ICL Technical Journal

Volume 2 Issue 1

May 1980

ICL Technical Journal

Contents

Volume 2 Issue 1

Security and privacy of data held in computers
J.M.M. Pinkerton 3

CADES – software engineering in practice.
R.W. McGuffin, A.E. Elliston, B.R. Tranter and P.N. Westmacott 13

ME29 Initial program load – an exercise in defensive programming
R. Lakin 29

Project Little – an experimental ultra reliable system
J.B. Brenner, C.P. Burton, A.D.Kitto and E.C.P. Portman 47

Flow of instructions through a pipelined processor
W.F. Wood 59

Towards an 'expert' diagnostic system
T.R. Addis 79

Using Open System Interconnection standards
John Brenner 106

ICL Technical Journal

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

Editor
J. Howlett
ICL House, Putney, London SW15 1SW, England

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

Annual subscription rate: £10 (cheques should be made out to "International Computers Limited, Corporate Communication, ICL Technical Journal Office, ICL House, Putney, London SW15 1SW)

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

Security and privacy of data held in computers

J.M.M. Pinkerton
ICL Marketing Division, Slough

Abstract

Wider use of computers is leading to concern in society for the privacy of personal data held by computers. The responsibilities of users of computers to take good care of personal or other data are distinguished from those of suppliers. The need for formal rules governing who may see what data under various specific circumstances is pointed out. A simple model is described to help users and computer managers analyse their particular requirements and thereby arrive at an appropriate set of house rules, it being contended that once the rules have been established it is practical and straightforward to give effect to those rules using facilities provided by the supplier.

1 Introduction

The subjects of computer security and especially of the privacy of personal data are of growing importance and public concern. Personal privacy is the subject of legislation in many countries already and will presumably be in the U.K. too eventually.[1,2,3,7] There is thus a need for better understanding of how security and privacy needs can be met, what and how much a computer user has to undertake for himself and exactly what help he can expect from his supplier. It is clear already that laws in most countries lay the duty of care for the privacy of data primarily on the user.

The objectives of this paper are as follows:

- to offer some clarification of the terms used in discussing this subject;
- to expose the limitations of what any computer supplier can provide and point out at the same time the obligations on users in various circumstances, including what may become legal obligations;
- to present an elementary model of security of the data at a computer installation;
- to outline the scope of a set of house rules applying general privacy guidelines to any given installation say; and
- to show how features controlling access to data can be used to achieve privacy, as well as for maintaining security in the military sense.

It will become clear that successful maintenance of privacy of data held by computers must depend ultimately on drawing up house rules or guidelines that must be laid down either by the individual user, or more generally by some administration for all users in a given class.[5] Examples of such guidelines for privacy are still very rare, although they are common enough for security purposes. It is the author's opinion that writing such guidelines will prove to be much more difficult than has been widely recognised. Though it is of course not the job of any computer supplier to draft such rules, once they are written he could comment on their practicality. He must be prepared to offer facilities for supporting any reasonable and foreseeable scheme.

2 Definitions

All the definitions necessarily are of very wide generality.

Access to data may involve reading, writing, marking, moving, altering or destroying them; it may also involve merely knowing that a particular item or items are held at all, or whether someone else has been able to read, write or otherwise affect them, and if so, when. We need to be concerned not merely with direct individual access at any instant, but also with access by a computer program or process that may be controlled by a person or group of persons indirectly.

Data: for privacy we are usually concerned with data items about individuals, but in the case of security the data can be of almost any class or category. Data may obviously also include any program code in machine-readable form.

Privacy is a term that has largely defied successful definition by lawyers and others, although everyone feels he knows what it means. In administrative practice, privacy control means granting of the right to see, record or alter one's personal data to persons other than oneself. Thus it is a right claimed by most persons – in very varying ways and circumstances. For the most sensitive personal data outside a computer, access to knowledge is generally regulated by some code of ethics, professional or social. We believe we can trust the discretion of those who know or may be told details of, say, one's health, business conduct, private life and so on. Once data is in a computer discretion is no longer a feasible guide as to whether knowledge should be passed on or not.

It should be fully recognised that computer privacy goes beyond both security and regulating who may have access to the data. According to the law in several countries, it may now mean that the subject of the data must be able to inspect and either correct or delete items of data about himself, if he can show them to be false or irrelevant. The law may also say that certain classes of personal data may not be held at all on computers.

Security requires that the system holding the data must be to an adequate degree resistant to attempts to get access to that data on the part of those not entitled to it. Thus security may be a primary objective, as in the case of national and military secrets, or it may be secondary to the main requirement to keep information private. It is generally assumed that the intruder looking for personal data will be very much less determined than the foreign spy. Thus the level of security precautions required to support privacy in computers can often afford to be less strict than in military or high-risk commercial applications. The sophistication of the arrangements for access control to ensure privacy may on the other hand conceivably need to be greater. With the rather limited practical experience of implementing privacy guidelines anywhere to date, this must still be conjectural.

3 Risks

Risks to security and thus to privacy are represented by an extremely wide variety of things that may happen accidentally or inadvertently, or that people may do deliberately.

Security of computers and data held by them have been extensively analysed and discussed; there is now a choice of sets of guidelines to help users to judge which risks are significant.[4,6] A risk is significant where the product of the consequential loss or damage and the chance of that eventuality in, say, a year, shows the user to be exposed to a serious degree. It is difficult or impossible sometimes to assess risks to personal privacy on a strictly numerical or financial basis. Political considerations will loom much larger.[3] The result is that careful judgment has to be exercised as to how serious certain threats of disclosure might be, and how much trouble or money it will be worth expending to counter them. Whatever level of precautions are taken it has to be recognised that 100% security is unattainable. All security measures are a nuisance and an irritation to the staff involved. Equally obviously, so long as human beings are involved somewhere they may represent a greater threat than the computer itself. Since security is no greater than the strength of the weakest link in the chain, it is thus important to avoid going to extremes in countering any given risk.

Now it is suggested security risks be considered under three broad headings:

those which users themselves must tackle because no specific facilities from manufacturers are relevant;
those which users can tackle far better if manufacturers' products are suitably adapted or extended; and
those which users can only tackle effectively if the products are suitably contrived.

Some examples may help to illustrate this subdivision.

No computer manufacturer responsibility:

staff integrity or security;
precautions against theft or fire;
errors in original data before entry;
dp application open to fraud;
natural disaster or calamity.

Shared responsibility:

checking of errors in *entering* data;
checking correctness of application *code* as such;

procedures for setting up and administering access regulations;
introduction of checks on personal identity, e.g. of someone using a terminal;
issue and control of cypher keys, passwords, ID badges, etc.

Prime manufacturers' responsibility:

hardware and software resilience against both faults and/or design errors;
systems designed to resist misuse or malpractice;
mechanisms for personal identification;
encrypting and decrypting means for, e.g. data communications;
access control *means* for applying access control *regulations*.

In the case of shared responsibility there would be something a user could do even if the manufacturer provided no facility. In the last case the user can do very little without some facility from the manufacturer.

Please notice that the word responsibility is not used to mean *legal* liability. It implies merely that there are some things a manufacturer has to tackle if his customers are going readily to maintain security or to guarantee privacy.

It is apparent that the degree of choice for a user and a manufacturer varies in opposite directions. In the first case a user is completely free but the manufacturer exercises no choice. In the second, he aims to give the user a wide range of choices to suit the user's circumstances. In the third he must make quite fundamental choices because features of the system architecture are in question. This could result in the user having less choice in the way those features are exploited, though the manufacturer must aim to leave this choice as wide as he can.

4 A conceptual model of access control

It follows from the above that it is for designers of systems or operators of computer installations to decide individually what sorts of risks they need to protect themselves against and what levels of protection are appropriate and effective in their own cases. The following description of a model is offered to help users in analysing their own needs. It was originally devised to help a group of ICL Computer Users Association members visualise the situations that have to be covered by their own house rules of access, and thus to frame appropriate procedures both for coding as programs for the computer and to be obeyed directly by their own staff. It is hoped too that consideration of such a model may help computer management to avoid creating some important security loophole by leaving some possibility of access out of account, which it is only too easy to do.

The model is described in relation to Fig. 1, which shows a simple situation in which one management is responsible for determining and administering the rules, and to Fig. 2, which shows a more complex situation as, for example, might exist in a bureau. The diagram is divided into two halves by a dotted line representing some physical barrier, for example, the enclosing walls, floor and ceiling of the computer room. It is assumed that the central equipment (mainframe, store and backing store) is within this boundary, and that the various groups of people identified will, under different circumstances, be either inside or outside it.

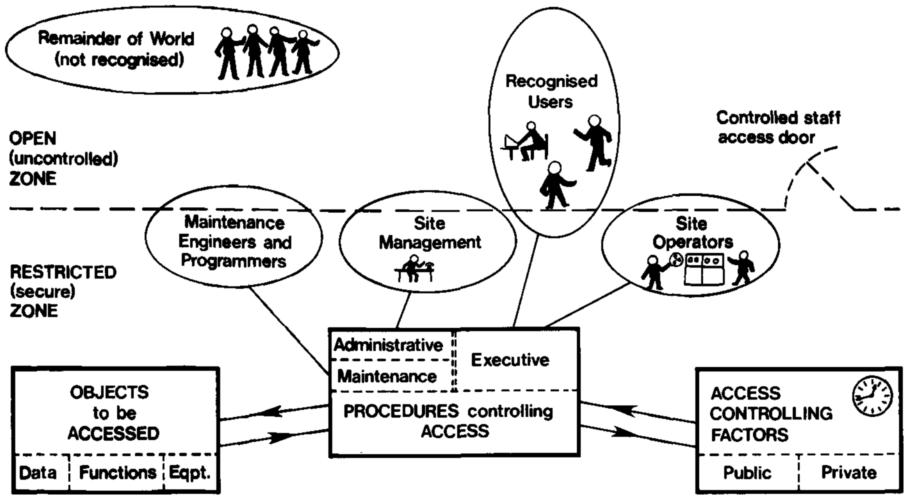


Fig. 1 Diagram of simple computer access model.

The model embraces the following components:

the physical *environment* and the *computer equipment* in it;
several classes of *person*;
three kinds of *entities stored* in the computer specifically:

(*a*) the valuable or sensitive *data* or other objects,
(*b*) programmed *procedures* embodying the *rules*,
(*c*) *controlling factors or parameters* that determine the interpretation of the rules.

Let us examine each component in turn.

It is assumed that there is physical protection of part of the computer environment against accidents, for instance fire or flood, and against deliberate penetration by intruders, i.e. the door can be opened only by say a magnetically encoded card. It is noted that some computing equipment, e.g. the terminals or at least the communication links to them, will be in uncontrolled zones where no physical security is practicable.

There are five significant *groups of persons* (ignoring designers and planners who may have been employed by the supplier, e.g. ICL). They are:

Installation management (IM) responsible for security of the installation and the data etc. held therein, and for the rules of the house.

Maintenance engineers and programmers responsible to the IM for repairing or modifying the system.

Site operators responsible for all routine operations on site, and for reporting to the IM evidence of any breach of the access rules and possibly having delegated authority for varying the rules of access in minor respects.

A *user population* for whose benefit the system is set up and operated.

The *rest of the world* who have no legitimate business with the system.

An analysis using the model to devise rules is concerned with what *authority* for access each of these group needs, how that authority is to be *delimited* and *defined* and how it may be *tested* by the system when an individual tries to gain access.

There are three sorts of entities stored, starting with the *'Objects to be accessed'*. These may be 'functions' provided by hardware directly or by software, or stored files, records or items of data. Occasionally the objects may be items of equipment. The model next assumes that access to the protected objects is to be regulated by *'Access-controlling procedures'*. These procedures are (in general) provided as software. To determine the outcome of each procedure there must also be a set of *'Access-controlling factors'* relevant to the particular house rules a user wishes to enforce and the circumstances, e.g. of the time or place. These factors may be subdivided into, 'externally known factors' (like the time, date and possibly details of some of the access-controlling rules) and 'internal or private factors'. Any genuine applicant for access knows his own identity and password and where he is at that moment; these are likely to be among the external factors. Internal factors could include, for instance, the security rating of the would-be accessor of the data, or the identity of the terminal from which access was being sought. They are said to be 'internal' because it should not be known to the accessor whether they are involved in deciding to grant access or not, or, failing that, what values they must have to ensure access is given. Thus security may be weakened if the accessor knows what these internal factors are precisely, or how they are taken into consideration.

The procedures will be broadly of three kinds:

administrative procedures for setting up and for checking routine operation of the access-controlling rules;

executive procedures for enforcing the rules laid down in day-to-day use of the system;

maintenance procedures that allow privileged persons, e.g. CED engineers, to repair and monitor the operation of the access-control mechanism – in the most general sense of those words.

It may appear to be difficult in practice to draw hard and fast boundaries between these kinds of procedure. Nevertheless to preserve strict security, every possible effort must be made to maintain the distinctions. This amounts to saying that the limits of authority granted to the various individuals and groups quoted must be well defined. The authorities ought also to be kept clearly distinct from the data or other objects to which access may be sought.

5 A more elaborate model

In many if not most practical situations the above model is too simple, chiefly because it assumes that a single management is involved, with complete authority for laying down and amending the rules, and though it has not been said, perhaps also with authority to have access to all stored data, including the authorities for access themselves and, say, the records of all actual accesses made by all users, engineers and operators. However, at many sites it is not permissible for access authorities to be based on a strict hierarchical principle.

A site management needs to be able to delegate some authority for access to independent management groups, each of whom will regulate its own groups of users in its own way. The independent (offsite) managements would in general not willingly give up their powers of regulating access by their own staffs. If they were forced to do so they would want to remove, or at worst destroy, their private data held on the system. This situation is now represented by Fig. 2. In this version of the model the direct users fall into separate groups each with its own management to regulate access for persons within that group.

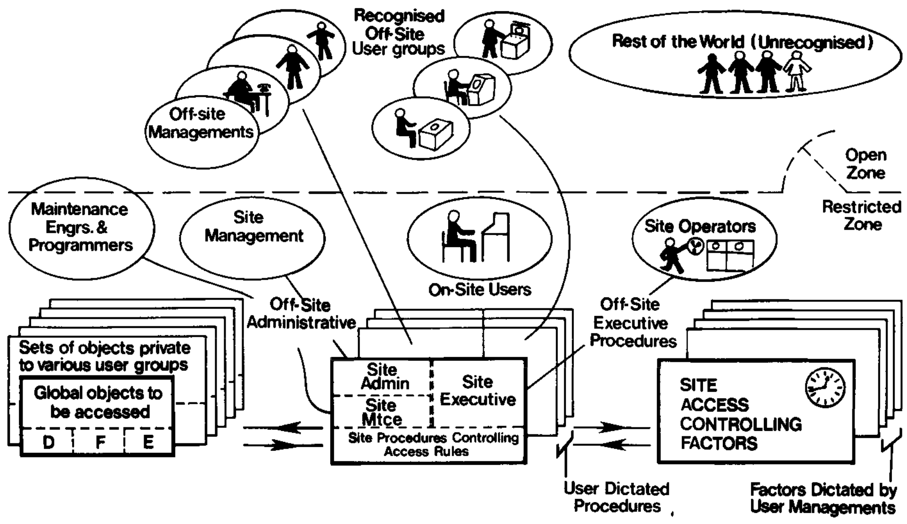

Recognised Off-Site User groups
Rest of the World (Unrecognised)
Off-site Managements
Open Zone
Restricted Zone
Maintenance Engrs. & Programmers
Site Management
On-Site Users
Site Operators
Off-Site Administrative
Off-Site Executive Procedures
Sets of objects private to various user groups
Global objects to be accessed
D F E
Site Admin
Site Executive
Site Mtce
Site Procedures Controlling Access Rules
SITE ACCESS CONTROLLING FACTORS
User Dictated Procedures
Factors Dictated by User Managements

Fig. 2 Diagram of more comprehensive computer access model.

Correspondingly there are now subsidiary groups of access procedures each subdivided into administrative and executive aspects as before. Again there will be groups of access-controlling factors mostly private to each user group but some global for all user groups including say the operators, maintenance staff and site management itself.

In practice it will be very important for security that no person is able to disobey the rules which must be framed to cater for all extraordinary situations including, of course, system crashes.

Obviously, any modern computer-operating system does in fact include facilities to allow users to implement a model such as has been described.

6 Multi-stage access control

It may be convenient to cater for a multistage access-control process – as in fact some systems already do. The concept is of successive levels of admission to access as follows:

(*a*) a general level determined at the start of a session and chiefly on the basis of external and publicly ascertainable factors; and
(*b*) a series of inner levels selectively and specifically controlled by internal and confidential or secret factors.

Obviously the way access rules apply to different groups of individuals have to be different; for instance the management on site will have very different responsibilities and authorities from those say of remote authorised users, who may not physically be seen by anyone in authority. Thus personal identification is fundamental to any scheme of access regulation. Since permitting access will nearly always involve recognising a person (either individually or as a member of some defined group), it is suggested that this be done in the first stage on the basis of certain externally known or knowable factors. After that, before access by a 'recognised' individual to this or that object is granted, more elaborate tests may be applied on the basis of more specific and detailed internal factors.

The decision process necessarily becomes more complex and elaborate, the closer an individual seeks to get to any highly secure or sensitive object. The rules laid down for a given installation must minimise any frustration or inefficiency that could result from over complex logical criteria designed to permit accesses successively to objects having varying security classifications.

Thus practical advantages of controlling access in stages are therefore:

that if the system is too busy to handle any more users, no time need be wasted identifying them or testing in finer detail for specific access permits;

that time is not wasted in going into detail over applications for access from those not recognised at all under the rules;

that, conversely, measures to apprehend those doing suspicious acts may be taken with a minimum of delay; and

that the information deducible by the unauthorised person from the reactions of the system to his efforts is at a minimum – he is simply aware of not being recognised.

7 Formulation of house rules

To set down rules to be applied in an actual case, say an Area Health Authority,[10] the procedure would be along the following lines:

identify the different data elements and other objects at risk of improper access and identify the different risks to be guarded against;

distinguish the different groups of individuals, e.g. doctors, nurses, administrators, technicians, porters, telephone operators, etc., and, considering their chief responsibilities, decide what is the minimum set of authorities for access to different data each needs and what right to delegate access authority a member of one group may need to have in respect of members of some other group;

analyse as fully as possible the opportunities for inadvertent error and deliberate improper action on the part of each group;

from that analysis, formulate a set of rules governing access to the system and data held, to protect privacy and the well-being of the using organisation generally.

The rules would have to include:

lists of authorised individuals and who may see those lists,
methods of *identification* of individuals,
access *criteria* and
logical combinations thereof,
records of both proper and attempts at improper access to be kept,
alarms to be given if rules are infringed,
authorities conferred on individuals or groups *to transfer access rights* to others,
the facilities and powers permitted for *cancelling access rights*,
the *records* of such transfers and cancellations that must be kept.

Before any rules are introduced, the detailed administration of the secure distribution to authorised persons of essential knowledge (e.g. passwords, cypher keys if used and proper procedures for using them) needs to be set out and carefully studied in detail to check that it too is free from loopholes. The computer system will in general be heavily involved in the day-to-day administration of the rules, including the secure distribution of key knowledge.

There are numerous practical points to notice in giving effect to the house rules, e.g. that making attempts at access visible or recording them inconspicuously is a security precaution in itself, that audits and regular spot checks need to be made on the data and the procedures and so forth. These topics are covered in a number of publications, e.g. Hoffman[11].

It has to be admitted that comparatively few examples of such rules or procedures (or of the implementation details of highly secure computer systems) have been described. The reasons for this are fairly obvious.

8 Acknowledgments

The author gratefully acknowledges the help of many colleagues in preparing this paper, especially Mr R.W. Jones and Mr D.J.Blackwell. However he takes full responsibility for the views expressed.

References

1 *Report of the committee on privacy* (The Younger Report), HMSO, London, Cmd. 5012, 1972.

2 *Computers and privacy* (Government White Paper), HMSO, London, Cmd. 6353, 1975.

3 *Report of the committee on data protection* (The Lindop Report), HMSO, London, Cmd. 7341, 1978.

4 WONG, K.: *Risk Analysis & Control* The National Computer Centre, Manchester, 1977.

5 KINGSLAKE, R.: 'Access control requirements for privacy and security', *Information Privacy*, 1979, 1, (7) p.312

6 PRITCHARD, J.A.T.: *Risk management in action*, NCC 1978.

7 'Protecting the Individual' – Report of the European Parliament Subcommittee on Data Processing and the Rights of the Individual. EEC reference PE56. 386. Also reprinted in *Information Privacy* 1979, 1, (8), p.335

8 DIFFIE, W. and HELLMAN, M.E.: 'Privacy and authentications: an introduction to cryptography' *Proc. IEEE*, 1979, 67 part 3, 397-427.

9 HELLMAN, M.E. 'The mathematics of Public-Key Cryptography' *Sci. Am.* 1979, 214, part 2, 146-157.

10 CLARKE, D.J., FISHER, R.H. and LING, G.: 'Computer-held patient records: the Exeter project'. *Information Privacy*, 1979, 1, (4), 164-171.

11 HOFFMAN, L.J.: *Modern methods for computer security and privacy*. Prentice Hall, U.S.A., 1977.

CADES – software engineering in practice

R.W.McGuffin, A.E.Elliston, B.R.Tranter and P.N.Westmacott
ICL Product Development Group, Technology Division, Manchester

Abstract

A software engineering problem – the development of a very large operating system, VME/B, the constraints placed upon its development and the objectives of the methodology and tools devised to support it (CADES) is described. The methodology (top-down, data driven) and the methods and mechanisms used to support the methodology are discussed in detail. The human aspects of the use of these are given and are supported by various statistics. From our experience, various lessons are drawn and the objectives are compared with achievements. In addition, the application of the techniques developed to hardware and to other software developments is described.

1 Introduction

The design, implementation and maintenance of complex software systems require the same kind of disciplined design methodology and automated, mass-production techniques that are normally associated with hardware developments. ICL, the major computer manufacturer in Europe, has extensive experience in the area of hardware design and manufacture. The design automation system, DA4, provides a service to some 400 users, from high level logic design through to manufacturing output and field support. However, a complementary service is also provided to aid the development and production of operating system software. This paper describes the inception, design and experience of this service.

In 1965, English Electric Computers and ICT set out to produce new operating systems, Multijob and George 3, for their new ranges of computers. They assumed that those systems could be developed using extensions of the techniques used on earlier systems, that is, small operating systems for small machines. They assumed that they could treat the operating systems as collections of programs and farm out the development of these programs to separate groups of programmers. They, like other manufacturers, found that this approach was inadequate. They discovered that the major problem they faced was that of linking the programs together to form a coherent, usable system rather than developing the individual programs. Also, it was found that the cost of the systems, compared with earlier systems, increased exponentially with size, and that with the inherent increase in complexity of the parts, the predictability of the completion date diminished considerably.

Thus, in 1970, when ICL (formed by the marriage of ICT and English Electric Computers) set out to produce a major operating system, VME/B, for its new range of computers, the 2900 series, it recognised that it had a major task on its hands.

VME/B would inevitably be a large system. It had to support all the facilities of general purpose systems produced for earlier machines (batch, interactive etc). It would also have to provide new capabilities, for example, virtual store management, protection levels, database management. Added to this was the requirement that it should be configuration independent. The 2900 series of computers was to be based on a radically new architecture – this added considerably to the difficulty of designing and producing the operating system.

This paper will describe the system designed and built by ICL to aid the development of VME/B; this system is called CADES – Computer-Aided Development and Evaluation System. Before discussing it in detail, it is of interest to consider both the constraints placed upon the development of VME/B and the objectives set for CADES.

1.1 Constraints

1.1.1 Timescales: Development of previous ICL operating systems had required teams of upward of 100 people. As VME/B was to be developed over a significantly shorter timescale, it was inevitable that an even greater number of people would be required. The work of this large team would have to be adequately co-ordinated and controlled.

1.1.2 Long life: The team would need to be able to identify and preserve the overall structure of the operating system. Experience on earlier systems had shown how difficult it was to protect a large system from structural decay. The team would need to be able to distinguish between features which affect the overall structure and those which were of merely local significance.

1.1.3 Resilience, enhanceability, maintainability: The methodology and computer-aided system would have to facilitate all stages of the operating system development process, that is high level design, low level design/implementation, construction, system generation and maintenance such that these constraints could be met.

1.2 Objectives

Given the target of producing a large new operating system on a new architecture with the constraints set out above it was decided to provide a design system which would aid development and positively encourage discipline. This system (CADES) had a number of well-defined objectives:

1.2.1 Method of working: It was recognised that human organisations are self-formative, that is, they will naturally be hierarchical – department manager, project manager, team leader etc. – and that the methodology should be such that the human organisation should mirror the design structure.

1.2.2 Communication documentation: People communicate but not very effectively. CADES had to provide a mechanism of formal communication, not just at the output level but between natural human (and logical) groupings such that, for example, common code was recognised.

1.2.3 Interfaces: VME/B design was to be data driven, consequently both vertical and lateral interfaces were extremely important and it was vital to be able to represent and stabilise these interfaces.

1.2.4 Formal capture of design: It was decided that a 'top-down' approach to design would be adopted and mechanisms had to be provided to support this approach. Further, at each level in the design hierarchy, it was very necessary that the methodology highlighted design-decision points such that these design decisions could be safely made, and recorded formally rather than in potentially ambiguous natural language form.

1.2.5 Performance validation: It was desirable that measures of performance could be established easily from the mechanisms provided to support the development. Further, it was essential that the performance implications of changes at high or low levels could be tracked.

1.2.6 'Own medicine': To provide the necessary incentive to hone the methodology and tools being applied to the development of VME/B, it was decided that these tools should be developed using the same techniques. Further, since the design of the operating system had commenced before the tools program was inaugurated, the tool providers had to make special efforts to catch up and overtake the operating system development.

2 The CADES system

The CADES system consists of a methodology and a set of mechanisms to support the methodology. That is, there is a defined method of approaching the design and implementation of the product which underlies all the individual activities which have to be performed to achieve the end result. This approach is then supported by detailed mechanisms which may be well-established rules for human action and interaction or software products to be used as tools by the designers and implementors.

The ultimate aim of the CADES system is to support the total software development cycle from receipt of the stimuli that initiate design changes (requests for new facilities, and reports of faults or of inadequacies in the product) right through to the release of software packages, complete with supporting documentation. The main elements of the development cycle are represented in Fig. 1.

2.1 Methodology

2.1.1 Formal design: A fundamental rule of the CADES system is that we design the product before we implement it, and we capture the design in a formal manner in an integrated database so that the design is preserved and is available for subsequent use, for example, for implementation, design amendments, etc. The design capture is the design method so that we do not produce a design in one manner and document it in another.

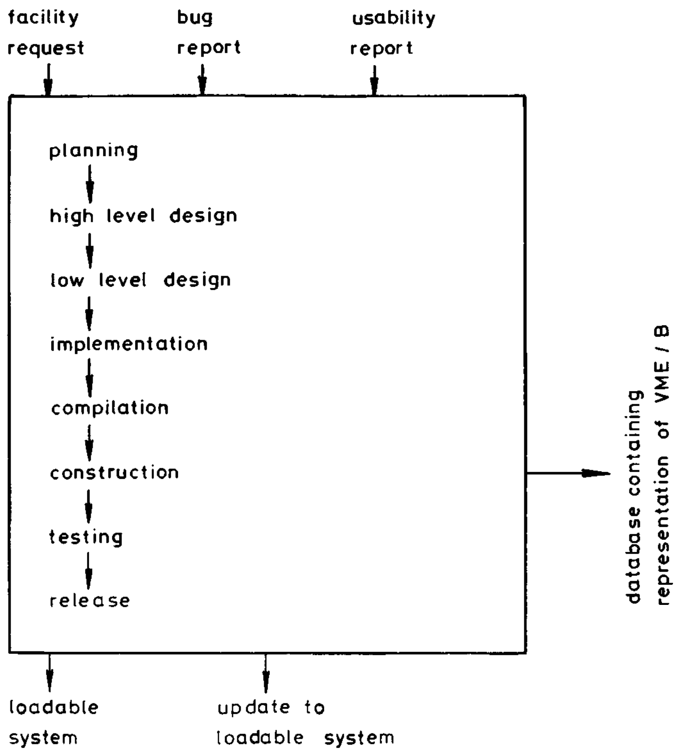


Fig. 1 The software development cycle

2.1.2 Top-down design: The design method we use, known as structural modelling, is one of top-down iteration. That is, we start at the top by defining the whole product in general terms based upon its specification. This definition is then partitioned at the next level down to provide a specification of its parts and each partition is defined in more detail. Each level of this structure defines the whole product, in interaction of the holon (see 2.1.3) and data hierarchy.

As we define each level, we re-evaluate the preceding (higher) level to ensure that all new decisions are consistent with those already recorded. If changes have to be made to a higher level to accommodate the new design decisions, then the level above is also re-evaluated and all the decisions in the level below the changed level are also re-evaluated in the light of the changes. This has the effect of a yoyo going up and down the hierarchy until no more changes are made. Then we proceed to define the next level of detail and repeat the process.

In this way, we do not make any decisions until we have to, but we end up with a consistent set of design decisions.

2.1.3 Data-driven design: The process of making the design decisions is data driven. That is, the entities we are dealing with first at each level are the data to be manipulated at that level. The process starts by partitioning the data of the higher level and defining their attributes and transformations at the new level. The complementary entities involved are the 'holons' (a term borrowed from Koestler, and used to represent objects in a neutral manner without any of the connotations of 'procedure', 'macro', etc.) which are the processes which control and use the data items. The higher level (parent) holon is partitioned in terms of its control of the data and the partitioned (child) holons are then defined in terms of their use of the data. The two hierarchies of data and holons are progressed in parallel, with data preceding holons at each level.

Each holon may have its own transient data dynamically allocated but its use of nontransient data, whether shared or local to the holon, is controlled via the interaction of the holon and data hierarchies. A holon may only manipulate data items which are children of the data manipulated by its parent and which are either owned by the holon itself or specifically made available by the holon which does own them. This system, as well as defining the control of the data, allows us to minimise the interactions of holons with data, and thus of holons with other holons.

As well as the data described above, we also wish the holons to manipulate parametric data. A holon may pass any data to which it has access to any holon which it calls. We need to ensure that the data as passed is consistent with the data expected by the receiving holon. This is achieved by having only one definition of the interface, which applies to both the called and calling holons.

2.1.4 Algorithms: The lowest level of the holon tree represents discrete functional elements of the product at the implementation level. That is, they will be coded in a compilable language and compiled into object code which will be the product. Only the function of the holon needs to be coded, as its environment (the data it manipulates) is already defined through the data hierarchy.

At higher levels of the holon tree, the coded algorithm is a formal specification of the partitioning of a holon's functions between its various component holons and of its use of external functions at the same level of abstraction.

2.1.5 Programming standards: We have maintained existing ideas on good programming practice along with the CADES methodology. The function of the implementation-level holons is coded in a high-level language with all the recognised benefits. In addition, the choice of S3 (a language related to Algol 68) allows full advantage to be taken of the 2900 architecture (for example, hardware stack) while producing extremely efficient object code. The partitioning of the product into many implementation-level holons leads naturally into the production of modular programs where individual routines may be amended and replaced.

2.1.6 Management structure: Since we have a structure of the product as defined in the holon and data hierarchies, when we need another hierarchical structure we mirror it off these. The next structure required is, in fact, a human organisational structure. The organisation of the production projects is based on the structure of the product they are to produce. Since we have defined the interfaces between the elements of the product, we have also defined and minimised the interfaces between the groups of people producing the product and the management structure for controlling them as they produce it.

2.1.7 Design authorisation: Of course, we need to keep an overall control on the product as designed and, therefore, we have a design team responsible for the total product and the interfaces between the parts of the product. No changes can be made which affect this structure without their authorisation.

In order to ensure that a change does not slip through because the production project did not realise it changed the structure, an authorisation cycle must be followed before any changes are made. This involves authorisation by the designer in the project and by the project manager and then by the overall design authority for every change. This way we can ensure that the full implications of each change are recorded and that the design does not decay because the actual product moves away from the design representation. As well as simply checking that the change is acceptable, we can check that the holon and data hierarchies are altered at every level to reflect the change, and that the documentation is also amended where necessary.

The normal working documents for designers and programmers are 'Tree pictures' of the holons and data. These are simply computer-produced listings of the hierarchical structures as described above. In addition, we use the interface definitions and charts of the interactions between the two hierarchies – that is, which holons use, or are allowed to use, which data.

2.2 Methods and mechanisms

2.2.1 Database: In order to support the methodology described above, we have a database system. The hierarchies of holons and data with their attributes and relationships are stored in the database, which is the authorised description of the product as it is being developed. The final content of the database is the product itself so that there is no break in continuity between design and production since the source code and compiled code are held in, and controlled by, the database. (In some cases, bulk data is physically held not in the database itself, but in conventional filestore addressed by records in the database; logically, however, such data are considered to form part of the integrated database.)

2.2.2 User interface: The interface for the user is an end-user language, specially designed to represent the structure we have chosen to describe the product. The structure is of target records and the relationships between them and the language is a navigational language. It allows the user to specify target records and to navigate the relationships between target records. The same language is used for updating and interrogating the database thus simplifying the interface for the users and removing the margin for error in effecting required changes to the database.

2.2.3 Language definition: The total set of target record types and possible relationships between them, and the attributes, with their descriptions, applicable to each target or relationship are also held in a CADES database. This database forms part of the definition and, therefore, the implementation of the CADES system. Thus, we are able to use the database navigation language to update the definition of the language itself. New target record types or new relationships between existing targets, or new attributes of targets or relationships can be defined, and we can then generate a new, improved version of the language. No only this, but we can hold different definitions of the language which we can make available to different users. In fact, we use a different dialect of the language for updating the language definitions from that used for producing other software systems.

2.2.4 Database software: Underlying the whole system is an IDMS database. IDMS is ICL's Integrated Database Management System. This system consists of programs and software components to handle the establishment, processing and administration of a database.

IDMS is an implementation of a subset of the COBOL database facility (1975) proposed by the Codasyl Programming Languages Committee. These committees have proposed standard languages for defining the content and organisation of a database and for defining the data manipulation functions available to COBOL programmers. Codasyl based database systems have been implemented on many different mainframes and operating systems and now form a 'de facto' standard.

We have produced a basic, general schema – the mechanism by which the particular database is described to IDMS. We then have a superstructure of CADES software which implements the user interface, and interprets the user's view of the database (in terms of specific target records and relationships) into the IDMS schema view, and vice versa.

As well as handling the language itself, this superstructure performs several other important functions including version handling and protection, which we provide on several levels.

2.2.5 Versions: The total product definition and implementation is held in the CADES database. The product is not static but is under continuous development for extension and enhancement of facilities. Therefore, we need to be able to represent different versions of the product. Some may be under development, some may be in the field, but we must maintain the definition of each one. We achieve this by the version-handling facilities which CADES supplies on top of the IDMS software.

Each target record and relationship in the database can exist in different versions. Whenever a new version is created it inherits the attributes of the preceding version so that we only need to specify the changes. If a particular entity is unchanged, then no new version is created and the new version of the product will inherit the preceding version of the entity in its entirety. Since we are dealing with a very large product, we have divided the total database into logical regions. Each region has its own independent version numbering and can be developed independently of all other regions. We then create a new version of the total product by incorporating a particular version of each region.

Once again we use the product structure as the main structure and mirror the region structure on this. In this way, we can develop a particular subsystem of the product and only incorporate it into the total product when we are ready. In fact, several subsystems can be under independent development without interfering with each other and may then be incorporated one-by-one into new versions of the product as they become available.

2.2.6 Protection: Since different groups of people are working independently on different parts of the product, we need to protect the data in the database against unauthorised changes – or rather changes by unauthorised people. Because the product structure is mirrored in the people structure and also the region structure, then we can use the region structure for protection. Therefore, when anybody logs in to the database, they log in to a region and they are prevented by the CADES software from altering anything in any other region.

The ordinary user in fact logs into a Database Activity which gives him access to one version of one region. Activities are allocated by the central design authority and give permission to create a new version of a region in response to a planned development. Thus, the user is freed from problems of region and version handling while the central design authority maintains control over the development of the product.

2.2.7 Documentation: The documentation of the product being developed is all held in a documentation system known as the Project Log. This consists partly of information extracted from the database (for example, the latest source listing of a holon) and partly of narrative descriptions. The whole is, however, controlled from the database by use of the project log reference numbering system which relates each part of the documentation to the entity documented, to its version and to its place in the hierarchies.

This forms a consistent set of maintenance, support and development documentation of the product. For convenience, it is held on microfiche as well as paper, with the parts extracted from the database being produced by a COM fiche (computer output microfiche) facility.

2.2.8 Tools: Since we put the definition of the product into the database, we obviously need to manipulate it in order to create the product. We have accordingly defined a set of software tools which complement the database software. During the development of CADES these tools have evolved considerably; descriptions in this paper are of the latest such tools unless otherwise indicated.

We have the data hierarchy with the bottom level defining the implementation level data, that is, basically in the form in which it needs to be defined for its manipulation in a high-level language.

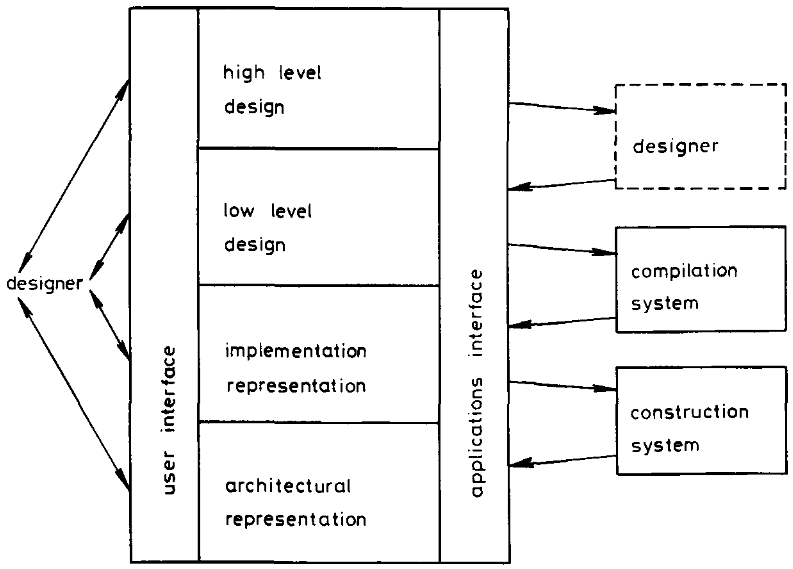
We have the holon hierarchy with the bottom level defining the implementation-level functions, that is, basically in the form in which it can be compiled as a high-level language. It also includes holon interaction by reference to the defined interfaces.

We also have the holon-data interaction defined in the database, as well as certain items of architectural information, that is, the relationships to physical hardware concepts such as protection levels. The sum total of the method of defining this is termed the system definition language (SDL), and basically comprises the database navigation language described above and the function specification.

In order to create the product, we need to go through such processes as compilation and collection. The method we employ for this is to use the standard ICL tools and to interface them to the database by means of applications which know about the structure and contents of the database and present standard input to the tools.

To present a module for compilation, we have an Environment Processor. This extracts from the database all the information about the holons which constitute the module – the functions, data and interface – and presents this in a form to be processed by the language compiler. Having compiled any new or amended modules, we can then collect them together with the existing product by use of a construction application which extracts information from the database about the modules which are to constitute the product and drives the standard ICL tools to collect together new and existing modules and apply any necessary architectural details.

The position of these tools in the CADES system, and the way in which they provide bridges between the orthogonal representations of the product in the CADES database, can be seen from the following diagram:



The applications we produce have progressed through several phases. Early ones interacted more directly with the IDMS database, using only the version-handling mechanism of CADES software. The later ones (including the current environment processor) use the navigation language as their interface to the database, but are themselves written in ordinary high-level languages such as COBOL. However, for our newest applications, we make use of a generalised tool which we have developed, the Report Generator.

This, as its name implies, was developed first for producing printed reports from the contents of the database and is extensively used for this purpose. In addition, however, we use it to generate the input data and steering data to drive the tools. The construction application is one of those that we have implemented in this way.

3 Man/machine interface

In order to illustrate the changes in man/machine interaction effected over the years, it is necessary to compare statistics relating to early CADES usage with those related to our current development environment using interactive facilities affording vastly improved information turnround.

In particular, software development under CADES has advanced to the stage where the apparent overhead of maintaining the central information database does not detract from the competitiveness of CADES software development from the point of view of an individual code production unit.

3.1 Development of the CADES system

The CADES system, since its conception around 1970, has been developed firstly on System 4-70 and then on the 2900 range after a bridging phase.

The principal machine used in this latter stage has been a 2970 upgraded to a dual system in recent months. The average human resource strength over this period has been some 20-25 personnel of which five have been strategic-system designers.

The CADES software development environment has consisted of 15-20 programmers interacting with the machine via four video terminals and remote job submission, the total level of machine usage per year exceeding a million seconds of central processor time on a 2970.

The extent of CADES software currently in existence is some 400 implementation-level holons each of which consists of approximately 400 lines of SDL, making a total of about 160,000 lines of high-level language source code.

In recent years, all CADES development has been performed using our own development applications and tools, that is, CADES itself. This has resulted in a high standard of software being released to internal customer units.

3.2 *Development of VME/B*

The development of VME/B has been performed from its earliest days using CADES by 100 to 200 designers and programmers. This includes a pool of some 20 strategists or strategic designers and a CADES service unit providing the front line CADES human and software interface to VME/B staff. This early development was on System 4 using an ICL produced database system called PEARL (Parts Explosion And Retrieval Language). Although the applications developed with this database were rather limited it still held 40 megabytes of data before finally being phased out after the transfer to 2900.

The transfer to 2900 was a major phase for VME/B CADES development and resulted in the establishment of an IDMS database containing 25 Megabytes of data; the conversion between two totally alien database structures was carried out with virtually no adverse impact on the users of the system. This IDMS database has a capacity of 183.4 Megabytes at the present time and consists of 59,952 IDMS pages each of 3100 bytes capacity. The content of this database has grown throughout, with the continual addition of new categories of information relating to VME/B design, production and release. At the time of writing, its data occupancy is 40 per cent.

On 2900 the predominant machines used for developing VME/B have been 2976 and single and dual 2970s. Some 30 video terminals attached to one or other of these machines are used by VME/B development staff.

The result of all this development has been a VME/B product consisting of some 6000 object code modules, derived from some 12,000 holons in total, arranged in a tree structure of typically five or six levels.

3.3 *Functions of the CADES service unit*

The prime responsibilities of the CADES service unit are to maintain the serviceability of the database, perform database updates, generate CADES reports and provide a documentation service.

The record of the VME/B database is such that it has not yet been necessary to restore it from the archive which is performed weekly. Database journals, however, which hold records of all modified IDMS pages are kept for limited periods in order that local intermediate recoveries may be performed.

Historically, the text specified by designers in order to update the database has been submitted to the service unit for batching together into a daily database update. The designer interface for functional code and documentation amendments is via a common design amendment form (DAF). Around 50 of these relating to VME/B are processed daily of which 25, on average, contain documentation changes which involve retyping and eventual microfiche updates and distribution to around 40 different centres within the company.

The update requests submitted by designers result in a weekly report summary on project and system information being generated by the service unit from the database. In this activity, some 60 to 80 reports are produced each week which summarise typically holon, data, literal and system message information. These reports are COM fiched and issued to centres throughout ICL for customer support in addition to being used for internal reference – an example centre being the Systems Maintenance Centre for direct customer support.

The introduction of interactive facilities for both database update and retrieval has improved the CADES service interface by enabling urgent design amendments to be processed on the spot and nonurgent amendments left over to be batched for the nightly update. This has produced a real-time response where previously no better than daily turnrounds were being achieved. It has been necessary to develop safety features related to interactive usage such that in the event of database failure, the service unit has recourse to an information diary. In addition, the recovery and contingency features of CADES are such that only hardware corruption of data necessitates returning to a database archive.

Further interactive usage extensions are planned which will allow VME/B designers unlimited direct access to the CADES database, within normal protection constraints, for updating and retrieval purposes; the selective archiving of regional information combined with information recall from archive on request; and the selective (and protected) deletion of information from the system. All these features will reduce the workload on the Service Unit.

3.4 Performance of the database

In order to perform any database interaction, it is first necessary to log into the CADES system, initialise data and generally gain access permission. Thereafter, database retrievals and, given appropriate access, updates may be initiated with very little overhead and may be intermingled with VME/B System Control Language statements.

Typically, the amount of updating of the database is of the order of 2000 lines of text per day; retrieval figures are less easy to quote since retrieval is not subject to the same authorisation checks as update – a conservative estimate of the order of 100,000 lines of text per day (including standard reports etc).

3.5 CADES report generation

It is necessary to provide designers with regular database information summaries. An advantage in this field is that the information content and format of these reports may also be defined in the database navigation language, the information then being extracted from the database using the CADES report generator. Some 20 standard CADES reports are currently available for use, and many others are under either design or development.

3.6 SDL compilation and environment processing

The major CADES application as far as VME/B development is concerned, is the

SDL compiler which is used to compile SDL code into object module format ready for build and release by the construction system.

The SDL compiler operates in two stages. The first stage is termed Environment Processing or fixing of design information into the source code; the second stage involves language compilation. Some 100 holons are SDL-compiled daily within the VME/B development environment.

This process may be performed directly off the CADES database such that a designer may update the database with a design modification interactively and initiate the generation of a new VME/B module immediately. The design information is, of course, database resident for other designers' usage and reference – a new version of VME/B may then be constructed.

4 Lessons and benefits

The use of the CADES methodology and mechanisms throughout the development of VME/B has been a success; a large and complex operating system has been produced over a period of several years by a large team of designers and programmers. It is, of course, impossible to tell how the development of VME/B would have differed, or whether it would have been produced at all, had structural modelling and CADES not existed, but an examination of the extent to which the various objectives outlined above were achieved indicates that VME/B is strongest in those areas where CADES gave most support.

4.1 Objectives of and constraints on VME/B

VME/B has met its major objectives of providing the users of ICL 2900 series computers with a powerful general-purpose operating system capable of running without change on any medium to large configuration.

The application of data-driven hierarchical design has led to a stability in the structure of the system which has enabled enhancements to be made in a controlled and predictable manner as new functional requirements or changed hardware interfaces demand. Use of the CADES system has proved invaluable, not only in costing such changes but also in ensuring that the changes were complete, by retrieving interaction information from the database.

While it would not be true to say that VME/B was produced in the short timescale originally envisaged, it was no mean achievement to have produced a viable major new operating system from scratch within a period of five years (approximately 750 man years). It is a moot point whether the use of CADES had any overall effect on the timescale for the production of the first version of VME/B – the absence of any comparable control project makes such measures impossible.

The reliability of the operating system was never considered to be within the scope of CADES, but the structure and documentation imposed by the CADES system has ensured that when faults are found, they can be corrected in a well-controlled manner without the ripple factor so often encountered in large scale software projects where clearance of one bug introduces another.

4.2 *Objectives of CADES*

If we consider in turn the objectives identified above for the CADES system, we see that on the whole, they were met, although in most cases quantification of the degree of success is difficult, if not impossible.

CADES has indeed allowed and encouraged the structure of the human organisation to mirror the structure of the product; communication between and among strategists, designers and implementors has been formalised and so constrained to exist; interfaces between and within subsystems are well controlled and defined. The formal capture of the design in a top-down manner has contributed significantly to the development of the system, the existence of formal descriptions at various levels of detail having facilitated iterations of design; however, the SDL functional descriptions at nonimplementation levels of the design have not been totally successful owing, no doubt, to the lack of tools which can act upon (or animate) such descriptions.

The version handling capability offered by CADES is such as to support concurrently several externally released issues of the product while at the same time allowing incremental development along many independent paths.

Use of a high-level language has been almost universal in VME/B there being no more than 20 holons containing any assembly language code out of over 10,000 holons.

The most significant area in which CADES has failed to meet its objectives has been the provision of a performance-prediction capability – this was omitted as a low-priority facility, although with hindsight we see that VME/B could have benefitted substantially from advance warning of areas of inadequate performance.

4.3 *Benefits not envisaged*

A number of problem areas have arisen during the development of VME/B, which, while not part of the original CADES concept, have been amenable to solutions based on the CADES database. Most significantly, the planning of developments within VME/B, whether for fault correction or for new facilities, can be tied in with the product representation in the database through the activities which already associate users of the system with developments within it. Inter-project dependencies, resource requirements, scheduled milestones and related data can all be held in the database and linked directly – where applicable – to records representing activities, groups of people (or individuals), subsystems etc. Thus, CADES now embraces the total development route.

A group of people as large as that producing VME/B is bound to generate a significant amount of documentation outside the realm of the project log. A database registration system for such documents has clear benefits, allowing retrieval of lists of documents by keywords or by author as well as straightforward reference lists; since the database used is also the design database we have been able to record relationships between, for example, strategic design documents and the subsystems and interfaces to be changed, and the activities in which the changes will take place.

Software faults are registered in the database linked to the holons in which the faults were manifested; when a solution (temporary or permanent) is found, it, too, is registered and associated with the fault record and with the holon to which the correction is to be applied.

The CADES version-handling scheme allows us to keep track of faults which may require clearance in different ways in different versions of the product.

The formalised filestore structure imposed by CADES for holding source and object code has proved invaluable in controlling the semi-automatic production of COM fiche support documentation.

In most of the above examples and in many other more mundane fields, the self-defining capability of the CADES database schema, together with the built-in report generator, have permitted very cost-effective extensions to the CADES system.

5 Future

In the preceding sections of this paper we have described the aims, mechanisms and achievements of CADES; we still have a long way to go before we can say that we have developed the full potential of the CADES methodology.

5.1 Further support for software development

Work is well advanced on enhancements to CADES to support the development and release control of hardware-test programs; this is an area very different from a large-scale operating system, consisting mainly of large numbers of relatively small, independent programs; success in this venture will support the belief that a CADES-style methodology and system is relevant to any branch of software engineering where there is a control problem, including the development of microprocessor software.

At the same time, we are not losing sight of those aspects of the original CADES concept which have not yet been developed: significant among these are a performance prediction capability; control over and possible automation of test systems; support for high level simulation (or animation of the structural model); and the formal representation of high level design concepts such as files and jobs.

5.2 Application to other disciplines

ICL's business is to manufacture and sell computer systems, the essential components of which are software, firmware and hardware. It is clear that the development routes for hardware and firmware have much in common with that for software; hence the methodology and control features of CADES can usefully be applied to them. Further, an integrated environment for total system development is essential, especially in the world of VLSI, and ICL has already commenced the design of such a system derived from its experience with CADES.

Bibliography

1 PEARSON, D.J.: 'CADES – computer-aided design and evaluation system' *Comput. Wkly*, July 26th, August 2nd, August 9th, 1973.

2 PRATTEN, G.D., and SNOWDON, R.A.: 'CADES – support for the development of complex software' *EUROCOMP* 1976.

3 PRATTEN, G.D., and SNOWDON, R.A.: 'CADES – computer aided development and evaluation system' ICL 1978.

4 PEARSON, D.J.: 'A study in the pragmatics of operating systems development' International Symposium on Operating Systems Techniques, Paris, 1978.

5 HENDERSON, P., GIMSON, R., PRATTEN, G.D., and SNOWDON, R.A.: The maintenance of software with multiple versions, in structured system design, INFOTECH 1979.

ME29 Initial program load – an exercise in defensive programming

R. Lakin
ICL Product Development Group, Southern Development Division, Kidsgrove, Staffs

Abstract

The ME29 system produced by ICL contains an initial load program which has been written to different criteria from its predecessors on other machines and systems. The concept of defensive programming has been used to write a loader which checks that an operation which has run to completion has done so corretly; and if an error is reported a support engineer should be able to predict with 95% certainty which component of the system is at fault. This paper describes the rules governing the writing of defensive programs in general and the ME29 loader in particular, and some of the tools which were used to achieve the aims of the program.

1 Introduction

The requirement for Defensive Initial Program Load (DIPL) arose from the problems encountered by field engineers when a machine failure was signalled. Until the present system, Initial Program Load (IPL) programs were governed more by a requirement for low store occupancy than one for resilience or good fault diagnosis. This stress on such a requirement arose because:

(*a*) IPL was written in assembler code or binary, therefore large programs were avoided;
(*b*) there was no executive to support IPL;
(*c*) IPL had to reside on a fixed place on the load media, consequently there were space restrictions;
(*d*) there was assumed to be an adequate supply of test programs to check out the machine before a load was attempted;
(*e*) error recovery normally consisted of a reload.

The (initial program) load process consisted of two phases:

(*a*) run the engineers' test programs. If any failures occur, repair the machine before proceeding;
(*b*) perform IPL.

Phase (*a*) was normally run at the start of day, after maintenance or machine modification or after a suspected hardware malfunction.

As operating systems became more complex, so did the requirements for IPL programs; they grew in size and complexity with obvious repercussions:

(*a*) the program became more difficult to follow in the event of a failure;
(*b*) more hardware was used by IPL to perform its function;
(*c*) a more complex system of recording IPLs and systems on load media became necessary;
(*d*) the test programs could not hope to simulate exactly the conditions under which IPL ran; when the test program would work, but not IPL, the reason was hard to isolate.

When an engineer was called out to repair a fault, extra time had to be spent determining which components were affected and a further visit was often necessary to supply the appropriate spare parts. The intention of DIPL is to enable an engineer remote from the site to determine from a telephone conversation with an operator whether a visit is necessary to repair the fault, and if so what spares need to be taken. In this way the machine can be reinstated quickly at less cost to the customer and to ICL. The principles and construction of DIPL are based on those given by Yau and Cheung[1] and by Dent[2].

Traditional programming practices do not provide a suitable method of writing initial load programs which give the required depth of information, for example:

(*a*) when the program has reported a successful run, what is the probability that it has performed its task successfully? Only test programs attempt to answer this question;
(*b*) it is often assumed that lower-level software or hardware will perform correctly;
(*c*) performance considerations often mean that reporting of errors is insufficiently covered; the same goes for sanity checks of apparently correctly running programs;
(*d*) programs written in assembly code often omit those checks which are automatically included in high-level language systems;
(*e*) the engineer often has to find his own way around an executive or microcode listing to be able to sort out unexpected hardware faults;
(*f*) independent test programs are used to check out the system prior to customer work. Often the system passes the tests, only to fail when used in anger: it is well known that systems 'learn' to perform test programs successfully;
(*g*) tools used by the programmer during debugging are removed when the program is put into production on the grounds that it is more efficient without the tools;
(*h*) failure messages are sometimes unhelpful except to the expert on the program that produces the messages.

Attempts to solve these problems gave rise to a code of practice and a set of tools to help to achieve the aims of defence.

2 Description of the ME29 system

Before describing DIPL it will be helpful to explain the nature of the machine on which it will be run. This is a physically small machine of conventional type in the sense that it includes a processing unit, peripherals and controllers. The processor is a fast architectural emulator with a special ability to create 'soft' machines; it has become known in ICL as M–1. The main features of the system are given in the following paragraphs, and the technical terms used here and in other parts of the paper are explained in the Glossary at the end.

2.1 Operator's panel

This is the means by which an operator or an engineer communicates with the M–1 Order Code Processor (OCP), which is described in para. 2.2. Fig. 1 shows this panel. It is situated on the cabinet which contains the OCP, peripheral control modules and a 1 Mb flexible disc. Information is sent from a hexadecimal keypad or control keypad and is read on an 8-digit hexadecimal display and a set of light-emitting diode displays. Within the panel is a microprocessor which handles the messages and controls the displays.

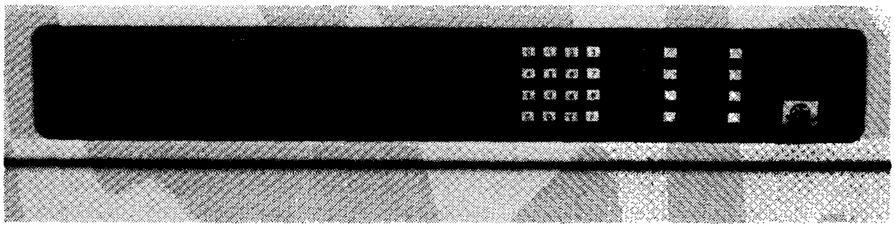


Fig. 1 Operator's panel

During normal operation the operator uses the panel to load an executive program which will communicate through a video console once it is running. An operator in possession of a key able to turn the lock on the panel is able also to perform functions designed to access the M–1 OCP more intimately: for example, to read and alter memory or to change the clock speed. Such an operator is known as a key operator.

The operator's panel software keeps a log of actions and failure reports so that DIPL and other software may retain meaningful information. This log can be called up and read on the display by an operator who need not be in possession of a key (a non-key operator) and who can send or telephone information to an ICL engineer who may be remote from the site. The engineer can then diagnose the fault.

Linked to the panel and on a separate board (printed circuit board, PCB) is a hardware diagnostic system for checking the ME29 hardware. Some of these checks are performed automatically as part of the load/dump sequence to ensure that the M–1 OCP and the stores work to specification.

2.2 *M–1 order code processor*

This is a state-of-the-art machine with an instruction set designed to aid the development and execution of emulators for existing target machines. One such target machine is the ICL 2903 but of course an emulator for any other can be loaded and machine architectures can be tailored to fit any specific requirements. The M–1 instructions are simple enough for it to be practical to write programs for M–1 itself rather than for a target machine, and in fact most of DIPL is written in this form.

2.3 *Associated peripherals*

DIPL is capable of loading an emulator from any of the following:

double density, double sided 1-megabyte flexible disc cartridges
60 Mb and 80 Mb exchangeable disc stores EDS 60 and 80

The ME29 system in its present form also supports:

operator station, DDE stations, multiple-access terminals
band printers at 360, 720 and 1130 lines per minute.

The format of files on system media is built around the Unified Direct Access Standard (UDAS) format used by many ICL systems. To assist in defence and to enable DIPL to use the same files without the need for it to know the structure of system files, a primitive operations file index (POFI) is formed on each volume. The volume header label (VOLH) exists in a fixed place on every volume and contains a pointer to the POFI from where the bootstrap, dump and microcode files may be accessed directly. If during the process of continuous development the structure of files within the UDAS area is changed, there will be no impact on the load system.

2.4 *Documentation*

The importance of good documentation cannot be over-emphasised. For this system the ICL engineer has three sources:

(*a*) the assembler program in the M–1 processor – the MICE assembler, described in para. 4.1 – produces program listings which are annotated with the engineer's needs in mind. Much use is made of a tracing aid known as *footprints*, described in para. 4.5. Each step of the program has a comment box in the listing containing the footprint number, a description of this footprint and error messages generated by it. Within the step each group of instructions performing a function is commented on with an English or an Algol-like sentence describing the function. On the right-hand side of the page, away from the code statements, there are comments referring to specific instructions. These comments do not describe what is done but rather how and why for sometimes a function is evaluated using a sequence of instructions which has been designed to avoid a set of instructions not used anywhere else in DIPL. An example of footprint structure and style of comment is given in Appendix 1 and more is said about documentation in para. 4.8.

(*b*) the technical literature which is made generally available shows the general construction and layout of DIPL. There is also a section detailing each footprint and this is duplicated in the program listing as comments.

(*c*) a manual provides the meaning of every failure message and footprint produced by DIPL, the meanings of the items in the information dump area, a possible diagnosis of the fault and suggestions for actions to be taken. This manual is in possession of the engineer at the time he receives the call from the customer.

There are other documents available to customers and to support programmers. Most important of these is the manual kept on the customer's machine, which details how to use DIPL and what to do if it fails. This manual is similar to that held by the engineer but is expressed in terms familiar to and more suitable for the operator. In the event of a failure which cannot be dealt with by the operator or installation manager, the customer engineer can be called and will tackle the failure using the documents described above.

2.5 *Method of loading*

The operator proceeds as follows. The required media are loaded. The operator presses the LOAD button on the panel. The default load device number and load parameters are displayed and they may be changed by the operator. When he is satisfied with the parameters he presses ENTER to initiate the load sequence.

There is a facility to dump the ME29 store to a magnetic device. The operator starts the dump process in a way similar to load, except that the DUMP button is used and a separate device may, if required, be used to perform the dumping.

During the load sequence the display on the panel will show footprint messages showing how far the process has gone. If the load is successful an appropriate message is displayed and the operator may use the video console to communicate with the executive which has been loaded. If a fault has been detected DIPL will stop and the panel will show the appropriate failure message. The operator may then refer in his manual to that message and take a course of action suggested there. If the fault can be rectified by the operator he may do so and restart the load sequence, otherwise he may extract the relevant information from the log and contact an engineer. The engineer uses the documentation detailed in para. 2.4, together with the information sent by the operator, to determine how to correct the fault.

Most important to the engineer is the probability that the diagnosis given or implied by the failure information is correct. If this is incorrect the cost to the customer is increased by return journeys to the site and by the carrying of large kits of spares; furthermore the customer's image of ICL is not enhanced when several attempts are needed to remedy a fault. In an attempt to reduce the number of spares carried on a visit the tests are designed to resolve faults to as few PCBs as possible. Therefore included in the resolution notes is a section devoted to the contents of the minimum spares kit required to repair the fault on a single call with an expected success rate of 95% or better. DIPL also attempts to make this minimum kit contain as few boards as possible, to save the engineer taking a complete machine to the site and to avoid the cost of retaining a large spares holding.

Sometimes the information in the error report gives a resolution to an integrated circuit. This arises because it is sometimes necessary to resolve to a great depth to distinguish between PCBs, and it does not cost much in size or speed to include this.

3 Construction of the DIPL system

The whole of DIPL consists of several parts:

(*a*) operator's panel ROM code;
(*b*) diagnostic card ROM code;
(*c*) DIPL code in the operator's panel ROM;
(*d*) disc bootstrap (microcode loader);
(*e*) emulator and preludes or TEST CONTROL package;
(*f*) executive bootstrap;
(*g*) executive start-of-day code.

When the machine is switched on or reset, the operator's panel code (*a*) checks that it is possible to communicate with the operator and with the diagnostic card (*b*). The load sequence starts with the panel checking that the data in its ROM is correct and the diagnostic card checking that the M–1 OCP can fetch and execute instructions. The diagnostic card is able to 'single shot', by software control, a sequence of instructions in the M–1 code (*c*) before it is run at full speed; we use this facility to check that jumps are executed correctly but it will be a simple matter to invoke the mechanism during fault resolution if experience shows that there is a need in future releases.

After the M–1 code has been loaded it checks the load device PCM and interrupt system before attempting a peripheral transfer. The M–1 code contains procedures to perform all peripheral transfers executed by (*c*) and by the disc bootstrap (*d*) which also is written in M–1 code. The differences between these are based mainly on the knowledge of the file formats on the load device: the ROM code (*c*) assumes only formats based on the primitive operations file index POFI, which means that it can load any bootstrap on any magnetic media device which contains a POFI. It is not necessary to restrict the system to use formats currently in use by the ME29 executive programs. New file formats may however necessitate a new bootstrap. More information on the place of POFI in the system is given in para. 2.3 above.

The disc bootstrap loads an emulator and its prelude (*e*). The latter is 'throw-away' code which checks and initialises the emulator and the hardware which it uses and is then removed from the processor. Should the emulator break because of hardware failure it will still be possible to load the package TEST CONTROL (*e*) which provides sophisticated test programs to resolve faults to any desired depth.

The executive bootstrap (*f*) is the first part of the system to use the target machine code. It performs the same functions as (*d*) but at a higher architectural level, loading in the executive (*g*). This executive contains some target code known as 'start-of-day' code which is equivalent to the emulator prelude. When this has finished running the machine is ready for use.

These components of the load process are in the main automatic and take only a few seconds to complete. All parts of DIPL from (*d*) onwards are written to magnetic media devices and can be termed 'software', although (*d*) and (*e*) are more 'firmware' in concept since in an emulator the 'software' is normally restricted to target machine code or higher.

4 Development tools and techniques

The following paragraphs describe some of the tools used by the teams developing the system; they are not made generally available outside ICL.

4.1 MICE assembler

This produces a binary file which may be transferred to a UDAS-formatted disc or converted to paper tape output for manufacturing read-only memory (ROM) programs. Any high-level language statements are converted by use of the macro scheme described in para. 4.2.

4.2 ML/1 macro assembler

This is a stand-alone general-purpose macro processor capable of converting statements from virtually any format into any other. DIPL preserves the format used by the MICE assembler but 'high-level' assembly statements are created to manipulate the stack (para. 4.4) or to control the interpretive driver (para 4.3). The use of such statements makes it more difficult to generate invalid statements or statements which would cause widespread corruption.

4.3 Interpretive driver

Since DIPL was concerned primarily with driving peripheral devices at the most primitive level, a generalised device driver has been written to accept a command program whose statements manipulate data buffers and action commands to the peripheral control modules (PCMs). The first facility this approach offers is to enable a new device to be driven with the minimum of development, using facilities already developed. In its defensive capacity it is able to detect faults and to report on them in a standard way, regardless of the nature of the device; this avoids replication of code with its possibility of error. The interpretive driver also helped the development of the ME29 system at a point when the method of conversation with peripherals was changed, by keeping those parts which were dependent on this change in a small part of the code.

It is intended to extend the interpretive capability to cover more facilities, such as block moves, sum-checks and redundancy checks.

4.4 Run-time stack

Temporary variables and procedural linkage information are held on a dynamic stack whose mechanism is similar to that of the ICL 2900 range. Appendix 2 shows this mechanism. The use of this stack will help the engineer to trace through a fault which defies normal resolution, as the stack will contain the previous values of variables which become overwritten during the course of loading, and also the procedure linkage information, so that if the program has stopped at an unexpected point it may be possible to trace how that point was reached.

4.5 Footprints

These are used as a simple trace and sanity check mechanism. Most run-time libraries attached to high level languages use a trace facility and it is usual in these cases to switch tracing off by suppressing the listing of the trace output file. This means that every run of a program can become a debugging run if the results are not as expected. Reporting faults in this way eliminates the need to construct a special program for debugging and also avoids the frustration one gets when the error refuses to recur on demand.

One way of tracing is to use a cyclic buffer which is not printed unless the program dumps itself. This method is inexpensive and is in principle similar to the in-flight recorders used in aircraft. The method used in DIPL is to divide the code into logical steps and to assign a footprint number to each step. At the beginning of each step a procedure is entered which plants the footprint number in the operator's display area, so that the operator is aware at all times of how far DIPL has progressed. Every error message contains the footprint number of the step which detected the error, so the engineer can point immediately to the section of code affected.

It is possible to keep a table of permitted sequences of footprints to check that the program is behaving sensibly; and to expand the footprint procedure so as to stop if an illegal sequence is being obeyed.

An alternative approach is suggested by Yau and Cheung,[1] using a 'baton-passing' technique by which a step finishes by planting a set value into a fixed location; the next step starts by checking that location for the correct value. If each step expects a unique value then an invalid sequence will fail before any damage is done.

A sanity check of DIPL has not yet been implemented, owing to space constraints.

4.6 Information dump area

This area is set up by DIPL during running and especially at the point of failure. When DIPL stops because of a failure the operator's panel log will contain a copy of the information in the dump area, so that an engineer may deduce the cause of the failure when it is reported by the operator. Typical items of information are:

device status
store address containing faulty information
expected value
observed value
footprint running at the point of interruption.

A manual describing the error messages (see para. 2.4(*c*)) will contain for each message the meaning of the values in the dump area.

4.7 Statistical analysis

A statistical program was used to give theoretical figures for resolution. The program is one that is used by test-program writers to present resolution and coverage figures for the units under test. DIPL was modelled for this purpose as a test program which performs tests in a fixed order and stops when an error is detected. If a test is completed successfully then the program can make assumptions about the coverage of any component and its probability of failure in a subsequent test. If a failure point has been reached the program uses the figures produced so far to estimate which components are likely to be at fault and suggests a spares kit to be 95% certain of repair. The output gives a list of figures for all tests and, at the end, estimates the overall probability of any test failing, the overall detection rate and the average size of spares kit required. The program makes no assumptions about any further information available to the engineer, so it has been necessary to insert additional tests to simulate the thought processes of the engineer. For example:

40 800AF Store and Check Read fails
Dump area: Failing Address
Expected Data
Received Data

would be modelled as

TEST 1	Both halves of data corrupt
TEST 2	Only least significant half corrupt
TEST 3	Only most significant half corrupt.

There are four components to be considered here:

Control store
OCP control
high order data
low order data.

For these tests the probabilities might be:

Component	Test 1	Test 2	Test 3
Control store	0·94	0·94	0·94
OCP control	0·06	0·02	0·02
high order data	0	0	0·04
low order data	0	0·04	0

This Table shows that if any of the tests fails the probability that the control store is at fault if 0•94, but if 0•95 certainty or more is required the engineer needs to look at the dump area to determine which two components to take with him to expect to repair the machine. In the case of Tests 2 and 3 he is expected to take the number of spares to give him the best chance of success, so for Test 2, for example, he might take a spare control store and low order data board, which gives a probability of 0•98; but if a spare OCP control board is available it will be worth while to bring it along to ensure a virtually certain repair.

The program is unable, without over complicating the model, to take advantage of all the information in the dump area, such as the failing address (to help the engineer to decide which control store board is suspect); therefore we expect to produce better resolution than the program might suggest. Like any other program it is only as good as the data that is provided, therefore it is important that the input be as accurate as possible. Some of the data is obtained from tables of 'mean time between failures' of components or estimated by counting the number of dilics (integrated circuits) on a PCB, or even by experience in the field of a particular device or component. It is not the author's intention to comment on the accuracy of the various methods of estimation except to state that a small variation in the figures may possibly affect the outcome by a significant amount. Therefore the use of a statistical tool to produce theoretical results must be checked in the field at the earliest opportunity. For this the designers need feedback from the field engineers.

4.8 Documentation aids

The most important and the most underrated aid in the program listing is the comment. Comments are vital in documenting assembly code programs, as the nature of the language permits us to see the particular only at the expense of the general. Comments are needed to redress the balance.

Most assemblers allow comments to occupy a complete line of text or alternatively to follow the operands of a statement. The author used the first format to present the code in a high-level algorithmic or plain language form, and the second to add extra information pertaining to the use of an instruction or sequence of instructions. In addition, each step in the code contains a comment in a box which links the footprint information in the supporting documents to the code. This is illustrated in the example of Appendix 1.

A further facility is to edit out all lines except the complete line comment to produce a design listing; it was then easy to check that

(*a*) The comments accurately reflect the code,
(*b*) the listings and the documentation do not conflict
(*c*) the engineer can follow the code from the documentation and vice versa.

4.9 File structure

Files use an executive-dependent structure to place and access them. DIPL attempts to be system-independent by accessing system files by means of the POFI. To assist in defence each block contains an identification string – 'VOLH' or 'POFI', for example – so that loading can terminate intelligently if the wrong block has been read.

Another defensive aid is the checksum of every block, to avoid undetected corruption of important data. Blocks that are moved from buffer to buffer are checked to ensure that information is not lost. Occasionally the data content can serve as checks for data corruption, examples are:

(*a*) address pointers being non-zero,
(*b*) disc addresses being within the range of the media,
(*c*) fixed length table entries containing their own (known) lengths,
(*d*) record headers containing some identifying word.

5 Results

DIPL has been developed at the same time as the rest of the system and has been useful in diagnosing faults during hardware development. However, it is vital that customers and customer engineers provide the implementors with information that will verify their estimates and enable them to provide an improved product in future releases.

A validation run was set up in which faults were injected into the M–1 hardware. DIPL reacted to the faults as follows.

	achieved	forecast
Successfully loaded exec. or test software	49	49
Success reported but load failed	0	0
Resolved fault correctly to 1 P.C.B.	84	108
Resolved fault correctly to one unit	87	110
Detected faults but could not resolve	65	24
Crashed or looped in DIPL with no error message	6	0
Total	291	291
Success rate (% of successful loads reported which were actually successful)	100	100
Resolution rate (% of detected faults resolved to 1 unit or better	71	90

Notes

(*a*) Definition of 'unit':
OCP (3 boards)
highway (3 boards)
Diagnostic cord (1 board)
Operators panel
Control store (2-4 boards)
Main store (4-16 boards)
Device & PCM (1-4 boards + drive).

(*b*) There were no media faults or mechanical faults injected into the system and there was no attempt to produce a statistical sample as would be found in the field. The faults were mainly generated by randomly selecting from the logic diagrams a number of signals to be 'grounded' to zero volts or allowed to 'float' by isolation of connectors.

(*c*) The 'achieved' figures are those obtained by the development system used for validation. The 'forecast' figures are those projected by the implementers for the system that is to be released to customers.

The extra space taken up by DIPL when it is loaded into M–1 control store makes it necessary to define an area of control store which does not contain information vital to dump analysis, therefore it is arranged that this area contains only code. Nondefensive loaders are usually sufficiently small to minimise this problem.

Let us now return to the problems outlined in the introduction and determine whether they have been answered.

(*a*) Every block of data is checked when it is read and again if it is moved in store; if DIPL reports a successful load then it is virtually certain that the store contains a correct image of the loaded program.
(*b*) There is no lower level software. The hardware is checked before and during use.
(*c*) Performance of DIPL is not so important as that of an emulator, so error reports are included. The only constraint is that of size.
(*d*) Checks have been consciously included.
(*e*) The comments on the listings are intended to help the engineer.
(*f*) The number of independent tests has been reduced to those necessary to check that M–1 will load and execute the DIPL ROM code. From the ROM code onwards any checks are part of the load process.
(*g*) No tools are removed from DIPL.
(*h*) The messages are specifically addressed to the operators and to the customer engineers.

6 Conclusions

It is the author's opinion that defensive programming should become second nature to programmers, like defensive driving: every program should be written with the assumption that the unusual can happen. A defensive program must have three aims:

(*a*) to perform its task correctly in a perfect environment;
(*b*) to inform the user of any exceptional circumstances;
(*c*) to provide aid for a systems programmer or an engineer if an obvious fault condition arises.

The following guidelines are presented in order to assist the writing of defensive software:

(*a*) Always design and plan the program assuming no trade-offs. It is easier to omit a facility at a later date than to try to insert it. A large number of enhancements lead to a condition known as architectural decay, where compromises made at each step produce a system that is neither good for the job for which it was originally written nor for the job it has been amended to do.

(*b*) Remember to whom your messages and supporting documents are addressed. Consider three kinds of addressees:

 a user who is using the program and is not expected to know how it performs;

 the author or support programmer;

 an 'ignorant expert' (programmer or engineer) who may be brought in to solve a problem detected by but not necessarily caused by the program. He is not expected to be a specialist on the program.

(*c*) Structure the program to help the expert and the support programmer. Tools such as footprints, procedural interfaces, stacks and standard error recovery methods help here. Imagine every run is a debugging run.

(*d*) Make sure the compilation listings contain enough comments to enable the engineer to link the program, its documentation and tools together.

(*e*) Remember that corners cut now will increase the amount of time spent supporting the program later; time consumed in site support costs the customer and ICL money and does not enhance the customer's image of ICL. Beware the 'I don't want it good, I want it Thursday' attitude.

(*f*) Do not assume that the hardware or lower level software is faultless. The hardest faults to trace are those not detected until the evidence has gone away.

(*g*) Check that tasks have been completed correctly.

(*h*) Do not remove any defensive tool unless absolutely necessary. When making trade-offs, count the cost of extra effort needed in support of a non-defensive program.

High-level language systems already follow many of these guidelines when a programmer is developing his software.

Error recovery and fault tolerance can be considered simply as extensions to the principles governing checking and detection, where the information that is gathered is processed by a machine instead of a man. The guidelines suggested are not restricted to a loading system.

A suggestion which is most useful when designing a defensive loader is to use the 'start small' approach described by Dent.[2] The first steps of the program are constrained to use the minimum set of system components so that if there is a failure the cause is suitably limited. If these first steps run successfully then further steps may test and use a wider set. DIPL uses 'start small' by checking that:

(*a*) the operator's panel can communicate with the operator and the diagnostic card;

(*b*) the DIPL ROM code is correctly loaded;

(*c*) the M–1 OCP functions correctly;
(*d*) theM–1 interrupt system functions correctly;
(*e*) the load device PCM functions correctly;

and so on.

The justification for defensively written utilities and applications programs is that corrupt data written by one program can cause wholesale corruption by being copied and trusted by others; and if one instance of corruption can go undetected for a long time, how many others are still undetected? Programs written in a high-level language can very easily be defensive if the run-time library contains standard checks such as array bound address ranges, and read-after-write, provided they are included by default. Once we have achieved this objective there has to be some justification to omit defence and that has to be weighed against the increased cost of error detection and of repair of a non-defensive program.

There is still the fact that defensive programs are inefficient. In practice it will be difficult for defensive emulators and executives to perform as quickly as those written non-defensively. If we use without proof of the rule that '90% of the time you are only using only 10% of the resources' then we can write a complete system defensively from the outset, with the expectation that at worst 10% will need to be traded against efficiency. With the greater emphasis on defensive techniques, micro-code or hardware assistance will be developed to improve the performance of defensive software rather than the removal of defence.

7 What next?

The immediate future development will cover an interpreter capable of supporting all the facilities needed for DIPL, containing within itself the defence which currently is explicitly coded. A target code will enable defensive programs to be written almost without the need for the writer to be conscious of defence; and in those cases where the writer is expected to supply some defence information it can be done with the minimum of effort.

Should the defensive interpreter live up to its expectations, the facilities may be increased to produce a defensive emulator or a microcode package to assist the implementation of defensive target machine programs. For this approach to be acceptable to customers, fault tolerance, error detection and recovery must receive as much emphasis as raw speed.

References

1 YAO, S.S. and CHEUNG, R.C.: 'Software error detection and correction', *Infotech State-of-the-Art Report 1978: System Reliability and Integrity.* Vol. 2, pp.96-121
2 DENT, J.C.: 'Diagnostic engineering requirements, Spring Joint Conferent, 1968, pp. 503-507.

Glossary

DIPL	Defensive Initial Program Load: an IPL (q.v.) coded in such a way as to give

	a measure of success if it runs to completion, and a prediction of a failing component if it should report a failure.
dump	The facility in DIPL to write the contents of store to a magnetic media device for future fault analysis.
emulator	A machine whose order code (known as microcode) does not itself mirror any particular physical machine but which may execute a program which presents the same interface to an executive (q.v.) or object-mode program as the machine which is being emulated (target machine, q.v.).
executive	Supervisory software which controls the running of user programs.
IPL	Initial Program Load: a program used to read an executive into a bare machine.
load	To read a program into a machine, whether or not supported by an executive. Here synonymous with IPL.
log	The operator's panel on a ME29 machine contains a store area (log) which can be displayed by a non-key operator on demand. This is used by DIPL to supply further information when a fault is detected.
M–1	The processor at the heart of the ME29 system. It is a machine which emulates the ICL 2903 code and is therefore capable of supporting a 2903-based executive. See also 'emulator'.
ME29	The system comprising: M–1 OCP associated peripherals 2903 emulator SUPREMO executive ICL-supplied utilities.
MICE	Microcode for M–1: the assembly language for ME29 microcode.
microcode	See 'emulator'
OCP	Order Code Processor
PCB	Printed Circuit Board, or simply 'Board': the smallest easily-exchangeable unit in a computer. An engineer will not be required to replace components on a board whilst on site.
PCM	Peripheral Control Module: a hardware module at the remote end of highway which controls one or more similar peripheral devices.
POFI	Primitive Operations File Index: see para. 2.3
ROM	Read-Only Memory: a nonvolatile memory device with information 'burnt in' to inhibit over-writing. In the case of DIPL integrated circuit ROM contains the first level bootstrap which is copied to the M–1 control store by the operator's panel firmware.
target machine	see 'emulator'
UDAS	Unified Direct Access Standard: see para. 2.3
VOLH	Volume Header Label: see para. 2.3

Appendix 1
Example of documentation footprints

```
Line no.
  10     ...   ***************************************************
  11     ...   description of footprints
  12     ...   all footprint boxes are enclosed in asterisks and start with three
               full stops
  13     ...   section number N is on a line which starts . . . N
  14     ...   section contents
  15    ...1      number of this footprint
  16    ...2      title
  17    ...3      hardware checks performed
  18    ...4      error messages and their meanings
  19    ...5      next footprint number
  20    ...6      detailed description
  21     ...   ***************************************************


2987     ...   ***************************************************
2988    ...1   7
2989    ...2   store write/read test, and control store and main store size check
2990    ...3   main store write then checkread
2991    ...4   40 00070 failure not reproduced on immediate retry but fails next
               time
2992     ...   40 80071 H3 fault
2993     ...            transmitted data wrong on write
2994     ...   40 80072 H3 fault
2995     ...            transmitted address wrong on write
2996     ...   40 80073 H3 fault
2997     ...            transmitted data wrong on read
2998     ...            but different in register
2999     ...   40 80074 H3 fault
3000     ...            transmitted address wrong on read
3001     ...   40 80075 H3 fault
3002     ...            store write/read and X2 reflect both fail
3003     ...   40 80076 store fault
3004     ...   40 80077 main store to control store read incorrect
3005    ...5   8
3006    ...6   Patterns 55555555 and AAAAAAAA are written to store and
3007     ...   then checkread. If error occurs, the check is repeated
3008     ...   but resolution is attempted by checking the contents of
3009     ...   the H3 registers:
3010     ...      transmitted data
3011     ...      transmitted address
3012     ...   on write and read, and then if necessary doing an X2
3013     ...   reflect to register FA.
3014     ...   A read from main store to control store is then
3015     ...   performed and checked.
```

```
3016    ...  The sizes of control store and main store are then found by writing
3017    ...  to and reading from each 4K word module until incorrect data is
3018    ...  read or an address error interrupt is received.
3019    ...  ****************************************************
3020    ...
```

Appendix 2
Stack mechanism

The standard names used in the 2900 range have been used where there is an obvious similarity.

The push-down stack contains a series of *local name spaces* which refer to an active procedure. Each local name contains:

(*a*) a pointer to the local name space of the procedure which is calling this one;
(*b*) the link for returning to the calling procedure;
(*c*) parameters passed to this procedure;
(*d*) local data for this procedure;
(*e*) temporary data.

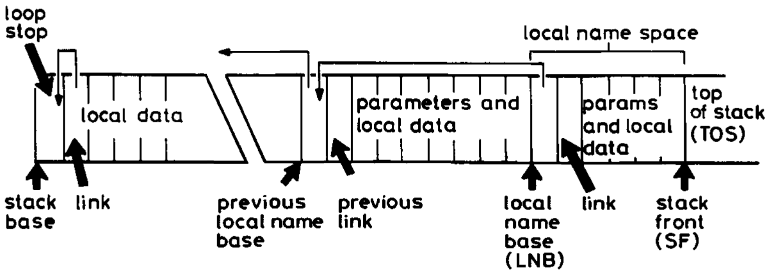


Fig. 2 Stack mechanism

Fig. 2.1 Layout

The lowest level is artificially set up to point to a loop stop, so that if the stack is emptied in error the program will stop.

A procedure call has the following effect on the stack: before the call the situation is as in Fig. 2.2. The *local name base* pointer LNB is stacked at *top of stack* TOS; the return link is constructed and stack at TOS. The parameters are evaluated in order and stacked. LNB is set to point to the stacked value of LNB.

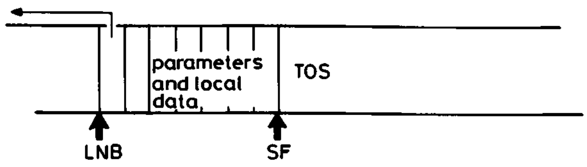


Fig. 2.2 Before the call

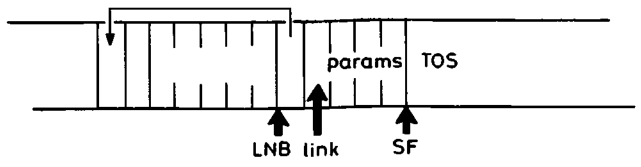


Fig. 2.3 On entry

The new procedure is entered. During execution of this, local data may be stacked, or temporary variables or a further procedure may be called.

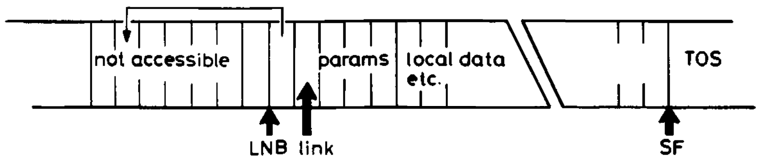


Fig. 2.4 Inside the called procedure

When the procedure is left, the pointers LNB and SF are returned to their values before the call and all information above TOS can be overwritten by further uses of the stack.

Acknowledgments

The techniques by which DIPL was written were prepared with the help of the following colleagues of the author: J.A. Bull, I.R. Greenaway, R.J. Hurst, C.J. Playford and W. Terrington. The author also wishes to thank M. Small for his statistics program.

Project Little – an experimental ultra reliable system

J.B. Brenner, C.P. Burton, A.D. Kitto and E.C.P. Portman
Advanced Development Centre, Technology Division, ICL, Manchester

Abstract

An approach to the design of simple, highly *reliable*, general purpose computer systems is described. An experimental *distributed* system has been implemented to examine the approach critically. This experimental system uses five single-board microcomputers, which send messages to each other via a bit-serial, contention-arbitrated bus based on Ethernet principles.

A representative user interface has been provided to allow application programs implemented in Pascal to be run.

The experimental system demonstrates *user concurrency* in accessing files, *fault containment* and *resilience*.

1 Introduction

1.1 Who are we?

Advanced Development Centre of Technology Division was set up in June 1979 to investigate future opportunities in a wide range of areas of interest to ICL with particular reference to the impact of technological advance on system structures and applications. The work described in this paper stems directly from the application of lower cost computing and storage hardware which is now becoming available to the solution of problems common in the computing business. The Centre includes engineers, programmers and system designers most of whom have well over ten years of experience.

1.2 Project 1980

The experimental ultrareliable system described here is one of the outcomes of research work started several years ago, as a result of two guesses:

a guess that it would be possible to devise technical means of achieving 'perfect' D.P. system reliability, availability and maintainability;

a guess that extrapolation of hardware technology trends would bring drastic changes of capability and cost, which might make such techniques practicable for everyday use.

We set out to discover, via a series of paper studies, how this might be done. This we called 'Project 1980' because at the time that seemed suitably futuristic.

A basic principle of this work has been the belief that computer hardware and software systems are unreliable mainly because they are complex, not because of intrinsic hardware unreliability. Therefore we see *simplicity* as being the key to reliability, by making systems easier to design correctly, and to test, diagnose and repair.

We forsaw that future abundance and cheapness of hardware technology had the potential to allow simplification by use of what we called the *'enough principle'.* If there were to be 'enough' mill, storage and interconnection bandwidth, then utilisation could be low enough to avoid some of the complexities of queueing, scheduling and performance tuning. If this abundance were cheap enough, then complex provisions to use it efficiently might become unjustifiable – e.g., multiprogramming, virtual storage and shared code.

We also speculated that this hardware might readily consist of large quantities of relatively simple computers. Each such *computer module* would be a self-contained functional and physical unit, comprising processor and storage and I/O in a simple physical package. To reinforce the modular separateness of each computer module, communications with and among them would always be by consenting exchange of messages via *high level protocols* on *bit serial interconnections.* This introduces further potential for simplicity, reliability, resilience and redundancy.

We then went on to propose a *federated system* style of control, in which computer modules all have essentially equal status, so that there is no critical failure dependency upon particular computers having some special position as master. With this go concepts of *distributed control* and local autonomy, plus the *location independence* of software functions, so that they can migrate to alternative hardware if there is a failure. There is a corresponding requirement for *data consistency,* such that data can be partitioned and replicated in different parts of the system to provide redundant and *resilient storage* of control information, programs and user data.

Software functions would be dispersed into *small modules*, communicating via high level protocols, in a way exactly equivalent to the hardware modularity, and with the same simplicity requirement. Comprehensive *fault containment* and *recovery* is also required.

The key to this distributed control, data consistency, fault containment and recoverability is the use of the *Commitment Unit* concept. We have described this in a previous paper in *ICL Technical Journal.*[1]

These various concepts are now to different degrees becoming visible in the guise of distributed systems, the electronic office and personal computers, but not (yet) in conventional mainframe D.P. systems. Neither have they yet been generally applied to achieve ultrareliability, as originally envisaged.

1.3 Systems modelling experiment

As a result of the paper studies conducted under Project 1980 and the subsequent discussions with logic design engineering colleagues in Technology Division it became apparent in September 1977 that a major stumbling block to the credibility of the proposed approach lay in the feasibility of implementing the system software to support the proposed techniques. Doubts were expressed both of the practicability of implementing a system so different from those with which we had experience and also of the technical viability since system resource demands might inhibit performance to an unacceptable degree. Accordingly, it was decided to simulate a minimal subset of the system on an existing service computer system and hence get a qualitative impression of the difficulties involved in detailing the design and implementing the chosen subset. It was also intended to use the modelled system to support some (simulated) workloads and to measure the activities in the various subsystems to answer the doubts about unacceptable overheads.

This work was started in October 1977. Two persons were involved part time in the system design and coding. A third person analysed part of the ICL corporate benchmark and provided synthetic workloads and the interpreter needed to execute them in the new environment. The programs were coded in Pascal and executed under the George 3 operating system on our 1904S service at West Gorton. We found that the facilities provided by George 3 for Communication Files formed a very suitable environment for linking the several concurrently executing programs together. By February 1978 the system was starting to function and it was possible to demonstrate its basic features. It was at this time that we decided to move on directly to a hardware-based experiment to provide a convincing demonstration at the earliest date rather than complete the software-based System Modelling that had been planned.

2 Pilot machine structure

2.1 System requirements

We wanted to show that use of the principles of 'simplicity' and 'enough' supported by the commitment unit concept enabled the development of a computing system which functioned, was able to contain errors and was able to recover from them. We also wished to show that this could be achieved at reasonable cost both in development and manufacturing terms. The system was therefore required to contain real computers that could be programmed to perform applications. It needed to support some sort of input of data and display of results and contain a 'file'-storage facility in which both data and program material could be kept. Since our principles implied a multicomputer system it should contain several computers linked by a 'loose coupling' and at least two separate storage devices which could be used for file backup when required. The computers supporting the storage devices would be responsible for maintaining records of all changes to the content and status of the files for which they were responsible, so that in the event of failure in the relevant commitment unit the changes could be undone. This information should be stored in a Local Integrity Journal in the volume whose changes it records. In order to use the system, the programs had to be compiled and placed in the system storage, and 'tools' to enable this had to be provided. The system software needed to be able to load and supervise the execution of application programs, support the usual file storage facilities – create, open, write, read, close and delete – in the commitment unit environment that we envisaged, as well as the less usual facilities of automatic, nonprocedural record level locking and file rollback. Finally the system software needed to support the recovery of jobs whose supporting hardware or software had malfunctioned and was no longer active in the system.

2.2 Constraints and limitations

We decided to build a system which, while capable of doing 'real work', was constrained in various ways to make the implementation more sure, less costly and more timely. Microcomputers were used to minimise hardware design and building costs. Pascal, with which we were all familiar, was to be the only programming language used for the application software and where possible for the system software. A monitor program was written in Intel assembly language and the Pascal Interpreter was written in PL/M. Random access storage was provided by flexible discs and hard copy output via a 30 c.p.s. hard copy terminal. Standard visual display terminals were attached via RS232 interfaces. In the software developed we took a number of short cuts which limit the utility of the present Project Little system as a general purpose computing system.

2.3 Hardware

The insistence on simplicity indicated that each computer module should be no more elaborate than could be packed into one physical unit, which we chose to be one standard 300 x 200 mm printed circuit board. This decision freed us from secondary decisions such as the size of racks and cabinets, physical expansion capability, power supply sizing etc., which would have had to be made if a more flexible approach to the computer module design was taken. To simplify the intermodule communication hardware, a straightforward 'open collector' bus type of connection was adopted, able to support a total bus length of only ten or so metres.

A search at that time, both in-house and OEM, to find a possible candidate to meet all our single-board computer module requirements, was fruitless. Consequently we designed and manufactured our own module, although the start-up costs of such an approach put a high overhead on the cost of the small number of modules which we needed.

Taking advantage of the then currently available LSI technology, it was possible to design the module with the following features:

type 8085 CPU using 3·072 MHz clock;
Direct Memory Access (DMA) controller with four channels;
16 kbyte of EPROM (2 kbyte increments);
64 kbyte dynamic RAM with no refreshing overhead;
6 programmable counter/timers;

type 2651 programmable Universal Synchronous/Asynchronous Receiver/Transmitter (USART) for intermodule bus communication;
Two hexadecimal digits display;
eight sense switches;
reset and interrupt pushbuttons;
two serial I/O channels;
1-byte, software-readable, changeable wired constant;
support logic for the intermodule bus, including a relay;
space for logic for personalising the module to support an I/O device. For example, several modules have a controller and interface logic for doublesided flexible discs.

The four channels of the DMA controller were allocated to the I/O device (e.g., flexible disc), to the USART transmit channel, the USART receive channel, and to the automatic recharge channel for the latter.

The 2651 USART was chosen because of its high speed – we use it at 768 kbaud, in synchronous mode. A phase-locked clock divider extracts the clocking instants from the intermodule bus data. By using crystal-controlled transmission at nominally the same frequency in all modules, this scheme allows error-free reception for the message sizes and data content used in our intermodule bus protocol.

The direct store addressing capacity of the 8085 is 64 kbyte. The first 16 kbyte of this space is program selectable to be either the EPROM or the first 16 kbyte of the RAM.

It was important that no individual hardware failure could prevent the rest of the system working. The outline of the intermodule bus interface (Fig. 1) shows a relay contact at the boundary between the system (i.e., the bus) and the module. Opening the relay isolates the (possibly faulty) module from the system. We accept for the purpose of the experiment that the relay contacts and the copper wire bus connections are sufficiently reliable to cause no significant problem. To ensure that only healthy modules remain connected to the bus, the relay will open after, for example, ten seconds, unless it is retriggered by the software. In the isolated state the software can test all the intermodule bus interface hardware out as far as the relay contact, prior to re-establishing connection to the bus.

The original intention was that each module should have its own small power supply unit. However, all experiments so far have used a bulk supply for all modules.

2.4 System software

2.4.1 Overview: The system software in each computer is divided into two parts, the support-level software and the control program. The support-level software is hardware dependent and is common to each of the computers whilst the control program is machine independent and characterises the particular service that each type of computer provides.

The control programs were written in Sequential Pascal and a large part of the support-level software, (written in PL/M), is an interpreter based upon the Concurrent Pascal interpreter of P. Brinch Hansen.[2] The rest of the support level software, called the kernel, is concerned with providing a simple interface to the hardware, mostly the transport mechanism for intercomputer communication. The contents of the messages sent between the computers is the responsibility of the control programs.

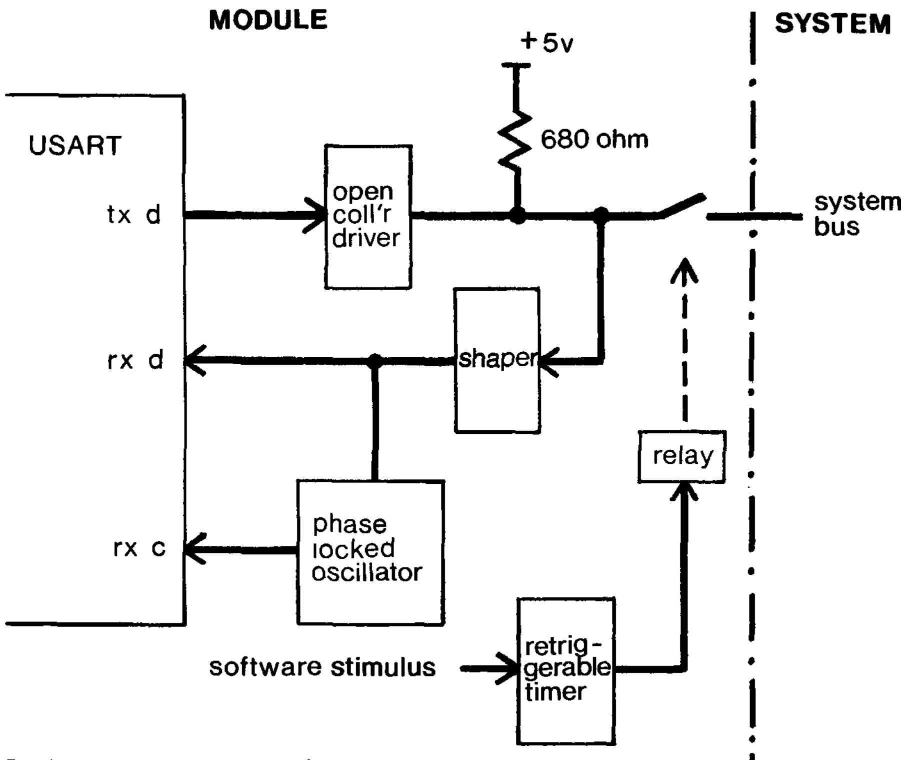


Fig. 1 Intermodule bus interface

A particular feature of Sequential Pascal is the program prefix. This is the mechanism by which an interface between the ideal machine and the real machine is defined in order that the Pascal programmer may manipulate the real machine resources. The kernel supports the prefix procedures defined for the control programs but a novel feature of our implementation is that a control program itself may also support a higher level of prefix procedures for use by user level programs. This provides a level of protection as user programs cannot access the message-generating code directly.

2.4.2 The kernel: The prefix procedures supported by the kernel include facilities to manipulate the Pascal heap, to transmit and receive data from local peripherals, to access timer facilities and provide the ability to transmit to and receive messages from other computers attached to the system bus. The code for this communications facility constitutes the majority of the kernel code and the mechanisms are described here in greater detail.

Three actions are necessary in order that a control program can start receiving messages from another computer. The control program must specify its own name, it must provide a set of buffers to receive the messages and it must cause the system bus relay to be closed to connect the computer to the system bus.

The control program provides an eight-character string which the kernel compares with the destination address of each message it receives, discarding those messages which do not match the name provided by the control program. This name can be changed at any time by the control program and may represent such things as the name of the disc volume which the control program manages.

As many buffers as required are provided by the control program to receive messages from other computers. The kernel initialises the DMA and the USART, so that any messages received are written into the control program buffers.

The control program indicates to the kernel when it wants the relay to connect the computer to the system bus. If the relay is to remain closed the control program must call the kernel at least once every ten seconds otherwise the computer is disconnected from the system bus. This safeguards against hardware or software failure causing the bus to be jammed.

The control program may also cause a message to be transmitted by calling the kernel which will dispatch it once the ability to receive messages has been established. The process of deciding which processor is to use the system bus next, that is the arbitration process, is distributed amongst all the participant processors and is based upon clash detection in the manner of Ethernet.[3] The kernel software is organised to detect clashes and to recover in an efficient manner by terminating its transmission quickly. If no clash occurs a message is sent out in full and on completion each computer which is connected will be interrupted by its DMA interrupt routine being entered.

There is no interrupt mechanism provided between the kernel and the control program, therefore the kernel must inform the control program of hardware activity via software flags associated with each message buffer and the control program must poll these flags.

The kernel is responsible for ensuring that messages passed to the control program contain no hardware-detected transmission errors. It is also responsible for detecting clash conditions when two computers try to send messages at the same time and to recover from them in an efficient manner (i.e. clash conditions are invisible to the control program). It is the control program's responsibility to recover from situations where messages are lost or misdirected when there is no hardware error detected within the computer supporting the control program. This is achieved by timeouts and message sequence numbers.

2.4.3 The control programs and their protocols: Two kinds of control program have been implemented, one to control processing and the other to control storage computers. These have been named EXEC and SVOL, respectively. Each EXEC supports one user program at a time, providing it with facilities similar to a typical microcomputer disc operating system, but with the additional facilities of check-pointing and concurrent shared access to the filestore managed by the SVOL computers. Each SVOL manages only one storage device but can converse with many active EXECs, creating and manipulating files on their behalf. The SVOL control program offers the normal file facilities of create, delete, open, close, read and write. In addition it provides data recovery for files, by automatic recording of before-looks of updated records, and file- and block-level locking. These additional facilities may, as a user option, be omitted for any file, with a consequential saving in overheads in exchange for the reduced level of security.

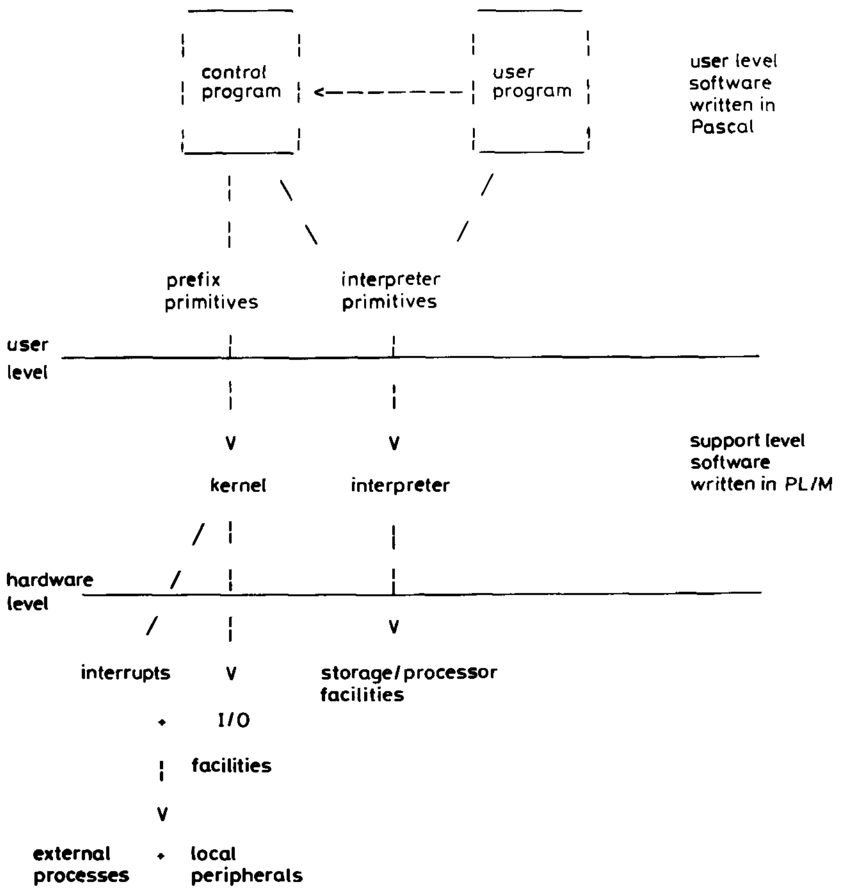


Fig. 2 Software structure

The underlying mechanism which allows EXECs and SVOLS to co-operate in safety is based upon the concept of commitment units which is at the heart of the system we are describing. When a user program starts, EXEC establishes a commitment unit series with each SVOL by sending a commitment unit introduction message which includes the starting transaction number for this series. SVOL will only allow the series to commence if the number is unique and, after commencement, will only allow EXEC to make file requests related to that number. File locks and recovery journals are related to the transaction number and in the event of the failure of an EXEC computer or the occurrence of deadlock situations, due to mutual exclusion of user programs from particular files, these trasaction numbers can be used to implement the necessary actions for recovery.

For performance reasons details of file manipulation may first be recorded in volatile store and only later recorded permanently in the Local Integrity Journal. The exact implementation of SVOL is invisible to EXEC but EXEC can ensure all its actions are made permanent by sending the SVOL a 'secure transaction' message. This is the first phase of the 'two-phase commit' sequence and EXEC may send such messages to a number of SVOLs before going on to the second phase. The second phase involves sending each SVOL a 'commit transaction' message which moves the commitment series to a new transaction number. The SVOL action on receiving a valid commit message is to release any file locks and to discard recovery journals associated with the old transaction number.

If at any point SVOL is involved in a failure causing data held in volatile store to be lost, the SVOL will be able to return to the previous secured and commited state. Also EXEC itself may request SVOL to revert to the previous secured and commited state by sending a 'rollback transaction' message which will cause the release of any associated file locks and the implementation of any associated data recovery as recorded in its journals. It should be noted that commitment series in a 'secured' state are immune to SVOL failures causing corruption of volatile store and EXEC has the choice to 'rollback' or 'commit' them as it chooses.

2.4.5 Job control: EXEC makes use of SVOL file facilities in order to provide job control and recovery as well as to meet the individual demands of the user programs. A shared file called 'tasklist' is maintained by each EXEC. It contains details of the user programs being run, including job parameters, program code file name and the last known status of the commitment series with respect to each SVOL. This file is duplicated on at least two SVOLs so that loss of one copy will not cause loss of control. In the event of EXEC hardware failure other EXECs may take over jobs detailed in the 'tasklist'.

The current implementation of EXEC only allows job initiation from a terminal attached to the EXEC computer but the 'tasklist' implementation will permit future versions of EXEC to provide 'batch facilities' as well as the current interactive facilities.

2.4.6 User interface and application software: The experiment was to model a general purpose system, consequently the user interface was not designed for any specific applications but simply to allow user programs to take advantage of the special system features, namely file sharing and recovery.

EXEC allows user programs to create files which may be recoverable after failure and which may be shareable with other programs. If files have shared access they can have locking facilities at file- and block-level. The files may instead be simply manipulated in the manner of a single-user computer system.

Two special procedure calls are provided to invoke checkpoint facilities:

CHECKPOINT causes EXEC to *commit* all filestore changes made by this transaction since the last checkpoint, or start of program if no previous checkpoint has been actioned;

ROLLBACK causes EXEC to request all filestore changes made since the last checkpoint to be undone.

Programs can ignore these special facilities in which case any filestore changes are automatically commited by EXEC when the program terminates normally or else automatically rolled back if the program terminates abnormally.

A number of application programs have been run on the system, either written especially for the system or originally for other systems. Two written for the system are described here. The purpose of the system development was to gain experience of implementing reliable systems and to demonstrate their abilities so there was a need for demonstration programs. The demonstration programs were to show that the system functioned, provided concurrent access to filestore, could recover from errors and would demonstrate fault containment to one computer. Performance was of much less importance for this experimental 'model' system.

The theme for these programs was that one or more files were modified by a number of separate programs which might also maintain separate files which needed to be kept consistent with the commonly accessed files. The simpler of the two programs was a 'cash transfer' program which allowed a number of bank balances to be credited or debited according to input from an interactive terminal. The records were maintained in one central file of which a duplicate copy was maintained. In the event of two transactions clashing in accessing a bank record they would automatically backoff and try again. In the event of the loss of one copy of the file, file updates would be suspended but balance enquiries would continue to be supported whilst a new second copy of the file was made available.

The second program was a 'Transport Pool Management' program. This manipulated a duplicated central file holding details of cars in a transport pool together with details concerning their maintenance and current ownership. In addition there were separate files for each department hiring cars from the pool which contained details of their present car holding together with their maintenance requirements. Cars could be hired from the transport pool and returned on or before a specified date. On return the files were updated to include the department's maintenance requirements. This program provided facilities to simulate hardware failure during critical periods when the various files were in an inconsistent state and to check the consistency of the files after error-recovery action had been taken.

These two programs have been demonstrated to run with a wide variety of system faults present, showing the system features that they were intended to show, i.e., concurrency, error recovery and fault containment. In addition they have provided initial information on the suitability of the user interface. One particular aspect was the choice of making the provision of duplex file copies a user responsibility. This was easily handled but there could be benefits in the provision of an automatic duplex file facility by EXEC. Any future study will involve providing an environment in which programs written for a normal environment can be made to make full use of the system's special resilience features without themselves being modified.

3 Development experience

The elapsed time of development described in this paper has been approximately 20 months but was spasmodic until the last five months, mid-July to mid-December 1979. The hardware design and commissioning took ten man-months, whilst the provision of software tools and implementation of the support level software took about the same amount of effort. These two activities were supported by four individuals. From July 1979 onwards a further eight man-months' effort was used in integrating the system. The control programs were based upon those written for the software-modelling exercise.

The system simplicity and modularity contributed substantially to the ease of development. The hardware development was progressed with a dedicated monitor program in EPROM which was retained in the final version of the computer. The support level software was developed in a modular fashion with separate development programs to test the message transport mechanism, Pascal interpreter etc.

The dedicated monitor EPROM was 2 kbytes in size and provided facilities to manipulate all the hardware on the board via a video keyboard as well as retrieving and storing programs on the flexible disc drives. The support level software totalled just more than 8 kbytes of which 6 kbytes was the Pascal interpreter. A bootstrap written in Pascal used to load control programs from a local disc drive or from another SVOL was programmed into 4 kbytes of ROM. This bootstrap also displays diagnostic messages in the event of control program failure.

The SVOL control program source was 2000 lines and compiled to under 20 kbytes. A test control program acting as an EXEC was implemented to test SVOL in a systematic manner, allowing SVOL to be fully tested when only two single-board computers had been commissioned. The EXEC control program source was 1500 lines and compiled to 15 kbytes. Test application programs were implemented to exercise the user interface and test the EXEC ability to detect and report user program failures.

During the last six months development was made particularly easy by the modular nature of the system and the only bottleneck was the reliability and availability of the mainframe link for down-loading of the control programs. There was sufficient opportunity for parallel activity that the entire system could have been developed in less than nine months from the end of the software modelling experiment.

The system performance was seriously impeded by the use of flexible disc drives; the slow speed of the Pascal interpreter was another significant performance factor.

The Pascal interpreter executes at about 1250 statements per second (4 to 5 op codes per statement). EXEC accesses file data in 256-byte blocks and each SVOL can supply three blocks per second or update two blocks per second on behalf of EXEC when no before-look journalising is involved. When journalising and block locking is involved file updates take almost two seconds per block. The current configuration of three EXECs and two SVOLs imposes only a very light load on the intercomputer transport mechanism which is capable of supporting 200 block transfers per second.

There are many ways of improving the performance such as the use of hard discs and by track buffering in RAM, also interpretation can be replaced by direct code execution. The object of the development exercise however was not to provide a high-performance system but to provide one which had the extra reliability features without significant extra overheads. The present system overheads for data recovery and checkpointing are not so great as to indicate the impracticability of the principles and further study should indicate more efficient control program message protocols. The current development has provided us with a system which is easy to program, to modify and to expand and will support such studies.

4 Conclusions

We have shown by this experiment that implementation of a small system based on our principles is feasible in a reasonable time with limited resources. We have shown that loosely coupled computers can co-operate in performing a particular type of workload with reasonable overheads and that many types of fault are contained by this system, allowing other work to proceed. We have shown that jobs which have suffered a failure may be returned to a previous state very simply and that they may then be restarted on the same or different hardware.

We have *not* shown that these provisions will support all types of workloads *nor* that a general-purpose operating environment such as VME/B with IDMS can be implemented on such a substrate – although we believe that it can. We have *not* shown that multiple eight-bit microcomputers are an adequate substitute for a medium-scale mainframe *nor* do we believe that to be generally true.

We intend to continue this work using our system. There is some tidying up to do and there are several interesting lines to follow. A fuller control environment supporting Batch, MAC and online applications will be developed. We shall extend the system to allow programs in languages other than Pascal. COBOL has some obvious attractions. We would like to attach hard discs as well as improve our implementation of SVOL with flexible discs. We have a high-resolution raster-scanned VDU which we intend to attach to provide text-and diagram-processing facilities. Further applications will also be written which will allow us to exercise the concurrent execution properties of our system usefully.

References

1 BRENNER, J.B.: 'A general model for integrity control', *ICL Tecn. J.* 1978, 1, 71.
2 BRINCH-HANSEN, P.: *The architecture of concurrent programs,* Prentice-Hall, London, 1977.
3 METCALF, R.M. and BOGGS, D.R.: *Commun. ACM,* 1976, 19, 39.

Flow of instructions through a pipelined processor

W.F.Wood

ICL Product Development Group, Technology Division, Manchester

Abstract

The paper describes the treatment of pipelined processors by both analytic and discrete-event simulation models and emphasises the importance of analytical models in decision making. The results obtained show that intrinsic delays in a pipeline may easily be erroneously attributed to extraneous factors such as store-accessing delays or jumps in instruction sequences.

1 Introduction: basic ideas on pipelines

A characteristic of many large computers or Order Code Processors is the simultaneous execution of several instructions. A common way of organising this is the use of a 'linear pipeline'; in this the execution of several instructions from a sequence is overlapped, so that at any instant each instruction is undergoing a different phase of its execution. The processor then consists of a number of distinct functional units, each corresponding to a different phase of execution.

An example of a linear pipelined machine is the ICL 2980. Its division into functional units depends on 2900 System architecture, in which each batch job or terminal session has its own set of virtual store segments. Access to these is obtained via 'descriptors' which contain information about the location and nature of data. Included within each set of virtual store segments is one segment, known as the primary operand segment, to which direct access is possible. This segment may be used as a last-in-first-out stack containing frequently-used operands and descriptors. It frequently happens that a descriptor in the primary operand segment is used to access data in other segments. Thus there are two phases in the execution where store access for operands may take place, and computation may be necessary for stack manipulation and descriptor modification as well as for the performance of operations explicitly described in user procedures.

The pipeline stages within the 2980 processor correspond to the following tasks:

(*a*) instruction fetch;
(*b*) instruction decode and the generation of primary operand address;
(*c*) primary operand fetch;
(*d*) generation and modification of addresses for indirect operands;
(*e*) indirect operand fetch;
(*f*) string handling and performance of arithmetic on operands;
(*g*) writing to store.

Linear pipelines are a feature of other 2900 machines also. These do not all follow closely the 2900 architecture as defined at the machine instruction level, or more precisely the primitive interface level, PLI. The instructions passed down the pipeline may be not PLI instructions but instructions which provide an interpretation of a PLI sequence.

2 Prima facie expectations of performance

We may expect *prima facie* that in the absence of delays due to extraneous factors the throughput of a pipelined processor is governed by the 'instruction rate per stage': one instruction leaves the last stage at this rate, while every other stage passes an instruction along the pipeline. The first evident drawback is that the tasks performed at the various stages do not take the same time. Recognising this leads to the view that the intrinsic rate of instruction processing is that of the slowest stage – the 'bottleneck'. This is a common view of pipeline processor throughput; for example, it is implicit in the equations for pipeline efficiency given by Ramanoothy and Li[1].

This manner of looking at throughput would be justified if the time to process an instruction were constant at each stage. In practice what tends to happen is that a high proportion of instructions are dealt with relatively quickly whilst progressively smaller numbers take longer and longer. For example, in a machine with a stack, code will be prepared in such a way that as far as possible access is made to the top of the stack or to other locations which are easily accessible by the use of pointers. Some instructions may however require arithmetical operations to be performed before the required location is computed, others may require stack manipulations and relatively infrequently a complete change of stack will be necessary.

Since instruction time per stage is not constant even the slowest stage is not always busy, and stages with a short average instruction time will sometimes cause holdups. Consequently the throughput is not the rate of the slowest stage but is determined by the products of the mean instruction time for a stage and the probability that that stage is usefully busy.

The danger in ignoring the variability in instruction time per stage is that, arguing *a posteriori* from the difference between calculated and observed performance, external factors such as store accessing or the presence of jump instructions will be held entirely responsible for failure to meet the 'theoretical' figure. Delays in accessing store do of course reduce processor performance and jumps can have a particularly important effect in a pipelined processor because a number of instructions following a conditional jump may already be partly executed when the jump instruction is executed. But if intrinsic pipeline delays are ignored the effects of these will be exaggerated and other possible ways of improving performance may be overlooked. For example, one way of reducing the idle time of pipeline stages is to place buffers between stages; if some sources of delay are ignored then the optimum placement of buffers will not be achieved.

Because there is overlap between the different types of delay, a reduction in any one will not necessarily produce a correspondingly large improvement in performance. Recognition of the fact that instruction time per stage is not constant is more realistic and more satisfactory theoretically than the 'bottleneck' approach.

3 How instruction time variability may influence throughput

In 1977 an analysis[2] was carried out to estimate the performance of a four-stage linear pipelined processor based on what was known in ICL as the 'primitive processor'. The primitive processor was developed as a general processor for use throughout the system, that is both within and without the order code processors (OCPs). An OCP could include one or more primitive processors, together with other units such as slave stores and decoders.[3] As part of the task of estimating system performance an estimate of the intrisic performance of the OCP was required.

The state of a pipelined processor is determined by the states of its component stages. In the absence of store delays and jump instructions a stage may be:

(*a*) busy (1), i.e. actively processing an instruction;
(*b*) waiting (0), i.e. awaiting the next instruction;
(*c*) blocked (b), i.e. unable to pass an instruction.

If there are no buffers between stages a stage will be blocked if on completing its work on an instruction it finds the next stage busy or blocked. In short, if there are no complicating factors such as conditional jumps a linear pipeline may be treated as a sequential service line.[4]

For the purposes of a theoretical analysis we have to make some assumptions about the distribution of instruction times, and these must be both in reasonable accordance with what we know about these times in real machines and such as to enable us to make some progress with the mathematical treatment. We make two assumptions here.

First, that the behaviour of successive instructions in the pipeline is statistically independent of that of other instructions, and that the times taken to service the same instruction at different stages in the pipeline are also statistically independent.

Second, that the instruction times have what is called the Markovian character:[5] that is, that if work on an instruction has already been in progress for a time t then the probability that this will terminate in the next small interval $\mathrm{d}t$ is proportional to $\mathrm{d}t$ and is independent of t. In other words, the time the system has spent in a particular state plays no role in th calculations. This leads to the negative exponential distribution of instruction times, as follows.

Let $P(t)$ be the probability that a stage is still busy working on an instruction at a time t after receiving that instruction; this is the same as saying that the instruction time is at least t. The probability that this will terminate in the interval $(t, t + \mathrm{d}t)$ is, by the assumption, $\mu\, \mathrm{d}t$ where μ is a constant, so the probability that it will not terminate in this interval is $1 - \mu\, \mathrm{d}t$. Therefore the probability that the stage is still busy at time $t + \mathrm{d}t$ is $P(t)[1 - \mu \mathrm{d}t]$ But this is $P(t + \mathrm{d}t)$, so we have

$$P(t + \mathrm{d}t) - P(t) = -\mu P(t)\,\mathrm{d}t. \tag{1}$$

Hence $\mathrm{d}P/\mathrm{d}t = -\mu P$ and $P(t) = A\mathrm{e}^{-\mu t}$ and because all instructions must have a time greater than zero, the constant of integration A must be 1 and

$$P(t) = \mathrm{e}^{-\mu t}. \tag{2}$$

This is the negative exponential distribution.

The proportion of instructions which have a life of between t and $t + \mathrm{d}t$ is $P(t) - P(t + \mathrm{d}t) = \mu P(t) = \mu \mathrm{e}^{-\mu t}$, so the mean time is

$$t_m = \int_0^\infty \mu t\, \mathrm{e}^{-\mu t}\mathrm{d}t = 1/\mu,$$

i.e. if the mean time for an instruction is t_m then

$$\mu = 1/t_m. \tag{3}$$

The Markovian assumption enables us to set up a mathematical model of the system which we can handle. The negative exponential distribution of instruction times is in reasonable accord with that we observe, which is that short instruction times are relatively frequent and longer times progressively less frequent. Also, experience of analysis of queuing systems (of which this is an example) shows that the assumption of negative exponential distribution of things like waiting times gives at least qualitatively sound results in a very wide range of circumstances.

All changes in the pipeline result from a busy stage completing an instruction, and as the rate at which a busy stage completes instructions is constant (the Markovian property assumed), a statistically steady state exists. This steady state can be calculated from a set of dynamic equilibrium equations which show how the probability of leaving any particular state is balanced by the probability of reaching it from all other possible states. We can now set up these equations.

As we are considering the intrinsic performance of the pipeline we may ignore the effects of any delays in accessing the store and regard the first stage, the 'instruction fetch' stage, as the generator of tasks for the later stages. The rate and distribution of instruction arrivals is then determined by the service-time characteristics of the first stage. The state of the pipeline, because of this decision regarding the first stage, is that of the three following stages and may therefore be represented by a triplet of symbols, each giving the state of one stage. Thus $(b,1,0)$ represents the state in which the second stage is blocked, the third is busy and the fourth is waiting. We denote the probability that the system is in this state by P_{b10} and correspondingly for other states.

Two factors simplify the calculation:

(*a*) in the absence of extra-processor holdups the final stage is never blocked; so no state $(*,*,b)$ can arise and therefore $P_{**b} = 0$. Here of course *can be any of 1, 0 or b;

(*b*) for this particular machine the first (instruction fetch) stage had a quicker and more even rate of processing than the following stages, because of the simpler tasks which it performed and the presence of slave registers. We were therefore able to ignore the possibility that the first stage kept the second stage waiting, so no state (0,*,*) can arise and $P_{0**} = 0$.

The possible transition paths between the various stages are shown diagrammatically in Fig. 1. The constants μ_2, μ_3, μ_4 are the mean instruction rates for the second, third and fourth stages, respectively; these are the reciprocals of the corresponding mean instruction times.

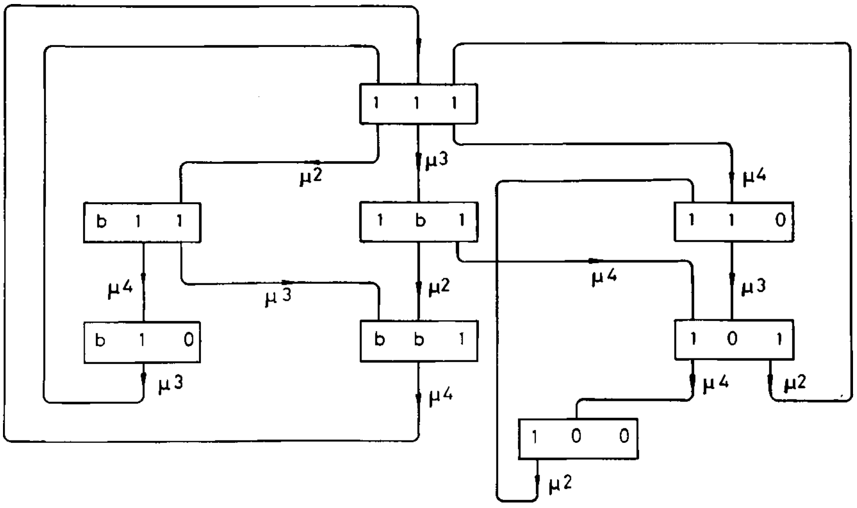


Fig. 1

With these values, the equilibrium equations are:

$$\mu_2 P_{100} = \mu_4 P_{101} \tag{4}$$

$$(\mu_2 + \mu_4) P_{101} = \mu_4 P_{1b1} + \mu_3 P_{110} \tag{5}$$

$$(\mu_2 + \mu_3) P_{110} = \mu_4 P_{111} + \mu_2 P_{100} \tag{6}$$

$$(\mu_2 + \mu_3 + \mu_4) P_{111} = \mu_4 P_{bb1} + \mu_3 P_{b10} + \mu_2 P_{101} \tag{7}$$

$$(\mu_2 + \mu_4) P_{1b1} = \mu_3 P_{111} \tag{8}$$

$$(\mu_3 + \mu_4) P_{b11} = \mu_2 P_{111} \tag{9}$$

$$\mu_3 P_{b10} = \mu_2 P_{110} + \mu_4 P_{b11} \tag{10}$$

$$\mu_4 P_{bb1} = \mu_3 P_{b11} + \mu_2 P_{1b1} \,. \tag{11}$$

With the restrictions on the first and last stages given above, every possible state is accounted for in the left sides of the set of equations; and each left side gives the probability of leaving a particular state whilst the right side gives the probability of reaching that state – the details of the derivation are given below. The equations as they stand, being homogeneous in the probabilities P_{100} etc., determine only the relative values of these; the absolute values are then determined by using the fact that because these represent the only states which can occur, the sum of the probabilities must be 1: therefore

$$\Sigma P = 1, \text{ summed over all states.} \tag{12}$$

3.1 Derivation of the equilibrium equations

As an example, consider eqn. 5 which deals with the state P_{101}. In this state the second and fourth stages of the pipeline are busy and the third is empty and waiting, and can change only if either the second or fourth stage completes an instruction. The probability that the first event occurs in an interval dt is $\mu_2\, dt$ and that the second occurs is $\mu_4\, dt$; and therefore, since they are independent, the probability that one or other occurs is $(\mu_2 + \mu_4)\, dt$. The probability of both occurring in the interval dt is of order $(dt)^2$ and is therefore negligible when dt is small. Therefore the probability that (*a*) the system is in the state (1,0,1) at the instant under consideration and (*b*) it changes to a different state in the next small interval dt is $(\mu_2 + \mu_4) P_{101}\, dt$, which is the left side of eqn. 5.

Consider now the states from which (1,0,1) can be reached directly: the only possibilities are (1,*b*,1) and (1,1,0). If the initial state is (1,*b*,1) and the fourth stage completes an instruction then the third stage becomes unblocked and the fourth immediately starts working on the new instruction it has received; this gives the state (1,0,1) because the probability of a simultaneous change in the second stage, which would lead to the state (1,1,1), is of second order. The probability of this change occurring in the interval dt is $\mu_4\, dt$ and therefore of the system being in the state (1,*b*,1) and changing to (1,0,1) in the next interval dt is $\mu_4 P_{1b1}\, dt$. The change from (1,1,0) to (1,0,1) corresponds to the third stage completing its work on an instruction and passing the instruction to the fourth stage which initially was waiting; and the probability here is $\mu_3 P_{110}\, dt$. So the total probability of changing into the state (1,0,1) is $[\mu_4 P_{1b1} + \mu_3 P_{110}]\, dt$ and equating this to the probability of changing from this state, and dividing by dt, gives eqn. 5. The other equations are obtained by the same type of argument.

3.2 Calculation of the throughput

Let us write P_2 for the probability that the second stage of the pipeline is busy, and similarly of P_3, P_4. Since a blocked stage cannot precede an empty stage these are given by the following relations:

$$\begin{aligned} P_2 &= P_{101} + P_{101} + P_{110} + P_{1b1} + P_{111} \\ P_3 &= P_{110} + P_{b10} + P_{111} \\ P_4 &= P_{101} + P_{1b1} + P_{bb1} + P_{b11} + P_{111} \end{aligned} \tag{13}$$

If we define the throughput of the pipeline as the mean rate at which instructions pass along it, then since the constants μ give the rates at which busy stages process instructions, the throughput is

$$T = \mu_2 P_2 = \mu_3 P_3 = \mu_4 P_4. * \tag{14}$$

We are more interested here in the effect of the variability of the instruction times on the throughput than in the details of the primitive processor, so we may reasonably consider the simplest possible case, in which the mean rates are the same for each stage.

If we put $\mu_2 = \mu_3 = \mu_4 = \mu$, say, in eqns.4–11 we get, on cancelling μ,

$$P_{100} = P_{101} \qquad \text{(4a) etc}$$

$$2P_{101} = P_{1b1} + P_{110}$$

$$2P_{110} = P_{111} + P_{100}$$

$$3P_{111} = P_{bb1} + P_{b10} + P_{101}$$

$$2P_{1b1} = P_{111}$$

$$2P_{b11} = P_{111}$$

P

$$P_{b10} = P_{b11} + P_{110}$$

$$P_{bb1} = P_{b11} + P_{1b1}.$$

Solving these in terms of ratios to P_{111} we get

P_{100}	P_{101}	P_{110}	P_{1b1}	P_{b11}	P_{b10}	P_{bb1}
2/3	2/3	5/6	1/2	1/2	4/3	1

(15)

From the fact that the Ps sum to unity we then get $P_{111} = \frac{2}{13}\mu$ and finally

$$P_2 = P_3 = P_4 = \frac{22}{39}\mu = 0{\cdot}564\,\mu$$

$$\text{so the throughput } T = 0{\cdot}564\,\mu \tag{16}$$

The point to be emphasised here is that if the instruction time per stage were strictly constant and equal to μ then the rate of processing for the complete pipeline, which is what we have called the throughput, would also be constant and equal to μ. The result we have just obtained shows that when these instruction times vary with a negative exponential distribution about a mean μ the throughput is only a little more than half that rate.

*The equality of the three products expresses the fact that there is no net accumulation of instructions at any stage; it can however be deduced from the equilibrium equations (see Appendix).

4 Effect of jump instructions

It is instructive to compare the loss in throughput resulting from variability in instruction times with that which results from the presence of jump instructions. To make sure that we do not underestimate this effect let us suppose that no jump is actually executed, and that no change in the instruction stream actually takes place, until a jump instruction has been processed by the last stage in the pipeline. That is, that it is not known until this stage whether or not the jump instruction will be obeyed and that therefore the interval between the jump instruction and the instruction which follows it is the time for the former to pass all the way along the pipeline. Thus if the mean instruction rates for the four stages are μ_1, μ_2, μ_3, μ_4, respectively, then the mean waiting time is

$$W_j = 1/\mu_1 + 1/\mu_2 + 1/\mu_3 + 1/\mu_4. \qquad (17)$$

This assumes that the mean times per stage for instructions which follow jumps do not differ from those for other instructions. The mean instruction time in the presence of jumps is therefore

$$W = W_j P_j + (1 - P_j)/P_c \mu_4, \qquad (18)$$

where P_j is the proportion of instructions which are successful jumps

P_c is the probability that the last stage is busy in the absence of jumps.

Consider again the case $\mu_2 = \mu_3 = \mu_4 = \mu$. Again to ensure that we do not underestimate the effect of jumps let us allow as much time for the instruction following a jump instruction to traverse the first stage of the pipeline as it takes for each of the succeeding stages; the mean instruction time is then

$$W = P_j . 4/\mu + (1 - P_j)/P_c \mu. \qquad (19)$$

As a reasonable value take the proportion P_j of jumps as one in five instructions. If the instruction time per stage is constant then $P_c = 1$ and the mean waiting time is

$$W = 1{\cdot}6/\mu$$

so the throughput is $T = 1/W = 0{\cdot}625\ \mu$. (20)

For the negative exponential case we found that $P_c = P_4 \mathrel{\overset{\frown}{=}} 0{\cdot}564\mu$ which with eqn. 19 gives

$$W = 2{\cdot}22/\mu$$

and throughput $T = 0{\cdot}45\mu$. (21)

In practice we have store access time to consider as well as jumps and any comparison between observed and calculated throughput would be complicated by this. However, comparison of the results obtained above for the effects of jumps and of instruction time variability, respectively, shows how easy it can be to attribute loss of throughput in the pipeline to the wrong causes; and how the ignoring of instruction time variability could lead to bad decisions on design. For example, a policy of fetching instructions in advance from two or more streams could be adopted as a consequence of putting too much emphasis on the effect of jumps.

5 Effect of buffers

It has already been remarked that a possible way to increase the throughput of a pipeline is to place buffers between some, or all of the stages. The effect of such buffers can be calculated by the same type of process as that just given. If the time which a buffer takes to pass on an instruction and its related data can be ignored, the new situation is one in which the number of stages remains the same but the number of possible states has increased. A stage may now be waiting, or if busy or blocked may have one or more instructions present. For example if the buffer capacity is one instruction the stage to which it belongs is either:

(*a*) busy with buffer empty;
(*b*) busy with buffer full;
(*c*) waiting;
(*d*) blocked with buffer empty;
(*e*) blocked with buffer full.

The equations for the statistical equilibrium can be set up as before and solved to give the probability of each state of the pipeline and hence the throughput. If the buffers introduce delays these can be treated as additional stages in the pipeline with short instruction times and with capacity to hold one or more instructions, the pipeline stages proper having capacity for one instruction only.

6 Example: throughput calculation based on observed data

Another investigation of pipeline throughput carried out in 1977 was undertaken as part of one proposal for a medium-sized processor.[6,7] This machine was designed to the 2900 PLI so that although instructions passing down the pipeline were not PLI instructions (see p.60) it was possible from analysis of programs run on earlier machines to estimate the frequency of each instruction type and hence instruction times within the pipeline units. A different method of calculation was used; this section compares the results obtained by that method and those obtained by using the linear service line method which has just been described.

6.1 Original calculation for the proposed machine

The machine contained two stages, an address generation unit, AGU, and a comp-putation unit, CU. It was a synchronous machine. The instruction time at each stage had a discrete probability distribution, each instruction taking an integral number of machine beats. Again the primary objective was to discover the intrinsic throughput of the pipeline, postponing consideration of complicating factors, such as cases where an instruction in the first stage could not be processed until the result of an instruction in the second stage was known. Thus our first task was to estimate those delays which occurred because the time needed to complete the instruction present in one unit at a given moment was not the same as that need for the instruction then in the other.

As a first step we may calculate either the time for which the AGU is blocked or the time the CU spends waiting. Choosing the latter, let

$\overline{W}$ be the mean number of beats per instruction which the CU is waiting

G_n be the number of beats which the AGU spends processing instruction n in a sequence

R_{n-1} be the number of beats which the CU spends processing the preceding ($n-1$th) instruction

$P(G_n=X,R_n=Y)$ be the probability that $G_n=X$ and $R_n=Y$ where X, Y, are integers

Then $\overline{W} = \sum_{X>Y} (X-Y)P(G_n=X, R_n=Y)$. (22)

Strictly, in this equation we should be talking of the duration of instruction processing in a stage only if work starts on an instruction in both stages at the same time. It does in fact happen that whenever the CU starts work on an instruction this has just been cleared from the AGU, so this also starts work on a new instruction.

A reasonable assumption is that the duration of an instruction is independent of that of its predecessor, in which case the events G_n and R_{n-1} are independent. With this assumption eqn. 22 becomes

$$\overline{W} = \sum_{X>Y} (X-Y)P(G=X)P(R=Y) \tag{23}$$

and now G is the duration of an instruction in the AGU and R the duration of an instruction in the CU.

Hence $\overline{W} = \sum_{n=0}^{\infty} P(R=n) \sum_{m=1}^{\infty} m\,P(G=n+m)$. (24)

When at work on a sequence of instructions, the machine processing rate is the rate at which instructions leave the CU, which is

$$T_C = 1/(\overline{W}+\overline{Y}) \tag{25}$$

and the probability that the CU is busy is

$$P_{Cb} = \overline{Y}/(\overline{W}+\overline{Y}). \tag{26}$$

Using eqn. 23 we can calculate these values if we have information on the distributions of instruction times in the AGU and CU. Two sets of such information were available representing two different types of work. The times for one of these sets are given in Table 1 and the corresponding probabilities in Table 2. From these, and assuming that 'variable' instructions all take only 16 beats, we get

$\overline{X} = 5{\cdot}18$ beats, $\overline{Y} = 6{\cdot}11$ beats

and from eqn. 24,

$\overline{W} = 1{\cdot}83$ beats

Table 1 Instruction times used in para. 6.1

Weighting (relative frequency)	AGU beats	CU beats
6•6	4	7
1•7	7	8
0•7	12	13
7•3	5	2
6•2	3	2
1•3	12	8
0•5	12	8
2•2	12	2
4•4	3	10
3•0	7	4
0•3	2	13
0•9	8	8
2•8	7	2
0•8	0	11
0•6	3	11
0•7	3	7
0•3	2	16
0•6	3	56
0•2	12	16
0•7	12	8
3•0	3	2
1•3	1	variable

Table 2 Frequency distributions of instruction times of Table 1

AGU beats X	$P(G = X)$	CU beats Y	$P(R = Y)$	
0	0•0174	2	0•4664	
1	•0282	4	•0650	
2	•0130	7	•1584	
3	•3362	8	•1106	
4	•1432	10	•0954	
5	•1584	11	•0304	
7	•1627	13	•0216	
8	•0195	16	•0108	
12	•1215	56	•0130	
		variable	•0282	(value 16 taken)
Mean $\bar{X}$	5•18	$\bar{Y}$	6•11	
Variance	9•24	variance	48•3	
Std. dev.	3•04	Std. dev.	6•95	

and hence, from eqn. 25 and 29,

throughput T_C = 0•126 instructions per beat

CU occupation P_{Cb} = 0•67.

6.2 Calculation based on linear service line model

In this case there are only two stages in the pipeline and the possible states are, with the previous notation, $(b,1)$, $(1,0)$, $(1,1)$.

The equilibrium equations are

$$\begin{aligned} \mu_2 P_{b1} &= \mu_1 P_{11} \\ \mu_1 P_{10} &= \mu_2 P_{11} \\ (\mu_1 + \mu_2) P_{11} &= \mu_1 P_{10} + \mu_2 P_{b1} \end{aligned} \qquad (27)$$

where μ_1, μ_2 are the mean instruction rates for the AGU and CU, respectively, and the normalisation equation is

$$P_{11} + P_{b1} + P_{10} = 1. \qquad (28)$$

From these,

$$\begin{aligned} P_{11} &= 1/[1 + (\mu_1^2 + \mu_2^2)/\mu_1 \mu_2] \\ P_{b1} &= (\mu_1/\mu_2) P_{11} \\ P_{10} &= (\mu_2/\mu_1) P_{11}. \end{aligned} \qquad (29)$$

Since the instruction rates are the inverses of the instruction times, these are equivalent to

$$\begin{aligned} P_{11} &= \bar{X}\,\bar{Y}/[\bar{X}\,\bar{Y} + \bar{X}^2 + \bar{Y}^2] \\ P_{b1} &= \bar{Y}^2/[\bar{X}\,\bar{Y} + \bar{X}^2 + \bar{Y}^2] \\ P_{10} &= \bar{X}^2\,/[\bar{X}\,\bar{Y} + \bar{X}^2 + \bar{Y}^2] \end{aligned}$$

and with the values of $\bar{X}$, $\bar{Y}$ used previously we get

$$P_{11} = 0{\cdot}330, P_{b1} = 0{\cdot}390, P_{Cb} = P_{11} + P_{b1} = 0{\cdot}720.$$

Then the throughput is $T_C = \mu_2 P_{Cb}/\bar{Y} = 0{\cdot}118$ instructions per beat.

This value for throughput has been calculated on the assumption that the instruction times are negative exponentially distributed about the observed means $\bar{X}$, $\bar{Y}$; and it agrees well with that calculated in the preceding section, using the observed distribution.

An indication of the extent to which the observed distribution times departs from the assumed negative exponential is given by the value of the coefficient of variation, which is the ratio of the standard deviation to the mean. For any negative exponential distribution this is 1 (or 100% as it is usually quoted), because the mean and standard deviation are equal. From Table 2, the values are 59% for the AGU and 114% for the CU.

The results for the two sets of observed data are summarised in Table 3.

Table 3

			Sample 1	Sample 2
AGI:	Mean beats	$\bar{X}$	5•18	4•5
	Variance		9•24	3•81
	Std. dev.		3•04	1•95
CU:	Mean beats	$\bar{Y}$	6•11	6•80
	Variance		48•30	86•60
	Std. dev.		6•95	9•31
CU	mean waiting	$\bar{W}$	1•83	1•46
CU	occupation	P_{Cb}	0•72	0•79
CU	throughput	T_C	0•118	0•116

6.3 Difficulties in the service-line treatment

Three difficulties encountered in this method of treating the problem have already been mentioned:

(*a*) jumps;
(*b*) distribution of instruction times departing from negative exponential;
(*c*) delays due to store accessing

A method for correcting for jumps has been described in para. 4. Instruction time distributions which are not negative exponential can be dealt with by treating a stage as an assembly of cells. Stages with a coefficient of variation smaller than that of the negative exponential (that is, less than 100%) can be treated as a sequence of exponential stages, only one of which may be occupied at a time. Stages with greater than 100% variation can be treated as a set of cells in parallel: each instruction enters one cell and one cell only, the probability of entering a given cell being fixed. Again each cell has a negative exponential distribution. In both series and parallel cases the man processing rates of separate cells differ and are fixed so as to simulate the characteristics of the original stage.[6] Once we have dropped the assumption of exponential services times, store accessing delays can be treated as part of the instruction time of the stage making the access.

There are however other problems:

(*a*) not all instructions proceed to the end of the pipeline;
(*b*) certain phases of some instructions keep two or more stages busy simultaneously;
(*c*) the presence of certain types of instruction may inhibit the progress of certain other instructions.

Instructions which do not go to the end of the pipeline are simply those whose execution requires only the earlier stages. Cooperation between stages may be required when, for example, a stage with limited arithmetic facilities needs to perform a multiplication which can be done more rapidly by recourse to a special unit. Inhibiting the flow of instructions will be necessary, for example, when one instruction needs to make use of a modifier and a later instruction is an explicit 'set modfier' command.

These problems can be dealt with in the sequential service line model, as has been indicated. But their combined effect is to make a different method of treatment more attractive, the method of *discrete event simulation,* which we now describe.

7 Treatment by the method of discrete event simulation

This method avoids certain of the mathematical difficulties inherent in the service-line approach by providing a model in which the significant events and the times at which they occur are represented directly. Problems will still arise however in ensuring that the statistical information used by the model is representative of what happens in practice.

7.1 ICL program SIPI

In 1978 a discrete event simulation program called SIPI was produced, to provide a means of studying the behaviour of linear pipelines.[7] When applied to OCPs the program could be used, for example, to help in deciding the optimum number of stages: that is, it was intended for use at relatively early stages of machine development. Previous applications of this method to the study of pipeline throughput had been concerned with particular machines and had used actual instruction sequences. SIPI was used in the early stages in the design of a machine known as S4.

SIPI, written in Algol 60, uses the event-scheduling approach.[8] By producing our own simulation control routines using a general problem-oriented language we were able to get direct access to tables of scheduled events. This was desirable because the method used for dealing with jumps required the removal of scheduled events before they were due to be executed. The simulation model embodied in the program is at the same level of abstraction as the service-line model but was more complete. That is, instruction sequences are represented by the instruction times required at each stage and the times of individual instructions are assumed to be independent. On the other hand each of the factors such as interstage dependencies which interrupt the smooth flow of instructions is dealt with directly not as a correction or as if an afterthought.

The essential inputs to the program for a simulation run are:

the number of pipeline stages;
the times to process instructions at each stage;
an indication of whether or not a stage has a buffer, and if so, the size.

There are two methods by which the user can give the distribution of instruction times for each stage. There are standard distributions such as the normal or the negative exponential which have explicit mathematical forms depending on a few parameters such as the mean and variance, or the mean only in the case of the negative exponential. In these cases the form can be specified and the parameters given, and sample values generated from random numbers. In other cases, where the distribution is empirical, the user may input details in a form similar to Table 2. SIPI then rearranges the data in a form suitable for addressing with the help of a pseudo-random number generator, and during a simulation run each delay is selected with the appropriate frequency. In this second type of input the mean and standard deviation are calculated by the program and provided as part of the output.

For store-accessing delays it is necessary first to specify which stages may access the store; then the delays may be stated in one of the forms used for presenting the instruction times. The tabular form of input is useful for representing store hierarchy, in which case there are discontinuities in the curve of delay against probability.

To deal with interference in the operation of two stages, the stages concerned are specified and the probability given that this particular form of check to instruction flow occurs. Other inputs give the proportions of instructions completed at each stage without passing to the end of the pipeline, the proportions of jump instructions and specify the stages on which the different classes of jump depend. Since input data are likely to have been obtained from extant machines and ratios are likely to be available in terms of completed instructions, the program can treat such things as the proportions of jump instructions as ratios between completed instructions. Adjustments must then be made within the program in applying the data to events which occur as the instructions reach the different stages of the pipeline.

The frequencies of occurrence of the different types of instruction are assumed to be independent. The corresponding events during a simulation run are generated using a pseudo-random number generator in accordance with the probabilities implicit in the input data. The state of the pipeline of the machine being studied is held as an array of integers describing in coded form each stage as 'blocked', 'busy', 'full' etc. The completion of an instruction at a stage is an event scheduled when work begins on that instruction within the stage.

7.2 Treatment of jumps and interstage interference

Execution of a jump involves clearing the instructions following it, which implies aborting scheduled events in the simulation model. The representation of interference between stages in the target machine takes the following form:

first the earlier of the two stages is 'forced' to remain in the busy state without passing on an instruction, until the later and all intervening stages are cleared of instructions by completing work on instructions in the normal way.

then the instruction in the earlier stage is allowed to pass along the pipeline.

Depending on the type of interference being simulated, the second phase may or may not involve preventing instructions moving down the pipeline

finally the later stages having been cleared by the instruction of interest, normal flow is again permitted.

As we enter further into the details of the ways in which the operations of different stages may interfere with one another, and of the problems of preserving consistency in our model (such as arise when a jump emanates from a stage between two co-operating stages), we leave the world of abstract models for that in which a detailed simulation of the execution of instruction sequences is required.

7.3 Example: some results of simulation runs with SIPI

To give an idea of the flexibility of the SIPI program and of the information it can provide, Tables 4–7 summarise the main results of four runs. These all refer to a fictious eight-stage machine with a beat time of 80 ns in which store accesses can be made from stages 1, 3 and 5. Stage 6 is simply a buffer of capacity nine instructions, including partial results, and a processing time of 1 beat per instruction; all other stages can be provided with buffers which do not contribute to the time taken to process an instruction. Interlocked stages always involve stages 2 or 4 as first stage and stage 7 as second stage.

Table 4 Run 1 Buffered
Time 8079 beats = 646μs
Instructions processed 2160 = 3•34 m.i.p.s
Jumps executed 268 Interlock instructions 167 Early-finish instructions 1074

Stage	1	2	3	4	5	6	7	8
Mean instr. time (beats)	1	1	1	1•5*	1	1	2*	1
Buffer capacity	1	1	1	1	1	9	1	1
State at end	full,busy	busy	empty	empty	empty	empty	busy	empty
Instr. in stage	2	1	0	0	0	0	1	0
Instr. processed	3104	2862	2418	2004	1654	1560	1501	818
Jumps	0	0	0	142	0	0	126	0
Instr. end here	0	170	700	146	0	0	557	818
Store accesses	453	0	75	0	103	0	0	0
Blocks begin here	0	68	0	98	0	0	0	0
Blocks end here	0	0	0	0	0	0	166	0
Percentage of time empty	0	33	44	47	69	76	64	90
busy	78	52	39	48	31	20	36	10
blocked	22	15	17	5	0	4	0	0

*Instruction times marked thus are the sum of a constant and a negative exponential term. Others are constant.

Table 5 Run 2 Unbuffered, instruction times as in Run 1
Time 8449 beats = 680μs
Instructions processed 2105 = 3•10 m.i.ps
Jumps executed 267 Interlock instructions 165 Early-finish instructions 1031

State	1	2	3	4	5	6	7	8
Mean instr. time (beats)	1	1	1	1•5*	1	1	2*	1
Buffer capacity	0	0	0	0	0	9	0	0
State at end	full, busy	full, busy	full, busy	full, busy	full, busy	empty	full, busy	empty
Instr. in stage	1	1	1	1	1	0	1	0
Instr. processed	2736	2560	2181	1922	1582	1454	1453	805
Jumps	0	0	0	131	0	0	136	0
Instr. end here	0	194	172	149	0	0	512	805
Store accesses	417	0	62	0	112	0	0	0
Blocks begin here	0	48	0	97	0	0	0	0
Blocks end here	0	0	0	0	0	0	165	0
Percentage of time								
empty	0	32	44	49	71	73	66	91
busy	68	45	33	44	29	18	34	19
blocked	32	24	22	6	0	9	0	0

The proportions of jumps, interlocks and early-finish instructions were the same in all four runs, as follows:

jumps	12•5% of instructions completed
interlock instructions	7•5% of instructions completed
early-finish instructions	50•0% of instructions completed

as were the proportions of instructions making store accesses, as follows:

from stage 1	12•5 % of instructions entering this stage
stage 2	2•5 % of instructions entering this stage
stage 3	6•25% of instructions entering this stage

The properties varied in these runs were the buffering and the instruction processing times. Runs 1 and 3 were for buffered systems, Runs 2 and 4 for unbuffered; Stage 6 remained the 9-instruction buffer in all four cases. Runs 1 and 2 used one set of values for the processing times, Runs 3 and 4 another. The details for each run are given in the relevant Table. Each table gives the length of time simulated, the effective throughput in millions of instructions completed per second (m.i.p.s), the conditions at the end of the simulated period and the percentages of the time during which each stage was in each of its possible states – Empty (i.e. idle), Busy or Blocked.

Table 6 Run 3 Buffered
Time 6499 beats = 520µs
Instructions processed 1034=1·99 m.i.p.s
Jumps executed 123 Interlock instructions 74 Early-finish instructions 522

Stage	1	2	3	4	5	6	7	8
Mean instr. time (beats)	2*	2*	2*	3*	2*	1	4*	2*
Buffer capacity	1	1	1	1	1	9	1	1
State at end	full, busy	empty	full, busy	empty	busy	busy	busy	empty
Instr. in stage	2	0	2	0	1	1	1	0
Instr. processed	1519	1375	1144	948	767	734	704	398
Jumps	0	0	0	61	0	0	62	0
Instr. end here	0	86	94	89	0	0	253	389
Store accesses	219	0	32	0	43	0	0	0
Blocks begin here	0	35	0	39	0	0	0	0
Blocks end here	0	0	0	0	0	0	74	0
Percentage of time								
empty	0	25	44	45	72	82	57	88
busy	71	59	40	53	28	12	42	12
blocked	29	15	17	2	0	7	1	0

Table 7 **Run 4 Unbuffered, *instruction times as in Run 3***
Time 7999 beats = 640µs
Instructions processed 1177 = 1·84 m.i.p.s
Jumps executed 144 Interlock instructions 87 Early-finish instructions 600

Stage	1	2	3	4	5	6	7	8
Mean instr. time (beats)	2*	2*	2*	3*	2*	1	4*	2*
Buffer capacity	0	0	0	0	0	9	0	0
State at end	full, busy	busy	empty	full, busy	empty	empty	empty	empty
Instr. in stage	1	1	0	1	0	0	0	0
Instr. processed	1524	1418	1216	1051	865	797	797	433
Jumps	0	0	0	67	0	0	77	0
Instr. end here	0	107	116	90	0	0	287	433
Store accesses	258	0	32	0	41	0	0	0
Blocks begin here	0	36	0	51	0	0	0	0
Blocks end here	0	0	0	0	0	0	87	0
Percentage of time								
empty	0	27	42	47	73	76	58	89
busy	63	49	35	48	27	11	41	11
blocked	37	24	23	5	0	14	1	0

8 Value of modelling pipeline throughput

Reference has been made in this paper to detailed discrete-event simulation models of OCP behaviour. Such models appear to offer an escape from the intractable problems presented by actual systems to provide a heuristic way of improving design. At the other end of the scale, when a policy such as the use of a linear pipeline is first adopted, only a very simple model of its operation may be presented. The object of the paper has been to show that it is possible to develop the thinking behind the original policy without being overwhelmed with detail.

Detail in discrete-event simulation tends to be full, because the inclusion of detail provides confidence in the model as a trustworthy representation of the system. The inclusion of detail is facilitated by the use of special-purpose simulation languages. However, the very inclusion of detail may obscure important characteristics of the system which is being studied; and any attempt to design consciously for better performance involves some kind of generalisation. A problem in the detail simulation of an OCP is the choice of instruction sequences to run on the model. To do justice to the work carried out in producing the model, we must be sure that these sequences are representative of those encountered 'in the field' in the use they make of the processor. A related problem is that of drawing general conclusions from what may appear to be unconnected sets of results. A third problem is that of deciding with what faithfulness to represent features of the system outside the OCP.

In addition to these difficulties there may be real dangers in resorting to a full simulation too early in the course of a design. In order to produce a detailed model a number of design decisions must be taken. In other cases assumptions about the design may be made in the course of producing the model, assumptions which may not be stated outside the simulation or concerning matters of detail which do not lend themselves to general statement. The result is that only a part of the problem is studied and only some of the possibilities are tried, while the simulation program may be defended as the best possible depiction of the problem and as the best possible representation of the machine which is being designed.

Analytic models use less computer time, provide faster turnround and are easier to set up and to modify than discrete event models. Above all they provide insight into the behaviour of the system being modeled. At a later stage in the design process a simulation model such as SIPI provides information about machine behaviour in a form which makes generalisation possible in a form meaningful to the design engineer. When subsequently a detailed simulation is used, those decisions already made will have been made consciously. The process may be viewed as a hierarchical structure of learning loops. While such a process is going on in connection with the Order Code Processor a similar process is taking place in connection with other aspects of the machine such as store accessing.

The sequential service line and pipeline models presented in this paper have been interpreted as Order Code Processors along which pass instruction streams. Pipelines occur elsewhere in computer systems where digital machines cooperate in the transmission and ordering of information, and the same models may be used to study their performance also.

Appendix
Solution of the equilibrium equations

Eqns. 4–11 being linear are easily solved. Solving in terms of the ratios of the probabilities to P_{111} and writing $P_{100} = P_{100}/P_{111}$ and similarly for the other ratios we get:

$$P_{100} = \frac{\mu_3 \mu_4^2}{\mu_2^2 (\mu_2 + \mu_4)} \cdot \frac{2\mu_2 + \mu_3 + \mu_4}{\mu_2 + \mu_3 + \mu_4}$$

$$P_{101} = \frac{\mu_2}{\mu_4} \cdot P_{100}$$

$$P_{110} = \frac{\mu_2 (\mu_2 + \mu_4)}{\mu_3 \mu_4} \cdot P_{100} - \frac{\mu_4}{\mu_2 + \mu_4}$$

$$P_{1b1} = \frac{\mu_3}{\mu_2 + \mu_4}$$

$$P_{b11} = \frac{\mu_2}{\mu_3 + \mu_4}$$

$$P_{b10} = \frac{\mu_3}{\mu_3} \left[\frac{(\mu_2 + \mu_4)^2 + \mu_2 \mu_4}{(\mu_2 + \mu_4)(\mu_2 + \mu_3 + \mu_4)} + \frac{\mu_2}{\mu_3 + \mu_4} \right]$$

$$P_{bb1} = \frac{\mu_2 \mu_3}{\mu_4} \left[\frac{1}{\mu_2 + \mu_4} + \frac{1}{\mu_2 + \mu_4} \right]$$

As a check on the algebra we find that putting $\mu_2 = \mu_3 = \mu_4$ in these expressions gives the same numerical values as were found in para. 3.2.

The throughput T is given by $T = \mu_2 P_2 = \mu_3 P_3 = \mu_4 P_4$ (eqn. 14). From the above solution we find, after some straightforward but rather tedious algebra, that the three products are indeed equal (which gives quite a powerful check on the algebra) and that

$$\mu_2 P_2 = \mu_3 P_3 = \mu_4 P_4$$

$$= \frac{(\mu_2 + \mu_2) [(\mu_2 + \mu_4)^3 + \mu_2 \mu_3 (\mu_2 + \mu_4) + \mu_3 \mu_4^2]}{\mu_2 (\mu_2 + \mu_4)(\mu_2 + \mu_3 + \mu_4)} .$$

Again putting $\mu_2 = \mu_3 = \mu_4$ this reduces to 22/39, agreeing with the value obtained in para. 3.2.

Acknowledgments

The author wishes to acknowledge the contribution of Mr. J.A.Jones in providing data on instruction time distribution. He would like to thank Dr. J. Howlett for his interest in the work, for checking the equations and for making improvements in the notation. He also wishes to thank Mr. R.J.Ball for encouraging him to carry out the simulation work, Mr. A.R.Brook for suggesting improvements to the program SIPI and Mr. C.M.Berners Lee for his helpful comments on the presentation of the work.

References

For the sake of completeness references are given to ICL internal reports and papers. These may not be available outside the company. Any reader who is interested is invited to write to the author.

1 RAMANOOTHLY, C.V. and LI, H.F.: 'Efficiency in generalised pipeline networks', National Computer Conference, 1974.

2 WOOD, W.F.: 'Performance of the pipelined "primitive processor"'. ICL Report, May 1977.

3 WOOD, K.W.: 'A set of macro components for primitive processor', Phase II, Technical Report TN 4035, ACTP Contract K/78B/388, September 1975.

4 MORSE, P.M.: *Queues, inventories and maintenance,* Chapter 4, New York, John Wiley and Sons Inc., 1958.

5 FELLER, W.: *Introduction to probability theory and its applications,* Chapter XVII New Yoek, John Wiley and Sons Inc., 1950.

6 MORSE, P.M.: *Queues, inventories and maintenance,* Chapter 5. New York, John Wiley and Sons Inc., 1958.

7 WOOD, W.F.: ICL internal memorandum, November 1977.

8 JONES, J.A.: 'A processor system: draft development proposal'. ICL, January 1977.

9 FISHMAN, G.S.: *Concepts and methods in discrete event simulation,* New York, John Wiley and Sons Inc., 1973.

Towards an 'expert' diagnostic system

T.R. Addis
ICL Research and Advanced Development Centre, Stevenage, Herts

Abstract

'Expert' systems are computer-based systems that are capable of acquiring knowledge from experts and making it available to the less skilled. The paper describes how these particular techniques can be applied to fault-finding in computers. After giving examples of successful implementations of 'expert' systems in other fields it describes diagnostic aids being developed within ICL. Two systems are dealt with known as CRIB and RAFFLES, respectively. The paper shows how information-theory concepts can be used to improve the efficiency of the search strategy and how the ICL Content Addressable File Store (CAFS) can be used in the implementation for part of the process. The diagnostic aids described, which can be related to different elements of the diagnostic skill, will eventually be united to form an 'expert' diagnostic system of considerable power.

1 Introduction

An 'expert' system is a program and database that captures the knowledge of experts in a recognised field, makes this knowledge coherent and then available to less experienced people within that field by a man–machine dialogue. One of the important properties of such systems is the ability to provide an explanation for any given advice. One method of providing a diagnostic aid for computer systems, both hardware and software, falls conveniently within this paradigm of an 'expert' system.

Modern hardware and software are becoming more complex and are demanding a much greater level of diagnostic skill and competence from engineers and system experts. However, the availability of highly qualified and experienced personnel is becoming less as systems proliferate. Apart from developing more reliable systems, a solution to this shortage is to reduce the time spent by the skilled in finding faults. This can be achieved by providing better fault-locating aids so that either the more competent can find faults quicker or the less competent (more available) and even untrained people can locate faults which would otherwise require greater skill. Compounding this requirement for greater skill is the need for rapidly communicating new modifications of systems and fault-locating techniques gleaned from others' experience.

The first part of this paper is a brief review of some successful implementations of 'expert' systems. The second part describes work being done by ICL that is moving towards an 'expert' system for finding faults in both hardware and software. This work is being done by ICL's Technical Services and the Research and Advanced Development Centre. It has evolved from two techniques, both of which are concerned with searching a known faults database. The first technique was developed in conjunction with Brunel University (Dr. R. Hartley) and is embodied in a system called CRIB. This system responds to described symptoms with recommended tests. The CRIB system was then modified extensively as a combined effort by Technical Services, RADC and Brunel in order to make it usable by software diagnostic specialists[1]. The other technique which is being developed by Technical Services was proposed by RADC to overcome some of the technical problems encountered with CRIB. This other technique is embodied in the RAFFLES system[2] and is based upon generating optimum search strategies. A third technique for dealing with previously unencountered faults is being explored by RADC. Each of these three systems is analogous to the three components of the diagnostic skill (see Fig. 1) and they will combine to form a commercially viable 'expert' diagnostic system.

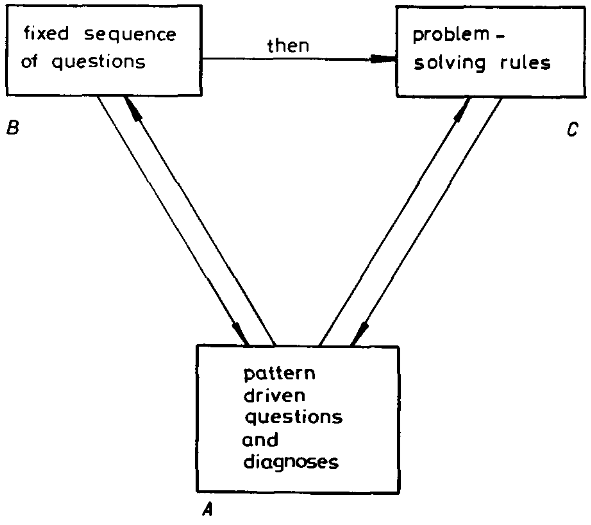

fixed sequence of questions
then
problem - solving rules
B
C
pattern driven questions and diagnoses
A

Fig. 1 Components of the diagnostic skill

2 'Expert' systems*

(Developed outside ICL.)

2.1 DENDRAL

One of the first practical systems built called DENDRAL utilised a 'production system'. This is a rule-based system where the rules are given as an unordered list of:

IF pattern THEN action.

The work on DENDRAL started in 1965 at Stanford and its task was to enumerate plausible structures (atom-bond graphs) for organic molecules, given two kinds of information:

* The section is an edited extract from Addis[3].

(*a*) analytic instrument data from a mass spectrometer and a nuclear magnetic resonance spectrometer;
and
(*b*) user-supplied constraints on the answers from any other source of knowledge available to the user.

The empirical theory of mass spectrometry is represented by a set of rules of the general form:

IF [Particular atomic configuration (subgraph) with probability, *P*, of occurring]
THEN [Fragmentation of the particular configuration (breaking links)].

Rules of this form are natural and expressive to mass spectrometrists.

DENDRAL also has an inference procedure dependent upon the Plan–Generate–Test method. The generate program called CONGEN generates plausible structures using a combinatorial algorithm that can produce all the topologically legal candidate structures. Constraints are supplied by the user and/or the 'plan' process. 'Test' discards less worthy candidate structures and rank orders the remainder. 'Plan' produces direct inference about likely substructures in the molecule from patterns in the data that are indicative of the presence of the substructure. Patterns in the data trigger the left-hand sides of the substructure's rules. 'Plan' sets up the search space so as to be relevant to the input data. 'Generate' is the inference tactician and 'Test' prunes the results.

DENDRAL's performance rivals expert human performance for a small number of molecular families for which the program has been given specialist knowledge. DENDRAL's performance is usually not only much faster but also more accurate than expert human performance and is used every day by Stanford chemists and collaborators. A program METADENDRAL was produced as an expert system for deriving rules of fragmentation of molecules in a mass spectrometer for possible use by DENDRAL.

2.2 MYCIN

There have since been many other expert systems as natural offspring of DENDRAL. The most important development was perhaps MYCIN, also produced at Stanford, whose task is to diagnose blood and meningitis infections and then recommend drug treatment. MYCIN conducts a consultation in medical English with a physician-user about a patient's case, constructing lines of reasoning leading to the diagnosis and treatment. Knowledge is acquired by confronting the user with a diagnosis with which he does not agree and then leading him back through this line of reasoning to the point at which he indicates that the analysis went wrong. The offending rule is modified or new rules added until the expert agrees with the solution.

EMYCIN (Empty MYCIN, again from Stanford) was developed to capture the framework of a general expert system. It just requires the addition of knowledge such as the production rules. Many different systems have been created from this essential expert. An example is the signal understander (SU/X) which forms and continually updates, over long periods of time, hypotheses about identity, location and velocity of objects from spectral lines of acoustic signals. Another is PUFF which is linked directly to an instrument into which a patient inhales and exhales. From the measured flow-volume loop of the patient's lungs and the patient's records PUFF diagnoses pulmonary function disorders. There is now work going on to produce PUFF on a chip.

EMYCIN has also been used to interface an excellent but complex mechanical structure analysis package to an engineer. The engineer interacts with the expert system (SACON) to describe his problem in general terms and this generates strategy recommendations for the software package. This in turn calculates and displays the physical behaviour of the structure. Other EMYCIN derivatives are GUIDON, an expert to teach the expertise captured by the systems and AGE, an expert to guide in the construction of expert systems.

2.3 PROSPECTOR

The PROSPECTOR system is intended to emulate the reasoning process of an experienced exploration geologist in assessing a given prospect site or region for its likelihood of containing an ore deposit. The system holds the current knowledge and information about classes of ore deposits. The user begins by providing a list of rocks and minerals observed plus other observations expressed in simple English. The program matches this data against a knowledge base and requests additional information of potential value for arriving at a more definite conclusion and provides a summary of the findings. The user can ask at any time for elaboration of the intent of a question, or for the geological rationale of including a question or for a trace of the effect of his answers on PROSPECTOR's conclusions. The intention is to provide the user with many of the services that could be provided by a telephone conversation with a panel of senior economic geologists, each an authority on a particular class of ore deposits. Currently, PROSPECTOR comprises seven such specialist knowledge bases.

The drilling site selection expertise for porphyry copper deposits derives its input from digitised maps of geological characteristics and produces an output of colour-coded graphical display of the favourability of each cell on a grid corresponding to the input map. The success of PROSPECTOR depends upon the combination of clausal statements with a strong inference engine and plausible statements framed as a production system using statistical techniques (i.e. Bayes' Rule).

2.4 Comments

The advent of expert systems is perhaps the most significant event to happen in Artificial Intelligence (AI) mainly because of the huge number of potential practical applications. There is tremendous pressure from industry and the military in the USA for such systems, but there are just too few experts on knowledge engineering to provide them[4]. Oil companies are pouring large sums of money into extending programs like PROSPECTOR and variants of EMYCIN are being used for many applications by U.S.A. Government organisations.

Expert systems of the kind described in this section are required within the computer industry to interface users to complex systems and help engineers or software specialists to find their way through involved mechanisms. In particular, they can be used as diagnostic aids of considerable utility.

3 Diagnostic aid systems

3.1 Diagnostic process

Production systems (see section 2.1) can be used to describe human problem-solving behaviour. Production systems simulate a wide range of tasks from those that are stimulus-driven, like driving a car, to those that are stimulus-independent, such as mental arithmetic. Production systems can illustrate the manner in which skills are accumulated by the growth of the related production rules and the generalisation of techniques to other tasks.

In particular, the openness of the production system enables it to capture the flexibility of control that is so typical of human behaviour and so hard to capture in a non-data-driven computer program. The diagnostic skill of engineers could conceivably be represented in this form. John Fox at Sheffield University[5] has recognised three distinct components of a doctor's skill (see Fig. 1).

A doctor's actions are often made on a pattern-recognition-like basis, in which a specific question to ask (or tentative diagnosis) is suggested by a particular configuration of symptoms (A). The doctor starts the diagnostic process, however, by asking a fixed sequence of questions in order to look for an emerging pattern (B). At some point in the sequence the information is sufficient to trigger one of the rules in the pattern-driven module, perhaps leading to the asking of a contingent question, the answer to which may well trigger another of the pattern-driven rules. If this chain of rule firing does not lead anywhere the initiative reverts back to the fixed sequence of questions. If such a system fails and the doctor reaches the end of his fixed list of questions then he adopts a problem-solving strategy interacting with the pattern-driven questions and diagnosis (C).

*3.2 The CRIB System**

(A diagnostic aid developed by Brunel University in conjunction with ICL.)

3.2.1 Description: The CRIB system (Computer Retrieval Incidence Bank) was initially conceived as an aid to diagnosis for the engineer by providing him with an accumulation of symptom patterns gleaned from many sources, including his peers (this approach can be equated with A in Fig. 1). The important aspect of this collection of symptom patterns is that it is used to generate further questions, according to certain rules (heuristics), so that the engineer can quickly be guided to the fault indicated by the symptoms. It is this feedback, above all, which differentiates this kind of diagnostic aid from an information retrieval system. Thus, the engineer gives the system a description of his observations and these descriptions are compared with the database. Groups of symptoms which match the incoming description are processed to produce a list of the best possible tests for the engineer to make next.

*This section is an edited extract from Addis and Hartley.[1]

The central feature of the CRIB database is where the action is designed to elicit symptoms from the machine via the engineer. Thus the *action* coded as A1036 and meaning 'run store test program X' results in the *symptom* 'store works' or its *inverse* 'store faulty'. The symptoms can be expressed either in code (e.g. S1036 or the inverse N1036) or in jargon English as above.

The action-symptom pairs (symptoms for short) are organised in groups which can be of three types:

(*a*) Total group. The accumulated group of symptoms observed to occur during many investigations of a single fault (T-group).
(*b*) Key group. A subset of symptoms of a total group all of which are necessary to indicate the location of a fault (K-group).
(*c*) Subgroup. A subset of symptoms of a key group which, by virtue of being an incomplete key group, can only indicate the fault location to a degree less than certain.

The contents of symptom groups reflect the experience of actual investigations and a group only exists if it has proved its usefulness in actual fault investigations. In particular, there is a floating population of subgroups which are compared for usefulness by a simple success ratio (the fraction of times the symptoms in the group have occurred in successful investigations relative to the total number of investigations in which the subgroup occurred). The more successful subgroups are considered as active, whilst the less successful ones are kept in reserve.

When a group of symptoms occur together (i.e. as observed by the engineer) during a sequence of actions or tests, then this group is associated with that part of the hardware in which the fault lies. The machine is modelled as a simple hierarchy of subunits (as in a parts explosion). A simple example of the first few levels is pictured in Fig. 2.

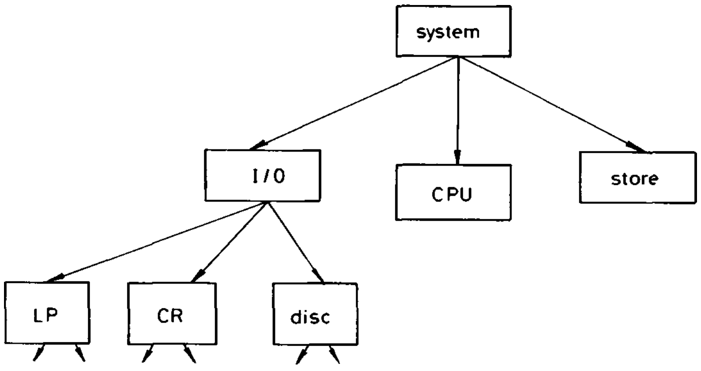


Fig. 2 A simple hierarchy of subunits of the patient computer

The point of the model is not to provide a 'true' picture of the organisation of the machine, but to provide some structure to the database so that a search can be done in an ordered fashion. The searching of the tree is done in a depth-first fashion, where a decision to drop a level is only made on the basis of matching a complete group (total, key or sub in that order) from amongst the set ot symptoms observed by the engineer. With only 50-100 nodes on the tree (subunits in the hierarchy) this is manageable.

By successively matching larger and larger symptom groups the CRIB system will eventually arrive at a terminal subunit which will be either repairable or replaceable. With the trend towards increased modularity, most fault investigations are eventually an exercise in board swapping. If a terminal subunit is reached and the fault is not cured this can mean one of four things:

(*a*) a path taken based upon a chosen subgroup did not lead to the fault;
(*b*) a key group is incomplete (more symptoms are necessary to be certain of success);
(*c*) multiple, transitory or 'soft' faults – these are difficulties in any diagnostic system, manual ones included;
(*d*) the hierarchy is incapable of reflecting the fault location precisely enough.

The system then backtracks automatically to the last decision point and tries to find another match. Since the search is depth-first the whole hierarchy can be covered if necessary, i.e. every symptom group in the database will eventually be examined. If this happens and the fault is still present then it is a fault which the system has not seen before.

When the system fails to find a match with even a subgroup at any level of the hierarchy, it asks for fresh information rather than backtracking immediately. This is not done in a blind fashion, but via a list of suggested actions put to the engineer, chosen on a heuristic basis using information about the diagnostic context current at that time. Several heuristics have been tried but all are aimed at attempting to complete incomplete subgroups thus enabling a drop in level, i.e. a more precise location of the fault. From the partially matched subgroups at that level, symptoms are compared for their usefulness according to four criteria:

(*a*) the time taken for the associated action to be carried out;
(*b*) the average amount of time the investigation is likely to take after this step has been taken;
(*c*) the number of incomplete subgroups in which this symptom occurs;
(*d*) whether the symptom is the last one in a subgroup.

These factors are combined in a heuristic procedure to order the relevant actions which have been extracted. The best five are suggested to the engineer.

The engineer is not obliged to carry out any or all of these actions, but having carried out one or more actions and having reported the symptoms he observes to the system, the search cycle is then repeated. Fig. 3 shows the diagnostic process without the backtracking.

3.2.2 Modifications to CRIB; use of CAFS: The CRIB system was constructed as described in the previous section but its testing was hampered by the lack of an efficient data-retrieval system. It was for this reason that the experimental CAFS MK I[1,6] was used which led to some interesting changes in method as well as performance.

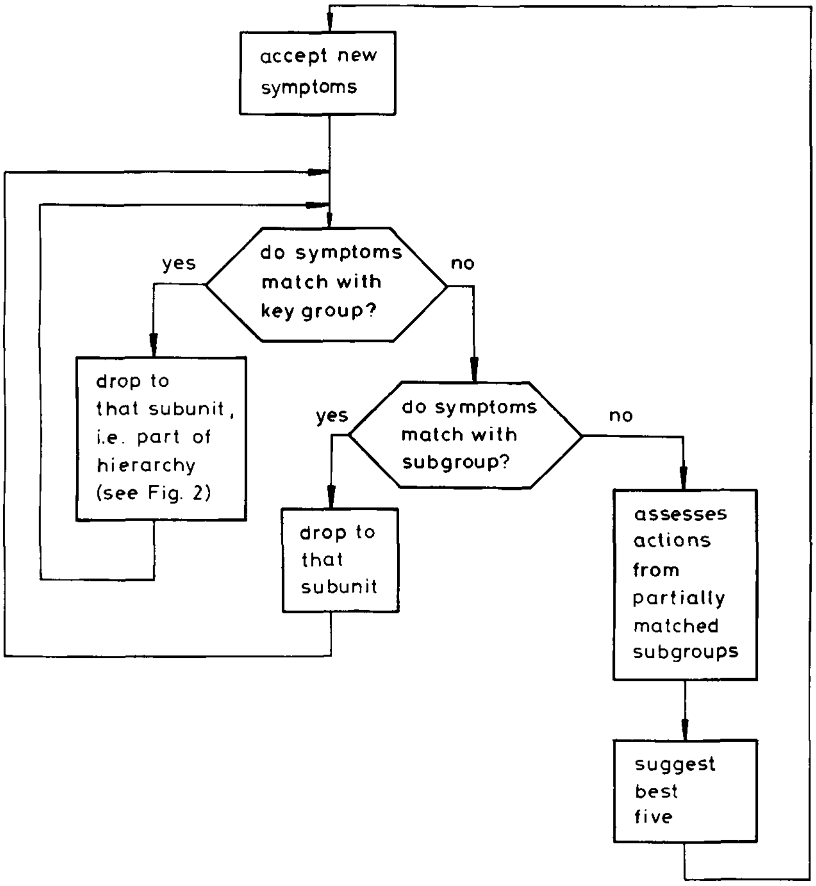

accept new symptoms
do symptoms match with key group?
yes
no
drop to that subunit, i.e. part of hierarchy (see Fig. 2)
do symptoms match with subgroup?
yes
no
drop to that subunit
assesses actions from partially matched subgroups
suggest best five

Fig. 3 Flow diagram of the CRIB diagnostic process

The early system operated as a depth-first search procedure where a match was sought only at the level below the current subunit, amongst its associated groups. Thus for all the symptom groups in the database, only those relating to the immediate subunits were examined. In this way the required search was contained.

CAFS, however, is capable of searching large databases globally, without any extra programming effort. Using this facility, the CRIB system can now look for a match with all symptom groups at all levels. The notion of a depth-first search then disappears. It is now possible for a fault to be located within any appropriate subunit in *one* step, as against the original step-by-step, level-by-level procedure. Since this is the case, the notion of backtracking out of unhelpful situations becomes unnecessary. Because of this major change in thinking, the organisation of the program had to be altered. In particular, group matching and action retrieval techniques had to be revised.

In the original CRIB system there were four files organised according to the required major search tasks. Speed of access was limited by the need to scan these files repeatedly. On the other hand, the CAFS MK I system depends upon all the data existing in *one* large file where each CAFS record contains one example of each kind of data item. This means that each symptom of a particular symptom group is contained in a separate record and all the symptoms of this group are scattered throughout the file[6, 12]. Each record also contains a series of flags (known as tags) which indicate its membership of one or more of 12 virtual files. Each of these virtual files also represents the natural join of each of their implied files thus providing a considerable amount of processing precompiled[12]. Eight of these virtual files are shown as relations where the attributes that can uniquely identify an item are to the left of the colon; the attributes to the right do not suffice for unique identification.

ACTION [A:TTP,NA,LOT,DES]	- is the list of all possible tests an engineer can perform. This contains information on how long each action takes (TTP), number of times used (NA) and level of training required to perform the task (LOT). DES is a character string describing the action.
SYMPTOMS [S:A]	- forms a list of all the symptoms and observations made by engineers in performing a particular test.
TOTAL-GROUP [T:SU,CTS]	- this represents the sets of groups of symptoms associated with each subunit and are called Total groups. CTS is an attribute showing the total number of symptoms associated with each Total group.
TOTAL-GROUP-SYMPTOMS [T,S:]	- this shows what symptoms belong to which total groups.
KEY-GROUP [K:T,CKS]	- is the set of necessary and sufficient symptoms within the total groups that can isolate a replaceable unit. These are called key groups. CKS is an attribute showing the total number of symptoms associated with each key group.
KEY-GROUP-SYMPTOMS [K,S:]	- as for TOTAL-GROUP-SYMPTOMS but for key groups.

SUBGROUP [G:K,SUCC,NG,SIT,CGS]	- is the set of necessary but *not* sufficient symptoms within the key groups. These are called subgroups. This also indicates the success of a particular subgroup in tracing the correct replaceable unit (SUCC), number of times used (NG) and investigation time (SIT). CGS is an attribute showing the total number of symptoms associated with each subgroup.
SUBGROUP-SYMPTOMS [G,S:]	- as for TOTAL-GROUP-SYMPTOMS but for subgroups.

The CAFS MK I hardware cannot, in itself, discover whether there exists a key or subgroup which is a subgroup of the observed symptom set (more than one symptom). However, it can be done by software simulation of bit maps[7] and count maps[1]. For purposes of effective use of the maps, the target identifier values (in this case the group identifier) must be transformed into a map pointer (each bit or count word in the map corresponds to one value).

All codes in CRIB such as symptom and group identification are four digit numerals which can be used as relative addressing to a word in store. Let KI and GI refer to the four digit numeric and subgroup identifiers, respectively. CKS, CGS are the group counts, i.e. the number of symptoms in the groups. *Sx* refers to a member of the observed symptom set (i.e. one of the symptoms observed by the engineer translated into the four digit code).

The CAFS tasks*

(1) LIST KI, CKS FOR (S=Sx) IN KEYGROUP-SYMPTOMS

(2) LIST GI, CGS FOR (S=Sx) IN SUBGROUP-SYMPTOMS

are performed for each number of the observed symptoms *Sx*. The retrieval group identifiers KI and GI each point to a word in their respective count maps and add in one to the count word associated with each group found (see Fig. 4).

These numbers in the count maps, after all the symptoms have been dealt with, represent the number of symptoms observed by the engineer that are found in each group and for comparison the maximum number of symptoms possible in each group (CKS and CGS) are also retrieved (the fact that in some cases it is retrieved several times does not matter). A match is found if the number in the count map is greater than zero.

---

*The terms used are intended to be reasonably self-explanatory, but Addis and Hartley[1], Maller[6] and Babb[7] and Addis[12] may be found helpful.

If a match is found, the remaining details of subunit (and keygroup in the case of a subgroup match) are needed. There are two more tasks:

(3) LIST SU FOR (KI = Kx or Ky or . . .) IN KEY-GROUP

(4) LIST SU, K FOR (GI = Gx or Gy or . . .) IN SUBGROUP

Here Kx, Ky . . . and Gx, Gy . . . refer to those groups which have been successfully matched. It is conceivable that more than one subunit will be found, although such ambiguities are not desirable since the whole idea of a keygroup is to uniquely identify a particular subunit.

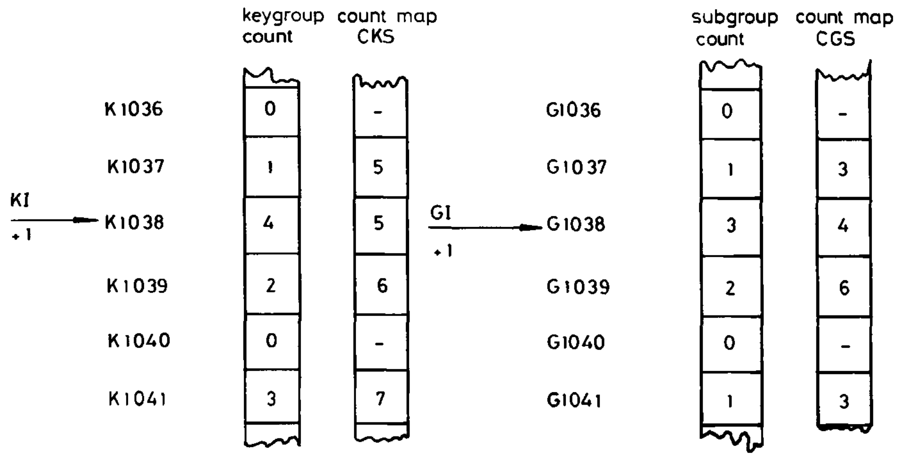


Fig. 4 Examples of count maps

To *minimise* the search those subgroups which are partially matched but which contain at least one symptom which is the inverse of an observed symptom, can be eliminated from further consideration. The task to do this is:

(5) LIST G FOR (S = N1 or N2 . . .) IN SUBGROUP-SYMPTOMS

where N1, N2, etc., are the inverse symptoms of the observed symptom set.

The action details for unmatched symptoms in partially matched subgroups are required for assessment. The count map used in (2) gives the partially matched subgroups (see Fig. 4); i.e. those with a count greater than zero and not equal to CGS. For each of these groups Gx, a task is needed:

(6) LIST A, TTP, DES FOR G = Gx & (S $\neq$ (S1 or S2 or . . .)) IN SUBGROUP SYMPTOMS

again S1, S2 . . . are the set of observed symptoms.

3.2.3 An example of the use of CRIB: Given that the CRIB system is ready for use the engineer will identify himself and then report the initial observed symptoms. This can be done either by a set of cryptic English statements or by an internal code. To illustrate the protocol consider an actual example of an engineer using the system to find a fault. The user's responses are underlined and comments are in brackets.

← LOG IN FRED, HERE
INVESTIGATION STARTED AT 10.35 ON 12.11.78

← PROGRAM LOAD FROM DISC NOT OK
SYMPTOM ACCEPTED AS N0022. ANY MORE SYMPTOMS?

← PROGRAM LOADED FROM CR NOT OK
SYMPTOM ACCEPTED AS N0026. ANY MORE SYMPTOMS?

← ALTER AND DISPLAY ONES OK
SYMPTOM ACCEPTED AS S0402. ANY MORE SYMPTOMS?

← NO (Having accepted three symptoms, the program starts its search and makes the following responses.)

SUBGROUP MATCH FOUND IN CPU.
(Action assessment starts.)
42 SUBGROUPS EXAMINED.
9 SUBGROUPS ELIMINATED.
59 ACTIONS ASSESSED.

SUGGESTED ACTIONS:

* CHECK FOR A PARITY ERROR - A0008
* LOAD MPD WITH 2 LOG MOVES - A0415
* CHECK HAND SWITCHES INDIRECT - A0418
* RUN WAMS - A0389
* CHECK RUN LIGHT - A0413

(Here five actions have been selected as being the most effective ones from the 59 which were considered; terms like MPD, WAMS are codes which the engineer would understand.)

← SO N0008, N0415, S0413 (This illustrates the short form of reporting observed symptoms where the 'A' of the action code given is replaced by S or N depending upon the result of the test.)

ANY MORE SYMPTOMS?

← NO (The second cycle begins.)

SUBGROUP MATCH FOUND IN CPU.

68 SUBGROUPS EXAMINED (Note that there is still insufficient information for CRIB to isolate the fault any further although the search has been widened.)

```
39 SUBGROUPS ELIMINATED.
58 ACTIONS ASSESSED

SUGGESTED ACTIONS:

*  CHANGE BOARD 9714 - A0405
*  CHECK HAND SWITCHES INDIRECT - A0418
*  CHANGE BOARD 9716 - A0421
*  CHANGE BOARD 9710 - A0422
*  CHECK STEPS OF DISC IPL ROM - A0406

LOG OUT OK
```

(In this case the first suggested action cleared the fault. The engineer was satisfied the diagnosis was complete and terminated the run as OK.)

```
INVESTIGATION FINISHED AT 10.40 ON 12.11.78
DURATION OF INVESTIGATION 4.5 MIN.
```

3.2.4 Assessment of CRIB: There were several difficulties which arose out of the use of CRIB. The first problem was perhaps the tendency of the CRIB system to spread its considerations over more and more groups every time a symptom was added. This had the double effect of, first, slowing the response because of the greater number of tasks to be generated and second, the insertion of apparently irrelevant actions into the suggested list because of it making contact with distant groups. This problem really reflects the method used in the CAFS search and could have been contained by using the hierarchical model (see Fig 2) to constrain the search to the relevant part of the machine under consideration.

A second problem was the uncertainty of how to answer some actions to select the desired symptom. This problem falls into two distinct parts.

The first is the problem of the jargon English interpreter which, though perfectly adequate when dealing with a handful of contextually dependent sentences, was found to be incapable of handling a large range of open-ended descriptions. The engineer could never be sure either how to phrase a sentence so that it would be accepted or if it was accepted whether it had been identified correctly. However, users of the system quickly learnt a few starting sentences and then employed the returned action code with the appropriate prefix to continue. The second part of this problem was that some answers to an action were ambiguous. If a symptom was observed there could be uncertainty as to whether a positive or negative response to the suggested action was expected.

Another (third) problem, which is really a consequence of the second, was that there was no way in which a set of symptoms in a new group (indicating a freshly discovered fault) could be checked to see if they already existed. Trial and error, or scanning down all the existing symptoms or using some kind of selector mechanism based upon the concatenation of words in a sentence (keywords) might help, but they are all uncertain methods of checking. There was also the related update problem in that no guarantee could be given that there existed a distinguishing symptom in each group that would ensure isolation of a fault. This last problem could be overcome by employing the diagnostic techniques during update.

Despite these difficulties (most of which have solutions) the system proved to be successful and in particular the suggested actions helped the engineer to consider areas of investigation which he would otherwise have overlooked. This had the useful result in that even if the fault being investigated was *not* to be found within the CRIB system it could still help the engineer isolate this new fault by suggesting avenues of enquiry.

3.3 A Simplified CRIB

3.3.1 Description: Shortly after the CRIB (CAFS MK I) system was implemented there was the suggestion that it could be used for system software investigations. In particular it could be used for VME/K and DME known faults by acting as the initial pre-scan before technical specialists were involved. With a considerable amount of help from those technical specialists and after several variants of CRIB and its database had been tested a simplified version of the CRIB system was constructed.

The concept of subgroups was abandoned, leaving just total and key groups. Each action was given a priority ranking instead of 'time to perform' (TTP) and marked either P (for patch) or Q (for question). Each key group (now called Trace) was given a 'likelihood' which acted as a priority modifier for all the actions within the key group. This priority was used as one of the factors in ordering the suggested list of actions returned to the user. The user then had the choice of the following: *either* having the returned suggested actions ordered according to the number of hits and (for those with the same number of hits) ordered according to the priority modified by 'likelihood' *or* having the suggested actions ordered according to the modified priority only.

The user also had the choice of listing out any number of the ordered actions he wished instead of just the first five. 'Likelihood' was intended to indicate the probability of a symptom group occurring so that as the database is updated so this parameter would be adjusted for each group.

The problem of restraining the CRIB system from contacting distant groups was left in the hands of the user by providing him with two distinct selection mechanisms. The first mechanism used the count maps in three ways:

(*a*) If the user has reported m symptoms then he can select from the count map only those groups which have a count greater or equal to $(m - n)$ where n is an integer of his choice. If n is zero then this is equivalent to insisting that *all* the symptoms given are contained in the groups retained for consideration.

(*b*) Irrespective of how many symptoms the user has reported the system can scan the count map looking for the maximum number of hits and only reporting on those groups with this maximum number. A simple extension of this idea is to allow (Max - n) as the selection mechanism.

(*c*) Count map selection can also be done relative to the total number of expected symptoms in a group (CKS in Fig. 4). The selection in this case is done by subtracting n from each group number (CKS) and if the count is greater than or equal to this then it is returned. Another way of describing this method is that it selects those groups which have at least a specified number of symptoms in total. Smaller groups than this number behave as (*a*), but the smaller the group the more symptoms are ignored. The number of symptoms given by the user + n = maximum total in groups.

The second mechanism employs a modification of the unit-subunit hierarchical structure used in CRIB (Fig. 2). In this case the hierarchical structure represented an organised approach to diagnosis constructed from the experience of the experts, where each level in the hierarchy represented sets of questions concerned with details of the level above. A five-figure code was constructed so that each character position represented a level and the character chosen gave the branch at that level. Fig. 5 shows part of this structure where the code at each level is given in brackets and each level is associated with a priority number (the lower the number the higher the priority). The user is then able to confine his search to any subsection of this hierarchy by typing in for example SU 32B** or SU **B*D where *represents a 'don't care' character.

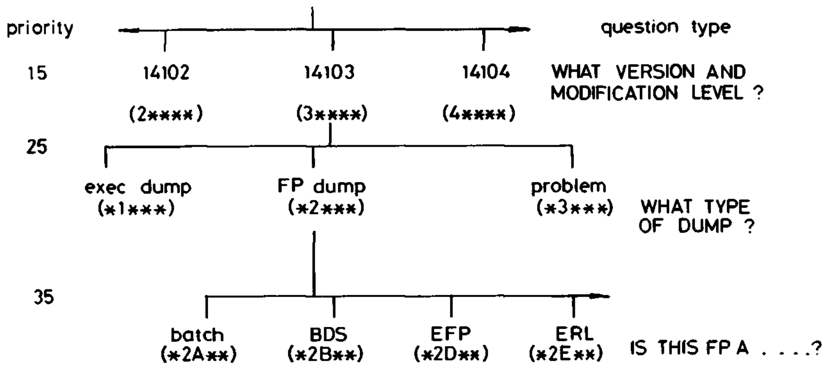


Fig. 5 Part of the VME/K faults database structure

The problem of ambiguous answers to actions (questions) was solved by simply allowing only one answer to be accepted - namely 'YES'. Thus the question 'is operating system version 14?' would either be answered 'YES' or not answered at all. This decision to have only one answer leads to other problems, but these other problems are easier to cope with for this application than ambiguity.

3.3.2 Assessment of simplified CRIB: The simplified CRIB is more of a tool in the hands of experts than the original since many of the strategies of pattern matching are decided by the user. The speed of response was considerably improved and the probability of success greatly enhanced with most of the original problems bypassed. The only outstanding problem was that of ensuring that symptoms were not repeated in another form when updating occurred.

This problem was partially overcome by classifying symptoms. Each class then consisted of a sufficiently small number of symptoms to be scanned by eye. It was intended to make an extended trial of this modified system but conflicting demands made it necessary to curtail this; however enough was done to show the importance of involving, at quite a detailed level, the experts and the end users in the development of such a tool. The final system required less than five minutes explanation to an end user for him to be using it efficiently. *The importance of such a tool as this is that it represents a medium by which an expert can express and record his expertise in a way that is self maintaining.* This is why it requires the specialist to create the database.

3.4 The RAFFLES system

(A fault-finding system currently being developed at Stevenage for use by Customer Services.)

3.4.1 Description: The CRIB systems were designed for a single user and depended upon fast data retrieval and processing, giving maximum control to the user. The objectives of RAFFLES, on the other hand, were to make a diagnostic aid available to a large number of people simultaneously without the need for expensive processing. As has been suggested in Section 3.1, and from observations of users of CRIB, the diagnostic sequences tend to be highly constrained. This results in much of the online working of CRIB being continually repeated at a time when minimum processing is required. Other problems of CRIB have been interpreting the symptoms (expressed in jargon English) and the control of symptom updates (see Section 3.2.4).

These problems can be overcome by the proposed Rapid Action Fault Finding Library Enquiry Service (RAFFLES) which can be considered as a compiled version of CRIB but with a test selection scheme based on information theory rather than heuristic methods. Since all the decision making is done offline this means a very simple online program. This scheme is designed to draw together the mass of different test procedures used and discovered by many engineers into a unified approach to fault finding. The system comprises principally two programs: Generate Tree and Display Tree as shown in Fig. 6. Generate Tree generates the optimum fault location procedure in the form of a multibranching decision tree. Each node of the decision tree is a test and each branch a result of that test. The decision tree is interpreted by displaying the tests at the top of the tree and according to the result provided by the engineer steers him down to a terminating node which represents the unit to be replaced. RAFFLES employs a technique which was successfully used for pattern recognition.

The RAFFLES system should have the advantage of being extremely fast; requiring minimum computer power, since all the hard work of selection and scanning is done offline by Generate Tree. The selection of test criteria is based upon the well understood and tried techniques of information theory which provide the fastest test sequences for fault location (which can also be issued in book form using program Print Tree), expose the existence of any equivalent or fallacious tests and avoid sequences of similar tests. The need for jargon English input becomes de-emphasised but is still capable of being supported when available, and the interactive protocol provides an ideal system for using speech as a means of communication. However, as with CRIB, the effectiveness of the RAFFLES system in locating faults depends entirely upon the quality of the data digested by the Generate Tree program. This will accrue over a period of time from actual results in the field as the file accumulates via program UPDATE (see Fig. 6).

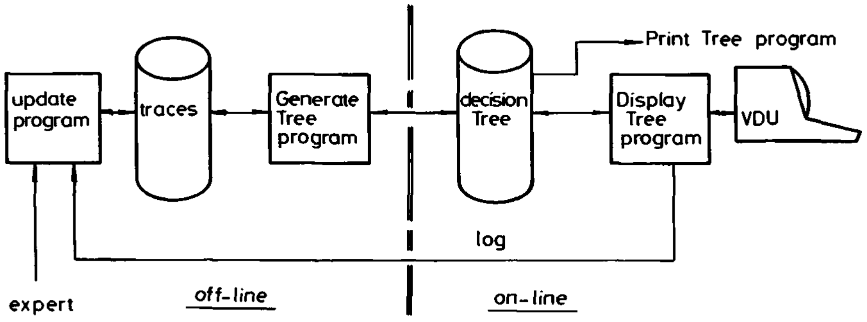


Fig. 6 The RAFFLES system

3.4.2 Theory of the Generate Tree Program: The objective of the Generate Tree program is to sift through examples of successful fault location traces (e.g. groups of symptoms that lead to a fault) associated with each replaceable unit and generate an optimum fault location guide in the form of a multibranching tree. The first node of the tree represents the best test which divides the classes into as near to equal-sized and stable groups as possible.

The second set of tests in the next level down the tree repeats this requirement for each divided group and so on until the final test in a branch isolates a replaceable unit. The measure of 'best' for a test requires that the test is both reliable and provides an equal division between the classes. In information terms, this represents the test that brings about the greatest reduction in relative entropy at each level in the hierarchy and in this case, where tests can take varying lengths of time, the greatest reduction in relative entropy per unit time.

The faults associated with each replaceable unit (or fault report in the case of software) can be considered as a series of independent symptoms that identify that unit. A trace is an actual example of a fault identification. Each symptom represents the result of an action where each action (usually a test in the form of a question) can have one or more possible outcomes (answers). The effect of dividing the database of traces (symptom groups identifying a fault) according to the result of any particular test is to reduce the entropy of the whole system with respect to the faults. That is, the faults become partially ordered. The best test will be the one that reduces the entropy by the greatest amount in the shortest possible time.

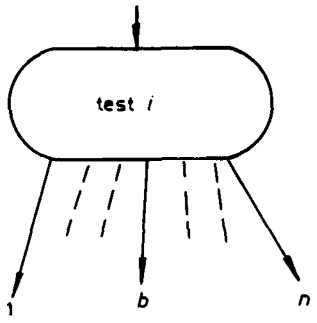


Fig. 7 Test i with n associated symptoms

The determination of such a test can be achieved by considering the statistics of each test when used without any constraints, such as in the CRIB system. Let the range of tests be $1 \leqslant i \leqslant T$, the range of possible outcomes of test i be $1 \leqslant b \leqslant n$ and the range of possible faults, or of replaceable units in the case of hardware, be $1 \leqslant k \leqslant K$.

We define the following probabilities:

$p(k/b, i)$	=	probability that fault k is present (or unit k is faulty) given that test i has outcome b.
$p(k, b/i)$	=	probability that test i will have outcome b and that fault k is present
$p(b/i)$	=	probability that test i will have outcome b
$p(k/i)$	=	probability that test i will give the conclusion that fault k is present.

It can be shown that given any outcome b for test i the entropy with respect to the set of K possible faults (or replaceable units) is

$$H_{b,i} = - \sum_{k=1}^{K} p(k/b, i) \log_2 p(k/b, i).$$

Then the expected entropy $E(H_{0,i})$ of the outcome, given the test i, is

$$E(H_{0,i}) = \sum_{b=1}^{n} p(b/i) H_{b,i}.$$

This becomes after suitable manipulation

$$E(H_{0,i}) = - \sum_{b=1}^{n} \sum_{k=1}^{K} p(k, b/i) \log_2 p(k, b/i) + \sum_{b=1}^{n} p(b/i) \log_2 p(b/i).$$

The input entropy (i.e. before the results of any test are known) is

$$H_{I,i} = - \sum_{k=1}^{k} p(k/i) \log_2 p(k/i).$$

The expected decrease in entropy $E(\Delta H_i)$ resulting from applying test i is $H_{I,i} - E(H_{0,i})$ and from the above equations this is

$$E(\Delta H_i) = - \sum_{b=1}^{n} p(b/i) \log_2 p(b/i) + \sum_{k=1}^{K} p(k/i) \sum_{b=1}^{n} p(b/k,i) \log_2 p(b/k,i)$$

If the test i takes t_i seconds to perform, then $E(\Delta H_i)/t_i$ is the entropic decrement per unit time.

Test i is chosen such that $E(\Delta H_i)/t_i$ is maximum for the set of traces under test. This having been chosen, the next set of tests is calculated independently for each of the resulting divisions of these traces caused by applying test i.

A full description of these calculations is given by Addis[2].

3.4.3 Using decision trees: The process of creating the decision tree is now simple. A database of actual test sequences that led to the location of a fault is analysed according to the above formula. This will provide every test with a measure of excellence in the form of the entropic decrement per unit time $E(\Delta H_i)/t_i$. These can then be ordered according to this measure of excellence and this ordered list of tests is now equivalent to the initial ordered list of actions that can be printed by CRIB. However, the Generate Tree program forces the issue by taking the test with the highest value (the test at the top of the list) and proceeds to use that test to divide the database of traces into n possible groups, n being the number of possible outcomes of the test. Fig. 8 illustrates this process.

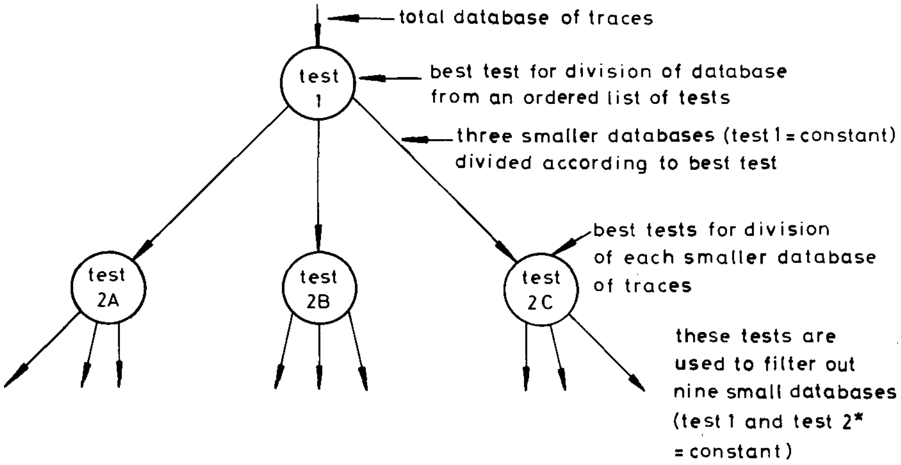

total database of traces
test 1
best test for division of database from an ordered list of tests
three smaller databases (test 1 = constant) divided according to best test
best tests for division of each smaller database of traces
test 2A
test 2B
test 2C
these tests are used to filter out nine small databases (test 1 and test 2* = constant)

Fig. 8 The Growing Tree

There are three useful byproducts of this technique which resolve a problem with the CRIB system. The first is that symptoms which carry the same or similar information irrespective of how they are expressed will have their measure of excellence reduced in sympathy with the chosen test. Suppose test z is chosen as the next test along a certain branch of the tree instead of say tests x or y. After the database has been divided according to that test, and if it is found that test x which had formerly a high entropic decrement value is now low, then it is reasonable to conclude that x and z are equivalent tests and should thus be combined. The second useful byproduct is that if a branch of a tree is reached where there are no more possible tests to distinguish between faults then this will be discovered and an appropriate request can be made to the overseeing expert for a distinguishing test.

The third byproduct is that tests which carry little or no information can be discovered and discarded.

The method of selection of tests in creating the tree does not take into account any other factors which might influence the choice of test. Certain tests may be too difficult for the user to perform, or too trivial to be worth considering because the general fault location is obvious, or just not possible because they depend upon certain equipment. Further, the engineer may just have additional knowledge of this particular environment which would guide his choice.

Some of these problems can be overcome by creating different kinds of trees for different purposes and by providing a mechanism for using the trees effectively with inadequate information. The existence of a test at any point in the tree is totally dependent upon the sequence of answers to the tests that led to it through the tree. A test out of context has little or no meaning to the system. However, there are at least three possible strategies which can resolve this problem.

(*a*) It is possible to assign each test to a classification representing the skill required to perform that test. Different trees can be created according to the level of skill of the users by selecting only those tests that are within an ability range.

(*b*) The 'patient' mechanism can be considered to be constructed in some hierarchical fashion as for CRIB (see Fig.2).If this is done then a tree can be grown which calculates the entropic decrement with respect to a level in this hierarchy instead of the final replaceable units (or fault document).

Fig. 9 illustrates a simple tree that decides the general area of the fault down to the first level. Once the general area has been selected another tree can be grown which calculates the entropic decrement with respect to the next level in the hierarchy. This is continued until a set of trees is formed that isolates all the faults (the lowest level in the hierarchy).

The advantage of such a system is that any expert user can jump *directly* to the appropriate level in the hierarchy thus bypassing the intermediate stage of answering the often tedious set of questions that would get him there. Each tree is constructed with the assumption that the user has correctly classified the general area of the fault. The method by which this classification has occurred is irrelevant so that any tree grown from that point has tests dependent only upon this correct classification. The disadvantage is the extra work required by the Generate Tree program to produce these different trees and also that the total number of tests to find a fault is likely to be more if a user were to start from the top level. However, this second point can be avoided by providing trees at all levels that home in on faults directly.

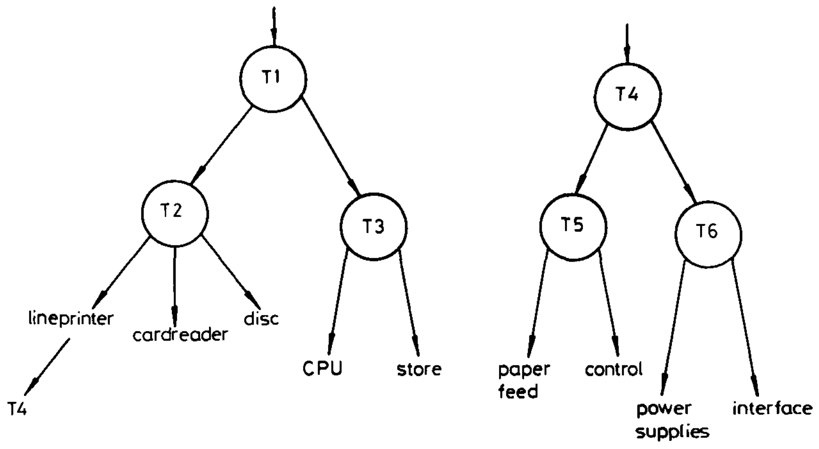


Fig. 9 A top and second-level tree

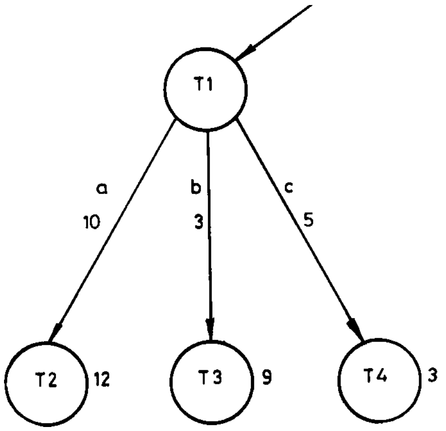
(*c*) When a question, given to a user from the RAFFLES system, is answered by 'Don't know' a problem arises in that the RAFFLES system then has to simultaneously progress through every possible answer for a given tree. If other questions are subsequently answered by 'Don't know' then this will create a very large number of simultaneous paths to be traversed. The management of such a multitude of traversals becomes extremely difficult.

A solution is *not* to traverse the paths simultaneously but consecutively choosing the most-likely-to-succeed path first. If a solution is found before all the paths have been traversed then the process can halt. This process requires that every answer to a question at a particular node has an order of precedence indicating which answer will lead to the most likely conclusion.

This order is directly related to the entropy associated with each branch H_{bi} (see Section 3.4.2). The greater the change in entropy of a branch the higher the ranking. The highest ranking answer which has *not* been tried before is automatically tried and marked. It is interesting to note that this gives a similar result to the CRIB heuristic methods in that any branch (symptom) that leads to the isolation of a fault (or level) will always give the greatest change in entropy. This is because the entropy will fall to zero. Consequently, as for CRIB, the last symptom in a group will always be suggested for testing before other routes are tried.

The user can then be given the option to continue until the final node of a particular path *or* backtrack to the last (last-1, last-2 etc.) marked node *or* both. The user can be given control to re-enter a path that was abandoned before the final node was reached with the option to continue (with perhaps a summary) from where the path was previously abandoned.

Rather than keep the user completely in the dark the next group of tests beyond the one under consideration can be presented in an ordered list according to the reduction in entropy taken to reach them (see Fig. 10).



T1

ΔH_b	ans	$E(\Delta H)/E$	Next test
(10)	a	(12)	T2
(5)	c	(3)	T4
(3)	b	(9)	T3

Fig. 10 A look ahead table for the end user

So far the assumption has been that every test provides a few distinct outcomes which can be represented as a branch in a tree. In principle any test can be broken down into a sequence of primitive tests with a binary outcome. If this is done then the problem of answering such sequences can be achieved by running these primitive tests as a program. For example, if a computer store address is required and the system is aware of a set of possible computer addresses then it would not be reasonable to expect the user to answer all the 'is the address = x' for every value of x. The principle of the Generate Tree program can still be applied to this sub-task so that the questions can be asked in the most efficient order with the user supplying only the address value (see Fig. 11).

Some results of tests can be potentially infinite but the range is limited. Such tests are the measurement of some voltage, current or length. In these cases the entropy of the system becomes a function of the result (see Fig. 12). The choice of tests to follow may depend upon threshold values of the function.

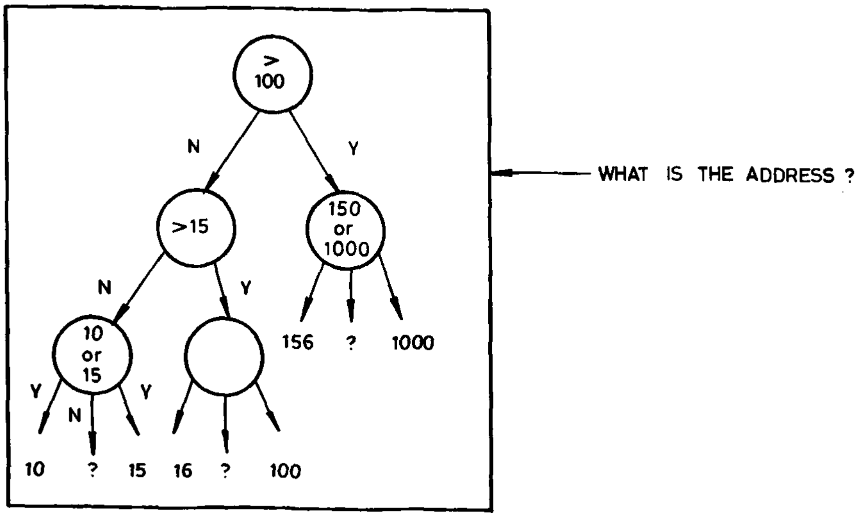


Fig. 11 A single complex test as a set of primitive tests

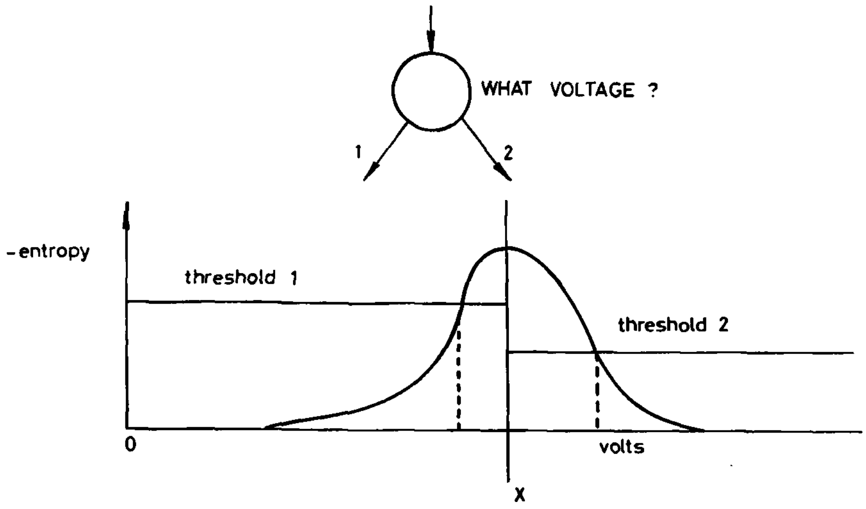


Fig. 12 A continuous function of entropy

It is important to note that any expert providing a sequence of tests for isolating a new fault for the RAFFLES (or even CRIB) system should choose tests which are going to provide a hierarchical division of the problem.

If most of the tests are too general or specific, then the Generate Tree program will generate a deep and imbalanced tree. Experts updating the diagnostic system should be made aware of both what is required and how their particular additions are assimilated.

3.4.4 Combining RAFFLES and CRIB: There are two problems with the

RAFFLES system which are not found in CRIB. The first problem is that the user is driven and constrained by the tree. This means that any extra knowledge about the problem the user may have cannot be fully utilised by the fixed diagnostic procedure. The advantage of this fixed procedure is that it sidesteps the difficulties of 'natural' language interpretation (see Section 3.2.4). The second problem is that towards the terminating tests in the tree, the statistics upon which the entropy measure is calculated start becoming unreliable because of the fewer number of examples upon which they are based. However, CRIB does not suffer from either of these difficulties because any errors due to statistical anomalies are adjusted by the common sense of the user. On the other hand, CRIB requires a considerable amount of processing power to provide this flexibility.

The solutions to these problems are not incompatible and it is possible to achieve a balanced solution by combining the advantages of both systems and reducing the disadvantages. Most of the CRIB processing is due to the quantity of data that has to be sifted and compared. One of the properties of a RAFFLES tree is to divide a very large database into many small databases at each level. So if a RAFFLES tree is constructed up to the point where the statistics become unreliable then this will provide many small databases each of which can be scanned with much reduced processing by the CRIB system. This combined system is illustrated in Fig. 13.

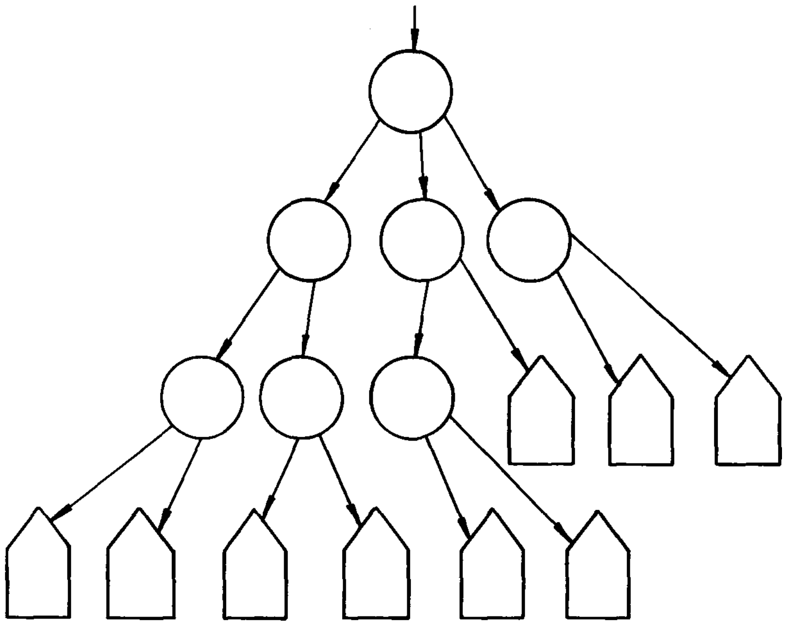


Fig. 13 Using a tree to create many small databases

3.4.5 Comment: In practical terms RAFFLES provides a predigested faults database accessed by disc. However, unlike any normal retrieval system it provides tests for the user to perform which have a high probability of success; the database is self maintaining and the online interactive program (Display Tree program) is simple.

3.5 New faults problem

RAFFLES and CRIB are concerned only with those faults that have been encountered previously. It is estimated that 30% to 50% of faults investigated in software and hardware are previously unencountered faults. It would be desirable to complete the diagnostic system by providing an aid for isolating these new faults. The difficulties, when anybody is faced with determining why a mechanism is not working, are what tests should be performed and what current observations are relevant. Previously, for encountered faults, this was simply repeating what had proved successful in the past. It now becomes necessary to generate new tests and make fresh observations in the light of what is already known.

For a system to generate tests on a mechanism (or to test the 'correctness' of current observations) requires that there is some representation (model) of the component parts of the mechanism; how these component parts interact and what happens when a component part fails in a particular way. Progress is being made on representing the medium to large range of machines down to gate level in order to generate diagnostic test programs[8]. This kind of precision, although leaving nothing to chance, can be much too detailed for a general fault finding guide. (However, this kind of detail is required for the automatic generation of test programs.) A more promising approach is a hierarchical description of the mechanism at the appropriate levels of coarseness. This description should be biased towards the task of fault finding rather than machine emulation.

A method for describing and running mechanisms in the abstract is currently being researched (known internally as the Broadside Programming project). This method is based upon principles developed by those involved in 'natural' language understanding by machine[9] and is currently in the feasibility study stage; whilst in this state of examination no details will be given in this paper. Machine understanding means that interpretation of any sentence, paragraph or story does not depend entirely upon grammatical constructs, but relies upon a working representation of what is being said in the light of previous accumulated knowledge. The difficulty of developing such representations depends upon choosing the correct elementary building blocks and primitive processes that can cope with a wide range of descriptions. Language analysis has shown that such elementary units exist but these units have to be modified quite extensively to fit our particular requirements of modelling mechanisms.

There are two provisional conclusions that can be drawn from this work. The first is that the concept of high and low-level languages vanishes. The objects manipulated by a high-level language are just different from those of a low-level language. Level, in fact, lies outside the language domain altogether since it describes the relationship between two sets of objects via an intermediary. The second is that there is no definable entity called 'natural' language. What comes to be called 'natural' language actually represents a pot-pourri of many different languages each of which is a highly honed tool for a particular task. The reason for the confusion is that there is a strong family resemblance between these languages since they were born out of the way man perceives reality.

This work, although only just begun, can expect to resolve the last of the problems experienced with the diagnostic system. It should now become possible to define a diagnostic language which will have some features of 'natural' language. This language would be used not only to describe the mechanism, but also to report symptoms. The products of the system (supporting such a language) will be useful views of the mechanism given by expert fault finders, which would be used for both generating suggested tests and checking the consequences of current observations.

4 Conclusions

This paper has described the 'potential' evolution towards an 'expert' diagnostic system. It is 'potential' because it has not yet been fulfilled. The CRIB system has been implemented and forms the basis of the pattern-driven-question-and-diagnostic component of the diagnosis skill (see Fig. 1 Section 3.1). The RAFFLES system is under construction and represents the fixed-sequence-of-questions component of this skill giving the most efficient fault finding guide. The NEW-FAULTS system is being researched and provides the problem-solving aspect still missing.

Independently, each of these three systems provides unique benefits to the user but they also have defects that are cancelled out by the existence of the other two systems. The defects with CRIB were essentially the need for describing symptoms so that they could be uniquely identified and the large processing requirements necessary for determining the next step. RAFFLES, on the other hand, cancels out all these problems by side-stepping the user symptom descriptions by printing out questions to be answered, identifies similarity of symptom descriptions on the database by their entropy behaviour and avoids doing the processing during diagnostic time. However, it loses the flexibility of CRIB that allows the experience of the user to be used to good effect. By judicially combining CRIB and RAFFLES, flexibility is to some extent restored without an excess of processing. The NEW-FAULTS system will probably also need a considerable amount of computation but it should only be required part of the time for NEW-FAULT resolution. The additional gain of the NEW-FAULT system is that it will provide a technique for describing symptoms uniquely and give lines of reasoning for any particular diagnosis*. The combined systems of CRIB, RAFFLES and NEW-FAULTS should interact to form a complete 'expert' diagnostic system.

The important difference between an 'expert' diagnostic system and that of a database retrieval system is that it provides guidance to the user, it has a database that is self-maintaining and it provides a medium through which expertise can be expressed. Without the 'expert' there can be no 'expert' system.

* It is interesting to note that such a complete system would not always be able to 'explain' why a certain sequence of tests will produce the desired result because the source of such information would come from the accumulated knowledge obtained from experts' experience. This is equivalent to any expert's 'rule of thumb'; knowing that a certain sequence works without knowing why.

5 Acknowledgments

Since this work was initiated by A.H. James (TS) and Prof. F. George (Brunel University) there have been many people who have provided invaluable help for this project. Thanks should be given particularly to Dr. R. Hartley who did an excellent job under difficult conditions to transfer his CRIB software to work with CAFS (Content Addressable File Store); D. Icke and his team of software specialists who worked overtime to capture their skills within the CRIB system and showed us clearly that you cannot have an expert system without the experts; A. Ingram and his team who are currently involved in the difficult task of making practical the theoretical concepts of RAFFLES; and G. Piper who has managed to make this co-operative venture work within an ever changing environment.

References

For the sake of completeness references to a number of ICL internal notes and reports are given. Not all of these are available outside the Company; any reader who would like to consult one of these is invited to write to the author.

1 ADDIS, T.R. and HARTLEY, R.T.: 'A fault-finding aid using a content-addressable file store'. Technical Note TN 79/3 June 1979.
2 ADDIS, T.R.: 'A feasibility study of RAFFLES' Technical Note TN 78/3 August 1978.
3 ADDIS, T.R.: 'Report on the AISB Summer School on expert systems in the micro-electronic age'. Internal Note TRA. 79/6 July 1979.
4 FEIGENBAUM, E.A.: 'Themes and case studies of knowledge engineering'. Published in 'Expert systems in the micro-electronic age' (Proceedings of AISB Summer School), Ed. Michie, D. Edinburgh University Press.
5 YOUNG, E.M.: 'Production systems for modelling human cognition'. Published in 'Expert systems in the micro-electronic age' (Proceedings of AISB Summer School), Ed. Michie, D. Edinburgh University Press.
6 MALLER, V.A.J.: 'The content addressable file store - CAFS' *ICL Tech. J.*, 1979,1 (3), 265-279.
7 BABB, E.: 'Implementing a relational database by means of specialised hardware'. *ACM Trans. Database Syst.* 1979, 4 (1), 1-29
8 CORRIN, P.G. and McLAREN, K.: 'System diagnostics – achivements and future work, July 1979'. STDC Information Note SOFTECH/IN/500, issue 0.
9 ADDIS, T.R.: 'Machine understanding of natural language' *Int. J. Man-Mach. Stud.*, 1977, 9, 207-222.
10 SHANNON, C.E. and WEAVER, W.: *Mathematical theory of communication*, Urbana, University of Illinois Press, 1964.
11 FU, K.S.: Sequential methods in pattern recognition and machine learning. London and New York, 1958, Academic Press.
12 ADDIS, T.R.: 'A relation based language interpreter for a CAFS'. Technical Report 1209 August 1979.

Using Open System Interconnection standards

John Brenner

Consultant, ICL Product Development Group, Technology Division, Manchester

Abstract

Open System Interconnection standardisation has important implications for ICL and its users. The paper assesses these, and discusses use of the ISO Reference Model and how it affects network architecture.

1 What is OSI?

Open System Interconnection (OSI) is the concept that any data-processing system should be able to communicate readily with any other, by use of 'Open System Interconnection' standards.

Interconnection is only directly possible among parties using the same interconnection rules, that is the same set of data-communication protocol and interface standards. Universal interconnectability implies universally used international standards. For acceptability and accountability reasons, the standards should be non-partisan and in the public domain. Therefore, the goal of universal and truly 'open' interconnection is only achievable by formal international standardisation activities.

The scope of OSI standardisation is necessarily very wide:

data-communication

data-processing support

applications.

Data-communications standardisation (e.g., network, data link, communications equipment and terminal standards) is only a small but vital component of the whole. The greater part is standardisation for DP support services so that meaningful interworking is possible, e.g. dialogue structuring, recovery, device-independence, data independence, file transfer, remote file access, file management, database, remote job entry, load sharing, application and system management, distributed system support, accounting, statistics, diagnostics. The scope of application standardisation for OSI ranges from general purpose application services such as text processing, message services, electronic mail and facsimile, to specific applications which may be openly accessible, such as electronic funds transfer or airline reservations.

For a few of these there are existing standards, some of which are unsatisfactory or mutually incompatible. Most of the work is new, and much of it is very urgent. Therefore, OSI is a very large, prolonged and challenging undertaking for those who devise and implement standards.

2 Why OSI?

One might equally ask 'why not OSI?'.

It is intuitively obvious that the world would be a better place without the incompatibilities which currently make interconnection so difficult. Conversely, it is equally obvious that such incompatibilities are inevitable at this early stage of data communications. A more meaningful question is 'why OSI *now*?'.

There are two main sets of reasons why the time is considered ripe for OSI. First, data communications is now sufficiently mature for the need for OSI to be properly understood, and for it to be technically and economically possible to satisfy. Secondly, for somewhat related reasons, there is likely to be an explosive growth of the quantity and variety of things to be interconnected, of the degree of their interconnectedness, and of the variety, complexity and intensity of activity via this interconnection. Therefore, standards are needed more than ever before, and if they do not come sufficiently quickly, there might be perpetual babel, with proliferating and ever more deeply entrenched incompatibilities. This has generated a real sense of urgency.

A review of the different motivations of the main parties involved indicates that there are very powerful interests working for successful OSI. They are:

users
manufacturers
PTTs
governments.

The main potential advantages of OSI to users are:

New applications: the readily available connectivity of OSI should make many new and extended applications feasible, with consequent business benefits.

Savings: Systems should be easier and cheaper to develop, install, maintain and use. Skills savings are particularly important.

Multivendor systems: OSI should allow to users a greater choice of suppliers and products; and more competitive procurement.

Although representation of user interests is somewhat fragmented, user needs can be seen as the most important factor in determining the outcome.

Vendors, such as ICL, have exactly parallel reasons of their own for wanting OSI:

New applications: OSI is a vital element of the large new markets needed to sustain the high growth rate of the industry

Savings: OSI promises less variety of interconnection methods than otherwise, and a consequent drastic reduction of product development and maintenance difficulties.

Multivendor systems: for each vendor, open-interconnection opens up the prospect of additional markets, which are otherwise closed to him and are usually much larger than his existing fraction of the total market.

This, of course, evokes the traditional counter argument that vendors do not want such standardisation because it opens their own supposedly captive customer bases to competition. This implied complacency and uncompetitiveness is not really viable in an era of increasingly fierce competition.

The PTTs have the clearest motivation for wanting worldwide open interconnection. The concept of open interconnection is fundamental to their traditional telecommunications business. It is basic to their new Public Data Networks and to their new services such as teletex, videotext and facsimile. The PTTs have an excellent track record of prompt and effective standardisation via their CCITT organisation. They have the power and the common interest to develop and enforce at least those aspects of OSI standardisation which are most directly concerned with telecommunications. They are visibly doing this with great urgency.

Governments have two sets of motivations strongly favouring OSI. First, OSI is potentially a means of regulating competition, monopoly power and international trade; but these are somewhat conflicting aims. Second, governments are among the largest and most powerful DP users. The benefits of OSI are therefore particularly attractive to them, especially at a time of economic stringency; and they have a means of enforcement via their procurement policies. Awareness of this is conspicuously visible in the US Federal Government.

We therefore have a potent conjunction of otherwise somewhat disparate interests, which points to some kind of OSI happening quite soon.

3 What kind of OSI?

The most important issue then is what kind of OSI standardisation will emerge. How good is the outcome likely to be; to what degree are the hoped-for benefits actually likely to be achievable?

In a previous issue of the *ICL Technical Journal*, Jack Houldsworth[1] has explained some of the technical complexities of this international standardisation and its many different committee structures. It is clearly apparent that the idealistic outcome of everybody worldwide converging quickly onto one unified set of OSI standards is highly improbable. Different aspects will be standardised by different bodies at different times, and there will be a long transitional period. We will all have to face high conversion costs. Proprietary 'own brand' communications will remain an important factor. Predictably, there will be some competing and incompatible standards from different sources, and certainly some lasting incompatibilities.

Here are some of the things which might go badly wrong:

proliferation and exclusive enforcement of different standards in different parts of the world or in different markets. This will impede more general connectivity and impose upon vendors, who now typically implement only 'own brand' protocols, the burden of supporting an excessive variety of different 'open' standards in order to trade in the various markets. This contradicts the anticipated variety-reduction benefits for users and vendors;

fuzzy standards which fail to achieve real compatibility. A common failing is compromise-by-superset; i.e. to include so many different options in a standard at the request of unreconcilably conflicting interests, such that incompatibility is almost certain in practice, either randomly or systematically by different subsetting;

unfair standards, which favour one party at the expense of others, sometimes aggressively. On the other hand, a less compromising standard is typically better and more timely.

Despite such concerns, it is generally believed that the outcome will on balance be highly beneficial. The half dozen or so most important standards are expected to emerge within a couple of years.

The best hope for compatible and effective OSI standardisation is that all the standards from whatever source should all genuinely conform to one agreed and viable architectural framework. That is the intended role of the International Standards Organisation (ISO) 'Reference Model of Open System Interconnection' which is why the initial standardising activities have concentrated upon developing the Model.

4 Reference Model

This is a brief introduction to the ISO Reference Model, and a summary of its main strengths and weaknesses.

After more than two years of intense debate, the Reference Model is essentially stable, and is generally accepted worldwide, although there are still important disputed issues. ISO and CCITT have both developed models using the same layering technique, and convergence between the two is now almost complete. ISO hopes to finalise the Model within about a year, and then to process it as a standard.

Fig. 1 is derived from the ISO document.[2] It illustrates the structuring of OSI functions into seven layers, each with a distinctive role, and the corresponding seven layers of protocol by which systems communicate with one another. The common architectural principle of layering and the sevenfold modularity are intended to provide a consistent and orderly basis for the technical and organisational subdivision of the standardisation work. The technical content is discussed in more detail later.

There is a curious paradox that the main weakness of the Model is at the same time its greatest strength. The weakness is that it contains many unresolved issues, even as fundamental as different views on what it is trying to model and to which purpose. The text is sufficiently large and varied for supporters of conflicting views to make their own different interpretations to their own satisfaction. On the other hand, this does serve to keep the negotiations going along the same track (or at worst on parallel ones), and allows much wider support. It must be recognised that different interests, such as different kinds of users, PTTs, and different kinds of manufacturers, genuinely have different needs. Also different technical experts have different perceptions of what they consider to be architectural truths. The great strength of the Model is that it allows all to see their different views positioned in a common framework, albeit somewhat imperfectly.

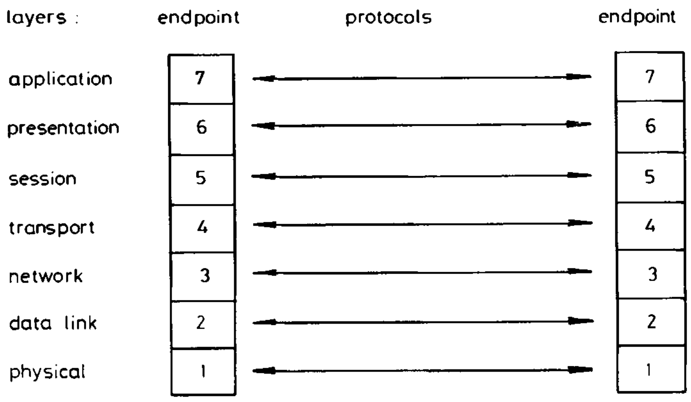


Fig. 1. ISO seven-layered Reference Model.

Another difficulty is that the document is now rather large (180 pages) and is difficult for even the experts to understand in its entirety, let alone for the intelligent lay user. But this complexity is to a degree unavoidable, and different presentations are obviously needed for different kinds of readers.

The consensus view among ICL experts working in this field is that the latest version of the Reference Model, although still requiring further development, can now provide a stable basis for OSI standardisation. Therefore, it should be frozen as soon as possible. Our main interest has now moved to participation in the development of the more urgently needed international standards, mostly via ECMA (the European Computer Manufacturers Association).

These are:

Transport Protocol Standard;
Session Protocol Standard;
Virtual Device Protocol Standard;
File Transfer Protocol Standard;

When completed, it is hoped that these will be successful candidates for ISO standardisation, as has usually happened in the past with other ECMA work.

5 Using the Reference Model

Given that the Reference Model is as yet somewhat imperfect, how is it to be used immediately to develop a consistent set of OSI standards? The approach described here has evolved in discussions with ICL colleagues, and with ECMA colleagues from other manufacturers; but it is not necessarily definitive. What one does is:

(*a*) start by defining precisely the particular use to which the Reference Model is to be put on the occasion in question;

(*b*) then extract from the Reference Model the (small) sub-set which is relevant to this immediate and restricted purpose.

A consistent view can then be obtained fairly readily, as we shall see.

We define the immediate interest of computer manufacturers to be: to understand how to use the Reference Model as a framework for developing internationally standardised protocols to interconnect their various systems and products.

This deliberately restricted viewpoint allows many extraneous issues to be weeded out:

one is not looking at the Reference Model as a 'systems architecture', i.e. as an alternative architecture requiring existing systems architectures to be replaced;

one is not seeking a structure for distributed applications, but only a standard data-path between the application sources/sinks;

one is not seeking the solution to the new technical problems posed by distributed systems, or the definitive 'distributed systems architecture' of the future.

one is not seeking resolution via the Reference Model of the controversial business issue of which services should be PTT provided and which PTT-user provided;

one is not seeking to position exactly in the seven layers such services as teletex, facsimile, file transfer and remote job entry. These issues can be left open without prejudice to their final outcome or to the immediate purpose;

one is not using the Reference Model to develop new theories or for research experiments;

one is not using the Reference Model to define standard programatic interfaces for purposes of software portability;

one is explicitly considering data-transmission via telecommunications; not within-machine communication or media exchange.

These excluded items are all in varying degrees legitimate aims reflected in the Reference Model text mostly as disputed issues which it leaves unresolved. This is a measure of the diversity of views which it seeks to reconcile and of the difficulties involved. For the specialised immediate purpose defined here, these other aims are ignored; but they are not forgotten, and some at least are likely to be the objects of other OSI standardisation.

Nobody is at present standardising the application layer so this can be excluded from detailed discussion. Standards for the lower three layers either already exist, or are in an advanced state. Therefore, primary interest centres upon only three of the layers: Transport layer, Session layer, and Presentation layer. The combined effect of this temporary narrowing of objectives is a drastic reduction of complexity and uncertainty in using the ISO Model.

6 Subset description

This is a brief description of the features of the ISO Reference Model which are relevant to the immediate purpose defined above.

The visible manifestation is layers of protocol, which will each be subject to standardisation. For modularity, each layer of protocol is kept as far as possible separate from and independent of the others. The principal means of layer independence is that each higher layer of protocol is encoded as data of the adjacent lower layer. Since each layer of protocol handles data transparently on behalf of the higher layer using it, there is a strong decoupling. Another means of decoupling between layers is for each higher layer of protocol to depend only upon the simplest possible set of services from its adjacent lower layer.

Different systems, typically with different 'own brand' native modes of communication, will then use these standard layers of protocol to achieve open interconnection among themselves.

6.1 Application layer protocols

Application layer protocols are the data-processing protocols entailed in the actual work on behalf of the end-user: his transaction processing, his system administration, etc. They are many and varied, reflecting the diversity of users and their applications. But they have one particular architectural characteristic in common: they should all be High Level Protocols (HLPs), independent of machine characteristics and the detailed mechanics of data transmission.

All the lower layers of protocol exist for the sole purpose of providing the support necessary to achieve these HLP characteristics in an open-interconnection environment.

6.2 Presentation layer protocol

Presentation layer protocol provides two specific functions: device independence and data independence. The aim is compatibility, by assisting HLPs and their associated software to be independent of functionally equivalent variations of device characteristics, data encoding and data format.

The device independence is provided by virtual device protocol (VDP). This defines a selected variety of standard device behaviours and associated standard protocol, onto which most real devices in common use can be mapped. This overcomes their detailed incompatibilities. Where this is not required, the device would be accessed in its native mode.

Several virtual device classes are needed to provide sufficient coverage. The most urgent being considered in ECMA are:

> interactive basic class character imaging device;
> interactive forms class character imaging device;
> bulk text device.

The basic class and forms class cover most interactive text communication devices, e.g. videos and keyboards. The bulk class covers printers, card readers, etc. Much of the protocol structure is common to the different classes.

There is a wide spectrum of requirements for HLP data independence: from simple code conversion and data compression, through to comprehensive Virtual Filestore Service (VFS) and DBMS-like provisions. The initial protocol standards will be for the more simple and well agreed functions.

6.3 Session layer protocol

Session layer protocol provides functions to support HLP dialogue on the transfer mechanism provided by the underlying transport layer. The main functions are:

> dialogue controls for conducting two-way alternative dialogue on the transport connection, plus use of its expedited flow; also controls for initialisation, reset and orderly termination of dialogue. Two-way simultaneous HLP dialogue is not supported (unnecessary complexity), but the two-way simultaneity of the underlying transport connection is needed for the various session layer control functions;
>
> delimiters which delimit HLP data units; those which delimit transfer units to enable efficient use of the transport service; and those which delimit error-control units and support their associated selective recovery and resynchronisation;
>
> segmenting and blocking to map the ‘infinitely’ variable sizes of HLP data units onto the actual transport service data units, whose size may be constrained by performance considerations, etc.;
>
> security by way of protocol to authenticate the HLP user identity, and by protocol encryption to preserve the secrecy and authenticity of HLP information content;
>
> a data unit typing service, which identifies the context in which the information content of a data unit is to be processed. This is what is sometimes called

'context switching', which is a very powerful and general tool for structuring HLPs;

parameter negotiation and selection;

transfer of the addresses which are used at session establishment to select the partners which will use the session;

flow control is not a separate function, it is provided by the dialogue structure and by use of the underlying transport service functions.

The session layer is designed to keep this set of functions modularly separate from functions which actually transfer data and are involved in telecommunications. These are in the adjacent lower layer.

6.4 Transport layer protocol

Transport layer protocol provides the standard means of data transcription between two separate systems. It uses the underlying network layer to provide the actual routing and transmission. This separateness from the network layer is a deliberate choice, with the aim of allowing the full variety of underlying network types and connections to be used, but without commitment to any particular one. This achieves flexibility, independence and resilience, and allows system performance optimisations and telecommunications cost savings by use of appropriate multiplexing and flow-control techniques, etc.

The alternative approach of integrating the transport service into the network layer, which has wide support, particularly among PTTs, would require endpoint systems to implement it in different ways specific to each network type. That would impose unnecessary implementation diversity and cost upon DP system manufacturers and users, and some sacrifice of user independence and total system functionality.

The main functions of the transport layer are:

to provide the standard transport service, with complete data transparency. Each transport connection has a two-way simultaneous normal flow, plus an expedited flow of restricted capacity to bypass transport normal flow control;

to provide this standard service efficiently and to an agreed quality, using whatever is the variety of underlying network characteristics. The higher layers of protocol need then not be aware of such details;

to provide flow control and delivery acknowledgement end-to-end between the source and destination systems, plus associated error management, connection, reset and disconnection functions;

to provide multiplexing onto network connections. Upwards multiplexing allows multiple transport connections between the same places to share the same network connection. Downwards multiplexing allows transport connections to use multiple separate or alternative network connections to increase bandwidth or resilience; either continuously, or dynamically in response to load fluctuations or network failures.

to provide a means of using whatever transport addresses identify endpoint systems independently from their telecommunications addressing; and provide run-time connection identifiers which are independent of all addressing schemes.

The ECMA transport protocol covers all these requirements. It has four classes:

Class 1:	basic class;
Class 2:	flow control class;
Class 3:	error recovery class;
Class 4:	full error recovery class.

The user of the transport service (i.e. session layer on behalf of the HLP user) specifies a quality of service suited to his needs and the transport mechanism then uses whichever protocol class is appropriate to the circumstances and the underlying network behaviour. This selectivity minimises the complexity and overheads of providing a suitable range of transport layer capability. Furthermore, transport connections using different classes can be multiplexed onto the same network connection.

The ECMA protocol uses a restricted set of network-layer services, chosen as being general to all types of network-layer connection. The purpose, once again, is to maximise the network independence of the transport protocol and its implementation in each computer, and to allow use of the widest possible choice of network types.

6.5 Network layer protocol

Network layer protocol provides two sets of functions: routing and data-transfer. First some clarification of the word 'network' is necessary. It is used here solely with the connotation of being that which provides the specific routing and data-transfer services. The mechanism is not necessarily a mesh of connections, nor a 'Public Data Network' (as in, say, PSS X25). The mechanism might equally be some set of dedicated star connections, or use of a single-dial-up line, or an X21 connection; i.e. any practicable routing mechanism.

The routing function makes a connection (or equivalently sends a datagram) to a destination system, which is specified by means of a network address of whatever kind is appropriate to the network. This routing includes intermediate gateway or relay functions if there are subnetworks in series between the endpoint systems. The connection is required to be two-way simultaneous (or functionally equivalent). Preferably there should also be a suitable expedited flow which bypasses network congestion. If not, the transport layer-of-protocol will construct its expedited flow on the network normal flow. There is also a network-layer disconnect function. The network connection is required to have complete data transparency.

The data-transfer function transfers data out of the source system into the network connection, and thereby to the destination system. No delivery acknowledgement is required, nor any end-to-end flow control. The responsibility of any network-layer acknowledgement or flow control is essentially to minimise congestion of the telecommunications medium and to control use of its interface. The delivery reliability requirements are simply that data in any flow is either delivered in sequence, or that it is dependably destroyed if not delivered within some known, time-out period. Given either of these assumptions, the transport layer above makes appropriate provisions. A connection reset (or 'purge') function is also required.

6.6 *Data-link layer protocol*

Data-link layer protocol provides a digital data link with an acceptable error rate, and characteristics which are independent of the actual physical medium and its analogue signalling. A typical example is HDLC.

6.7 *Physical layer protocol*

Physical layer protocol is concerned with provision of the electrical and mechanical characteristics of the accessible transmission medium. A typical example is V24.

7 Summary

We have argued the case that Open System Interconnection standardisation is of urgent and vital importance to all who are concerned in the provision and use of data communications, and that it is potentially highly beneficial. Despite some hazards, the prognosis is favourable for successful OSI standardisation in the not too distant future.

There are clear indications that computer systems manufacturers recognise a need to align their future business strategies and products with forthcoming Open System Interconnection standards. ICL certainly does. The relevant technical characteristics of the ISO Reference Model have been described in outline.

References

1 HOULDSWORTH, J.: 'Standards for open network operation', *ICL Tech. J.*, 1978, 1, 50-65.

2 ANON: 'Reference Model of Open System Interconnection (Version 4 as of June 1979)', ISO/TC97/SC16/N227. Copies available from National Standards Bodies.