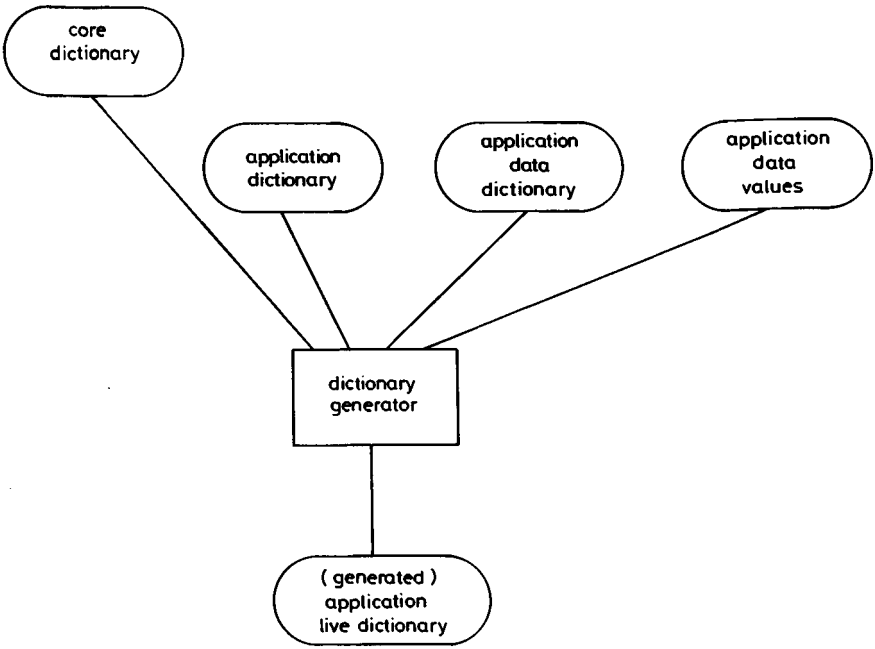


Fig. 2 Generation of application live dictionary

4.1 The representation of natural language grammar

4.1.1 The grammar compiler. Fig. 3 illustrates the generation of the parser. The *grammar compiler* generates it from the *grammar rules*. All the components involved are application independent.

The grammar rules are expressed in a short-hand form of PROLOG which is more readable and concise than the full form (for a detailed discussion see Clocksin and Mellish).[3] Some PROLOG implementations support grammar rule notation, but QPROC contains its own, more general, grammar compiler to generate the full form. Extra features in the grammar compiler include:

- left recursion is permitted in rules
- the start and end points of the current phrase are available for error reporting: see for example Section 2, query 9
- the ability to look ahead at subsequent words in the sentence.

4.1.2 Semantics. Because QPROC is only attempting to recognise the restricted semantics of queries that can be put to a relational database, the analysis of the sentence is more limited than would be necessary for general linguistic purposes.

A sentence has a verb and a number of verb modifiers. The verb modifiers are noun phrases or adverbial phrases that modify the verb. In the sentence:

Smith contested an election at Worthing

the verb is 'contested', and the verb modifiers are 'Smith', 'an election' and 'at Worthing'. The verb modifiers can be distinguished by their grammar. If one appears before the verb it is the subject. After the verb it is an object.

QPROC's dictionary associates with each verb a database relation and with each proper noun a data value. Thus we can illustrate a simple example of mapping a natural language query onto a database:

sentence:	Bolton contested which election?		
grammar:	subject	verb	object
meanings:	'Bolton'	contest	X (variable)
database:	contest	person	election
		'Smith'	'elect1'
		'Bolton'	'elect1'
		...	...

In PROLOG grammar rule notation this analysis can be expressed as:

```
sentence (Meaning) →   subj(Value),
                       verb (Relation, Attributes),
                       objects (List),
                      {Match ([Value | List] , Attributes, Args),
                       Meaning = . . [Relation | Args]}
```

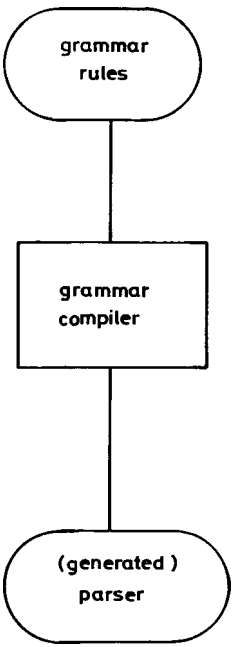


Fig. 3 Generation of the parser

This grammar states that a 'sentence' comprises:

'subj' and 'verb' and 'objects'

Each of these must also have a grammatical definition. The goals in curly brackets are ordinary PROLOG goals, which derive the meaning.

For the query 'Bolton contested which election?', the 'meaning' would be the PROLOG structure:

'contest ('Bolton', X)'

and when this is executed against the database it yields the answer X = 'elect1'.

4.1.3 Using database domains. In the query:

Who contested an election in 1974 in Newham?

the two verb modifiers 'in 1974' and 'in Newham' are grammatically indistinguishable. Their order of occurrence within the sentence might equally well be reversed so even this cannot be used to make the distinction between them. However, verb modifiers can be distinguished not only grammatically, but also semantically. When a verb modifier is interpreted onto the database it is associated with a database domain. In the election database, 'Newham' is mapped to the data value 'Newham' which belongs to the domain of 'constituencies'. '1974' is a 'date'. Thus the example sentence is mapped onto the database as follows:

sentence:	Who contested an election in 1974 in Newham?				
grammar:	Subject	verb	object	in-object	in-object
meanings:	person:X	contest	election:Y	date:1974	constituency:Newham

(the name before a colon is a domain name, which is usually the same as the attribute name)

database: contest	person	election	date	constituency
	'Smith'	'elect1'	1974	'Newham'
	'Jones'	'elect1'	1974	'Newham'
	...	...	...	...

4.1.4 The noun phrase. The simplest noun phrase is a name, 'Bolton'. QPROC deals with compound noun phrases, as well, which may include:

a determiner	'the', 'a'
a number	'10'
adjectives	'southern', '50 year old'
classifiers and headnouns	'Newham election'
postmodifiers	'The vote *for Smith*'
relative clauses	'The seat *where Bolton was the MP*'

The components of a noun phrase are to some extent reflected in the syntax of descriptions and qualifiers, given in Table 1, Section 4.3.1.

Adjectives, classifiers, postmodifiers and relative clauses are only meaningful if they can be linked with the headnoun of the noun phrase. To forge this link QPROC uses the data dictionary. Thus if the database has a relation 'candidate' with attributes 'vote' and 'person' then 'the vote for Smith' means 'the value of the "vote" attribute in the candidate relation, where the "person" attribute is "Smith" '.

To illustrate the preceding discussion we now give one of the PROLOG clauses for 'nounphrase' which actually occurs in QPROC; the lines have been numbered for reference in the notes that follow:

```
nounphrase (Dom: Desc, Num & Case) →                    (1)
nounphrase (Dom: Desc1, _ & Case),                      (2)
(explicit (findrefs (Dom: Desc1, Desc, F, L), F, L);    (3)
nounmod (Dom: Desc1, Desc, Num))                        (4)
```

Notes

(1) 'nounphrase' returns a description ('Desc'), with an associated database domain ('Dom'). For grammatical purposes it also returns a number ('Num') and case ('Case') (see also Section 4.2 'inflection').

(2) The clause is left recursive, its first goal being a call to 'nounphrase' itself.

(3) 'findrefs' checks if the nounphrase is definite (e.g. '*the* candidates at Newham'), and if so finds all the references in the database.

'explicit' makes available the start and end points ('F' and 'L') of the nounphrase in the sentence which contains it. If 'findrefs' fails, it notes an error (see Section 2, query 9).

(4) If 'findrefs' fails, the goal 'nounmod' is called to look for postmodifiers.

4.2 The natural language dictionaries

The core, application and live dictionaries have already been introduced in Section 3. The form of dictionary entry used is described below (the live dictionary is slightly different).

Each dictionary entry is a PROLOG fact for the predicate 'dic' and has four elements:

part of speech
word
inflection
meaning

An example is

dic (relative pronoun, "who", _ , person: _)

QPROC supports the following:

Part of speech.	*Example(s)*
sentence (example)	bye
verb	be, list, win
noun	seat
noun phrase	Mrs. . .
adjective	old
value (names a data value)	tory
relative pronoun	who
interrogative pronoun	who
determiner	the
interrogative determiner	how many, which
preposition	for
particle	not
conjunctive	and, or

Pronouns, adverbs and interjections are not yet provided.

Word. This may be a single word, e.g. 'who', or may have a continuation, e.g. ' "how" _ _ "many" ', ' "what" _ _ dic (verb, _ , _, be)'. The last expects any verb with meaning 'be' after the word 'what'.

Inflection. Only irregular inflections are held in the dictionary as regular ones are deduced by the lexical analyser. Many words have no inflection, indicated by '–'.

For verbs, the possible inflections are:

present, infinitive, perfect, past participle ('–en')

so there are entries like:

dic (verb, "did", perfect, dic ("do"))

For nouns, the possible inflections are

number & case

where number is singular or plural or both and case is nominative or accusative or both, or genitive. An example is

dic (relpron, "whom", _ & accusative, person _)

Meaning. Several types of meaning are illustrated in the above example:

function word, recognised by the parser, e.g. 'be'

synonym, e.g. 'dic("do")'

variable over a specified domain, e.g. person: _)

4.3 The generation of formal database queries

4.3.1 Descriptions and qualifiers. The formal query language used, called 'DESCRIPTIONS AND QUALIFIERS', is recursively defined in terms of descrip-

tions which refer to sets of data items and qualifiers which assert how the items are related. This structure is close to the structure of natural language, as descriptions interpret nounphrases and qualifiers interpret clauses. It is intended to be independent of the particular natural language.

Once the query has been interpreted by the formal query language interpreter, not only will the meanings be available but also the references in the database of each nounphrase. This potentially helps with the interpretation of succeeding queries which operate within the context of preceding ones or refer back to them, but QPROC does not yet take advantage of this. It also helps with the diagnosis and reporting of presupposition failure (see Section 2 query 9).

The syntax of DESCRIPTIONS AND QUALIFIERS is shown in Table 1. A query is expressed as a PROLOG structure, using the functors 'desc' and 'qual', certain operators (↑, &, or, not, is, =) and some atoms ('true', 'false', 'any' etc.). The RELATIONS and ATTRIBUTES are those of the application database. VARIABLE stands for a PROLOG variable and LITERAL for a PROLOG atom or integer.

Table 1 The Syntax of DESCRIPTIONS AND QUALIFIERS

Notation
Terminals appear in lower case, nonterminals in upper case.

Alternatives are separated by a vertical bar.

{ }... indicates optional repetition.

Query

QUERY	→	RULE

Qualifiers

QUALIFIER	→	qual (VARIABLE, RULE)
RULE	→	true | false | RELATION ↑ [ARGUMENTS] | not RULE | RULE & RULE | RULE or RULE | DESCRIPTION is QUALIFIER
ARGUMENTS	→	ATTRIBUTE = VALUE {,ATTRIBUTE = VALUE}...
VALUE	→	VARIABLE | LITERAL

Descriptions

DESCRIPTION	→	LITERAL | desc (DETERMINER, INTEGER, QUALIFIER) | DESCRIPTION & DESCRIPTION | DESCRIPTION or DESCRIPTION
DETERMINER	→	any | the | what | no | all

4.3.2 Two natural language queries and their interpretations. The interpretation of a sentence or clause is a formal QUALIFIER. A simple sentence comprises a verb and a number of verb modifiers. The verb is interpreted as a formal relation, and the verb modifiers supply attribute values for that relation. The sentence

Every Tory candidate contested an election

has the interpretation

<Desc 1>is qual (V1, <Desc 2> is qual (V2, contest↑[person = V1, election = V2]))

where <Desc 1> and <Desc 2> are defined below.

The relation 'contest' interprets the verb 'to contest', and the verb modifiers, 'Every Tory candidate' and 'an election' supply values for the two attributes 'person' and 'election'.

'<Desc 1>' is the formal DESCRIPTION which interprets the nounphrase 'Every Tory candidate'. As this is a definite nounphrase, the parser finds all its references and returns the formal DESCRIPTION (<Desc 1>): 'Smith' & 'Disraeli'.

'<Desc 2>' is the following DESCRIPTION:

desc (any, 1, qual (V, true))

This is a vacuous DESCRIPTION which matches any value. The constraint that it must be an election is imposed through the variable 'V2', which can only match values in the 'election' attribute of the relation 'contest'. The whole formula is satisfied if for each value 'Smith' and 'Disraeli' there is a tuple in the relation contest with that value for the 'person' attribute. There is no constraint on the 'election' attribute as any value will do.

A similar sentence is:

'One election was contested by every Tory candidate'

which is interpreted as:

desc (any, 1, qual (V, true)) is qual (V2,

'Smith' & Disraeli' is qual (V1, contest ↑ [person = V1, election = V2]))

The descriptions are just as before but their order within the formal query is reversed. This formal QUALIFIER is only satisfied if there is some election, 'E', such that each of 'Smith' and 'Disraeli' occur as the 'person' attribute of some tuple in the 'contest' relation whose 'election' attribute is 'E'.

4.3.3 Generating formal subqueries during parsing. PROLOG's pattern-matching capabilities enable formal subqueries to be generated during the parsing process. This is best illustrated by another example from QPROC (the lines have been numbered for reference in the notes that follow):

```
(1) sentence  (Desc is qual (V, Qual), Mods) →
(2)           qnph (Dom: Desc, Num & accusative),
(3)           sentence (Qual, [modf (acc,Dom:V) | Mods] ).
```

This is one of the grammar rules for sentences. It deals with queries like:

'Which seat did Bolton win?'

(1) 'sentence' has two arguments. The first argument, 'Desc is qual (V, Qual)' is the formal QUALIFIER returned if the clause succeeds. 'Desc' is the formal DESCRIPTION returned from (2), and 'Qual' is the formal QUALIFIER returned from (3).

The second argument, 'Mods', holds a list of sentence premodifiers, e.g. *'In 1974 at Worthing* who stood for the Liberals?'

(2) 'qnph' is a grammar predicate which recognises question elements, such as 'Which seat'. The arguments are the same as for the 'nounphrase' (see Section 4.1.4 above). Notice that the 'case', which is generally returned as a result from 'qnph', is an input value, 'accusative', in this call. This reflects PROLOG's facility to use arguments indiscriminately for input or output.

(3) 'Sentence' is called recursively. The list of premodifiers now includes a new element, modf (acc, Dom:V) on the front of the list. This will be used to fill in the object of the verb 'win' in dealing with the remainder of the sentence '. . . did Bolton win?'.
The final interpretation of the sentence is:

```
desc (what, 1, qual (W, true)) is qual (V, 'Bolton' is qual (X, win↑[person = X,
     constituency = V] ))
```

4.4 The convertor

To interface QPROC to an independent database management system (DBMS), formal queries must be expressed in the form required by the DBMS and passed to it rather than to the formal query language interpreter. QPROC contains a convertor which converts the 'DESCRIPTIONS AND QUALIFIERS' formal query into the form required by the ICL Personal Data System (PDS)[9]:

list ATTRIBUTES where RULE

where ATTRIBUTES names which attributes of which relations are required and RULE defines which tuples of the relations are to be selected.

A simple example of conversion is the query from Section 4.3.3 above:

Which seat did Bolton win?

The formal interpretation is:

```
desc (what, 1, qual (W, true) is qual (V,
'Bolton' is qual (X,
win ↑ [person=X, constituency = V] )).
```

This is converted to the PDS list command:

```
list win. constituency where win. person = "Bolton".
```

PROLOG's pattern-matching capabilities make it a particularly appropriate language for implementing such convertors.

5 PROLOG portability

The main changes needed to bridge QPROC from UNIX PROLOG to York PROLOG were:

(*a*) Explicit uses of ASCII character codes were replaced by their EBCDIC equivalents. It is unfortunate that PROLOG implementations still require character codes rather than characters in some constructs (e.g. as arguments of the 'put' predicate)

(*b*) The notation A.B for the list with head A and tail B had to be replaced by [A | B]. There is no international standard for the language yet, so such differences between implementations can be expected.

(*c*) Owing to limits in UNIX PROLOG, QPROC was too large to run as a single process under UNIX and was split into three processes communicating via UNIX pipes. Under VME on ICL 2900, QPROC runs as a single process. Different PROLOG implementations can be expected to vary considerably in their limits.

6 Conclusion

The main aim of this paper has been to demonstrate the practical use of PROLOG on an application of some complexity. The current version of QPROC offers the casual user a subset of English covering a wide variety of the queries relevant to a database. The power of the PROLOG language is indicated by the compactness of the implementation. The total size of all the application-independent components is only 2500 lines, with the election database components accounting for another 400. Our experience indicates that the use of PROLOG can reduce complex tasks like natural language understanding to more mangeable proportions.

References

1 WALLACE, M.G.: 'QPROC: a natural language inquiry system', Ph.D. Thesis, University of Southampton, 1983.

2 BABB, E.: 'The logic language PROLOG-M in database technology and intelligent knowledge-based systems', *ICL Tech. J.*, 3, (4), 373-392.

3 CLOCKSIN, W.F., and MELLISH, C.S.: *Programming in Prolog*, Springer-Verlag, 1981.
4 DAHL, V.: 'Un Système Déduct if d'Interrogation de Banques de Donnees en Espagnol', Ph.D. Thesis, University of Marseilles, 1977.
5 PASERO, R.: 'A dialogue in natural language' Proceedings of the First International Logic Programming Conference, University of Marseilles, 1982.
6 WARREN, D.H.D. and PEREIRA, F.C.N.: 'An efficient easily adaptable system for interpreting natural language queries', Research Paper 155, Department of Artifical Intelligence, University of Edinburgh, 1982.
7 CLOCKSIN, W.F. and MELLISH, C.S.: 'The Unix Prolog system', Software Report 5, Department of Artificial Intelligence, University of Edinburgh, 1979.
8 SPIVEY, J.M.: 'The University of York portable prolog system: user's guide', University of York, 1982.
9 International Computers Limited: 'The personal data system (PDS70)'. Restricted Publication 2230, ICL House, Putney, London, 1982.

Modelling software support

P. Mellor
ICL Customer Service & Quality Division, Stevenage, Hertfordshire

Abstract

The paper describes a mathematical model for use in forecasting the cost of providing support for software products, which takes into account (i) software reliability (ii) support techniques (iii) commercial policy and (iv) engineers' time consumed in responding to incidents. The representation of reliability is based on the work of Dr. B. Littlewood of City University, in which failure is treated as being due to a number of independent Poisson sources having different rates. Three different support techniques are modelled: fix-on-fail, systems maintenance file release and use of known error log (the terms are explained in the paper). The effects of commercial policies such as the application of a warranty period can be studied, and customer queries are included as well as product failures.

A first version of the model has been programmed in Pascal for the ICL Personal Computer, and samples of output are used to illustrate points in its operation and application. Work is continuing on improvements and refinements to the model, using information gathered by the ICL software-support organisations. An accurate simulation of actual product failure data is presented.

1 Software failure and repair, and the need for modelling

1.1 The peculiar nature of software failure

Why does software fail, and how can we measure its reliability? The same questions can be asked about hardware and reasonable answers can be given, but for software the situation is far more complicated and far less well understood; there is not even general agreement on how to define software reliability, even though achieving high reliability is one of the major challenges facing the computer industry. Defining failure is equally problematical. One definition of failure in a product is 'deviation from specification', but the specification of a piece of software as complex as an operating system will almost certainly be incomplete and contain ambiguities, to the extent that such a simple definition will be inapplicable. Alternatively the product may be within the formal specification but may fail in the sense of not fulfilling in some way what the customer regards as his reasonable expectations; or there may be an obvious deviation from the specification, but this has so trivial an effect on the customer's work that it is not worth while to go to the trouble of a query.

Software does not fail as a consequence of age or wear. It fails because a human mistake was made in the design, or in the writing of the code (which can be regarded

simply as a more detailed stage of the design process). The result of the mistake is what is usually called a bug in the product. This is not necessarily a localised error in the code; it may result, for example, from conflicting interpretations of an interface specification being made by separate writers of different modules of code, or from an error made by the writer of the user manual which misleads the customer. Then, since all software faults are essentially design faults and are either present in the product or not, can they be said to be in any sense random? The treatment of software faults that will be described in this paper is based on assumptions of randomness, which can arise in either of these ways:

- A bug will only generate a failure in response to a certain subset of all possible inputs to the system. Encountering an input from this 'bad' subset is a random event. The subset which causes failure is different for each bug, and the frequency of failure caused by a given bug depends on the size of its bad subset.
- The total number of bugs and their individual frequencies are not known, and therefore we can treat them as random variables: this is the Bayesian approach. We have certain expectations of what the frequencies of bugs in a new product will be before making any observations of failures, which we express by assigning a probability to a bug having a frequency in a given range (the prior probability distribution). The observation increases our knowledge and changes our expectation, so that we can then assign different probabilities to the frequencies (the posterior distribution): this is so even if the bugs are removed as they are found.[2]

An important point, made most clearly by Campbell and Ott[13], is that a record of product failure distribution in time contains information *even about events which have not happened,* provided that they have been exposed to the 'risk' of happening during the relevant period. Realising this, we can use our observations to deduce a frequency distribution for the faults that are left in a product as well as those that have already caused a failure.

Admitting that different bugs may result in different frequencies of failure has the important implication that software with many infrequent bugs may be more reliable than software with a few frequent bugs. The important thing to determine is not 'the number of bugs in a program' but 'how sure are we that this program will perform satisfactorily for a given period of time?'

Mostly, software faults are removable; and software differs from hardware in that repair permanently changes the reliability of the product. Repair of hardware essentially replaces a failed component, which has ceased to perform to specification, by one that performs correctly; but the new component has the same reliability as the one it replaces and will itself eventually fail. Hence the reliability of the product is unchanged by repair. By contrast, a software product (with a static specification) becomes more and more reliable as faults are removed, and once right will perform correctly for ever. Further, while a hardware repair corrects only the single example of the product on which it is carried out, a software repair can be very easily applied to every example of the product in the field. A countereffect is that a software repair can itself introduce other faults, so that the reliability of

the product can be reduced as a consequence of the repair. This applies equally to any change made to the product, such as the addition of new code to provide enhancements; and since almost every software product does require enhancements from time to time, this implies that repair must be properly managed so as to ensure that on average reliability improves. Table 1 summarises the main differences between software and hardware from the points of view of reliability and repair. A final comment here, however, is that the 'hardening of software and softening of hardware' is tending to blur these distinctions; design faults in complex hardware bear a strong resemblance to software faults, and faults in 'firmware' or software distributed on ROM are in many respects like those of hardware.

1.2 The need for, and value of, a model

The users of most products expect the supplier to offer a service of support which will at least keep the product in good working order and repair failures quickly and effectively. To plan a strategy for such a service for a new product the supplier must know the likely demand for the service and the effect of technological and

Table 1 Differences between hardware and software from the support point of view

	Hardware	Software
Specification	simple, complete	extremely complex, almost certainly incomplete
Design	perfect. Design faults removed prior to production	imperfect. All faults are design faults
Manufacture	imperfect. Substandard components removed by 'burn-in'	perfect. New faults not introduced during copying and distribution
Effect of structure on reliability	well-defined, beneficial	poorly understood. 'Integration faults' frequent
Quality measures	MTBF well defined and completely describes failure characteristics	MTBF may not exist in some circumstances Reliability = Pr { system will fail before a given time } and failure rate should be used
Cost of failure	predictable, containable	unpredictable, may be catastrophic
Serviceability	MTTR can be ascertained for each component.	each fault different. Some not repairable
Cost of repair	predictable from cost of spare and MTTR	depends on diagnosis and fix time. Widely variable
Reliability after repair	as before failure	permanently altered. Probably better, possibly worse!
Effectiveness of repair	around 100%	can be as low as 50%
Repair requires:	movement of personnel and materials	movement of information
Cost of product update throughout field	prohibitive	relatively small

commercial decisions on the cost of providing it.

This is essentially a question of forecasting the costs of future operations on the basis of whatever relevant information is available, and what is needed if this forecasting is to be a rational process is a theoretical model – a mathematical model, in effect – that simulates the behaviour of the complete system formed by the product in the field combined with the supporting service. When such a model has been developed and its validity established by checking it against historical data, it can be used not only for forecasting but also for what is usually called sensitivity analysis: this means observing how the output – usually a predicted cost – varies as input parameters to the model are varied, and showing how the cost would be affected by changes that might be made in the real world in which the product is being used. The paper describes a model developed for this purpose in the ICL Customer Service & Quality Division.

1.3 Relevant published work

Because of the recent interest in ultra reliable systems in which software plays a crucial role, and of the inherent difficulty of defining, measuring and predicting software reliability, the literature on the subject has grown enormously over the past few years. Many of the published papers propose mathematical models of software failure and fault removal and the researcher is in danger of getting lost in this thicket; fortunately guidance is available in a group of excellent survey papers by Dale and Harris.[4,5,6] The earliest of these models is that of Jelinsky and Moranda,[9] published in 1972, the basic assumptions of which have persisted to a surprising extent: as Littlewood[1,3] has pointed out, most of the later models are essentially variations on this with corrections and refinements added. One particularly questionable assumption made by Jelinsky and Moranda is that all bugs in a product contribute equally to the failure rate. Littlewood abandons this assumption in his proposed models and uses Bayesian techniques to represent the different effects of different, individual bugs; his work on stochastic reliability growth[2] is the basis of the failure-rate part of the ICL model.

All these published models deal with the debugging of a single program in a stable environment; they are concerned solely with the effects on reliability and not at all with the cost aspects of the repair process. To be useful to a vendor of computer systems a model must take into account the following:

- 'customer effects': change in the apparent failure rate due to a change in usage of the product; variations in customers' skill in fault recognition and avoidance; reluctance to report failures, so as to save time; making general queries unrelated to failure
- presence in the field of many copies of each product, and variation of this population with time
- effect of delays in fault diagnosis or repair
- differing cost to the customer of different failures, i.e. variation of the severity of the effect
- differing cost to the vendor of different failures, i.e. variation in the resources consumed in diagnosis and repair.

Most published models, including the present one, treat the software product as a 'black box' and aim to predict its future reliability from past observations. In designing the present model it was felt that it would not be profitable to try to relate the detailed structure of the product to the bug content; the work of Kitchenham[7], for example, has shown that the use of complexity metrics may give no better prediction of bug content of the various modules of a big software product than does a simple count of the number of instructions in each module.

1.4 Background to the ICL model: functioning of the software support service

The terminology in use is defined in this Section. The usage of the terms 'fault' and 'failure' is consistent with that of Anderson and Lee.[8]

Any software product may contain bugs, which can lead to failure in use, and features which can lead the customer to make direct queries to the support service. These are referred to together as *event sources*; the issue of software to the field distributes the full set of sources with each copy of the product. Each copy is called an *instance* of a source, and an event generated by any instance is called a *manifestation* of that source. A manifestation can be either a *product failure* (PF) or a query related to something other than a failure, termed a *nonfailure query* (NFQ). A source can be *turned off* by being dealt with by one of the mechanisms described below; until turned off it is *active*.

Each source has its own *intrinsic failure rate*, but the failure rate observed in the field will vary from one customer to another, depending on the type and intensity of his use of the product. This effect of usage is represented by a *stress factor* which is combined with the intrinsic rate to give a *local failure rate* for each individual instance. The use of stress factors is explained later in the paper; it is usual[10,11,12] to classify installations into three *stress levels*.

The essential function of the support service is the resolution of events generated in the field. The ICL support organisations record information relating to faults and their repair in the maintenance database (MDB). An important part of this is the known error log (KEL), which contains symptoms by which the manifestations of known faults may be recognised.

Software repairs or patches are held in another part of the MDB and cross-referenced from the KEL.

The service is organised hierarchically, and for the purposes of the model three levels are assumed. The initial contact with the customer is a query to the first level; this can be due to a product failure or can be a *nonfailure query*; in either case the event may not pass beyond the query stage – which may still entail considerable work within the support organisation – or may lead to a *bug report* being raised. This will be passed to the higher levels and will be *closed* by being put into one of a number of *closure categories*. Four categories are used in the model, although in practice a much finer division is used. The four are as follows:

– *New product error (NPE).* A new fault is found in the software or in the docu-

mentation, and can be corrected. This improves things for all customers for that product.
- *Usability.* Includes user error, operator error, request for explanation. Solves the one customer's problem.
- *Known error (KE).* The recurrence of a fault previously entered in the known error log (KEL).
- *Unresolved.* Relates to a situation in which there is insufficient evidence for any decision to be made, or that the query is withdrawn later. Queries in this category may have taken a considerable resource from both customer and vendor, but there is no apparent value to either.

In the handling of these, the first (NPE) would be expected to consume most time, the third (KE) the least, and the others to be more variable.

The event source will be dealt with by one or other of several support mechanisms. The model takes into account the three following:

- *Fix-on-fail (FOF).* The single instance of a source which has manifested itself is turned off by software repair. The response to a *nonfailure* query should have the same effect, since the customer can be expected not to repeat the same query.
- *Known error log (KEL).* Once a new product error or usability problem has been disposed of and a fix, if applicable, is available it is entered on the known error log. Subsequent manifestations of other instances of the same source will then be closed as 'known error'.
- *Systems maintenance file (SMF).* This is also known as bug clearance release. It consists of the issue to the whole field of software repairs for all known product faults, or of a version of the software with all these faults source-cleared, which is identical from the point of view of the model. This turns off all instances of known products faults, but does not affect the nonfailure query sources.

2 The ICL model

2.1 Objectives

The aim is to provide a means of calculating, at each moment in the period simulated, the expected rate of generation of support events by all the systems in the field, corrected for turnoff by one or other of the support mechanisms or by expiry of a warranty. The events are then classified as known/new errors and product failure/nonfailure query events, allocated to closure classes and their costs assigned; accumulated costs are calculated, and classified in various ways. Successfully resolved events have a feedback effect on the field via the fix-on-fail, known error log and systems maintenance file mechanisms. An estimate of the expected reliability of each system can be calculated.

2.2 Basic assumptions

The theoretical basis on which the model rests is the assumption that each instance of an active fault or nonfailure query is an independent Poisson source with its

individual rate. It follows from this that the flow of events from the whole field is a nonstationary Poisson process whose instantaneous rate is the sum of the rates of all those instances that are active at that moment; this rate varies with time because the number of active instances varies. If the rate at time t is $f(t)$ then the accumulated number of events up to that time has a Poisson distribution with mean $F(t)$, where

$$F(t) = \int_0^t f(u)\,du$$

Appendix 1 gives a derivation of the nonstationary Poisson distribution from assumptions which are in fact less restrictive than those made here.

From this we can calculate the expected accumulation of events and the probability of the number accumulated in any period lying within any given bounds. It can be helpful to use the fact that a Poisson distribution with a large mean F approximates to a normal distribution with mean and variance both equal to F.

In simplified cases, such as a constant population, we can derive explicit formulae for the accumulation of events; but in the general case this is not possible and a computer program which treats the period under study as made up of a number of time slices has to be used.

2.3 Time and population

The period being studied is divided into intervals, typically of length 1 year; time-dependent quantities are specified by their values at interval boundaries and it is assumed that linear interpolation suffices to give intermediate values.

Each interval is divided into a number of time slices, short compared with the interval length and typically of the order of a few days; variables are treated as if they changed stepwise at the start of each slice.

Let M_I = initial population of systems in the field; i.e. the number present just prior to the start of the first interval
M_i = population at the start of interval i; the first interval is $i = 0$, so $M_0 = M_I$
D_i = number of systems delivered during interval i
P_j = probability that a system is still in the field j intervals after delivery

Then $$M_i = M_I P_i + \sum_{j=0}^{i-1} D_j P_{i-j-1} \qquad (1)$$

Let L be the length of the interval, h that of the slice and T_i (= iL) the start time of interval i. Then the assumption of linear interpolation within an interval gives the population $M(t)$ at a time t in the interval i as

$$M(t) = M_i + (M_{i+1} - M_i)(t - T_i)/L \text{ where } T_i < t \leqslant T_{i+1} \qquad (2)$$

We need to divide the population into stress levels and commercial categories, and the proportions in each will generally vary with time. A warranty may apply in some commercial categories, and a system whose warranty has expired will take no part in support operations. Other systems are called *active*. Let

$M_g(t)$ = total number of systems in stress level g at time t
$M_{cg}(t)$ = number of active systems in stress level g at time t
$P_s(k,t)$ = probability that a system present at time t will still be present and active at time $t+k$

Also let $Y(t)$ be the total accumulated operational time for the population up to time t: this is the total running time for all systems, weighted for stress and corrected for warranty. The weighting for stress level g is represented by a stress factor f_g; if the accumulated time for stress g is $Y_g(t)$, then

$$Y_g(t) = \int_0^t M_{cg}(u) f_g du \text{ and } Y(t) = \sum_g Y_g(t) \quad (3)$$

The stress factor f_g must take into account the ratio of running to elapsed time as well as the effects of customer usage; it must be made clear whether estimated failure rates refer to elapsed time or to running time and the stress factor used consistently. Time can be measured in any units – hours, days or years – so long, of course, as the same units are used throughout.

The field population is assumed to be given prior to the simulation, and is not changed by events occurring during the run. There is no representation of, for example, increased sales due to improved product reliability.

Fig. 1 gives an example of a population graph.

In this example, population is initially zero. The number of systems delivered in each of the six successive years is 100, 200, 250, 200, 100 and 50, respectively. The standard withdrawal rate is used, i.e. normal with mean 5 years. Percentage of systems left in each year after delivery is 99, 95, 87, 63, 37, 13, 5, 1 and 0, respectively. Note that population peaks and then declines as deliveries tail off.

2.4 Event generation and effect of support feedback

2.4.1 Probability of event manifestation: The standard notation Pr { statement } is used to denote the probability that the statement in the brackets is true, and Pr $\{a|b\}$ the probability of a, given that b is true.

If events are generated by a Poisson source with rate r, then

$$\Pr\{n \text{ events are generated in period of length } t\} = \frac{(rt)^n}{n!} \exp(-rt)$$

so

$$\Pr\{\text{no events generated}\} = \exp(-rt)$$
$$\Pr\{\text{at least 1 event generated}\} = 1 - \exp(-rt)$$

We group the sources into classes according to intrinsic rates, and say that all sources in class a have the same intrinsic rate r_a; then an instance of a class a source on a system in stress level g will have a local rate $r_a f_g$ and

$$\Pr\{\text{no event from this source in slice of length } h\} = \exp(-r_a f_g h) \tag{4}$$

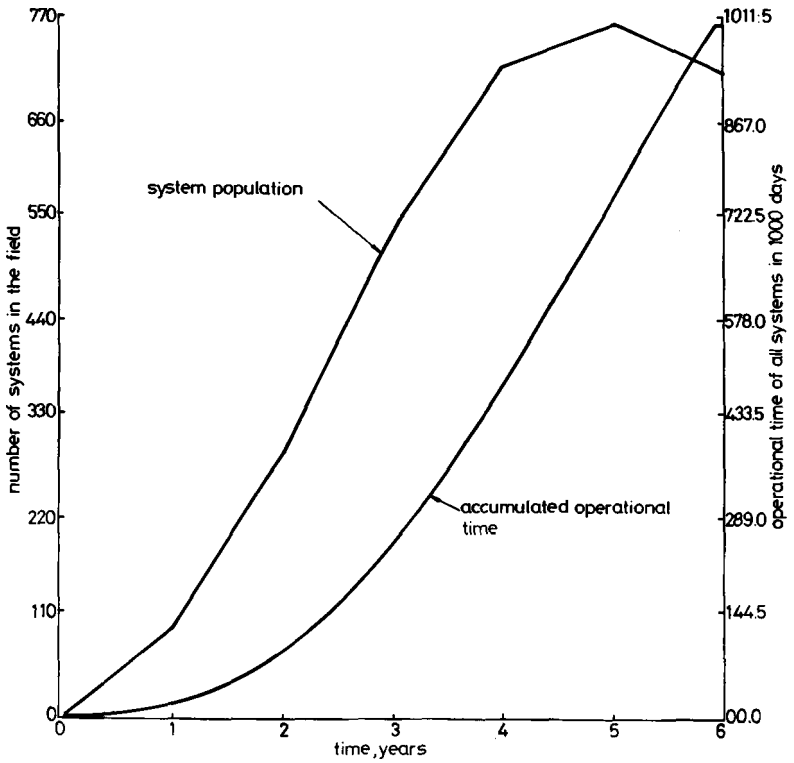


Fig. 1 System population and accumulated operational time

Consider now the manifestation of a source with rate r_a on *any* system in the field up to time t. The total accumulated exposure time is $Y(t)$, and so

$$\Pr\{\text{no manifestation of source in class } a \text{ on any system by time } t\} = \exp[-r_a\, Y(t)] \tag{5}$$

To be turned off by fix-on-fail, an instance of a source must manifest itself. Eqn. 4 therefore gives the probability that any instance of a class a source on a stress level g system will not be fixed as a result of manifesting itself in a given time slice. Eqn. 5 applies to all instances of a class a source and gives the probability that the source:

– is not on the known error log at time t, or
– if t is the time of the last systems maintenance file release, has not been turned off by that SMF.

2.4.2 Effect of imperfect fix and delay: There will be a delay between the manifestation of a source instance and its fix-on-fail, entry on the known error log or inclusion in the systems maintenance file. The model in its present form treats the average delay as constant, k, so that t must be replaced by $t-k$ in eqn. 5 above and $Y(t)$ taken as zero for $t<k$. In fix-on-fail delay means that the fix is applied $[k/h]$ slices further on, where as usual $[x]$ means the integral part of x. Fix-on-fail simulation is generally complicated and is treated separately below.

The case of the imperfect fix is handled by introducing the parameter

$$P_f = \Pr\{\text{any source instance is successfully fixed after manifestation}\}$$

Taking P_f as constant is equivalent to assuming:

(i) all fix attempts have the same probability of success
(ii) this probability is not affected by manifestations of other instances of the same source
(iii) only working fixes are included on the systems maintenance file

and to ignoring the facts that:

(i) later attempts to fix an instance will be more likely to succeed than the earlier ones
(ii) use of the known error log will spread this effect over all instances of the same source
(iii) once a good fix is on the known error log subsequent manifestations are fixed with certainty
(iv) bad fixes can get on to the systems maintenance file
(v) a bad fix may not merely fail to clear the original bug but may also introduce a new source.

It is part of the quality control function to ensure that we are able to ignore (iv) and (v). A support world in which these were major factors would be chaotic.

Accepting these limitations we proceed as follows (with a slight change of notation for the sake of simplicity).

If p is the probability that a fix is good (so that $q = 1-p$ is the probability that it is bad) then for a source of rate r and exposure time y

$$\Pr\{\text{source is not fixed}\}$$

$$= \Pr\{\text{source is not manifest}\}$$

$$+ \sum_{n=1}^{\infty} \Pr\{\text{source manifest } n \text{ times and is fixed badly every time}\}$$

$$= \exp(-ry) + \sum_{n=1}^{\infty} \frac{(ry)^n}{n!} \exp(-ry) q^n$$

$$= \exp(-ry) \exp(ryq) = \exp(-ryp)$$

Eqn. 4 now becomes (substituting $r_a f_g$ for r, h for y, P_f for p in the above):

$$\Pr\{\text{class } a \text{ source on stress } g \text{ system generates no fix in any slice } h\} = \exp(-r_a f_g P_f h) \quad (6)$$

If T_v is the time of the vth systems maintenance file release, then eqn. 5 becomes (substituting r_a for r, $Y(T_v - k)$ for y, P_f for p):

$$\Pr\{\text{class } a \text{ fault still active after } v\text{th SMF}\} = \exp[-r_a P_f Y(T_v - k)] \quad (7)$$

For nonfailure query sources this probability is 1 always.

The symptoms which will assist the recognition of future manifestations of a source can be on the known error log *before* a good fix is available, and so we ignore P_f when calculating the probability that a source is known. We still assume that there is a delay before the KEL entry is made, however, so that:

$$\Pr\{\text{class } a \text{ source is not on KEL at time } t\} = \exp[-r_a Y(t-k)] \quad (8)$$

If a source is not on the known error log it cannot have been dealt with either by fix-on-fail or by system maintenance file release. Eqn. 8 therefore gives the expected proportion of all class a sources which are still active and unknown.

2.4.3 Representation of differing source frequencies: We have already made the rather crude assumption that event sources can be grouped into classes, with all sources in one class having the same intrinsic rate. Given this, any chosen frequency distribution of source rates can be simulated by assigning different proportions of sources to the different classes and any overall failure rate by choosing the appropriate total number of sources. The rates in each class having been chosen, the number of sources in each class is called the *quality profile* of the product. Products with different profiles behave very differently, even though they may appear to have the same failure rate when issued.

Fig. 2 gives a theoretical example; it shows the rates of support events generated by the whole population in the field of two products with different quality profiles, on the assumption that:

- the system population is that of Fig. 1
- each product runs on every system in the field

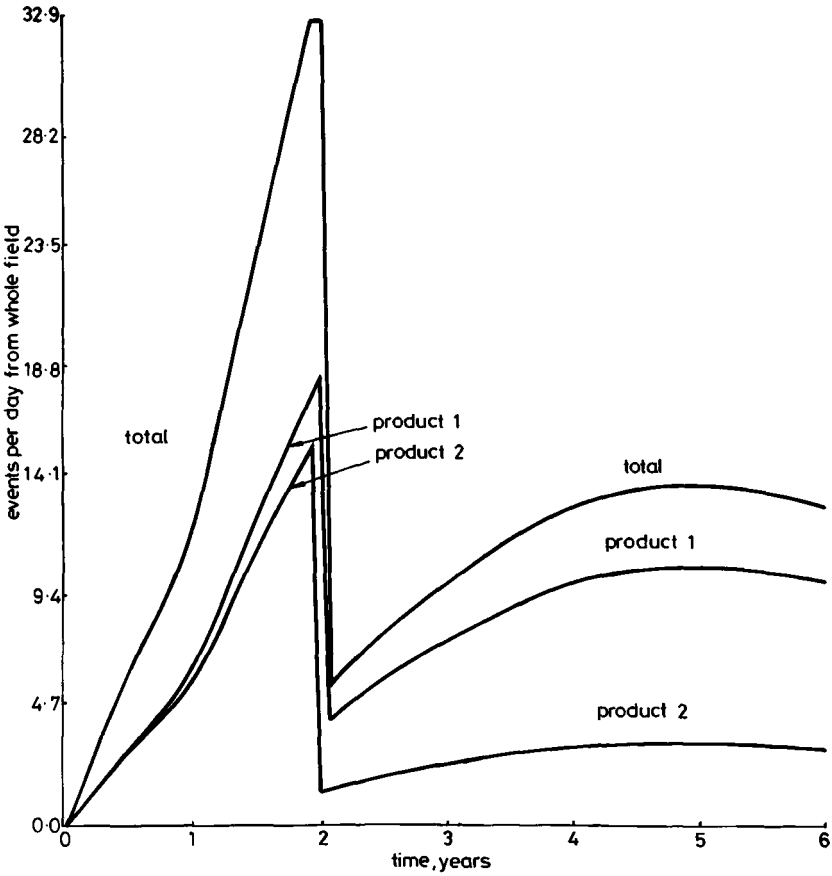


Fig. 2 *Rates of support event generation: total and for each software product*

Table 2 Rates and product profiles

Class	1	2	3	4	5
Rate (events/day)	2×10^{-3}	$6{\cdot}3 \times 10^{-4}$	2×10^{-4}	$6{\cdot}3 \times 10^{-5}$	2×10^{-5}
Product 1 (sources)	0	0	140	240	420
Product 2 (sources)	14	24	42	100	206

Class	6	7	8	9	10
Rate (events/day)	$6{\cdot}3 \times 10^{-6}$	2×10^{-6}	$6{\cdot}3 \times 10^{-7}$	2×10^{-7}	$6{\cdot}3 \times 10^{-8}$
Product 1 (sources)	1000	2060	3560	5760	6840
Product 2 (sources)	356	576	684	0	0

- a systems maintenance file is issued for each product around the end of the second year
- only product faults are considered, and possible nonfailure queries ignored.

Ten classes of event source are defined, with the rate decreasing by a constant factor

of $\sqrt{10}$ from each class to the next; the rates and the product profiles are in Table 2.

The example has been constructed so that Product 1 has 10 times the number of faults as Product 2, but these have only 1/10 the frequency; so the initial failure rates are identical. Since the numbers of copies of the two products are always the same, the differences in the numbers of support events shown in Fig. 2 can arise only from the differences between the two profiles. The following points shown by Fig. 2 are worth noting.

- The initial rate is zero, since the initial population is zero, and increases rapidly as the population grows.
- Early rates from the two products are almost identical. The fewer, more frequent, sources in Product 2 are cleared more rapidly by fix-on-fail and its rate declines a little more rapidly, relative to population, than Product 1.
- Overall effect of fix-on-fail is slight; when it is the sole mechanism in operation the rate follows changes in population almost exactly.
- The effect of systems maintenance file release is dramatic, and much greater for Product 2 than for Product 1. Bearing in mind that the effect of SMF is to remove from the whole population all known faults at the time of issue, this shows that far more are known for Product 2 than for 1.

The graphs have been smoothed slightly from the original printer output; there would be some more smoothing in a real-life case because the SMF release is not applied to all systems simultaneously.

Fig. 3 shows the accumulating life-cycle cost of supporting each product, on the assumptions that a known fault costs 0·2 engineer days per event and a new fault 10. The significant points to note here are:

- since proportionately more of the events from Product 2 are closed as 'known error', and these cost much less than 'new error', the difference in cost is even greater than the difference in event rate. The two products have the same initial failure rate but after 6 years Product 1 has cost more than four times as much to support as Product 2.
- since SMF release affects known sources, it has less effect on the cost rate than it has on the event rate. There is in fact a kink in the graph at 2 years, but this is barely discernible.

Figs. 4, 5, 6 illustrate the effects of differing quality profiles in a different way. These are based on the assumptions that the field population is constant at 100 systems and that the only support mechanism in force is fix-on-fail. The graphs show total product failures, those closed as 'known error' and those closed as 'new error'. Six source classes are assumed, with rates and profiles as in Table 3.

Significant points here are:

- total event rate decreases by half during the six years with Profile 1, decreases slightly with Profile 2 and stays virtually constant with Profile 3

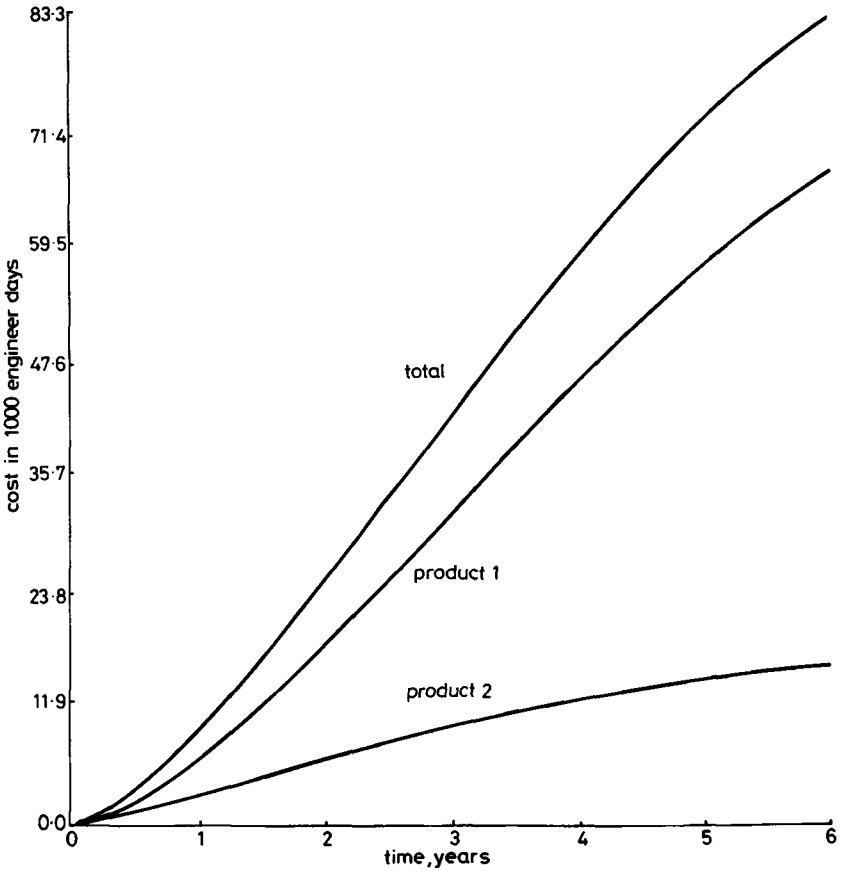


Fig. 3 Accumulating life-cycle cost of support: total and for each software product

– with Profile 1 new faults decrease to 10% of the total in 1 year, whereas with Profile 2 they are still more than 20% of total, and with Profile 3 more than 50%, after 6 years.

This example was constructed artificially for purposes of illustration. The following paragraph is intended to give some idea of what quality profile to expect in real life and to introduce the concept of a continuous distribution of probability of source frequency.

2.4.4 Exponential frequency distribution: Consider the application of eqn. 7 to the period of testing the product before it is released to the field. This is equivalent to a systems maintenance file release at $t = 0$ but with $Y(0) = H$, where H is the total exposure of the product during the testing period. The equation shows that all but the very infrequent faults will be removed with virtual certainty before the release. This conclusion is supported by such observations as have been made, which suggest that frequencies of individual sources in the field are in fact very low, and that high failure rates are caused by there being very many, very infrequent sources. (A bug

Table 3 Rates of source classes and profiles

Class	1	2	3	4	5	6
Rate (events/day)	6×10^{-4}	6×10^{-5}	6×10^{-6}	6×10^{-7}	6×10^{-8}	6×10^{-9}
Profile 1 (Fig. 4)	100	400	1600	4000	0	0
Profile 2 (Fig. 5)	0	1000	4000	16000	40000	0
Profile 3 (Fig. 6)	0	0	10000	40000	160000	400000

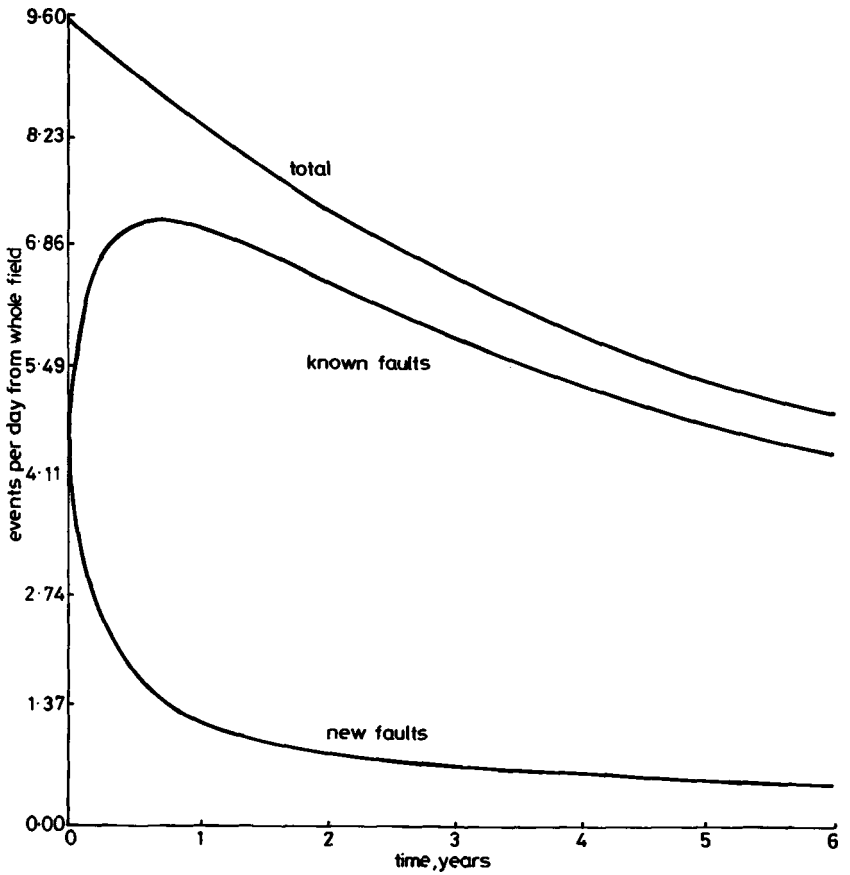


Fig. 4 Known, new and total event rates. Quality profile 1

with a frequency of 1 event in 10^6 years has been reported on a widely used operating system which had clocked up that order of operational time.)

To quantify this qualitative judgement, consider a continuous distribution of failure rate. Writing as usual Pr $\{ x/y \}$ for the probability of x, given y, and pdf for probability density function, let $p_0(r)$ be the prior pdf for r at $t = 0$ and $p_H(r)$ the posterior pdf at $t = H$. Then for a single source at $t = H$, ignoring delays and other effects,

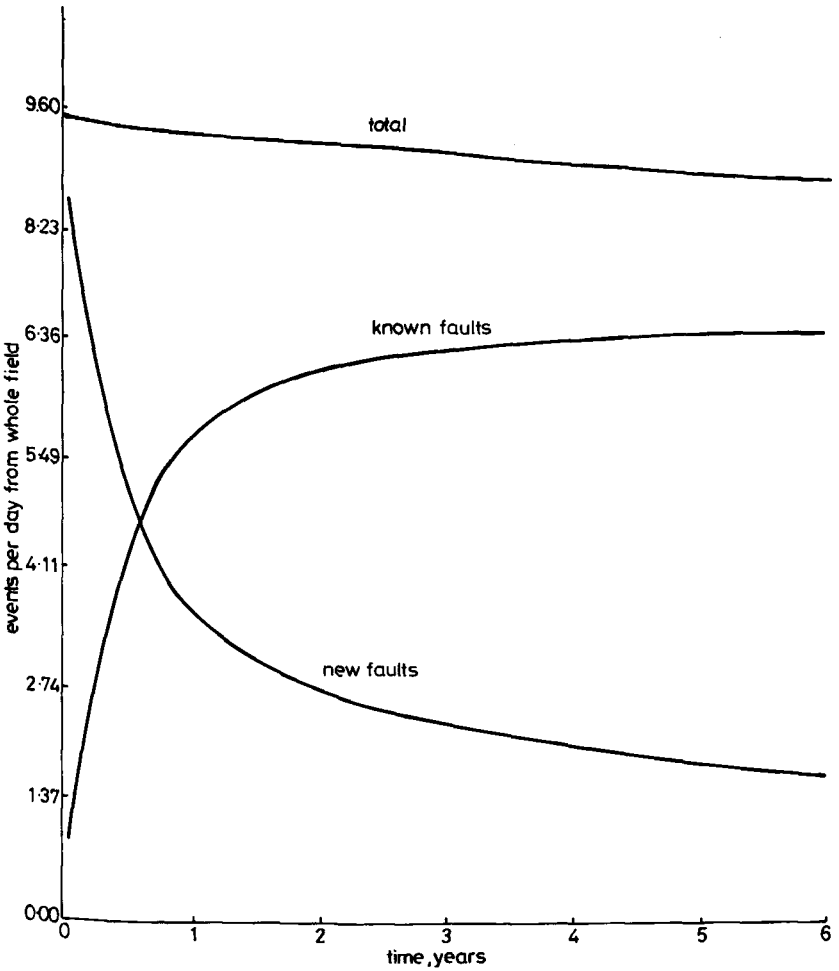


Fig. 5 Known, new and total event rates. Quality profile 2

$$p_H(r) = \text{pdf (rate of source} = r \mid \text{source not manifest during period } H)$$

$$= \frac{\Pr\{\text{source not manifest during } H \mid \text{rate} = r\}\, p_0(r)}{\int_0^\infty \Pr\{\text{source not manifest during } H \mid \text{rate} = r\}\, p_0(r)\, dr}$$

by Bayes Theorem

$$= \exp(-Hr)\, p_0(r) \Big/ \int_0^\infty \exp(-Hr)\, p_0(r)\, dr$$

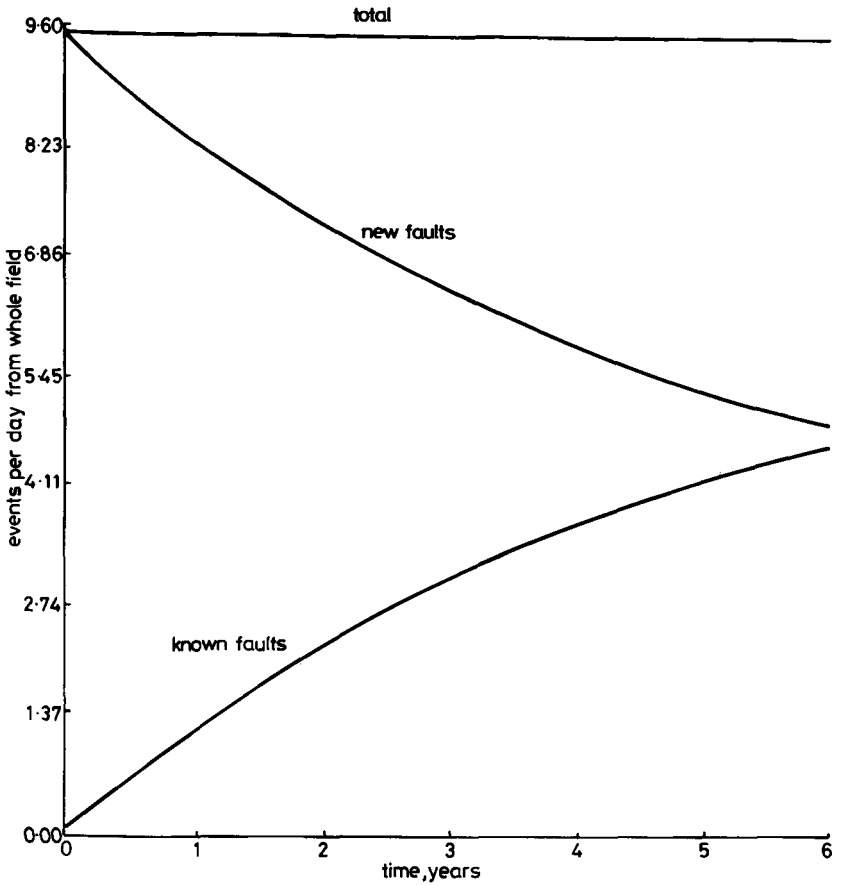


Fig. 6 Known, new and total event rates. Quality profile 3

If we now assume $p_0(r) = 1$ (an improper uniform distribution) it follows that $p_H(r) = H \exp(-Hr)$, an exponential distribution with expectation $1/H$. This is intuitively appealing since we should expect a fault with a rate less than $1/H$ to show up less than once during the exposure period H, and one with rate greater than $1/H$ to show up at least once.

I am indebted to Dr. B. Littlewood for this proof, which corrects and generalises an earlier attempt to justify exponential distribution of frequencies.

Littlewood criticises the assumption of the improper prior distribution, and also points out that the exponential is a special case of the gamma distribution considered in his 1981 paper[2]. Gamma distributions form a two-parameter family, each being defined by a 'shape' parameter and a 'scale' parameter, the latter defining the mean of the distribution. The exponential distribution is a gamma distribution with the shape parameter 1. A more rigorous treatment would be to assume a gamma distribution and use observed failure rates to estimate the parameters; the resulting

distribution could then be used to predict future reliability. It is likely that different distributions would be needed for product failures and nonfailure queries, respectively.

To investigate the behaviour of a certain type of 'uniform' distribution of fault frequency, a run of the model was done with 1000 faults in each of 10 classes. The rates were in the ratio $\sqrt{10}$ between adjacent classes. The number of sources still active in each class after a 10-year run on one system is given in Table 4.

Table 4 Remaining sources in each class after 10 years running time

Class	1	2	3	4	5
Rate	3×10^{-2}	$9{\cdot}487 \times 10^{-3}$	3×10^{-3}	$9{\cdot}487 \times 10^{-4}$	3×10^{-4}
Sources	0	0	0·02	31	334
Class	6	7	8	9	10
Rate	$9{\cdot}487 \times 10^{-5}$	3×10^{-5}	$9{\cdot}487 \times 10^{-6}$	3×10^{-6}	$9{\cdot}487 \times 10^{-7}$
Sources	707	896	965	989	996

2.4.5 Simulating fix-on-fail: Since this applies to individual source instances spread over a changing population its simulation is complex. First, the number of fixes generated in any slice is calculated for each instance of a class a source on a stress level g system; this is then added to the fixes to be applied $[k/h]$ slices later, carrying forward the fixes from the previous slice and allowing for withdrawals from the field and expiries of warranties.

Further manifestations of the same instance in later slices, but before the 'target' slice for application of the fix, must not generate fixes. Finally, if T_m is the time of the mth systems maintenance file release all fixes generated prior to $T_m - k$ are deducted from the accumulating total; and the number of active instances at any time is obtained by subtracting the number of fixes from the number of instances, treating instances from each class at each stress level separately.

Let $A_{ag}(t)$ and $F_{ag}(t)$ be the number of active instances remaining, and the number of fixes to be applied, respectively, for class a sources on level g systems at time t

T_m = time of most recent (mth) systems maintenance file release, or zero if none has been made

$P_a(t)$ = Pr {class a source still active after last SMF}

= 1 if there has been no SMF release, or if the source class consists of nonfailure query sources (see eqn. 7)

$$\text{then } F_{ag}(t) = \sum_{z=T_m-k}^{t-k} A_{ag}(z)\, P_s(z,t)[1 - \exp(-r_a f_g P_f h)] \qquad (9)$$

where the summation is slice-by-slice

and $A_{ag}(t) = M_{cg}(t)\, P_a(t) - F_{ag}(t)$ (10)

2.4.6 Expected rates and accumulation of support events: If $R(t)$ and $E(t)$ are the expectations of the total event rate from the field and the total accumulation, respectively, then

$$R(t) = \sum_a \sum_g n_a\, r_a\, f_g\, A_{ag}(t) \tag{11}$$

$$E(t) = \int_0^t R(u)du = \sum_{z=0}^{t} R(z)h \text{ (approximately)} \tag{12}$$

where n_a is the number of class a sources in the product, and, as before, the summation is slice-by-slice.

The expectation of the rate of generation of new errors is $R_N(t)$ where, from eqn. 8,

$$R_N(t) = \sum_g f_g\, M_{cg}(t) \sum_a n_a r_a \exp\left[-r_a\, Y(t-k)\right] \tag{13}$$

and the expectation of the rate for known errors $R_k(t)$ is

$$R_k(t) = R(t) - R_N(t) \tag{14}$$

2.5 Calculation of costs

The present version of the model treats the calculation of costs very crudely: a constant average cost in engineer days per event is assumed, with different values for each of four event types, the split being product failure and nonfailure queries, and known/new sources within each. Different probabilities of closure in each of the four main categories described in Section 1.4 (these probabilities depending on event type) and the treatment of the resulting costs must await a later version of the model.

A further refinement which should be included is that the calculation must take into account the probability of not recognising a known fault from the entry in the known error log. That this is significant is shown by the large number of reports closed eventually as 'known error' at the top level of the support hierarchy. With perfect recognition we should expect all known faults to be closed at the lowest level, or even not to appear at all since in practice the customer has access to the known error log and could be expected to recognise these himself; there would then be no cost to the support organisation other than that of providing the log.

An alternative approach would be to treat cost as a random variable with a given distribution, and log-normal has been suggested as a starting hypothesis. This leads to studies of the sums of random numbers of random variables; the mean and variance of the distribution of total cost resulting from this approach can be found without too much difficulty.

2.6 Input to the model

The maintenance database is the main source of product reliability data, together with the monthly returns from the field collected by the product service evaluation organisation. In particular, failure during validation of a product is recorded in the database. By matching bug reports to entries in the known error log it is possible to estimate the frequency of manifestation of each known fault, correcting for suppression of some known errors before they reach the database as mentioned in Section 2.5. Proportions of fault reports closed in various categories and at various support levels can be estimated from statistics provided by the software support centres.

Information on costs is more of a problem, telephone queries in particular being notoriously difficult to cost. The best sources are the audits carried out by the first and second lines of support.

3 Comments and conclusions

3.1 Failure rate and frequency distribution of sources

The same instantaneous failure rate could result from any number of different frequency distributions over an appropriate number of sources. The behaviour of the product when exposed in the field and subject to the improving effect of support mechanisms will, however, be completely different for each distribution. This is illustrated by Figs. 2-6.

Basically, the presence of many infrequent sources results in slow growth in reliability, whereas a few frequent sources lead to rapid growth. What is observed in practice is closer to the former than the latter.

3.2 Implications for software-reliability measurement

When software is being validated, targets for reliability have to be set in order to ensure that a specified support cost is not exceeded after the product has been released to the field. It is obviously insufficient to specify either a simple failure rate or MTBF (mean time between failures), or a maximum bug content – even if this latter could be estimated during validation. In fact, software reliability can be fully specified only by using some conceptual model of the process of generation of events; the model must take into account the distribution of source frequencies and targets must be specified in terms of the model.

As an example of how this might work in practice, consider the special case of the

exponential distribution of frequencies discussed in Section 2.4.3. The mean was found there to be $1/H$, where H is a measure of the total exposure of the product so far. This will not generally be known when the product arrives in validation because up to that time it will have lived in a fairly uncontrolled environment. However, the total exposure V during validation can be measured and the problem is to use this and other measurements to estimate H and the total number of faults or bugs in the product N. On the exponential assumption, future behaviour can then be predicted.

We assume that faults are removed as they are found during the validation. R_1 and R_2, the failure rates at the start and end, respectively, of the validation process, must be measured.

One possible way to estimate these is to use a graphical technique described by Campbell and Ott[13], and adapted by Dale and Harris[4]. The accumulated number of faults found is plotted against time and what is called the least convex majorant is drawn: this is the 'smallest' smooth convex curve lying above all the points. The slope of the tangent at any point is an estimate of the instantaneous failure rate at the corresponding time.

Given V, R_1 and R_2 the calculation of H and N is as follows.

The expected rate at the start is $R_1 = N/H$. (15)

The expected number of faults found during validation

$$= N \Pr \{ \text{fault manifests itself during period } V \}$$

$$= N \int_0^\infty \text{pdf}(r) \Pr \{ \text{fault manifests during } V \mid \text{rate} = r \}\, dr$$

$$= N \int_0^\infty H \exp(-Hr)\, [1 - \exp(-Vr)]\, dr$$

$$= NV/(H + V)$$

The expected number of faults remaining after validation is therefore

$$N - NV/(H + V) = NH/(H + V)$$

Since the total exposure time is now $H + V$ the expected rate of each bug left is $1/(H + V)$, and so the total rate after validation is

$$R_2 = NH/(H + V)^2 \qquad (16)$$

Substituting $H = N/R_1$ from eqn. 15 we obtain the equation for N:*

$$(R_1 - R_2)N^2 - 2R_1 R_2 N - R_1^2 R_2 V^2 = 0$$

from which, taking the positive root

$$N = VR_1 [R_2 + \sqrt{(R_1 R_2)}]/(R_1 - R_2)$$

and hence H from eqn. 15.

This derivation is based on the assumption of the exponential distribution; for a more general gamma distribution there would be a shape parameter also to be estimated.

Use of a model in this way makes it possible to define reliability targets in meaningful terms. In this example the validation could continue until the expected number of faults remaining was such as to guarantee that support costs would stay within prescribed limits. Without a model, development and validation teams have no way of relating their efforts to product quality and support cost.

3.3 Effectiveness of support mechanisms

It is evident that the two mechanisms that rely on input from the whole field, systems maintenance file and known error log, are vastly more effective than fix-on-fail, which affects only individual sites. In fact, SMF release is so effective that if well timed it should be possible by this mechanism to reduce the flow of fault reports concerning any widely-dispersed stable product virtually to zero, given a well managed support effort. Unfortunately, it is a fact that complex software products such as operating systems must evolve to remain useful, and it is the necessary continual updating that defeats the improvement process.

Fig. 7 illustrates the effect of product reissue. It shows graphs of event rate for two products, in this case with identical profiles. Both are subject to SMF releases after 2 years and 4 years, but only Product 2 is reissued. The assumption is that reissue introduces a certain percentage of new code (in this case 20) with the same quality profile as the original, into the field.

The two big drops in event rate are due to SMF release. The fairly large increase is due to reissue.

The quality profile for both products is given in Table 5. Note that this is similar to the profile obtained by running the model for a time on an initially 'uniform' distribution over similar classes.

The population is the same for Figs. 1 and 2, but the lower profile means that the

* The convexity of the curve fitted to the observations guarantees that $R_1 > R_2$. If a convex curve could not be fitted, the failure rate would not be decreasing during the validation and the process would be meaningless.

Table 5 Quality profile for products in Fig. 7

Class	1	2	3	4
Rate	3×10^{-3}	$9{\cdot}487 \times 10^{-4}$	3×10^{-4}	$9{\cdot}487 \times 10^{-5}$
Sources	1	3	33	70
Class	5	6	7	8
Rate	3×10^{-5}	$9{\cdot}487 \times 10^{-6}$	3×10^{-6}	$9{\cdot}487 \times 10^{-7}$
Sources	90	96	99	100

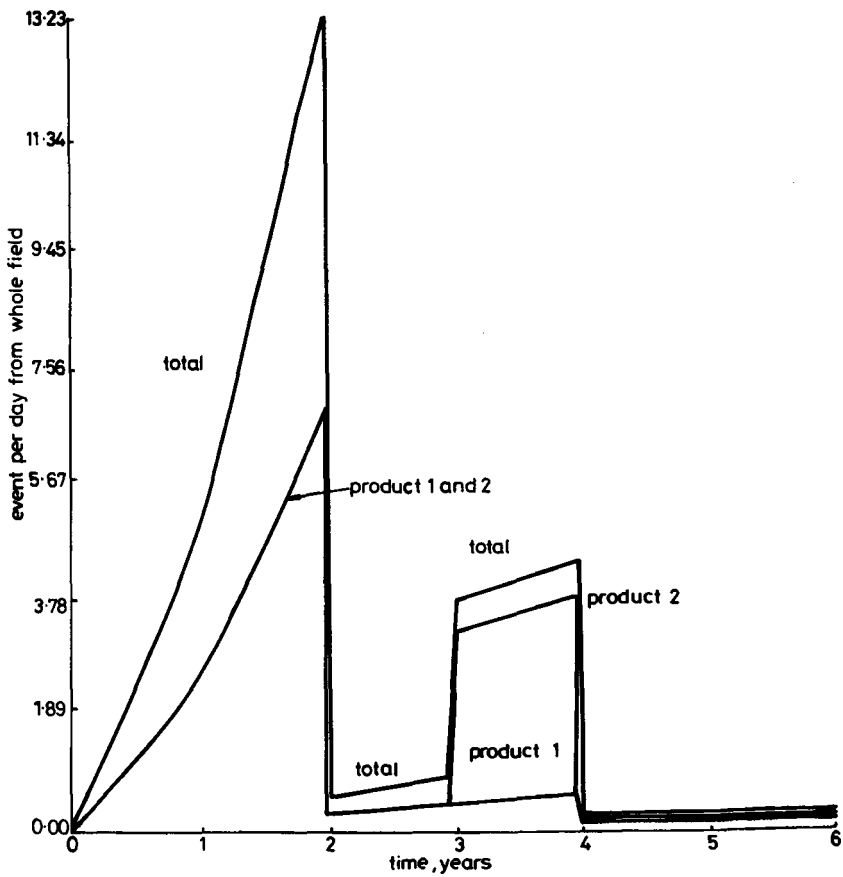


Fig. 7 Event rates: total and for each product. Both products have SMF at years 2 and 4. Only one has 20% reissue at year 3

highest rate attained before the first SMF is less than half that in Fig. 2.

An important consequence of the effectiveness of the SMF mechanism is that the event rate will be relatively very high in the early part of the life of any product.

This will cause problems in providing support resources, for example, diagnosticians, and the consequential queueing effects will become important in these early stages. Nonfailure queries, on the other hand, are dealt with only by fix-on-fail types of mechanism, so a steady workload, more or less proportional to the population in the field, must be expected.

The issue of SMF releases and the maintenance of the known error log are important overheads on the support operation and must be included in the life-cycle costs of the products.

3.4 Some shortcomings of the present model

1 The basic hypothesis that software failure can be described by a discrete set of independent, removable Poisson sources is assumed almost universally in software reliability modelling. It should not be forgotten that this is only a hypothesis, to be modified in the light of experience. It is common experience, for example, among software writers and supporters that faults are sometimes not independent.
2 The treatment of sources as belonging to discrete categories is at best a poor approximation to the use of a realistic frequency distribution.
3 Stress factor is a blunderbuss weapon for attacking the problem of customer effects. The effects are not usually uniform over all sources, the difference being particularly marked between validation and user environments.* Stress is difficult to measure using historical data and even more difficult to predict. Littlewood suggests the use of 'explanatory variables' instead.
4 Effects of queues are omitted, and it is inaccurate to treat the delay in applying a fix as constant.
5 Introduction of new faults into the product by bad fixes is not modelled.
6 Two different uses of the known error log are not properly distinguished. These are
   – to recognise a fault even before a fix is available
   – to recognise a fault and extract the relevant fix from the database.
7 It does not take into account the structure of a complex piece of software.

4 Possible developments of the model

The following suggested developments would redress some of the shortcomings listed.

1 Treat the source frequencies as having a continuous distribution. This will simplify the model (as well as making it more realistic).
2 Treat similarly the delay in applying a fix. The probability that a source is fixed within a given time is then the sum of two random variables, 'time to manifestation' and 'time to fix'; its distribution is the convolution of the distributions of these.
3 Treat the distribution of costs by the method mentioned in Section 2.5, the

* This is expressed in 'Sparrow's Law': the fault that is so unlikely that it is not expected to manifest itself in the whole life cycle of the product always appears twice on the first day of issue.

'sum of a random number of random variables'.

4 The model deals with the effect of the support mechanisms on the flow of events. It should deal also with the reverse effect, which can manifest itself as queues forming in the support organisation and causing increased delays in applying fixes. This is not amenable to classical queuing theory which provides a manageable analytic solution only for the steady state of a system, because here there is no steady state. The time-slice approach should allow an adequate simulation, given an input defining the available support resources.

5 Represent priority. At present the ICL organisation assigns priorities in five categories reflecting the cost of the failure to the customer. This will complicate the queuing effect referred to in (4) above and will require the distribution of the cost of failure to be modelled.

Acknowledgements

I would like to thank Dr. B. Littlewood of City University for his help and advice, particularly on the software-failure aspect; Dr. Barbara Kitchenham of ICL's Mainframe Systems Development Division, Kidsgrove, for encouragement and help; J.R. Sparrow, until recently on 3rd-line support in ICL's Distributed Systems Development Division and now the proprietor of the company he calls 'diggers', for his wit and wisdom on the subject of support, which are now sadly missed; and E.D. Link, Manager, Service Operations and Methods, and other colleagues in Customer Support & Quality Division, for not allowing me to forget the urgent practicalities of support costing.

References

1 LITTLEWOOD, B.: 'How to measure software reliability and how not to', *IEEE Trans.*, 1979, R-28, 103-110.

2 LITTLEWOOD, B.: 'Stochastic reliability growth: a model for fault-removal in computer-programs and hardware designs', *IEEE Trans.* 1981, R-30, 313-320.

3 KEILLER, P.A., LITTLEWOOD, B., MILLER, D.R. and SOFER, A.: 'On the quality of software reliability prediction', Proc. NATO Advanced Study Institute of Electronic Systems Effectiveness and Life Cycle Costing, 19-31 July 1982, Springer-Verlag

4 DALE, C.J. and HARRIS, L.N.: 'Approaches to software reliability prediction', British Aerospace plc, Dynamics Group, Stevenage Division. 1982 Proceedings Annual Reliability and Maintainability Symposium.

5 DALE, C.J. and HARRIS, L.N.: 'Reliability aspects of microprocessor systems'. BAe Report ST 25358, 1981. Available from Department of Industry, report T816164.

6 DALE, C.J. and HARRIS, L.N.: 'Software reliability evaluation methods'. BAe Report ST26750, Sept. 1982.

7 KITCHENHAM, B.A.: 'Measures of programming complexity', *ICL Tech. J.*, 1981, 2, (3), 298-316.

8 ANDERSON, T. and LEE, P.A.: *'Fault tolerance – principles and practice'*, Prentice-Hall International, 1981.

9 JELINSKI, Z. and MORANDA, P.B.: 'Software reliability research. *Statistical computer performance evaluation'*, Academic Press, New York, 1972.

10 DRURY, M., STRASS, P., BOWMER, R.A. and GODDING, J.F.: 'A model for software failure rates', ICL Quality & Statistics internal report CQA/A/R293. 28/6/76.

11 DRURY, M., STRASS, P., BOWMER, R.A. and GODDING, J.F.: 'Effects of workload and processing power on software failure rates', ICL Quality & Statistics internal report CQA/A/R.294. 28/6/76.

12 DRURY, M., STRASS, P., BOWMER, R.A. and GODDING, J.F.: 'A prediction of VME/B failure rates. ICL Quality & Statistics internal report CQA/A/R.295. 28/6/76.
13 CAMPBELL, G. and OTT, K.O.: 'Statistical evaluation of major human errors during the development of new technological systems', *Nuclear Sci. & Eng.*, 1979, **71**, 267-279.

Appendix 1

Nonstationary Poisson process

Consider an accumulating discrete process, such as the number of bug reports received during the life-cycle of a product subject to 'random' failures.

Assume:

1 Events are independent

2 The average rate at which events occur varies with time, but over any sufficiently short interval δt the probability of an event occurring is $f\delta t$ where f is in general a function of t.

It follows from the assumption of independence that the probability of more than one event occurring in δt is of order $(f\delta t)^2$, and that therefore the probability that no event occurs in δt is $1-f\delta t$, to order $f\delta t$

Let $P_n = P_n(t) = \Pr\{n \text{ events have occurred up to time } t\}$

Then $P_n(t+\delta t) = \Pr\{n \text{ events have occurred up to time } t+\delta t\}$
$+ \Pr\{n \text{ events up to } t, 0 \text{ events between } t, t+\delta t\}$
$+ \Pr\{n-1 \text{ events up to } t, 1 \text{ event between } t, t+\delta t\}$
$+ \Pr\{n-2 \text{ events up to } t, 2 \text{ events between } t, t+\delta t\}$
etc.

$= P_n(t)(1 - f\delta t) + fP_{n-1}(t)\,\delta t + \text{terms of order } (\delta t)^2$

i.e. $P_n(t+\delta t) - fP_n(t) = fP_{n-1}(t)\delta t$

Dividing by δt and letting δt tend to 0 gives the differential equation

$$\frac{d}{dt}P_n + fP_n = fP_{n-1} \tag{A1}$$

If the counting of the events starts at $t = 0$, the initial conditions are $P_0(0) = 1$, $P_n(0) = 0$ for all $n > 0$.

The equation for $P_0(t)$ is

$P_0(t+\delta t) = \Pr\{0 \text{ events up to } t, 0 \text{ events between } t, t+\delta t\}$
$= P_0(t)(1 - f\delta t)$ leading to

$P_0' + fP_0 = 0$ (writing P' for dP/dt), with $P_0(0) = 1$ (A2)

the solution of which is

$$P_0(t) = \exp[-F(t)] \text{ where } F(t) = \int_0^t f(u)du \tag{A3}$$

With this definition of $F(t)$ the general equation, eqn. A1, can be written

$$\frac{d}{dt}(e^F P_n) = f(t)(e^F P_{n-1})$$

i.e. $Q'_n = fQ_{n-1}$ where $Q_n = e^F P_n$ (A4)

Since $Q_0(0) = P_0(0) = 1$ and $Q_n(0) = P_n(0) = 0$ for $n > 0$

$$Q_0(t) = e^{F(t)} P_0(t) = 1$$

and for all $n \geqslant 1$,

$$Q_n(t) = \int_0^t f(u)\, Q_{n-1}(u)\, du = \int_0^t Q_{n-1}(u)\, dF(u) \tag{A5}$$

Hence

$$Q_1(t) = \int_0^t dF = F(t)$$

$$Q_2(t) = \int_0^t FdF = F^2/2$$

$$Q_3(t) = \int_0^t F^2 dF = F^3/3!$$

and in general $Q_n(t) = F^n/n!$ (A6)

giving the general solution for P_n

$$P_n(t) = \frac{1}{n!}[F(t)]^n \exp[-F(t)] \tag{A7}$$

Putting f = constant, k say, gives $F(t) = kt$ and the ordinary stationary Poisson distribution $(kt)^n \exp(-kt)/n!$, for which the mean and variance are both kt. In the general form (eqn. A7), $F(t)$ replaces kt, which is the result quoted in Section 2.2.

Appendix 2

Comparison of output from the event generation part of the model with actual product data

Objectives

(i) To see if the model is capable of simulating observed product behaviour (in this case the manifestation of new faults).
(ii) To see if the model will predict later product behaviour taking as input only data from an early period.

Method

A particular version of a suitable product was selected. All entries in the known error log relating to it were extracted and the dates of entry recorded. Entries were divided into product faults and nonfailure query sources. To see if the simulation would work for a subset of the total sources, a separate record was kept of those product faults in a particular identifiable subset. Since the accumulated running time for the product at each date in the chosen period was known, it was then possible to plot graphs of the accumulated number of faults found against total running time for the product. The method of Section 3.2 could then be applied to estimate the total number of sources, and the expected frequency of an individual source, assuming an exponential distribution of frequencies. Note that although Section 3.2 describes the debugging of a single copy of the product, the method applies equally to the multicopy case provided total running time is used and only the first occurrence of each fault is recorded.

Having so obtained the parameters defining the exponential distribution, it was necessary to define source classes to represent the situation. The procedure was as follows:

Probability function for the distribution is $\Pr\{\text{rate of source} \leqslant r\} = 1 - \exp(-Hr)$

where H, the inverse of the mean, has been estimated from the graph.

The probability distribution for source frequencies was divided into ten equal segments, i.e. the probability of the frequency being in any segment is 0·1, by defining boundary frequencies

$$\frac{1}{H} \log \frac{10}{10-i}, \text{ where } i = 1,2,\ldots,9$$

It can be shown that the mean of any segment (a, b) of an exponential distribution is

$$1/H + \frac{a \exp(-Ha) - b \exp(-Hb)}{\exp(-Ha) - \exp(-Hb)}$$

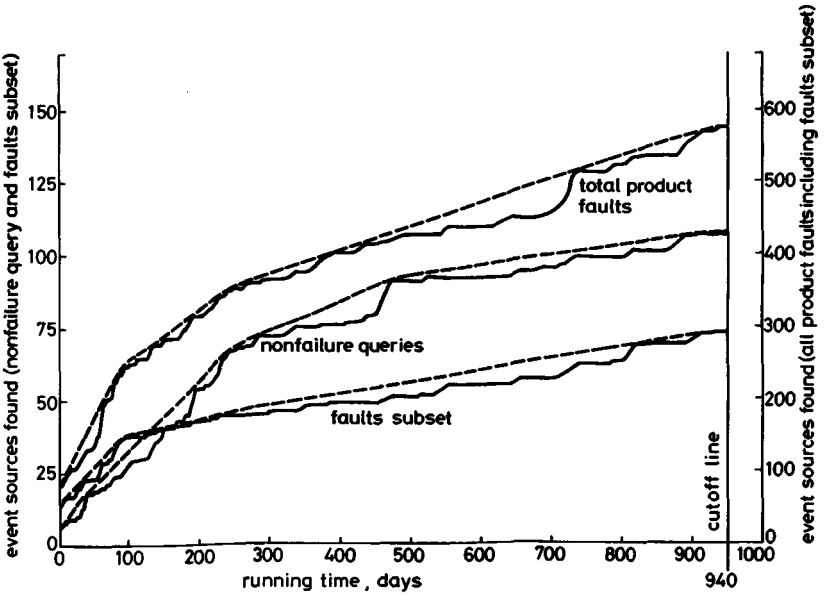


Fig. 8 Accumulated faults found (in three categories) plotted against running time

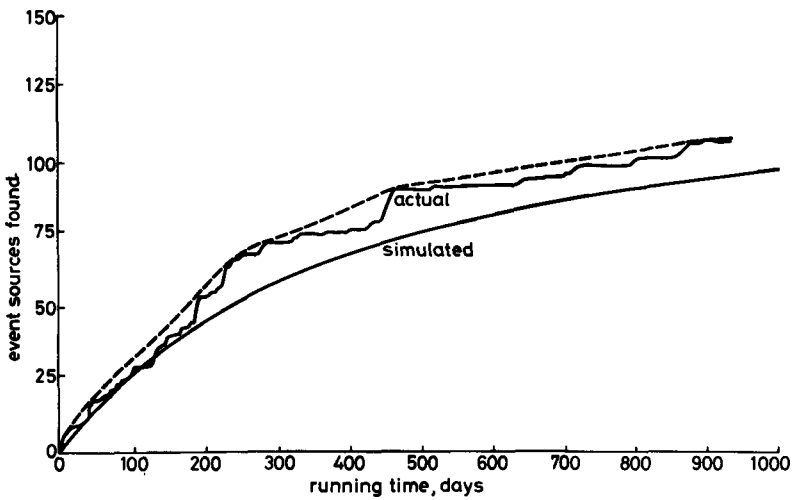


Fig. 9 Simulation of accumulation of known faults: nonfailure queries

and from this was found the mean of each of the ten equal segments. The resulting values were taken as the rates of ten source classes each containing 1/10th of the total sources.

Using this quality profile and a constant population of one system the model was run for a suitable time period (1000 days). The accumulated number of new failures generated by the simulation was then plotted against time to the same scale

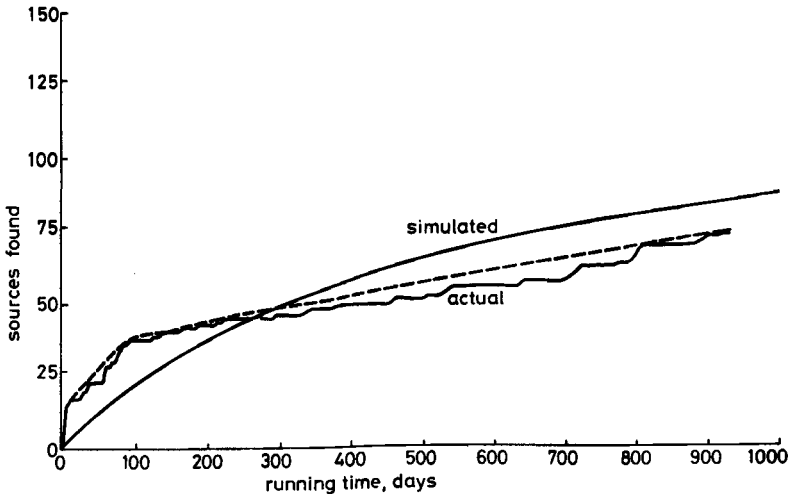


Fig. 10 Simulation of accumulation of known faults: subset faults

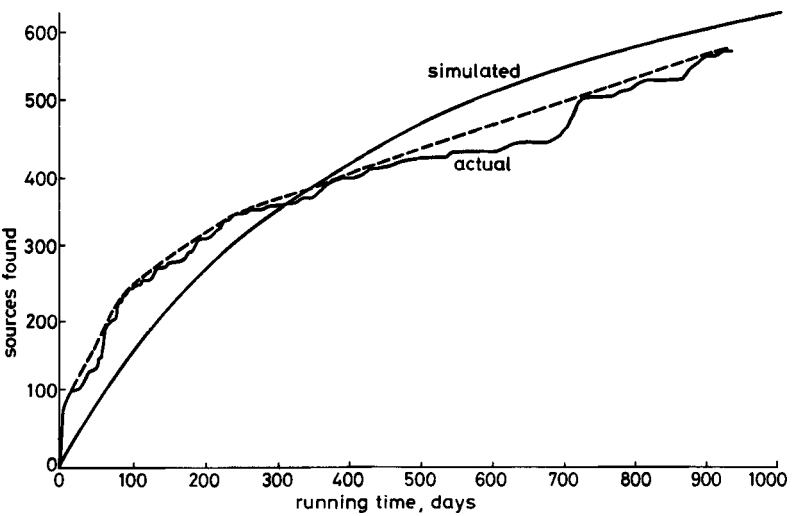


Fig. 11 Simulation of accumulation of known faults: total product faults

as the original graph of actual data, and simulation and actual experience were compared.

The above procedure was followed for each of the three categories of fault, estimating the quality profile from the slope of the graph as close to the start and end of the data as possible. Finally, a run was done for the total product faults, taking the second rate from the slope of the graph at 80 days, to test the predictive ability of the model.

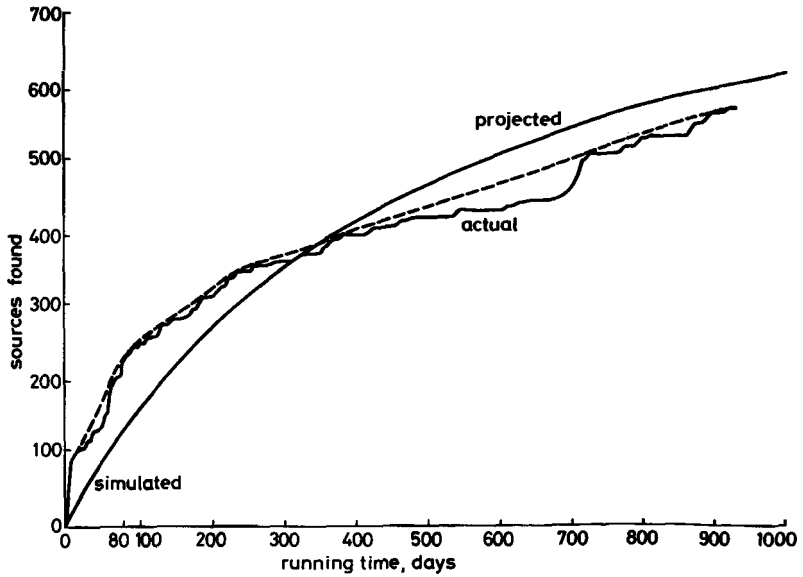


Fig. 12 Projects of accumulated known faults from first 80 days data = total product faults

Table 6 Results of estimating quality profiles

	Nonfailure query sources	Subset faults	Total product faults	Total product faults (based on first 80 days only)
Start (running days)	4	8	8	8
End (running days)	910	940	940	80
Operational time, V	906	932	932	72
Start rate, R1 (events per day)	0·33	0·26	1·92	1·92
End rate, R2 (events per day)	0·03	·032	·23	1·46
Sources at start (estimated)	140	131	947	938
Sources at end (estimated)	44	46	327	817
Sources found (estimated)	96	85	620	121
Sources found (actual)	100	58	478	140
H	419·0	503·7	493·3	488·4

Table 7 **Source classes and rates used to simulate estimated profiles (all rates x 1000)**

Source class:	1	2	3	4	5	6	7	8	9	10
Nonfailure query (14 sources per class)	7·88	4·57	3·32	2·51	1·91	1·43	1·03	0·688	0·389	0·124
Subset faults: (13·1 sources per class)	6·56	3·80	2·77	2·09	1·59	1·19	0·86	0·57	0·32	0·10
Total faults: (94·7 sources per class)	6·69	3·88	2·82	2·14	1·62	1·21	0·88	0·58	0·33	0·10
Total faults: (based on first 80 days) (93·8 sources per class)	6·76	3·92	2·85	2·16	1·64	1·23	0·88	0·59	0·33	0·11

Results

Fig. 8 shows the graphs of the actual accumulated faults. The cutoff line at day 940 is due to lack of subsequent running time information. The dotted line above each graph is the 'least concave majorant', whose slope gives the instantaneous failure rate at any time.

The event rates measured from the graphs and the calculated numbers of faults are shown in Table 6. The difference between the estimated numbers of sources at the start and end of the period considered agrees fairly well in all cases with the actual number of faults observed.

Table 7 shows the calculated quality profiles input to the simulation, and Figs. 9-12 show the simulated and actual graphs superimposed for each of the four exercises.

Conclusion

Agreement is very close between simulated and actual graphs for nonfailure queries and total product faults (Figs. 9 and 11) and not quite so good for the subset of product faults (Fig. 10). What is particularly striking is that the quality profile based on the first 80 days of data alone is almost identical to that derived from the whole 940 days, and the 'projected' graph of Fig. 12 is indistinguishable from the 'simulated' graph of Fig. 11.

This indicates that the model is capable of predicting the generation of support events from software products with reasonable accuracy, and that the assumption of an exponential distribution of source frequencies is adequate for the simulation, at least to a first approximation and for the product considered.

ICL Technical Journal

Pages contained in each issue

1 (1) 1–96
1 (2) 97–192
1 (3) 193–300
2 (1) 1–116
2 (2) 117–220
2 (3) 221–316
2 (4) 317–424
3 (1) 1–116
3 (2) 117–228
3 (3) 229–344
3 (4) 345–452

Subject index
Volumes 1-3

X *X.25*

Author index
Volumes 1-3