Technical Journal

Volume 3 Issue 1

May 1982

ICL Technical Journal

Contents

Volume 3 Issue 1

Information Technology Year 1982
Dennis J. Blackwell 3

Software of the ICL System 25
M.A. Cave 5

Security in a large general-purpose operating system: ICL's approach in VME/2900
T.A. Parker 29

Systems evolution dynamics of VME/B
B.A. Kitchenham 43

Software aspects of the Exeter Community Health Services Computer Project
The Exeter Project Team edited by D.J. Clarke and J. Sparrow 58

Associative data management system
L.E. Crockford 82

Evaluating manufacturing testing strategies
M. Small and D. Murray 97

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

Editorial Board
J. Howlett (Editor)
D.W.Davies
(National Physical Laboratory)
D.P.Jenkins
(Royal Signals & Radar Establishment)
C.H.Devonald
D.W. Kilby
K.H. Macdonald
B.M. Murphy
J.M. Pinkerton
E.C.P. Portman

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

1982 subscription rates: annual subscription £11.50 UK, £13.50 ($32.00) overseas, airmail supplement £5.00, single copy price £6.75. Cheques should be made out to 'Peter Peregrinus Ltd.', and sent to Peter Peregrinus Ltd., Station House, Nightingale Road, Hitchin, Herts. SG5 1SA, England, Telephone: Hitchin 53331 (s.t.d. 0462 53331).

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

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

This publication is copyright under the Berne Convention and the International Copyright Convention. All rights reserved. Apart from any copying under the UK Copyright Act 1956, part 1, section 7, whereby a single copy of an article may be supplied, under certain conditions, for the purposes of research or private study, by a library of a class prescribed by the UK Board of Trade Regulations (Statutory Instruments 1957, No. 868), no part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means without the prior permission of the copyright owners. Permission is however, not required to copy abstracts of papers or articles on condition that a full reference to the source is shown. Multiple copying of the contents of the publication without permission is always illegal.

© 1982 International Computers Ltd.

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

Information Technology Year 1982

Dennis J. Blackwell

ICL Director for Quality Assurance, Bracknell, Berkshire, UK

'Information Technology (IT for short) is the fastest developing area of industrial and business activity in the western world. Its markets are huge, its applications multitudinous, and its potential for increasing efficiency immense. Without doubt it will be the engine for economic growth for at least the rest of the century. Britain's economic prosperity depends on the success with which we manufacture its products and provide and exploit its services. This is the message that must be got over to everyone in this country – the general public, school children, as well as industry, trade and commerce.'

These are the words with which Kenneth Baker, Minister for Information Technology, launched 1982 as Information Technology Year.

Information Technology is not easy to define; fortunately there is no need for me to attempt a definition for the readers of this Journal, who will all be conscious of the enormous range of ideas, techniques, technologies and applications which the name covers. In ICL I have the task of directing the company's program for IT82 and I want here to offer some thoughts stimulated by the occasion and this responsibility.

It has become almost a cliché to say that the electronic digital computer is one of the great inventions of all time. No-one who knows anything about it is in any doubt about this. But there is still a great weight of negative and hostile attitudes: one hears with great regularity that computers destroy jobs, invade privacy and open up new pathways to fraud, and stories still circulate about gas bills for £1000000 and demands for payment of £0.00 and similar nonsenses – all accompanied by 'it's because the computer doesn't know any better'. I very much hope that IT82 will do a great deal towards combatting these attitudes and replacing them by positive ones. The computer is the vehicle which carries information technology into use in all our lives. It puts possibilities for enormous power into our hands and of course this, like any other power, can be used to serve bad ends. But it can be, and of course already is being, used to very many very good ends, many

of which are a long way from what one usually thinks of as technological – leisure activities, the arts, in fact the general enrichment of life. What is needed is more education in the nature and uses of this new technology, spread right through the population because it is for everyone, not for just a select few, so that we get a better understanding of the possibilities which it offers and are better able to control its powers and to exploit these for socially good ends.

One of the most striking and encouraging features of the IT scene is the strong appeal it has for the young. There is clearly something which captures their imaginations and releases latent talents. Any number of examples could be quoted of young – often very young, under 10 years old – children showing skills which no-one had any idea they possessed. It has often been said that there are only three fields in which infant prodigies appear – music, mathematics, chess: is IT a fourth?

As a company whose business is entirely in the field of information technology, ICL welcomes the initiative of IT82. My personal aim and responsibility are to ensure that our programme for the year will support and enhance the national programme and will enable ICL, its staff and the users of its products to contribute to the social and educational aims of the year.

Software for the ICL System 25

M.A.Cave
ICL Distributed Systems Development Division, Bracknell, Berkshire, UK

Abstract

The paper complements that by Walton[1] on the hardware and architecture of the ICL System 25 small-business machine. It describes the organisation of the facilities provided by the standard software for the machine. This software has been developed from, and is compatible with, that for the earlier System Ten, extended to handle new peripherals and to provide extra services, including transaction processing and comprehensive communications facilities.

1 Introduction

As was stated in the paper by Walton[1] on the architecture of System 25, the machine is the successor to the ICL System Ten, originally a Singer product, of which approaching 10,000 are in use world-wide, mainly for small-scale business data-processing. It is a feature of both System 25 and System Ten that there is no operating system or executive in the usual sense of these terms, the functions normally associated with such software being provided here by the basic hardware architecture.

A comprehensive set of software packages was provided for System Ten, collectively called DMF (Data Management Facilities)-II and including an input/output package LIOCS (Logical Input/Output Control Software), a disc maintenance and file handling system CSM (Conversational System Manager), a program loader with job control facilities, an assembler, an RPG (Report Program Generator) compiler, a SORT package and a range of program development aids such as editors and testers. A corresponding package DMF-III has been developed for System-25, backwards compatible with DMF-II, and in addition new software as follows:

(i) Disc management software to support new discs with 512 byte sectors, and to offer new access methods giving improved performance and larger disc capacities.
(ii) New communications software aimed at integration with ICL's IPA and IBM's SNA.
(iii) A TP (transaction processing) system, offering standard video and printing facilities.
(iv) A COBOL language system
(v) Additional utilities such as job-logging and screen processing.

2 Overview: main components of System 25 software

This Section gives a brief statement of the main components of the software and shows how they fit together, concentrating on the run-time aspects. Later sections give more details and deal briefly with the development tools.

2.1 Main-store architecture of System 25

The paper by Walton[1] describes this in some detail. A very simple view, which will help in the understanding of the present paper, is that there is a variable number, up to 20, of program partitions, each of which has access to a single Common store partition and has direct access to the disc I/O channels.

2.2 Principal software components

These are broadly grouped as follows:

LIOCS Logical I/O Control Software.
The I/O package, offering access to a wide range of magnetic media such as floppy diskettes, fixed and exchangeable discs, magnetic tape. The facilities offered permit the opening and closing of files and the reading, writing and up-dating of records serially or by number or key. Most of the code is resident in Common and is therefore directly accessible from a program partition.

CSM Conversational System Manager.
This creates and maintains the environment in which LIOCS and the rest of the system run. It contains the following elements:

Filestore maintenance facilities
Conversational loader
Job-stream file support

It can normally be accessed via any video workstation and uses a private interface to the operator. Most of the code is non-resident and is run as an application program.

IAS Interactive Applications Support.
Provides support to applications programs written in a high-level language. It provides the control software and run-time library for the TP system and the corresponding support for batch programs.

CAM Communications Access Manager.
This provides the wide range of communications facilities which are available on System 25, allowing inter-working with both ICL and IBM equipment. The ICL facilities permit connections using ICLC-03 and X-25 protocols, those for IBM the use of BISYNC (3270) and SNA (SDLC) protocols.

Fig. 1 shows how these fit together in the total system.

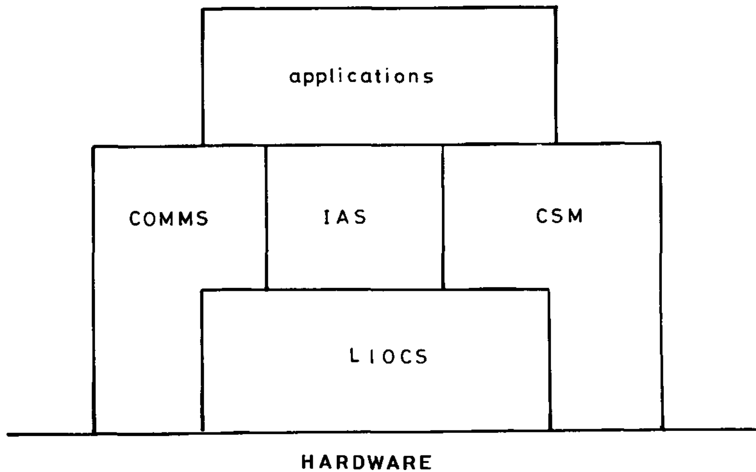


Fig. 1 Software interfaces

2.3 *Further components*

Screen and printer drivers. These permit any combination of standard printers and videos to be driven, up to 10 on a single partition. There are four sets of facilities:

printer handling
basic video handling
screen processing
conversational loader

The printer handling software supports the standard printer functions required for the spooler; it contains only the basic device drivers, a spooler print application being expected to interface between the spool file and this handler.

Basic video handling supports the display of pre-formatted screens (templates) and further text, control over the keyboard and the input of data, basic validation of input data, etc.

Standard applications. A wide range is offered, including:

development aids
'batch' utilities, e.g. SORT
communications packages, e.g. video emulation
Industry and Retail packages

These normally run in and use a single partition.

3 General housekeeping package DMF-III

This provides the facilities on System 25 for structuring and accessing data on magnetic media and for loading and running programs. It consists of three separate elements:

a set of data structures on disc
the run-time access package LIOCS
the general-purpose utility CSM

This Section of the paper is concerned with the details of the data structures and of the workings of the LIOCS package.

3.1 Data structures on disc

Each physical disc is generally defined to be one *volume,* which consists of a contiguous set of physical *sectors* each of 512 bytes. The sectors are numbered and can be arranged by the user (or management) so as to give the best performance when accessed serially.

Each volume is split into one or more *pools,* each of which consists of a set of contiguous sectors, and contains an index of all the pools into which it is split. One volume, designated PRIMARY SYSRES, normally contains an index of all pools on all volumes and additional bootstrap software for initial program loading, IPL. Not all of these volumes need to be online at once; CSM permits a new volume to be introduced and all the pools on it to be linked to the PRIMARY SYSRES. A partition may, however, be assigned via CSM to *any* volume, from which it may access only the pools on that volume or linked to it. Each pool contains one or more *files,* each consisting of a set of *records,* to which the user can gain access via LIOCS.

There are various kinds of pool and file, in which the way records are held and physically accessed differ. This gives the user significant freedom in choosing the type of pool and file for optimal performance. Records may be blocked, such that a number of records are transferred at any one time, thereby reducing the total number of disc accesses over a series of LIOCS calls.

DMF supports the following types of pool:

Relative pools are the main kind to be used by application programs for holding data records, and consist of a set of blocked records which can be accessed by record or block number. They always contain only one file.

Linked Sequential pools are normally used only for holding source programs where records have to be frequently inserted and deleted. They contain one or more files, each of which consists of a chain of records which can (normally) only be accessed serially. LIOCS supports only unblocked records, although the user is able to pack records outside the control of LIOCS.

Program pools contain object program files in such a manner that they can be loaded and run efficiently.

Superimposed upon these pool types is an indexing facility. This permits a separate index to be created outside the data to be indexed. This facility is used by the new loader in accessing program files by name, as well as offering indexing of user files in relative and linked sequential pools. Each index is held in a pool whose basic organisation is that of the relative pool.

3.1.1 Relative pools. Each relative pool can contain only one file which can be of any of the following types:

- (i) Relative – fixed length records, accessed sequentially, or randomly via record number
- (ii) Mapped Relative – as relative, but also permits insertion of records
- (iii) Direct – fixed length records, accessed randomly by key
- (iv) Variable Sequential – fixed or variable length records, accessed sequentially

Records are blocked as follows:

```
blocks:   -------b--------,- ----b+1----------,-
sectors:  ----p---,---p+1---,-- p+2---,----p+3-----,-
records:  n , n+1 , n+2 , X , n+3 , n+4 , n+5 , X ,     (X denotes
                                                           end-of-
                                                        block marker)
```

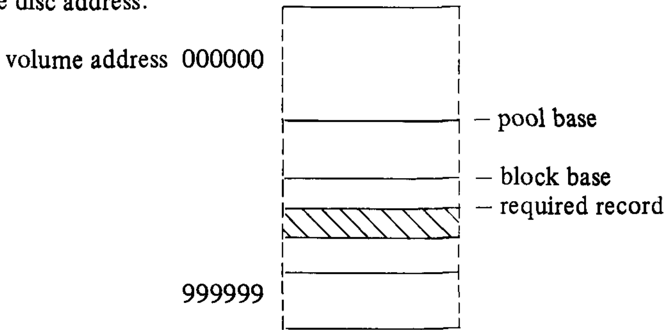
The block is always of a fixed size and is the basic unit of transfer for the file. One block contains an integral number of sectors, and there is therefore sometimes an unused area at the end of each block. At run time it is possible when accessing relative files to request the basic unit of transfer for the file to be a multiple of the block size – this is referred to as the bucket size. The use of buckets is transparent to the data structures, other than that the size of the pool must be a multiple of both block and bucket size.

Files which have only one record per block are referred to as being *unblocked.*

Relative Files. Records are always of fixed size, and can be accessed via record number or sequentially by ascending record numbers.

The record number can be converted to a block number by knowing the number of records to a block. This can be converted to a sector address by knowing the

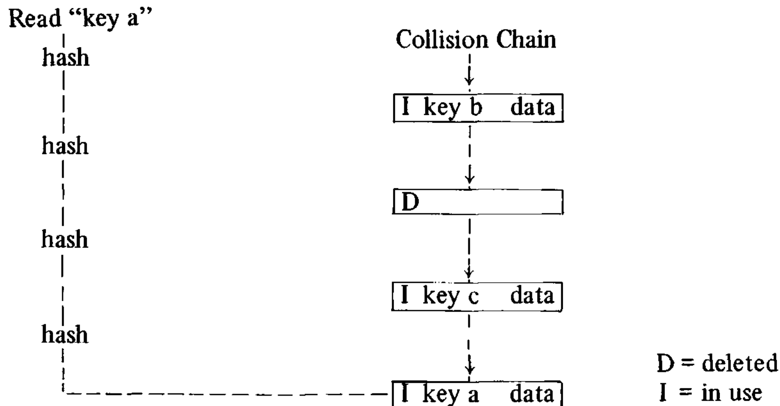
number of sectors to a block, and this, added to the base address of the pool, gives the disc address:



Optionally, records can be flagged as 'available', 'in use', or 'deleted' – permitting functions equivalent to 'insert' and 'delete' to be provided. When records have status flags the file must be initialised when it is created, to mark all records as 'available'.

Direct Files: Hashing. The structure of direct files is very similar to that of relative files, with fixed length records and blocking. Records must have status flags. The main difference is that the records are accessed via a key which is converted to a record or block number, using a key transformation (hashing) algorithm which can be either system-supplied or user-written.

Hashing allows a non-unique number to be generated from a character string (i.e. the number of possible keys is normally more than the number of records in the pool). Where two records with different keys result in the same hashed number a 'collision' is said to have occurred. The larger the ratio of the number of possible keys to the number of records or blocks in the file the greater the chance of a collision occurring. When a collision occurs a further hash number is derived to yield an alternative record or block number, which could result in another collision and further hashing. This sequence of numbers is referred to as a collision chain. The performance of access to a direct file is therefore determined by how long the collision chain gets. The following is an example of a collision chain in an unblocked file:

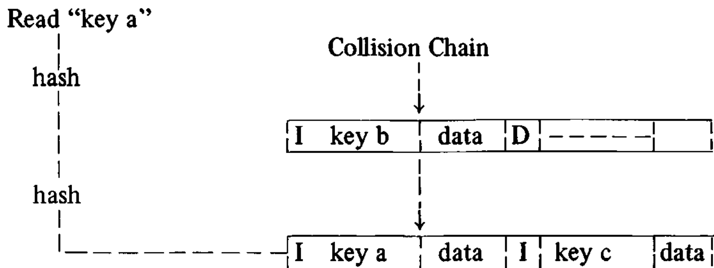


As a guide, when an unblocked file is 80% full two accesses are needed on average to find the required record, and it is recommended that such files be 20-30% larger than necessary for all the records which are likely to be held.

Records are flagged as 'in use', 'available' or 'deleted' and such files therefore have to be initialised.

The end of the chain is denoted by an 'available' record, and 'deleted' records are ignored when searching for a key. If a record is to be written and no such key already exists then the first 'deleted' record, if any, will be overwritten, otherwise the 'available' record will be overwritten. If the key already exists then the previous version will normally be overwritten. Optionally, the first 'deleted' record can be overwritten, if one occurs before the extant version of the record. This will result in multiple copies of records to be present in the file, and possible confusion when a later version of the record is deleted.

The above descriptions referred to unblocked files (one record per block). DMF also permits blocked records to be handled, where the hashing process yields a block number. All records in the block are effectively in the same collision chain, but are available in the one disc transfer. The following example is very similar to the previous one, but blocked:



The number of records in the block is called the *blocking factor;* in this case, for a blocking factor of two, the file can be up to 93% full and still require only two accesses on average to find the desired record. As the blocking factor increases, so the number of records which can be found in two accesses increases.

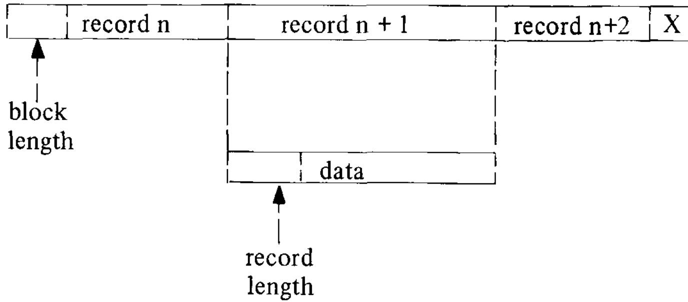
Note that the collision chain described above is unique to each key. That is, the collisions occur where two chains *cross.* This is achieved by using the original key for rehashing.

The performance of access to direct files is also obviously affected by 'deleted' records which extend collision chains, and the user will frequently have to tidy such files. Failure to do so may cause a drastic reduction in performance. Consequently direct files are not appropriate where records have to be regularly inserted and deleted.

Variable Sequential Files. Records in variable sequential files can be of variable length and are always blocked. Variable sequential files are normally accessed

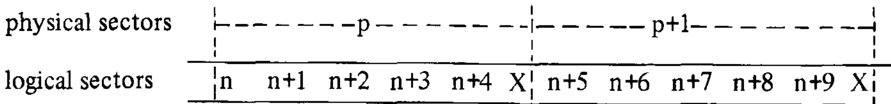
serially, although the user is given the opportunity to record the block number for later use.

The format of a block in a variable sequential file is as follows:

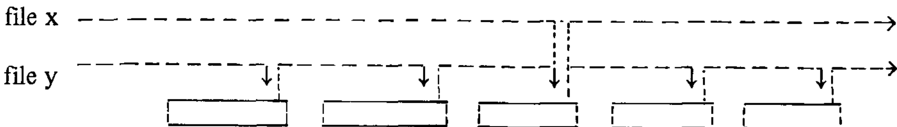


The block and record lengths are four digits and include their own size.

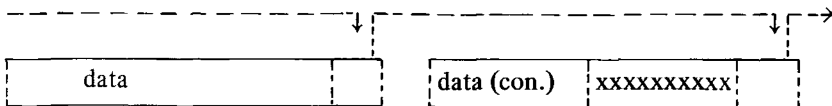
3.1.2 Linked Sequential Pools. A linked sequential pool can consist of one or more linked sequential files, all of which must have the same record length. The pool consists of a set of continguous 512 byte sectors, each of which consists of five 100 byte logical sectors:



The 12 bytes at the end of each physical sector are unused. Each linked sequential file consists of a chain of logical sectors:



Each logical sector contains a 6 byte chain pointer and contains either a whole record, or part of one if the record is more than 94 bytes long. In the latter case the record will be split:



The chain pointer can be in one of two forms:

unpacked – consisting of a 5 digit unpacked logical sector number, giving up to 10 Mb

packed — consisting of a 7 digit packed logical sector number, giving up to 1000 Mb

Insertion and deletion of records is achieved by manipulating the chain pointers. A chain of unused logical sectors is held at the pool level.

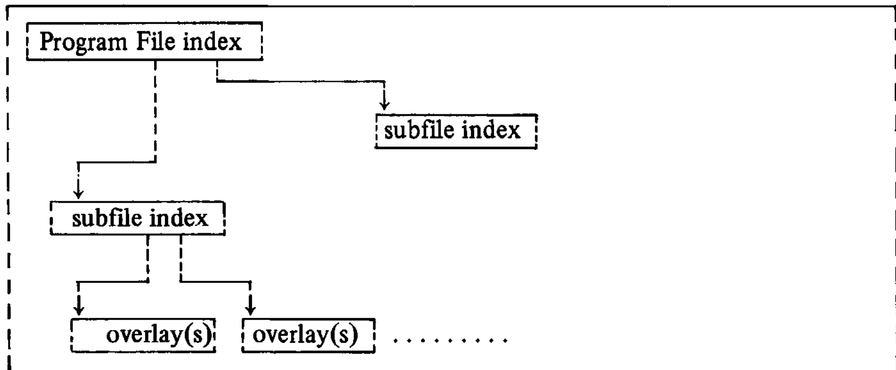
Doubly linked sequential files can be created where both forward and backward pointers exist. This type of file is used by the text editor. LIOCS does not offer any facilities for writing such records.

3.1.3 Program pools. Each program pool may consist of one or more program files. The structure of the pool and the files is designed to optimise performance and exploit the string-read facilities of System 25.

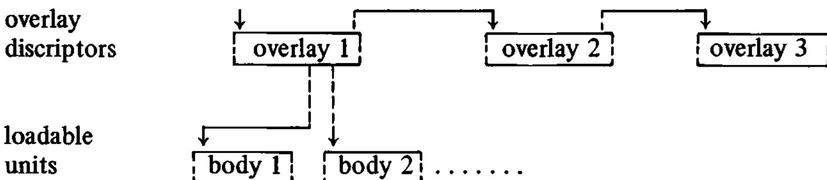
Each program file consists of a number of subfiles, each of which can be referred to by name. Each subfile contains one or more overlays which can be individually loaded.

The pool is accessed by the loader in a number of different ways; and roughly corresponds to a relative file with 512 byte unblocked records, i.e. one per sector.

The format of the program pool can be shown as follows:



A file is initially located by searching the index for the required name. Once located this gives access to an index of subfiles which can again be located by name. This results in a pointer to the overlay(s) in that subfile, which can be serially accessed:



The overlay descriptions may take either a single record or a number of consecutive records and contain pointers for up to 71 data/code bodies, each up to 9999 bytes long, as follows:

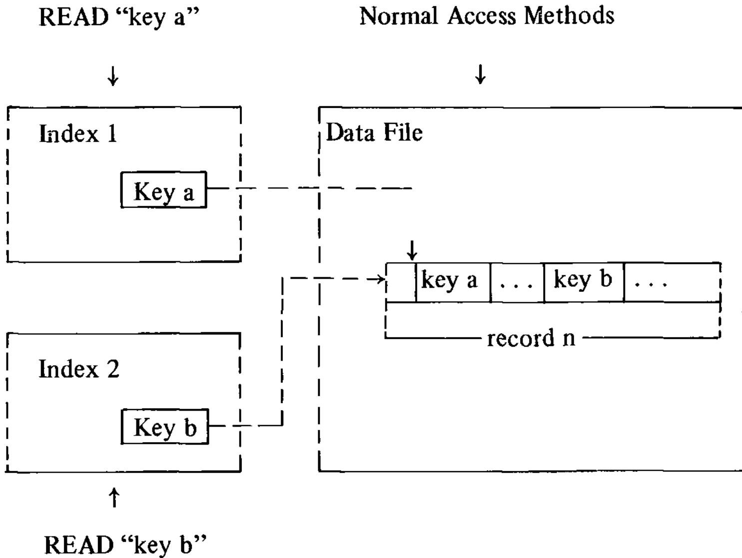
Properties list	Body list(s)	Relocation List(s)	Comment(s)
Description of the overlay	Pointer to the bodies, includes the size and address of each, relative to Common or Partition origin	Those location(s) which have to be modified, if the bodies are relocated involves double relocation as the addresses are relative to origin	General comments – ignored at load time.

Simple overlays will need one descriptor record and one body – such overlays will therefore require only two disc accesses to load them.

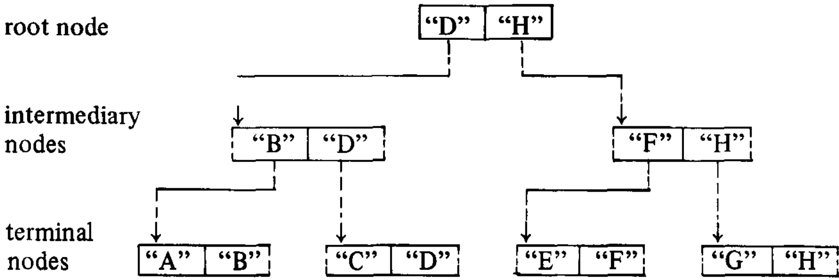
The user is given access to record numbers of each overlay descriptor and therefore is able to bypass the logic to locate by file or subfile name, avoiding repetitive accesses to the indexes when an overlay has already been loaded during the run.

3.1.4 Indexing. This Section describes the structures built upon the basic file types for the index access method, which permits records in one file to be accessed via a key search through a separate index file. The index file is always in a relative pool, as a relative file whose structure is described below. The organisation of the index file permits both random access (by key) or serial access (ascending key order).

The index access method also permits the one file, which may be a relative, direct or linked sequential file, to be accessed by more than one index. Such files can be directly accessed via their normal access method(s):

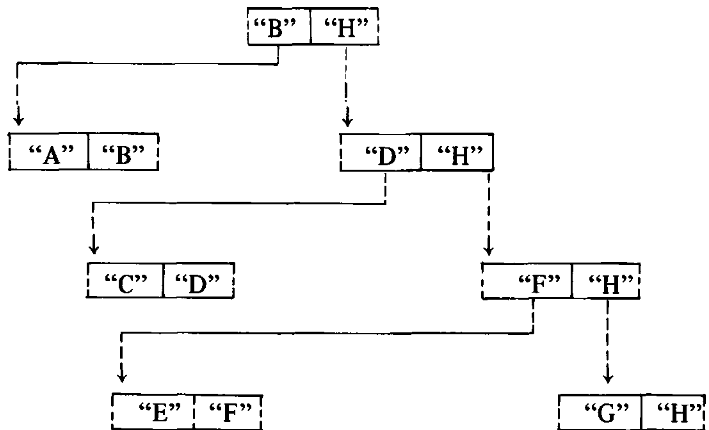


Structure of the index. The structure is based upon that described by Knuth.[2] It is best described by means of a simple example:



The required entries for each of the keys "A" to "H" are designated as terminal nodes. The tree is structured such that the required key can be compared at each level of the structure, starting at the root node, looking for the branch on which the key should exist. In the above example there are two keys/branches at each level of the tree. A branch is followed if the required key is less than or the same as the key associated with the branch. Three accesses are needed to access a terminal node in the above example.

The above tree is balanced, with the same number of nodes between each terminal node and the root. Consider then the example where the tree is *unbalanced:*



In this case the average number of disc access goes up from 3 to 3.25, and is obviously undesirable.

DMF uses a tree structure similar to the above but with more than two keys/branches per node. The number is determined by the size of the key, given that each node uses a single 512 byte record/sector. Each node may have spare entries to permit keys to be inserted and deleted. This is useful when the tree has to be re-organised to keep it balanced (e.g. the above example of an unbalanced tree is prevented from occurring by the algorithms for creating, inserting and deleting). DMF will normally ensure that each node is at least half full. When an entry is

deleted and the node becomes less than half full an attempt will be made to redistribute the entries amongst 'adjacent' nodes at the same level in the tree and remove that node. Similarly when a node becomes full a new node has to be introduced and entries moved to it from the full node.

DMF requires only a small overhead of each key. For example, about 24 entries can be held in a node for 15 byte keys, and about 42 for 6 byte keys.

The number of disc accesses n for a full balanced tree with m entries per node and a total of k keys satisfies the relation

$$m^{n-1} < k < m^n$$

For normal usage the nodes will be 75% full on average. The Table below indicates for three key sizes, for 75% and 100% full cases the number of keys which can be held and accessed within 3 and 4 disc transfers, excluding the final transfer on the data file if present.

Key Size	75% full, 3 transfers	100% full, 3 transfers	75% full, 4 transfers	100% full, 4 transfers
6	27,800	68,800	864,000	2·9 million
10	11,600	29.800	267,000	894,000
15	5,500	13.200	99,000	318,000

3.2 LIOCS access methods

LIOCS consists of code permanently resident in Common, some fixed areas in low Partition and a range of macros embedded in the user programs. The code in Common is therefore shared and not self-modifying. The macros for opening and closing files, and accessing them, all result in branches into common. In the case of programs written in COBOL these details are transparent to the user, being contained within the COBOL run time library. In the case of TP applications still more is done for the user in the opening and closing of files. COBOL does not offer access to all of the facilities described below, e.g. it cannot take advantage of some shortcuts possible when interfacing directly to LIOCS.

3.2.1 Access to files. All access to files is by means of a file control block (FCB). This is used in the process of opening a file, and thereafter in accessing it – the format of the FCB changes in the OPEN process to contain the physical location of the file, etc. The FCB is held in the user program's workspace, and the application is able, via macros, to access and amend fields in the FCB. The macros for opening and closing the file and accessing it all require an FCB. The FCB holds pointers to two further areas to be supplied by the user: the block and record work areas. The block area is used only when the file to be accessed is blocked. The record work area contains the record to be transferred.

Filenames can be introduced into the FCB prior to OPENing the file either via the compilation process, or via a *control file.* This is a linked sequential file of a specific format.

The FCB itself is not of fixed format and must be tailored to the specific access method(s) to be used on the file. Each file has an associated access method. The user may specify that one of three general access methods may be used, which will work with more than one type of file:

Basic Sequential – Relative, Direct, Variable Sequential, Linked Sequential
for non-shared serial access to files

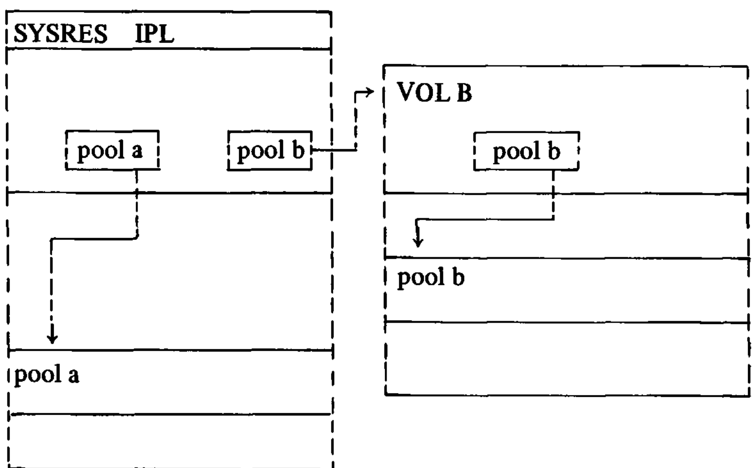
Mixed Sequential – Relative, Direct, Variable Sequential, Linked Sequential
for shared access with updating permitted

Keyed Access – Direct, Indexed Files

These are only of relevance to LIOCS itself. Appendix 1 gives a list of the principal macros available and the access methods for which they are valid.

3.2.2 Red tape and flawed discs. Each volume, pool and file on disc has a red tape area associated with it, which is used by LIOCS in locating and accessing files and handling flawed areas of disc (*blots*).

Volume Red Tape. The start of each volume has a reserved area for holding information about the volume, including an index of all pools on that volume and their locations. If the volume is designated for SYSRES then it must also contain an index of all pools on all volumes. A fixed area of this volume also contains the IPL software which is required at cold startup and can be loaded via an appropriate workstation:



Pool red tape. The nature of pool red tape depends upon the type of pool:

Relative pools. The first sector contains a file label for the one and only file. It is followed by a set of alternate sectors which can be used when sectors in the data portion of the pool are found to be defective (normally when the disc is initialised). Following the alternate sectors is a blot table listing each sector in the data area which is defective and its alternate:

file label
alternate sectors
blot table
data sectors

When a sector is accessed via LIOCS a check is made to see if the sector is flawed. If so the alternate sector is used.

Note that with direct files the hashing algorithms may be used to generate numbers which do not encompass the whole file/pool. The rest of the file is available for relative access, e.g. for overflow records.

Linked Sequential Pools. The first sector of the pool contains the pool label, which includes a chain of free sectors and a pointer to the rest of the file labels. Flawed sectors are automatically excluded from the chain of free sectors.

4 Basic communications facilities

This Section necessarily mentions two particular items of System 25 hardware, the C-Coupler and the MTIOC interface. These are described in Walton's paper.[1] Briefly, the C-Coupler provides for the connection of communication lines to the machine and MTIOC (Multi-Terminal I/O Channel) is the main slow peripheral interface, allowing connection of up to 10 slow peripherals onto a single twisted pair.

Three types of communications facilities are offered by System 25:

PCA – Programmable Communications Adaptor, special firmware for the C-Coupler which converts the protocols of remote devices to those of their local equivalents. This permits remote devices to emulate those connected via MTIOC, primarily Model 84 videos to emulate Model 85C. The protocol supported on the remote link is an extension of that used between Model 81 videos and System Ten. All this is transparent to the user and to the system software which drives the device, but some initialisation may be required.

SCA – Synchronous Communications Adaptor, a combination of special firmware for the C-Coupler and code residing in a private partition, permitting System 25 programs to use private BISYNC (binary synchronous) protocols for transmission of data to and from "alien" mainframes or terminals. This is achieved via a low-level interface in System 25 PLI and can be used in two ways –

(i) System Ten compatible usage, for programs which were written for the 6-bit interface of System Ten SCA
(ii) Full 8-bit usage, in essence very similar to (i) but avoiding the intricacies of generating 8-bit control codes via a 6-bit PLI interface, and therefore somewhat simpler

CAM – Communications Access Manager, a combination of special firmware for the C-Coupler and associated partition-resident software, offering higher-level facilities for transferring data between System 25 and a mainframe. The protocol used for any mainframe link can be split into two major layers –

(i) Basic transport services, responsible for getting messages safely across the link. This is therefore concerned with the low-level protocols used to pass messages across a line, with the format of those messages (the framing standards) and the addressing mechanisms.

(ii) Access level, responsible for the content of the messages and for implementing end-to-end protocols. For instance, a video access level determines the actual character set to be used and the meaning of any embedded control characters. Two different access levels may use the same control characters but attach different meanings to them.

CAM offers the basic transport service, while the application program is responsible for most aspects of access level. The video access levels for IBM and ICL versions of CAM are different, in terms of both character set and control characters. Because of such differences it is not generally possible to develop applications for one version of CAM and then to move to another version without significant work.

The facilities permit one or more applications in different positions to share the use of a single mainframe connection. The interface between application(s) and CAM is via messages passed through Common, using LIOCS-type calls. This interface offers a range of primitive functions for reading and writing the data on a logical stream, and some flow control mechanisms. While there are several very different versions of CAM they all support the same primitive functions.

The following Sections describe the various versions of CAM in terms of what the user will see of the CAM subsystem.

4.1 CAM/C03

CAM/C03 supports communications to ICL mainframes via ICLC-03 line protocols and is the basis for offering various facilities under the general title of IPA – Information Processing Architecture. ICLC-03 is an *asymmetric* protocol; that is, the dialogue is between a *primary* and one or more *secondaries*. The mainframe will always act as primary. The secondary is addressed as a *group*, with a number of associated *subsidiaries*. A group/subsidiary corresponds to a logical stream supporting an end-to-end access level dialogue. Normally one secondary acts as a single group.

4.1.1 IPA. An account of this important strategic concept is given in a paper by Kemp and Reynolds[3] in a previous issue of this Journal. The System 25 communications software provides four facilities:

RSA – Remote Service Access, permitting videos connected to one system (the Gateway) to access not only the service(s) in that system but also remote (Host) services.

DTS – Distributed TP Service, permitting TP applications in one service to initiate enquiries in another and to receive replies.

GFT – General File Transfer, permitting the transfer of files from one filestore to another.

ADI – Application Data Interchange, permitting applications in different machines to communicate.

Both RSA and DTS are associated with data originating from or destined for a terminal. For use across ICLC-03 lines they are based on the existing video access level.

Both GFT and ADI assume the existence of a fully transparent data path between the end points, called DIAL (Device Independent Access Level). X25 offers this automatically but ICLC-03 does not; a mechanism therefore is built above ICLC-03 to achieve the appearance of transparency.

4.2 CAM 3270

CAM 3270 provides connection to IBM machines and to others which support the IBM 3270 Binary Synchronous protocol. It is a combination of firmware for the communications coupler and software within System 25. It is not intended to provide 3270 screen emulation, though a suitable application can provide simple screen

management; it is aimed rather at allowing System 25 to function as a powerful distributed processor within an IBM network.

CAM 3270 has been used both to provide interactive filestore access to distributed System 25s from a central IBM machine, and to provide batch transfer of files at end-of-day for processing on the central machine.

4.3 CAM-SNA and the ISO 7-Layer Model

The CAM-SNA product will allow System 25 to become part of an IBM SNA network and fulfil its potential as a distributed processor. To achieve this CAM-SNA makes the System 25 look like an IBM 3274 or 3276 cluster controller with a number of terminals (Logical Units, LUs) attached. A cluster controller is an SNA Physical Unit (PU) Type 2, and up to 32 LUs can be supported by each PU. A maximum of four lines with one PU per line can be supported and each line can run at up to 9600 b.p.s.

CAM-SNA consists of software in the 8085-based communications coupler and software in the System 25. The communications coupler handles the SDLC (Synchronous Data Link Control) line discipline and the modem/line connection, equivalent to Layers 1 and 2 of the ISO 7-Layer model. The System 25 software provides the Path Control, Transmission Control and Data Flow Control layers of SNA, equivalent to ISO layers 3, 4 and 5. A macro interface at the Data Flow Control level allows programs to be written to provide the particular end-user facilities required, equivalent to providing ISO Layers 6 and 7. An operator facility is provided to configure, load and control an SNA system.

5 The TP system

The TP system on System 25 offers an environment for running applications normally written in a high-level language largely divorced from the physical attributes of the system in which it is run. It has been implemented using a similar style of program structure, interfaces and facilities as for larger machines. The major benefits to ICL and to users are the evolution of a house style such that applications developed for one regime can be moved to another fairly easily, and the removal of the need for intimate hardware knowledge from the applications.

TP control takes responsibility for routing messages between devices (videos or specialised terminals) and applications, permitting applications to be written as though driving a single device. The interface software for driving terminals removes all device characteristics. The TP system is designed to support a range of 'intelligent' and 'dumb' terminals, with appropriate development aids offered via the Service Definition Facilities.

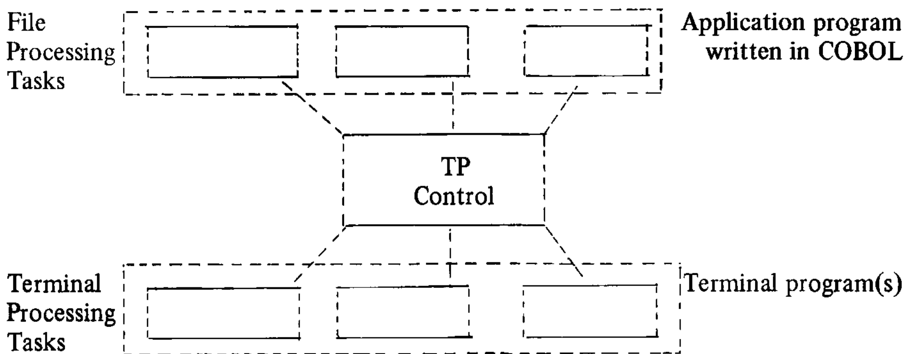
TP control also offers comprehensive facilities via a device-independent interface similar to that used for interactive devices. Via Service Definition, the TP service will run in or 'own' a set of partitions. Supervisor facilities are offered to dynamically affect the number of partitions being used.

The following Sections describe the main aspects of the system in terms of applications written in a high-level language.

5.1 Basic TP system

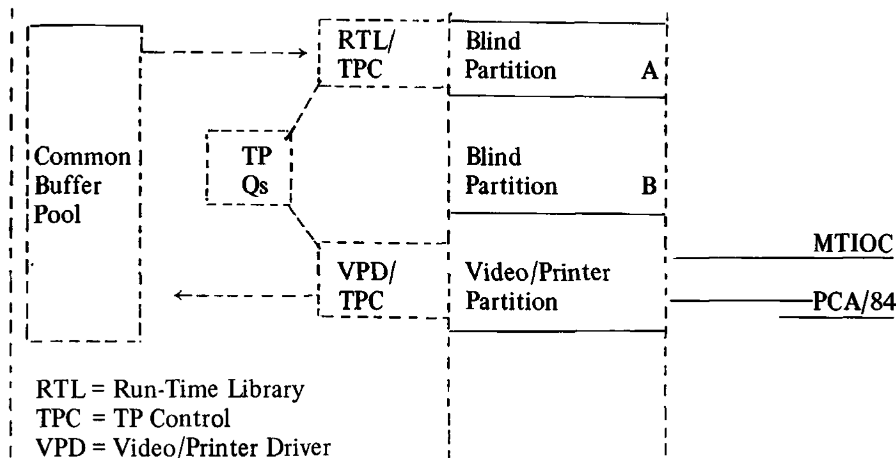
System 25 TP is designed to separate the processing of data in the application from the detail of device driving, thereby simplifying and speeding up the process of the program development.

An application is split into two separate sets of elements:



The application program (in COBOL) is split into one or more components, each of which can perform one or more different types of task. Each terminal has an associated task, which can send messages via TP control to an associated file processing task. Such messages consist of a header and data, and are specified so as to obviate the need for the application to use specific device control characters – any such controls are parameterised in the header. The file processing application is able to reply to the terminal, as well as pass messages to other applications or to a remote TP service, by means of the DISPLAY verb in the COBOL program. An application component is only able to process a single message at a time, and therefore will normally run for a discrete period of time, terminated by means of the EXIT PROGRAM verb in the COBOL program.
An operator is normally given access to several applications via a menu selection facility offered by the basic TP system. Each will have an associated set of files, which are opened at 'start of day'. Most applications will require some extra files and context information (*partial results*) to be available for each operator using the application. These are opened/initialised when the operator first gains access to the application via the menu-select facility. Whenever a component of that application is run because a message from the terminal has been received, both sets of files are automatically 'opened'. Partial results are then available via the RESTORE verb. At the end of processing the message, partial results, e.g. running totals, can be stored using the SAVE verb.

The way the system is implemented can be shown as follows, where it can be seen that terminal processing tasks run in the driver partition and application programs run in other (blind) partitions:



The video/printer driver passes messages to and from buffers held in Common. Applications are able to run in any of a number of blind partitions allocated to TP working. When a partition is idle a check is made to test if any messages are in the queue for an application component. If there are, then the first overlay of the component is loaded and entered. Any messages generated (via DISPLAY) are queued in Common in a similar fashion. The application will run to completion (the EXIT PROGRAM verb), at which point the partition will again become idle and look for the next message to process. There is therefore an element of TP software 'residing' in each blind partition and device driver; in fact, such code may be in Common and shared.

The TP software associated with loading and running an application component retrieves the file descriptions for the transaction, ready for COBOL file I/O verbs to access the associated files defined with the application.

5.2 COBOL run-time library (TP)

The design and implementation of System 25 COBOL requires a package to be available which consists of a number of procedures called from the code generated by the compiler. Some of these routines are associated with more complex language constructs and are common with those for batch COBOL. Other routines exist as system interfaces to TP and relate to particular verbs:

(i) DISPLAY – calls specific TP software to carry out message queueing as already described.

(ii) EXIT PROGRAM – enters specific TP software to look for the next message.

(iii) SAVE/RESTORE – calls specific TP software to save or restore partial results.
(iv) File I/O – calls standard COBOL file I/O software in the run time library.

5.3 *Additional TP facilities*

5.3.1 Screen processing/templates. The screen driver software takes complete control over all aspects of device handling. The screen is split into one or more display windows. The application can direct the display of a template within one of those windows, where the template consists of protected text (for the benefit of the operator) and various fields which can be classified as:

(i) input fields (unprotected) for entry of data by the operator

(ii) reply fields (protected) for display of additional data from the application

(iii) mixed fields which are for both input and replies.

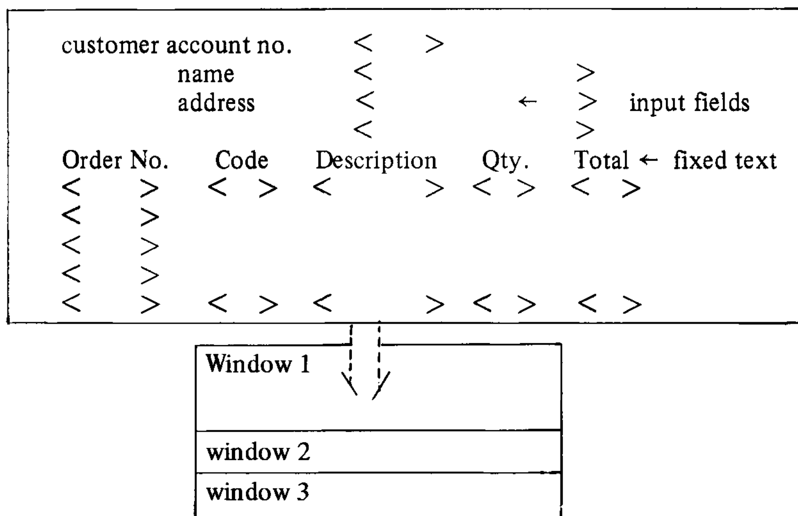


Fig. 2 Example of template and screen with three windows

The splitting of the video into windows permits separate areas to be reserved for system messages and broadcasts, and allows multiple templates to be output.

Associated with the screen template facilities are a range of supporting features for the control over the video, cursor movement and field validation.

5.3.2 Spooler/printing. The DISPLAY verb in COBOL also offers printer facilities. These permit messages to be sent to a system application which is able to 'compose'

pages of print, which are then given to the printer driver. The facilities permit the use of printer templates which save considerable programming at the COBOL level.

The facilities control the input of several documents at once and handle printer exception conditions.

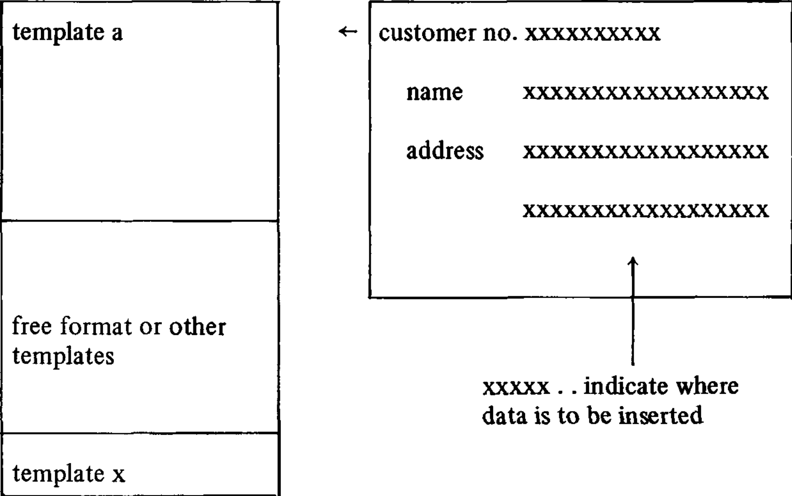


Fig. 3 Example of page or print

5.3.3 Supervisory video. An interface is offered via DISPLAY to send a message to the supervisory console, which is another video also under control of the TP system. Because of the separation of the video driving from the application it is possible to fit the interface into the video driver while permitting the supervisory video to be used also for running applications at the same time.

This video is used to control the TP service, open and close applications, provide status information etc. when not required for normal data entry.

6 Program development tools

System 25 offers a wide range of development tools for the production of applications written in Assembler or the supported high-level languages. The following are the most important:

CSM, already referred to in various parts of this paper, provides a powerful tool for the manipulation of files.

TEXT EDITOR: provides the facilities for displaying and context-editing line-by-line

LINK EDITOR: resolves the cross-references between related modules of applications object code.

ASSEMBLER: the basic assembler of System Ten has been enhanced to permit the new System 25 instructions to be compiled; this is a powerful macro assembler system.

RPG: The RPG-II compiler of System Ten has been enhanced to run under System 25 and takes advantage of the new object program format.

COBOL: a compiler is offered, together with a run-time library.

TESTER: this permits System 25 object code to be tested *in situ*, instruction by instruction or up to fixed breakpoints. It can be used to test applications where a video or workstation is available on the partition under test, or it can be used in a "blind" partition where the tests are to be run from a video connected to another partition. TESTER can be used also to amend areas on disc outside the constraints of DMF files or pools.

ALF (A Loadable Format) PATCHER: permits programs and overlays in a Program File to be amended. It also permits overlays and bodies to be inserted or deleted. Patches are check-summed to protect the system against mis-applied patches.

7 Concluding comment

While the foregoing account has of necessity dealt only superficially with some of the features of the System 25 software, the author hopes that it has made evident the power, richness and sophistication of the software provided for a modern 'small business computer'.

References

1 WALTON, A.: 'Architecture of the ICL System 25', *ICL Tech. J.*, 1981, 2(4), pp.319-339.
2 KNUTH, D.: '*The art of computer programming, Vol. 3*', Addison-Wesley Ltd., pp.473
3 KEMP, J. and REYNOLDS, R.: 'The ICL information processing architecture IPA', *ICL Tech. J.*, 1980, 2(2), pp. 119-131.

Appendix 1

Summary of LIOCS File I/O Macros

Macros	Description	can be used on file types:						
general purpose		R	D	VS	LS	IR	IMR	ILS
OPEN	open file	Y	Y	Y	Y	Y	Y	Y
CLOSE	close file	Y	Y	Y	Y	Y	Y	Y
WRTEOF	write end-of-file marker	Y	Y	Y	Y			

sequential								
BOF	reset pointer to start of file	Y	Y	Y	Y	Y	Y	Y
EOF	reset pointer to end of file	Y	Y	Y	Y	Y	Y	Y
QEOF	quick version of EOF				Y	Y	Y	Y
GET	read 'next' record, no lock	Y	Y	Y	Y	Y	Y	Y
GETUP	read 'next' record, locked	Y	Y	Y	Y	Y	Y	Y
PUT	write 'next' record	Y	Y	Y	Y		[Y]	Y
INSRT	insert a record and index entry				Y		[Y]	Y
DELETE	delete a record				Y		[Y]	Y
keyed								
READ	read record by key, no lock		Y			Y	Y	Y
READUP	read record by key, locked		Y			Y	Y	Y
WRITE	write record by key		Y				[Y]	Y
QWRITE	quick write by key, duplicates		Y				[Y]	Y
DELETD	mark record deleted		Y					
PUTD	write new record by key		Y					
QPUTD	quick write of new record		Y					
mixed								
UPDATE	rewrite current record	Y	Y	Y				
DELTR	rewrite current record	Y						
indexed								
BSP	get previous record, no lock				Y	Y	Y	
BSPUP	get previous record, locked				Y	Y	Y	
SCAN	get 'next' record, same key, no lock				Y	Y	Y	
SCANUP	get 'next' record, same key, locked				Y	Y	Y	
DELETI	delete record/index entry					[Y]		
UPDATI	rewrite current record by key				Y	Y		
PUTI	write new record by key					[Y]		
QPUTI	quick write of new record					[Y]		

Summary of File Characteristics and Access Methods

Characteristics	File Types						
	R	D	VS	LS	IR	IMR	ILS
fixed length records	Y	Y	Y	Y	Y	Y	Y
variable length records			Y				
blocked	Y	Y	Y		Y	Y	
buckets 1 block	Y						
minimum record length	10	10	10	10	10	10	10
maximum record length	9800	9800	9800	9300	9800	9800	9300
records can be in sequence	Y		Y	Y	Y	Y	Y
more than one file per pool				Y			Y
normal access (R = random, S = serial)	R or S	R	S	S	R	R	R
access by key		Y			Y	Y	Y
access by record number	Y	Y					
access serially	Y	Y	Y	Y	Y	Y	Y
starting not at first record	Y	Y			Y	Y	Y
insertion of records				Y		Y	[Y]
deletion of records	Y	Y		Y	[Y]	Y	[Y]

Legend

[Y] refers to index only
Y refers to data file only

R = Relative
D = Direct
VS = Variable Sequential
LS = Linked Sequential

IR = Indexed Relative
IMR = Indexed Mapped Relative
ILS = Indexed Linked Sequential
Y = Access via Primary index only

Security in a large general-purpose operating system: ICL's approach in VME/2900

T. A. Parker
ICL Security Consultant
Kennet House, Le Marchant Barracks, Devizes, Wilts., UK

Abstract

Now that computers are being used more and more frequently for the storage and manipulation of sensitive data, it is becoming increasingly important that the operating system that presumes to protect that information from any forms of unauthorised access should be capable of doing so. This paper describes how ICL has tackled the problem by designing the operating system VME/2900 so as to take advantage of the powerful support provided by the basic architecture of the 2900 range.

Security facilities, such as privacy mechanisms and the checking of log-in passwords, are of little use if they can be corrupted or by-passed. The paper therefore describes not only these but also the advanced software technology that ICL has used in the production of the system and the further work on security assurance that has been done to maximise the integrity of the security features: security correctness is a recurring theme throughout the paper.

The paper is a modified version of a presentation made to the Fourth American Department of Defence Seminar on Computer Security, held in Washington D.C. in August 1981. Some of the background material has been described already in papers in this Journal, but has been included here so as to keep the paper self-contained.

1 Preface

It is always difficult to describe specific features of an operating system in a way that is independent of the release state of the system. VME/2900 is no exception; many different system options are available and not all of the facilities described in this paper are available on all of them. Readers to whom correctness of such details for their system is important should consult the relevant technical manuals.

2 Background

ICL was formed in 1969 as a result of the merger of what were then the two major and competing British computer manufacturers, ICT and English Electric. It was

then realised that the new company would soon need a new range of machines to replace the many, varied and mutually incompatible ones inherited from the merger. Further, these inherited machines had architectures and hardware technologies dating from the 1950s and early 1960s, and both hardware and software technologies had moved on a lot since then. Out of all this, after an appropriate gestation period, came the first of the new 2900 series. This was a revolutionary step rather than an evolutionary one, a rare thing in the commercial computing world.

The design of the new series was of course influenced by that of existing in-house systems, and the architectures of other machines on the market were studied – for example, a number of the concepts of MULTICS were very influential, particularly in the protection sphere where some aspects still represent the state of the art even after 12 years. Details of this history are given in the book by John Buckle[1] and also in his paper[2] in the November 1978 issue of the *ICL Technical Journal.*

3 2900 Architecture

So what kind of machine did we produce? This Section looks at some of the architectural features of the 2900 and the next, Section 4, describes VME/2900 and explains how it uses these features.

3.1 Virtual addressing

Central to the architecture of the 2900 series are the complementary concepts of *virtual store* and *virtual machine*, and their common basis of *virtual addressing*. All addressing is in terms of virtual addresses mapped on to real addresses by hardware using *segment* and *page* tables, as shown in Fig. 1. The real addresses can be in the real main store or in the secondary store on drum or disc: thus we have a straightforward virtual store implementation. Each process runs in its own virtual machine in which it has its own unique local segment table and also shares a public segment table with all other virtual machines. It can have also *global* segments, which it shares with chosen other virtual machines. It is well accepted that this kind of hardware-supported separation of the address spaces in which different processes run is important to good system security. The separation provided by the 2900 architecture is not merely a matter of the operating system performing checks on the base address and displacement of an operand but is a fundamental feature of the architecture and ensures that no process can meaningfully address the local space of any other.

3.2 Descriptors

The primitive instruction code makes extensive use of *descriptors* for indirect addressing. A descriptor is a 64-bit entity which formally describes an item of information in store. One half contains the base address of the item in terms of the segment number and the displacement: that is, the virtual address. The other

half contains information relating to the unit size of the item, the number of units it contains, whether modifiers added to the item's address should or should not be appropriately scaled, and so on. Descriptors are also typed according to what kind of information they are addressing.

The principle is illustrated in Fig. 2. This shows a *descriptor descriptor* pointing to a row of *byte-vector descriptors*, each of which is pointing to a bounded area of virtual store: the 2900 architecture provides for the automatic checking of

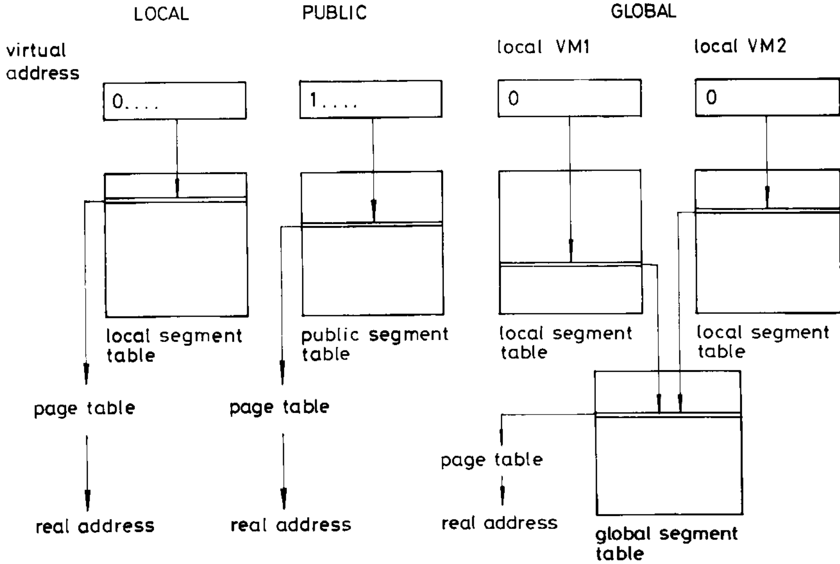


Fig. 1 Virtual addressing

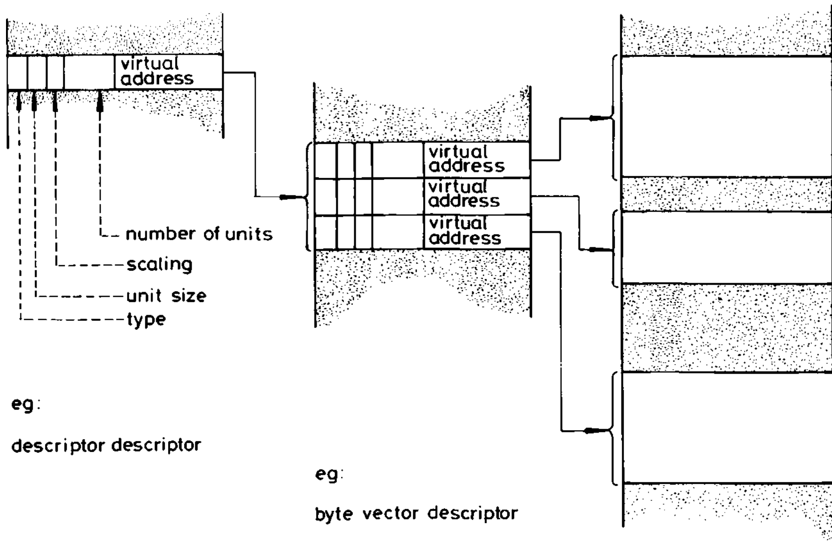


Fig. 2 Descriptors

address bounds after modification, which is clearly a most important integrity feature. Other types of descriptor include *code descriptors, semaphore descriptors* and *system call descriptors.*

A common source of security and other errors in non-descriptor systems is their inadequate and sometimes omitted reference pointer validation. This is from the point of view of not only the actual address referenced, but also the type of object being addressed. The use of descriptors enables the detection of many of these classes of error to be made naturally and automatically; and because descriptors are an integral part of the system's architecture, the consequent performance cost is low.

3.3 Privilege

In every machine architecture there are certain low-level operations whose misuse could be prejudicial to system security. One example is the primitive input/output operation, which communicates directly with the system's peripheral devices; if a normal user program were able to utilize this directly it might be able to by-pass any file privacy system imposed by the operating system.

In the 2900 architecture, potentially dangerous operations like this are controlled by means of a system status which is called *privilege*. Only a privileged procedure may perform privileged operations. Privileged status is obtainable only by a hardware interrupt mechanism. When the VME/2900 operating system is present, such interrupts cause entry to the most trusted part of the system, the *kernel*, the place of which in the system is described later in Section 4.1.

3.4 The ACR protection system

A process's level of trustedness is defined by the contents of a protected hardware register, the Access Control Register or ACR. The level is called the ACR level and the lower its numerical value the more trusted is the process. There are 16 ACR levels, that is, 16 possible levels of trustedness.

The protection of a segment in store from unauthorised access is effected by the combination of information carried by the segment on the one hand and by the process attempting access on the other. The segment's information gives the maximum level at which it may be accessed (i.e. the minimum level of trustedness) and the modes of access that are allowed – for example, any form of modification may be forbidden; and this is compared with the ACR level of the process and the mode of access being requested. One mode of access to a segment is 'Change Access', which concerns the ability to change the access-permission fields themselves. There is also an Execute Permission bit which is used to prevent the accidental execution of data (as program).

If a procedure attempts to enter another running at a different, usually lower, ACR level, the hardware invokes a 'system call' mechanism which polices the

availability of the called procedure to the caller. An important feature here is that the mechanism will enforce entry, if permitted, at the proper entry point. It is vital to the security of the system that a trusted program when called by a less trusted code shall properly validate any parameters passed to it. A weakness here might enable a malicious user to penetrate the system completely, so a special primitive instruction, called the *validate* instruction, has been provided to enable validation to be performed correctly and efficiently. This important matter is also discussed further in Section 4.2.

3.5 Recap

It is helpful now to collect together the features described so far and to identify how they combine together to give two dimensions of protection:

Protection between processes is provided by

- virtual addressing, supporting
- virtual store and
- virtual machines
- global segments and the control of public segments

Protection within a process is provided by

- descriptors with automatic bound checking
- a mechanism to protect input/output and other privileged operations
- a 16-level ACR protection with
- an associated mechanism for policing the transfer of control between levels

These are all basic architectural features, supported by hardware and present in the raw machine.

3.6 Process stack

This is one further architectural feature which merits brief mention. It has no direct security connotations but makes such an important contribution to the overall flavour of the 2900 architecture that it would be misleading to miss it out.

The instruction code at the primitive level is based on the use of a LIFO or 'Last In First Out' stack. This is used for parameter passing and for local name space purposes and each virtual machine has its own stack. Nested procedure calls will cause the usual succession of name spaces to be built up on the stack, and to be deleted on the 'last in, first out' principle as the procedures exit. ICL has found the process stack on 2900 to be an elegant and natural aid to the procedure call mechanism.

These then are the main architectural features of the 2900 series machines; they form a firm basis for the development of a secure operating system.

4 VME/2900

One such development is VME/2900: VME stands for Virtual Machine Environment. It is a large, mixed work-load operating system which caters for batch, multi-access and transaction-processing applications.

4.1 Structure

VME/2900 divides into three distinct parts, separated by error handlers; this is shown in Fig. 3. The error handler at each level traps errors occurring at any ACR level above it and takes the appropriate action. The one at ACR 6, for example, confines errors occurring above it to the virtual machine concerned, the rest of the system remaining unaffected.

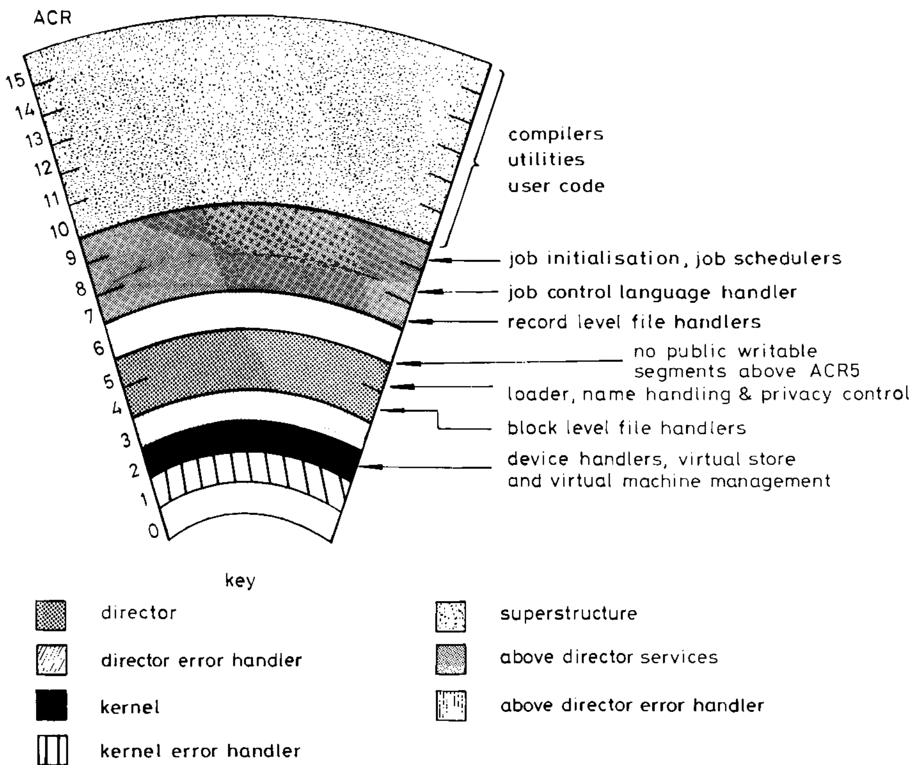


Fig. 3 VME/2900 structure

At the most trusted level is the *Kernel*, which handles real system resources like store and peripheral devices. It runs mainly 'out of process', that is, not in a virtual machine but on a public stack; and helps to support the virtual store/virtual machine image of the basic architecture.

Next above this and occupying ACR levels 4 and 5 is *Director*, responsible for the handling of a more abstract view of the system's resources. At this level are the block-level file managers and the major security-related functions of the operating system like the Loader, Name Handler and Privacy Controller. Uncontrolled communication between virtual machines is prohibited above ACR level 5 by disallowing the existence of public segments with write-access keys greater than ACR 5 and by controlling the availability of global segments.

Levels 7 to 9 contain the *Above Director* software: from the security point of view this can be considered as a sort of 'trusted superstructure'.

Above Level 9 is the user, who must be treated as potentially malicious. Facilities are provided for user installations to structure the levels at which the various application programs may run between ACR 10 and 15, and these can be used by the management to cut off unprivileged users either partially or completely from direct use of the operating system, if this is required. I return to this point later. Notice here that, in contrast to what is done in some contemporary machines, compilers and general utility programs are no more trusted by the operating system than is user code.

All operating system code segments are established with write-access key of zero; so all operating system code is necessarily pure.

4.2 Interface control

Fig. 4 shows a selection of operating system procedures and VME/2900's use of the system call mechanism. The small boxes are the procedures, drawn at their execution ACR level. The lines with blobs on the end show in each case the highest ACR level from which the procedure can be called. In the actual system the vast majority of procedures cannot be called at all from outside their own level, but there remain a substantial number that can be called directly from user ACR levels; these provide the total set of VME/2900 facilities available to the user.

As has been said already in the paper, the proper validation of parameters passed across the interface between the user and the operating system is critical to security correctness in any system, and VME/2900 is no exception. Much time and effort has therefore been spent in ensuring that this validation is complete. In particular, a package has been written for the analysis of object code which searches for discrepancies and presents these for manual analysis and correction. It examines actual loaded code and so detects flaws which might be introduced by post-compilation patches and repairs which at that stage have been fully applied. The checking is therefore as near to the 'engine' as possible.

Another approach has been to reduce the number of procedures available to unprivileged users at particular installations. A package has been written which will monitor the use of system code so that the installation manager can make unused interfaces unavailable or restrict the availability of little-used interfaces as required. This package has a very low performance overhead and so can be left permanently

in the system. Any reader who is interested in further details should consult the author.

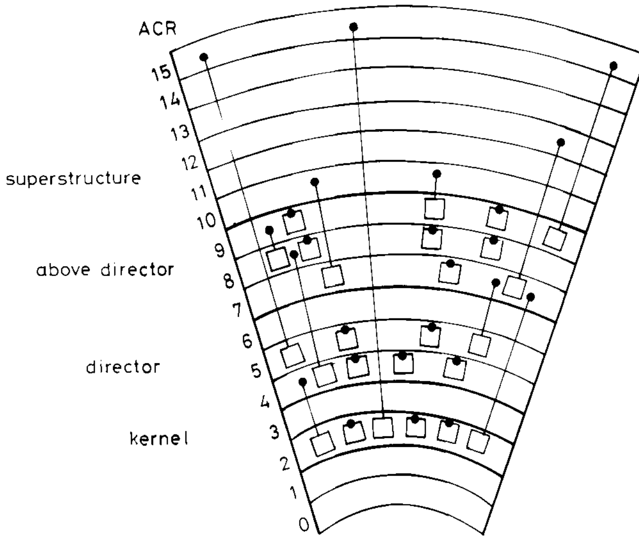


Fig. 4 Access to operating system interfaces

Installations have a considerable degree of control over the availability of operating system procedures. The ACR level from which each procedure can be called is defined at system load time and is held in a control file interestingly called the 'recipe' file; this is amendable only by the installation management. There is also a mechanism for giving trusted users access to specified additional, more powerful, procedures, the availability of which can be tailored to meet the specific functional requirements of the chosen trusted user classes – for example, support engineers, operators and the system manager. There are in fact a number of areas in VME/2900 into which hooks and options have been put and which give installation security authorities a great deal of flexibility in deciding what they want for their system. Indeed, the extent to which particular installations can adapt VME/2900 to suit their individual security requirements is itself a major security feature of the system. To give a very simple example, a class of multi-access user could be defined whose only commands were, say:

INPUT
EDIT
COBOL_COMPILE
COBOL_RUN

with no low-level code or direct use of any operating system interface being allowed at all.

4.3 *Catalogue*

All major system objects are recorded in a central filestore database known as the

Catalogue, which is controlled from ACR 5. It is organised in terms of *nodes* and *relationships:* entries for named objects are located at the nodes, which are connected by the relationships. Objects catalogued include Devices, Volumes, Files and other specialised VME/2900 objects. An example of a VME/2900 object is shown in Fig. 5, a Job Profile; the set of job profiles available to a user will determine the kind of work he can run on the system and will normally be controlled by the system manager.

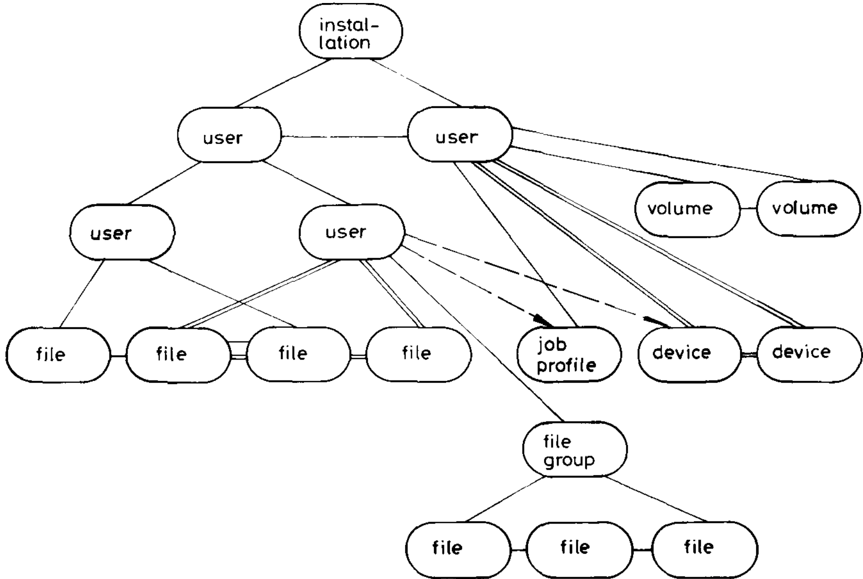


Fig. 5 VME/2900 catalogue structure

Privacy controls can be applied to any or all of these objects and access can be constrained on a general, specific or hierarchic basis. A wide variety of access types is supported, distinguishing for example between access to a file's contents and access only to its name and description. All attempted violations of privacy are logged in a security journal; an installation can arrange that such messages are output immediately to the journal rather than held in a buffer and output only when the buffer is full. Particularly important in the access control features is the ability which the system manager has, to prevent users other than chosen individuals from accessing the system from a specified terminal. Alternatively, all users *except* certain named individuals can be allowed to use a particular device: this is useful, for example, in preventing the system manager's identity from being used at any terminal except a particular one, and is additional to the protection provided by the manager's password.

4.4 User authentication

Users are identified by a catalogued username which can have one of three security levels: low, medium, high. A high-security user may not submit batch jobs; high

and medium-security users must submit a password when logging in for a multi-access (MAC) session. Such passwords can be up to 12 characters long and are irreversibly encrypted when stored at the catalogue node for comparison on logging-in. In this way, sight of the stored version is made useless to the would-be penetrator.

The log-in sequence is very tightly controlled: The would-be MAC user, having once started the sequence, will either obtain legitimate access within a certain time or will cause the terminal to be locked out with an immediate security alarm at the master operator's terminal. Line breakdowns, for example, at this stage are treated as security violations, so pulling the plug out won't do the user any good.

A *reverse password* facility also is available, with which the system can be made to identify itself to the user. By this means the user can be sure that he is signing on to the system that he intended to sign on to.

4.5 ACR control of file-access

Another feature is program-controlled access to files, using ACR levels. By this means the management can force access to chosen files to be through software specially written by the installation which can perform extra protection checks, for example can require passwords for individual files. This feature has been utilised by the ICL database management system IDMSX, developed from the Cullinane Corporation design. In this way IDMSX has become one of the few securely protected database systems available commercially today.

5 Production methodology: CADES

Most modern computer operating systems offer a reasonable range of security facilities, but few of their designers have seriously addressed the problem of ensuring that these facilities are correctly implemented. Security features that can be by-passed or corrupted by knowledgeable users give a false sense of security and can therefore be a danger rather than an asset.

It is here that the use of proper software engineering design and production techniques becomes of paramount importance. If the design of the system is well structured and well documented it is easier to compartmentalise it and to perform the validation processes that form the core of any theoretical analyses of its integrity. And, of course, the system is more likely to be right in the first place. It is particularly important for these purposes that the operating system is written in a high-level language. This imposes a fine-grain discipline and structure, and the compilation process performs automatically many of the correctness checks which would have had to be performed manually and which are probably too numerous for that to be feasible in a low-level language.

This Section describes the methodology used by ICL, called CADES – Computer

Aided Development and Evaluation System – and gives a flavour of the languages used in the implementation of VME/2900.

CADES is a methodology and a set of mechanisms to support that methodology. The VME/2900 design is top-down, data-driven and hierarchic; the prime objective of CADES was that the product was designed before it was implemented. Everyone knows how difficult that is in the pressures of a commercial production environment. The design methodology is supported by mechanisms which may consist either of well-established rules for human actions and interactions (called the CADES Design Rules) or of software products to be used by the designers and implementers. The hierarchies of modules and data structures, together with their attributes and relationships, are stored in the CADES database and this forms an authorised description of the product as it is being developed. The final content of the database is the product itself, so there is no break in continuity between design and production. The source code and the compiled code are controlled by and are logically a part of the database itself.

Thus CADES supports the total software development cycle from the initial design right through to successive releases of the system, with supporting documentation; and as a result, VME/2900 is extensively documented in a structured manner in a micro-fiched multi-volume library known as the Project Log. For example, system-wide cross-reference listings of the usage of data objects and of the structures of procedure calls are available in the CADES database and can be reproduced automatically for all new releases. There is a very good description of CADES in the paper by McGuffin *et al.*[3] in the May 1980 issue of this Journal.

It is important to say at this point that CADES does not have the richness of a design language nor the degree of formalisation enjoyed by formal languages like SPECIAL or GYPSY.[4] It was never intended to be used as a basis for the later formal verification of the correctness of the design. Nevertheless there is no doubt whatever that CADES has proved an invaluable practical tool.

The implementation language of VME/2900 is called S3 and is a development of Algol 68; it is a well-structured high-level language with moderately strong typing and a block structure which is very suitable for the 2900 stack architecture. The production teams, however, actually code in what they call SDL – System Description Language – which is automatically converted into S3 by the CADES system; this enables basic essentials like complex data mode declarations, interface parameter specifications, constant and failure code values, macro expansions and so on to be looked after automatically by CADES.

Fig. 6 shows an example of some implementation level SDL. It is actually the code for a module which is part of the CADES system itself – we use CADES to design and build CADES.

It is not the intention of this paper to describe Fig. 6 in detail; it is there just to give a flavour of the language. At the top are the EXT and IO sections which list the procedures that this procedure calls and the external data areas referenced. The

54.1.2.2.E.311
PHASE 50

```
 1   HOLON      : EN_PARAMETER_VALIDATOR;
 2   VERSION    : 001;
 3   EXT        : COMMON : CHECK,
 4                         MOVE,
 5                         SCANUNQ,
 6                         ;
 7                SCLMAC : TRANSLATE_HIERARCHIC_NAME,
 8                         ;
 9                       : EN_OUTPUT_MESSAGE,
10                         EN_EXTEND_HEAP,
11                         ;
12   10         : E :
13                    EN_OUTPUT_PHASE,
14                    EN_OUTPUT_LIBRARY_OPEN,
15                    EN_TRUSTED_USERNAME,
16                    EN_TRUSTED_USERNAME_BUFFER,
17                    EN_TRUSTED_USERNAME_CURRENCY,
18                P : HOLON_NAMES,
19                    SELECTORS,
20                    NON_STD_PHASES,
21                    GP,
22                    ENV_DE,
23                    ACTNAME;
24   FUNCTION       :
25   BEGIN
26         (32)BYTE ANB : =
27   *(L)             EN_ALPHANUMERIC_BREAK,
28                       AN  : =
29   *(L)             EN_ALPHANUMERIC,
30                       NRMC:=
31   *(L)             EN_NUMERIC;
32         RESULT_CODE :=
33   *(L)             SUCCESS;
34         UNLESS
35                    HOLON_NAMES IS NIL
36         THEN
37                    FOR 1 to BOUND HOLON_NAMES
38                    DO
39                          REF() BYTE HN IS HOLON_NAMES(1);
40                          UNLESS
41                                LENGTH HN LE 32
42                                AND
43   *                            CHECK(ANB, HN, 0, NIL)
44                                AND
```

```
45                 IF
46                       LENGTH HN = 0
47                 THEN
48                       TRUE
49                 ELSE       @ ALPHABETIC 1ST CHAR ?@
50                       HN(0) GE "A"
51                       AND
52                       HN(0) LE "Z"
53                 FI
```

Fig. 6 Example implementation level SDL

interface definitions and modes of these items are all held in the CADES database: the existence of centrally held definitions of system-wide objects like these automatically reduces risks of mismatches in all of these areas. The asterisks at the beginning of some of the lines in the FUNCTION section trigger off various substitution and validation actions when the system is converting the SDL code into S3.

One final point of interest, to do with the CADES design: there are no GO TOs in VME/2900 – well, hardly any!

6 Additional security work

Obviously, some of ICL's customers will have special security requirements; a substantial number of extra security features have been developed to satisfy these. Work has been done also on 'hardening' VME/2900, not only from the point of view of providing extra facilities but also from that of security correctness. Of course, everybody benefits from the correctness; and some of the additional facilities have since become standard product line items.

In the pursuit of correctness ICL has been subjecting the primitive architectural and low-level operating system features to theoretical analysis, backed up where appropriate by actual tests. This is a continuous process because new releases of the system are continually providing new areas to be examined; and for this reason ICL has attempted to automate the analysis as far as possible. Most of the tools developed in this work are incorporated into a 'security test package' which includes also tests of those standard security facilities that are visible to the user. This package is now being applied during the acceptance testing phase of each new release of the product.

ICL also maintains close relationships with its major secure users and conducts regular meetings devoted to the examination and discussion of security issues at both technical and general levels.

The security of the VME/2900 system can be more clearly appreciated by standing back and looking at the overall security structure. One current example of this is a development of an idea put forward by Linde[5] of SDC, that of the 'security control object dependency graph'; it is hoped that this will be found useful in identifying areas requiring most attention.[6]

An early 'hardened' version of VME/2900 was subjected to a 'tiger team' attack a few years ago, with encouraging results. In that attack the system demonstrated a reasonable degree of security in that the team failed to achieve their major penetration objectives, though there were a small number of known defects which were declared as 'no go areas' and others which had to be compensated for by rather restrictive procedural controls. These defects have of course been removed since then, but it would be foolish to claim that the system is now therefore totally secure. What it does show is that any claim that a modern well-structured commercial operating system is easy to penetrate has to be examined very carefully. The great majority of successful penetrations of other systems have been by teams consisting of top-class systems penetration specialists.

A system that has been so penetrated, as VME/2900 might well be one day, cannot be dismissed as insecure. Security is not a binary property that is either present or not, as has been clearly recognised in Nibaldi's valuable work.[7,8] ICL takes a pragmatic approach to security: the approach has to be pragmatic with so large a system as VME/2900. Absolute security cannot be claimed: all that can be done is to eliminate as many potential loopholes as our expertise and the state of the art allow.

References

1 BUCKLE, J.K.: *The ICL 2900 Series*, McMillan, London, 1978.
2 BUCKLE, J.K.: 'The origins of the 2900 Series', *ICL Tech. J.*, 1978, 1(1), pp.5-22.
3 McGUFFIN, R.W., ELLISTON, A.E., TRANTER, B.R. and WESTMACOTT, P.M.: 'CADES – software engineering in practice', *ICL Tech. J.*, 1980, 2(1), pp.13-28.
4 CHEHEYEL, M.H. *et al.*: 'Secure system specification and verification: survey of methodologies', MITRE Corporation, Bedford, Mass. USA MTR-3904, Feb. 1980.
5 LINDE, R.L.: 'Operating system penetration', Proc. US National Conference Feb. 1975, pp. 361-368.
6 PARKER, T.A.: 'Analysis of the security structure of a commercial operating system', ICL, internal paper, July 1981.
7 NIBALDI, G.H.: 'Proposed technical evaluation criteria for trusted computer systems', MITRE Corporation, Bedford, Mass. USA M79-225, Oct. 1979.
8 NIBALDI, G.H.: 'Specification of a trusted computing base', MITRE Corporation, Bedford, Mass. USA M79-228, Nov. 1979.

System evolution dynamics of VME/B

B.A.Kitchenham
ICL Product Development Group, Northern Development Division, Kidsgrove, Staffs

Abstract

The development of the ICL Operating System VME/B, was investigated over successive system releases in the light of the theory of program evolution dynamics developed by M.M.Lehman and L.A.Belady.

The results were mainly in accord with those predicted by the theory, thus providing an independent verification of the general applicability of the theory.

In addition, the particular results for VME/B base code show none of the undesirable characteristics associated with severe complexity problems. VME/B evolution shows a stable work rate with growth being maintained; there is no evidence of increasing system complexity as the system ages, nor is there any systematic increase in inter-release interval.

1 Introduction

1.1 Background

The theory of program (or system) evolution dynamics, as proposed by Belady and Lehman,[1,2] is concerned with the effect of change on widely used programs such as operating systems. They take as a premise that large-scale programs undergo a continuing cycle of maintenance and enhancement to keep pace with the changing requirements of users and with technological innovation. Belady and Lehman[2] proposed three 'laws' of evolution dynamics which were later expanded by Lehman to five.[3,4] They were able to model the evolution dynamics of a particular system using five parameters derived from four measurements. They advocated the use of such models as a means of improving the planning and control of system development.

It is evident that a theory of evolution dynamics could be of great potential value to the monitoring and control of a large operating system such as VME/B*. This paper reports the results of an investigation of the evolution dynamics of VME/B, intended to assess the relevance of the theory to VME/B, and if it did prove relevant, to assess the current and future status of VME/B.

The problems associated with the development and maintenance of large operating systems have already been documented.[6] Belady and Lehman's work is a first

*Since the time of writing of this article System VME/B has been renamed VME/2900.

attempt to bring the development of large systems under intellectual and managerial control. The practical importance of their theory demands that it be subject to serious study under as many and as varied conditions as possible.

However, as yet the theory has only been tested on a limited number of systems.

Lehman and Parr[5] identified three systems which have provided the basic data for program evolution investigations. One was a large operating system with many customers and two were smaller developments intended for only a limited number of users. This study represents only the second attempt to investigate the evolution of a large operating system. The results therefore have important implications for the generality of Belady and Lehman's theories and will be discussed in the context of their ability to validate or invalidate the proposed laws.

1.2 The laws of program evolution dynamics

The theory is encapsulated in five basic principles or laws which may be summarised as follows:

(i) The law of continuing change: A large system that is used undergoes continuous change or becomes progressively less useful. The change or decay process continues until it is considered more cost-effective to provide a replacement system.

(ii) The law of increasing complexity: If a system is subjected to continuous change, its complexity, which is a result of structural deterioration, will increase unless work is done to halt or reverse the trend.

(iii) The fundamental law of large-program evolution: Measures of global project and system attributes are cyclically self-regulating with statistically determinable trends and invariances.

(iv) The law of conservation of organisational stability (invariant work rate): The global activity rate in a large programming project is statistically invariant.

(v) The law of conservation of familiarity (perceived complexity): For controlled development, a large system undergoing changes must be released at intervals determined by a safe maximum release content. Exceeding that maximum causes problems in integration with cost overruns. These unpleasant results help to regulate excessive growth and thus maintain statistical invariance.

2. Data collection and analysis

2.1 The data

Data were available, in computer filestore, concerning the content and development of a number, but not all, of VME/B Base releases.

The data, relating to the development of a release, resulted from the process of incremental development under which VME/B is developed. Incremental development allows many versions of VME/B to be constructed between two major releases in the following manner: when a module (or group of modules) is considered to be in a state suitable for in-house exposure, it is incorporated into the current version of VME/B to create a new version (or system increment) and is used to provide the in-house computer system. If, as a result of the in-house exposure, the module requires correction or modification, a new version is created which in turn is incorporated into a new increment. Thus, data were available to identify not only which modules were handled between releases but also how many incremental versions of a module were required before it reached a state suitable for release.

VME/B Base (or Operating System) modules divide into three main groups:

Kernel-loader loaded modules, often designated Kernel; Supervisor-loader loaded modules often designated Director; and conventionally-loaded modules often designated Above Director.

Kernel and Director are loaded during the system load sequence, Above Director is loaded when a module is called. The data used in this investigation were limited to Kernel and Director information spanning seven Kernel releases and ten Director releases. Two of the Director releases and one of the Kernel releases were 'special' releases but because of the limited number of releases they were retained in the analyses. The releases covered by this analysis are (in an increasing time sequence):

Release	Director	Kernel
5X23	DOF70	not available
5X27	DOM25	not available
5X32	DOXOH	KDCO2
5X33	DOZ08 (ATS1)	KDM90
5X33	DOZ70(DAP)	
5X36	D0123	KDUF1
5X37	D0132	K0026
6.10	D0160	K0068
6.11	D0170	K0089
6.21	D0179	K0102

The name given to the Kernel and Director comprising a released version of VME/B Base code correspond to the incremental construction identifier. There are releases of VME/B earlier than 5X23 but information concerning their detailed content was no longer available in computer filestore; this was also the case for the 5X23 and 5X27 Kernels.

The two 5X33 releases were specials but because they were also part of the sequential development of VME/B, their inclusion was additionally justified.

Because of the discrepancy between the availability of information between Kernel and Director, it was necessary to analyse the two systems separately. It is not an

arbitrary decision because the production of Kernel and Director modules continues separately and incremental versions of Kernel and Director are constructed quite separately.

2.2 System measurements

Belady and Lehman[1,2] defined a number of system measurements that they used to investigate the behaviour of a particular system. The following five measurements are the primary data obtained for each release; the first four measurements correspond to those used by Belady and Lehman,[1,2] the fifth is an additional measurement available for VME/B.

(i) Release Sequence Number (RSN): this identifies the releases in sequence from the base release, DOF70 for Director and KDC02 for Kernel.

(ii) Inter-release Interval (I): this is the interval in weeks between the construction of two incremental systems. Because this interval is based on system construction rather than customer release, the intervals between two Directors and two Kernels, which relate to two consecutive customer releases, may be different.

(iii) System Size in Modules (M).

(iv) Number of Modules Handled (MH): this is a count of the number of modules amended or new since the previous release.

(v) Number of Module Versions Handled (MVH): this is a count of the total number of different versions of modules handled during the development of a release.

Also, from the five primary measurements four additional statistics were calculated.

(vi) Actual Work Rate (AWR) = MVH/I: This is the average number of modules handled per week including multiple versions.

(vii) Effective Work Rate (EWR) = MH/I: this is the average number of modules handled per week excluding multiple versions.

(viii) Fractional Change (FC) = MH/M: this is the fraction of the system altered to produce a given release.

(ix) Release Handle Rate (RHR) = MVH/MH: this is the ratio of the total number of different module versions handled to the total number of different modules. It is a measure of the degree of difficulty associated with establishing the content of a release.

2.3 The analysis

Because there are only a limited number of data points (i.e. releases) available,

the results of any statistical or graphical analysis must be treated with caution. The procedure used in this study was to plot the measurements of interest against either release number or system age in order to observe whether any system evolution trends could be identified and to use statistical techniques to investigate specific hypotheses about the data.

Thus, system size was plotted against system age and release number; and work rate (actual and effective), fractional change and release handle rate were plotted against release number. Relationships between all the system measurements were investigated using correlation analysis. In addition, the relationship between system size and release number, for both Kernel and Director, were subjected to regression analysis.

An additional attempt was made to investigate the complexity of VME/B base code by comparing the average module handle rates for comparable groups of new modules and amended modules within releases. It was expected that serious complexity problems would reveal themselves in a higher average handle rate for amended modules than for new modules. Comparable groups of modules were identified by obtaining average handle rates within VME/B subsystems. Subsystems that were subject to development involving at least 10 new and 10 amended modules within a release were selected and the average handle rates for each group of modules were calculated and compared.

3. Results

The basic program evolution measurements for Director are shown in Table 1, with the derived measurements shown in Table 2. The equivalent results for Kernel are shown in Tables 3 and 4.

Graphical representation of the evolution of VME/B Director and Kernel systems is shown in Figs 1 – 10. Figs 1 and 2 show the gross sytem size of Director and

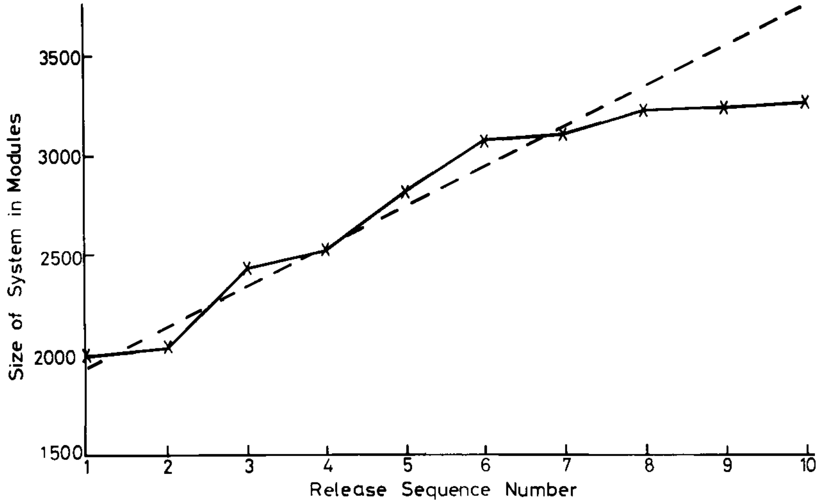


Fig. 1 Size of Director releases subsequent to DOF70 in relation to Release Sequence Number

Table 1 Basic program evolution metrics for the VME/B Director

Director	RSN	M	MH	MVH	I (weeks)
DOF70	1	1991	–	–	–
DOM25	2	2032	847	1267	34
DOXOH	3	2442	1345	2686	51
DOZ08	4	2516	466	581	15
DOZ70	5	2798	894	1295	19
DO123	6	3061	1562	3240	44
DO132	7	3086	601	748	14
DO160	8	3226	1011	1668	23
DO170	9	3228	213	241	11
DO179	10	3239	260	331	9

Table 2 Derived metrics for the VME/B Director

Director	AWR = MVH/I (modules/week)	EWR = MH/I (modules/week)	FC = MH/M	RHR = MVH/MH
DOF70	–	–	–	–
DOM25	37	25	0·42	1·50
DOXOH	53	26	10·56	2·00
DOZO8	39	31	0·19	1·25
DOZ70	68	47	0·32	1·45
DO123	74	35	0·51	2·07
DO132	53	43	0·19	1·24
DO160	73	44	0·31	1·65
DO170	22	19	0·07	1·13
DO179	37	29	0·08	1·27

Table 3 Basic program evolution metrics for VME/B Kernel

Kernel	RSN	M	MH	MVH	I (weeks)
KDCO2	1	493	–	–	–
KDM90	2	745	514	1500	34
KDUF1	3	877	620	1783	34
KOO26	4	900	344	679	28
KOO68	5	960	375	768	21
KOO89	6	971	140	222	11
KO102	7	996	310	426	10

Table 4 Derived metrics for the VME/B Kernel

Kernel	AWR	EWR	FC	RHR
KDC02	–	–	–	–
KDM90	44	15	0·69	2·92
KDUF1	52	18	0·71	2·87
KOO26	24	12	0·38	1·94
KOO68	37	18	0·39	2·05
KOO89	20	13	0·14	1·59
KO102	43	31	0·31	1·37

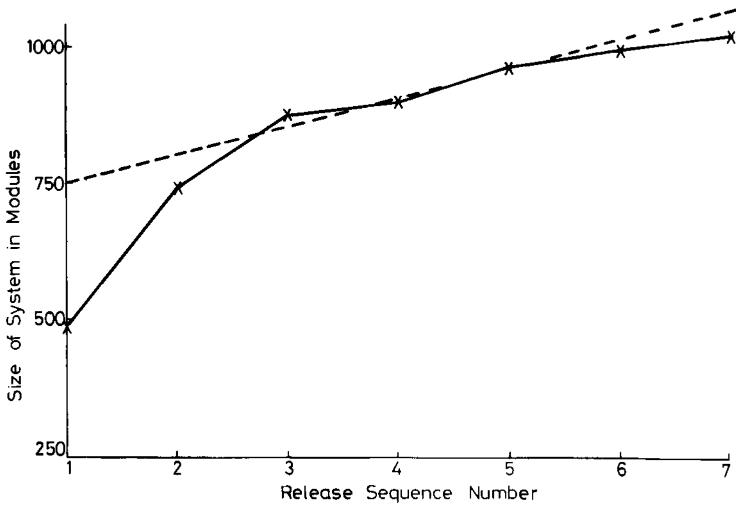


Fig. 2 Size of Kernel releases subsequent to KDCO2 in relation to Release Sequence Number

Kernel, respectively, plotted against release number. Lehman and Belady[2] observed a linear relationship between size and release number. For the Director system a linear relationship appears a good fit to the first eight data points as shown on the graph but the last two releases show very different growth characteristics. For the Kernel system, the last six releases appear to be following a linear relationship while the first releases have a different trend.

Figs 3 and 4 show the gross system size of Director and Kernel, respectively, plotted against system age. A change of variable from release interval to system age appears

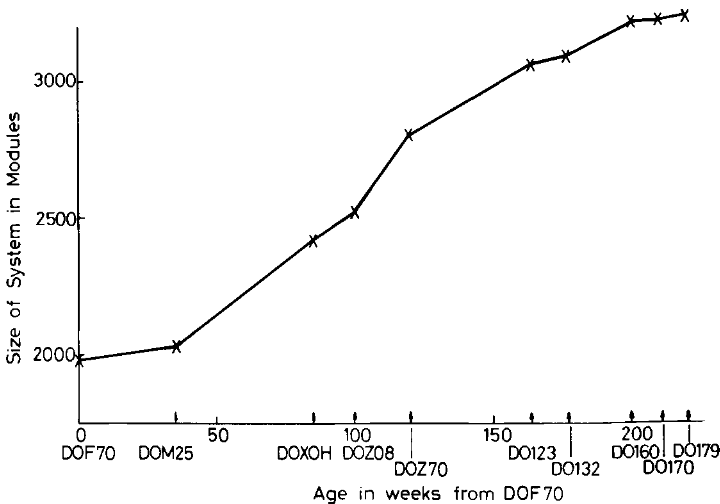


Fig. 3 Size of Director releases in relation to system age

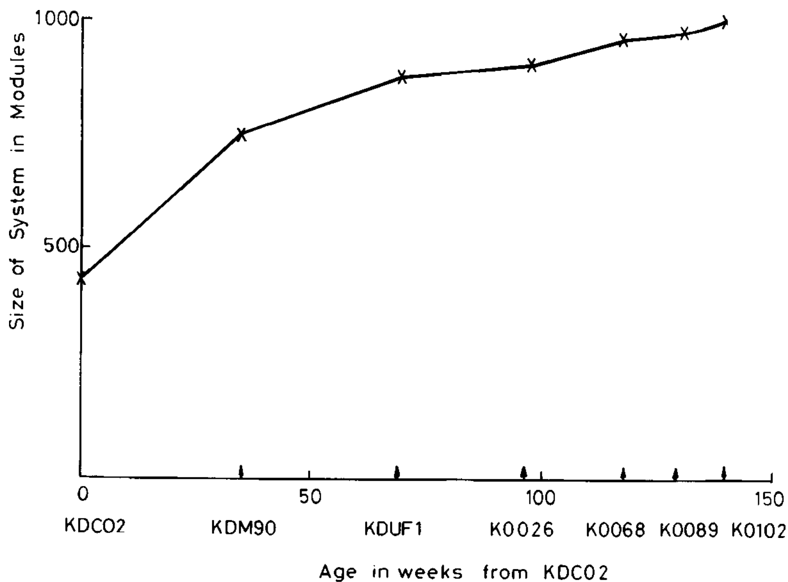


Fig. 4 Size of Kernel releases in relation to system age

to smooth the trends seen in the Figs 1 and 2. This indicates that system size is influenced by the combined effect of age and inter-release interval.

Figs 5 and 6 show the fraction of the system changed for Director and Kernel, respectively. The Director results appear to show a cyclic pattern of peaks and troughs imposed on a declining trend. The Kernel results appear to show a generally declining trend. The fraction of the system changed per release is believed to be related to inter-release interval.[5] The correlation between fractional change and inter-release interval was 0·96 for Director and 0·90 for Kernel. In spite of the small number of releases both these correlations are significant ($p < 0.01$ for the Director correla-

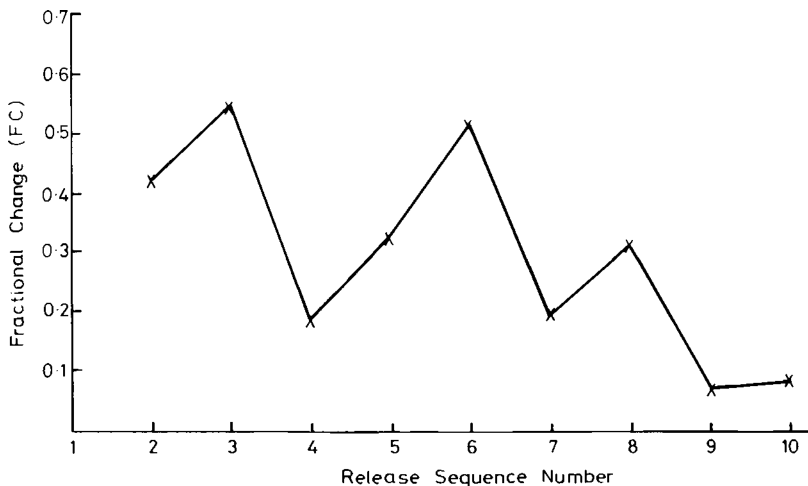


Fig. 5 Fraction of Director altered in each release subsequent to DOF70

tion, p <0·05 for the Kernel correlation). The apparent decrease in change with release number is probably related to the fact that the later releases have, in general, had small inter-release intervals. Thus, although both Director and Kernel show significant negative correlations with release number (−0·69 and −0·87 respectively), Director shows a high but not significant negative correlation between release number and inter-release interval (−0·63) and Kernel shows a highly significant negative correlation (−0·97).

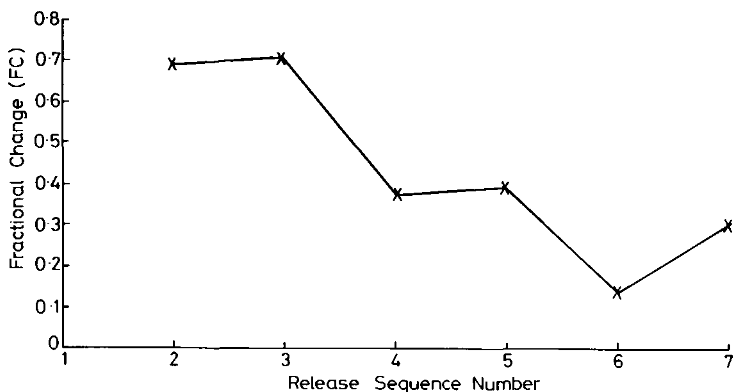


Fig. 6 Fraction of Kernel altered in each release subsequent to KDCO2

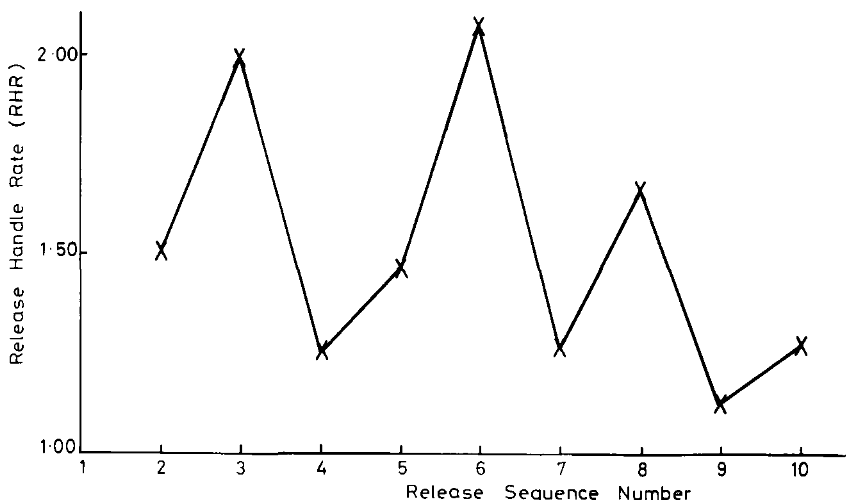


Fig. 7 The Release Handle Rate for Director releases subsequent to DOF70

Figs 7 and 8 show the release handle rate for Director and Kernel, respectively. The pattern for Director appears to be a cyclic pattern of troughs and peaks, while the pattern for Kernel appears to be one of a general decrease in handle rate with increasing release number. Relationships between release handle rate and inter release *interval* revealed highly significant correlations for both Director (0·93) and Kernel (0·94). Kernel revealed also a highly significant negative correlation between release number and release handle rate (−0·95), but this result is complicated by virtue of the correlation between release number and inter-release interval. Director,

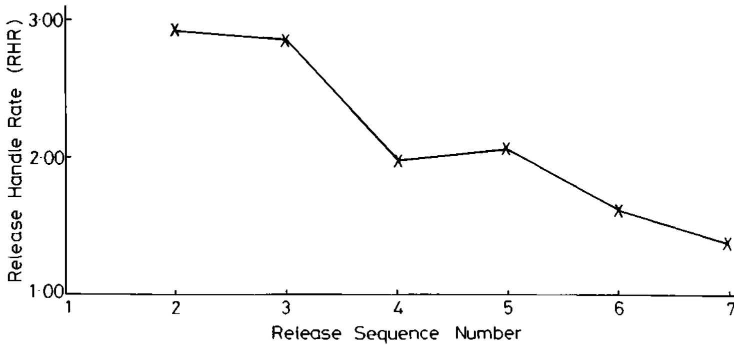


Fig. 8 The Release Handle Rate for Kernel releases subsequent to KDCO2

however, revealed only a small non-significant correlation between release number and release handle rate (−0·40), in contrast to the results for fractional change.

Figs 9 and 10 show the actual and effective work rates for both Director and Kernel. The pattern for Director and Kernel is a widely fluctuating work rate and a more stable effective work rate.

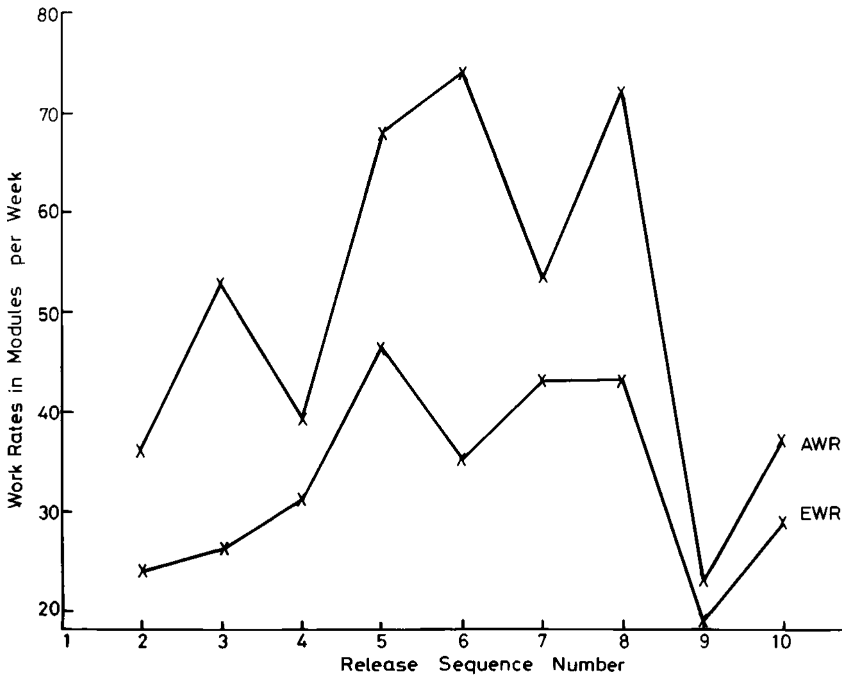


Fig. 9 Actual Work Rate (AWR) and Effective Work Rate (EWR) for Director releases subsequent to DOF70

The variation of actual work rate is over three times that observed for effective work rate for both Director and Kernel. In addition, the Director work rates were over twice as variable as the Kernel rates. Neither actual work rate nor effective

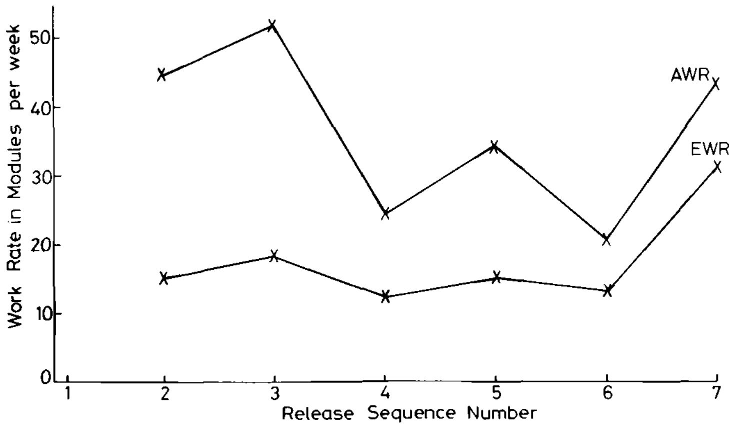


Fig. 10 Actual Work Rate (AWR) and Effective Work Rate (EWR) for Kernel releases subsequent to KDCO2

work rate revealed any significant relationships with system size or release number. For both Director and Kernel, effective work rate was not related to the number of modules handled or the number of module versions handled, neither was it related to inter-release interval. However, actual work rate showed a significant correlation with number of modules handled (0·77 for Director and 0·84 for Kernel) and for Director also showed a significant correlation with the number of module versions handled (0·71). Like the effective work rate, the actual work rate was not significantly correlated with inter-release interval.

All the correlations observed between the system measurements are shown in Table 5 for Director and Table 6 for Kernel.

Table 5 Correlations observed among the Director measurements *

	RSN	M	MH	MVH	I	FC	RHR	AWR
M	0·94							
MH	−0·49	−0·25						
MVH	−0·41	−0·19	0·98					
I	−0·63	−0·49	0·90	0·92				
FC	−0·69	−0·52	0·95	0·93	0·96			
RHR	−0·40	−0·21	0·96	0·99	0·93	0·92		
AWR	−0·10	0·17	0·77	0·71	0·43	0·59	0·66	
EWR	0·08	0·28	0·28	0·16	−0·13	0·10	0·16	0·80

* Correlations greater than 0·67 are significant at 0·05 probability level
Correlations greater than 0·80 are significant at 0·10 probability level

In order to investigate the complexity of new and amended modules, each release was investigated at the level of its constituent subsystems. A number of subsystems which experienced major software developments during the production of a release

Table 6 Correlations* observed among the Kernel measurements

	RSN	M	MH	MVH	I	FC	RHR	AWR
M	0·93							
MH	−0·78	−0·66						
MVH	−0·87	−0·76	0·97					
I	−0·97	−0·84	0·86	0·90				
FC	−0·87	−0·80	0·98	0·98	0·90			
RHR	−0·95	−0·86	0·88	0·96	0·94	0·94		
AWR	−0·38	−0·36	0·84	0·76	0·45	0·79	0·59	
EWR	0·55	0·43	0·02	−0·15	−0·46	−0·08	−0·38	0·51

* Correlations greater than 0·81 are significant at 0·05 probability level.

Correlations greater than 0·92 are significant at 0·01 probability level.

were selected and analysed further in terms of the handle rates obtained for new and amended modules.

The results of this investigation are summarised in Table 7. There was no evidence of any significant trends either within release or comparing results sequentially across releases. Thus, in terms of handle rate, the effort involved in establishing new modules in VME/B subsystems is essentially the same as the effort involved in establishing code enhancements.

4. Discussion

The theory of program evolution dynamics relies to a great extent on the validity of the five laws proposed by Lehman.[3] Although the results from a limited number of releases on one particular system cannot be said to support or disprove the theory, it is possible to investigate which, if any, of the particular laws, or principles, appear to be of practical use.

Table 7 Handle rate for new and amended modules for Director and Kernel subsystems within release

System identifier	Number of subsystems contributing to averages	Average handle rate for amended modules	Average handle rate for new modules
DOM25	5	1·64	2·15
DOXOH	6	1·84	1·99
DOZ08	3	1·23	1·30
DOZ70	2	1·25	1·21
DO123	7	2·22	2·07
DO132	1	1·11	1·43
DO160	4	1·62	2·79
KDM90	3	2·42	2·82
KDUF1	4	3·71	2·11
KOO68	2	1·78	1·85
KO102	1	1·53	1·92

Part of the first law (of continuing change) can readily be seen to apply to both Director and Kernel systems; by observing the increase in size with increasing age, and the continuing amount of change to which the systems are subject, in terms of the module handle counts and observed work rates. The additional statement in the law that systems become progressively less useful if they are not changed cannot be directly verified, it can only be implied by the slightly circular argument that the systems would not be changed if they were in a useful state.

The second law (of increasing complexity) is difficult to investigate explicitly without some objective measure of system complexity. Originally[1,2] the fraction of modules handled during a release was used as a complexity measure but later[5] it was pointed out that fractional change was likely to be influenced mainly by release interval. The results in this study certainly confirm the relationship between release interval and fractional change. The measure used in this study as an indication of complexity is the ratio of total number of module versions to total number of modules handled, the release handle rate. This measure was chosen, not only because it is plausible to suggest that in a complex system more attempts will be required before a module is suitable for release than in a simple system, but also because handle rates of individual modules have been shown to be related to various program complexity measurements.[7]

Even after identifying a potential complexity measure, it is still not possible to test the law explicitly. However, both the Director releases that had large release handle rates also exhibited some of the problems Belady and Lehman attribute to complex systems. The DOXOH release (release number 3) slipped five months compared with original estimate. The DO123 release (release number 6) required major redesign such that certain facilities originally planned for DO123 were included in later releases; and even so the reduced release was a month late and required three remakes. Anecdotal evidence cannot prove or disprove an hypothesis but it does offer some support.

In an attempt to assess the complexity of VME/B, the handle rates of new and amended modules were compared. Thus, at present it appears to be no more difficult to amend old modules than to write new modules; this implies that VME/B is avoiding the worst implications of continuing change and structural decay.

The third law asserts that system evolution trends exist and can be identified. It may be considered in conjunction with the fourth law. The fourth law is a particularisation of the third law which states that the work rate observed during the evolution of large systems will be statistically invariant. Lehman and his coworkers[2,3,5] demonstrated the invariance of work rate by demonstrating a linear relationship between cumulative number of modules handled and system age. Because there is an *a priori* possibility of observing a relationship between two monotonically increasing measurements like age and cumulative amount of change, the method used in this study was to establish that work rate was independent of system size, modules handled per release and inter-release interval. For the effective work rate (i.e. the average number of modules changed per week per release), the results of the correlation analysis support the hypothesis of independence for both Director and Kernel. For the actual work rate (i.e. the average number of

module versions handled per week per release), the results were more complicated. The AWR showed no significant correlation with inter-release interval or system size for both Director and Kernel systems but did show a significant correlation with the number of modules changed.
Thus, in general, the results of the analysis of VME/B do support the hypothesis of an invariant work rate, implying support for both the third and fourth laws of system evolution theory.

The fifth law states that there is a limit to the amount of change that a system can readily absorb during the development of a new release. Although this law is also a particularisation of the third law, it can only be given anecdotal support by this study. The two Director releases that experienced major problems did correspond to the two releases that experienced the largest fractional change: i.e. more than 50% of modules in each release were changed. It is, therefore, possible to suggest that Director releases should limit themselves to less than 50% change. Lehman[3,5] also suggests that self-regulatory phenomena will be observed in this case, i.e. that releases which exceed the limit will be followed by small 'tidy-up' releases. The releases subsequent to Director releases 3 and 6 were small in terms of fractional change but this result is complicated by the fact that the inter-release interval was also small.

From the above discussion it is clear that historical records of the development of the Director and Kernel systems of VME/B demonstrate some of the trends and invariance predicted by the theory of system evolution. It is, therefore, useful to consider the implications of the theory to VME/B.

Belady and Lehman[2] observed a break-down in the development of a large operating system, which they attributed to excess complexity. They observed a fall in growth rate and work rate, an excessive increase in fractional change, and an unplanned increase in inter-release interval prior to the complete break-down in development. The results observed for VME/B base code do not exhibit any such severe problems and, therefore, suggest that VME/B is developing in a controlled manner and keeping potential complexity problems in check.

The particular trends in the development of VME/B must be considered in the context of management decisions. It is the stated policy of management to keep to regular, small releases; this policy has been a management goal since the 5X36 release and has been established for the last two releases of Director and Kernel. The policy of small inter-release intervals when coupled with an essentially invariant work rate would be expected to restrict the fractional change per release. This decrease in fractional change was observed for the last two Director releases and the last but one Kernel release. Nonetheless, some unexpected results have occurred. The last but one Director release suffered an extremely low work rate; this may be attributable to the fact that it was primarily a bug clearance rather than a facility release. Also, the last Kernel release exhibited a dramatic increase in effective work rate which resulted in a higher fractional change than would have been expected. There is no obvious explanation for the Kernel result in terms of changes in working methods or staffing levels.

The results for the latest Director and Kernel releases may reflect the need for a

period of adjustment when large systems adjust to new working conditions. However, the Kernel result does highlight a potential danger when relying on small inter-release intervals, if the committed release content exceeds the expected capacity for change. The results observed for the release handle rate indicate that the Kernel system has absorbed the increase in work rate without an increase in complexity, but it will be important to check that the next Kernel release has not been adversely affected.
The current state of VME/B having been established, it is also necessary to consider the implications of system evolution dynamics for future development. The current strategy is to continue development of VME/B on the basis of small release intervals with the flexibility of allowing certain releases to be primarily bug clearance or 'tidy-up' releases. This management policy should both prevent the build up of excessive system change between releases and permit the containment of structural decay and complexity.

The important constraint on this policy is the invariance of work rate. This invariance implies that the amount of change that can be implemented between two short release-interval releases is strictly limited. It is, therefore, important that the content of successive releases be planned in the light of this constraint.

Finally, the exercise of monitoring the system evolution of VME/B will be continued in the hope that any *potential* problems will be identified by the system measurements and subject to correction before they develop into *actual* problems.

Acknowledgments

I should like to thank Ms V. J. Newman for developing the programs required to automate the comparison of successive system releases.

Thanks are also due to Mrs C. Lodge and Mrs R. A. Fry for their help in the preparation of the manuscript.

References

1 BELADY, L.A. and LEHMAN, M.M.: Programming system dynamics or the meta-dynamics of systems in maintenance and growth, *IBM Research Report RC 3546*, 1971.
2 BELADY, L.A. and LEHMAN, M.M.: The evolution dynamics of large programs, *IBM Research Report RC 5615*, 1975.
3 LEHMAN, M.M.: The software engineering environment, Infotech State of the Art Conference 'Beyond structured programming', 1978.
4 LEHMAN, M.M.: Laws of program evolution – rules and tools for programming management, Infotech State of the Art Conference 'Why software projects fail', 1978.
5 LEHMAN, M.M. and PARR, F.N.: Program evolution and its impact on software engineering, Proc. 2nd Int. Conf. Software Engineering, 1976.
6 BROOKS,F.P.: The mythical man month, Addison-Wesley, Reading, Mass., 1975.
7 KITCHENHAM, B.A.: Measures of programming complexity, *ICL Tech. J.*, 1981, **2**, 298-316.

Software aspects of the Exeter Community Health Services Computer Project

The Exeter Project Team
Edited by
D.J. Clarke, Chief Programmer, and
J. Sparrow, Director

Royal Devon & Exeter Hospital, Exeter, Devon, UK

Abstract

The Exeter Project is a full scale application of real-time computer techniques to the provision, co-ordination and management of health services for a population of about 300,000 patients in the Exeter District of South West England. Both Primary Care and Hospital services are involved, a particular feature being the use of Visual Display Units by general practitioners in their surgeries and by nurses on their hospital wards.

This paper describes the structure and main function of the software systems which have been developed for the project using an ICL 1904A. Applications have been running progressively on this machine since January 1974 and are now being transferred to ICL ME29 and DEC PDP11 equipment.

1 Outline of the Exeter Project

1.1 History and Aims

The Exeter Community Health Services Computer Project was one of a number of experimental projects initially supported by the Department of Health and Social Security (DHSS). It was set up in 1969 following feasibility and preliminary studies which had demonstrated the practicality of using real-time techniques, with Visual Display Units (VDUs), to assist general practitioners in the surgery, and the further possibility of joining medical data from a number of sources (GP, hospital, local authority services) and making these available where required.

The objectives of the Exeter Project were (and still are):

(i) better patient care
(ii) increased clinical and administrative efficiency
and (iii) improved facilities for management and research.

The project firmly believes that the patient should be the centre of any well directed system of medical care. Although several different agencies may be respon-

sible for different activities, there is an obvious need, both in the clinical care of the individual and in the planning and management of health services generally, for information about patients to be brought together and considered as a coherent entity. Thus, in addition to the direct improvement of particular services, one of the main functions of the Project has been to set up a community-based medical information system.

At the Exeter Project the heart of the system is the file of patient records – the database. The database designed by the project has been in use since early in 1974, and is directly accessible on-line to two health centres (one rural, one urban), two specialised hospitals (orthopaedic and eyes), and one District General Hospital organised in two separate locations. In addition three other urban health centres and five small hospitals access and amend the database via terminals located in the general hospital.

The first patient records (for the rural health centre) were added to the file in January 1974. Prior to this the patient medical records had been summarised, and stored on an interim file. With the availability of a real-time system this interim file, then containing some 10,000 patient records, was transferred to the database using conventional batch processing methods. In the following August the orthopaedic hospital was accepted into the system, followed rapidly by the general hospital and the first of the urban health centres. The remaining health centres started registering patients in October 1976. Since the initial registration of health centre patients, all additions to the database have been made using VDUs. At health centres it is the normal practice for a team of 'paramedical summarisers' to assist GPs with the task of converting patient case note folders, and for clerical staff to register the patients on the database using VDUs. The registration of all patients at health centres takes place intensively over a relatively short period of time. At hospitals such an approach would be prohibitive and it has been the practice to register patients only as they come into contact with the hospital, i.e. through outpatient clinics, admissions from waiting list, and accident emergency admissions. Hospital registrations are therefore gradual, and growth is relatively constant. When both hospitals have been fully registered the database will contain records for some 300,000 patients. At the beginning of 1982 it had records for 260,000 patients, growing at the rate of 2,000 per month.

1.2 Applications

The following is a summary of the main applications implemented. They all involve the use of VDU and hard copy terminals by NHS staff at their normal place of work (i.e. surgery, ward, department).

(A) Primary Care:
- Registration
- Summary medical history
- Medication spectrum and repeat prescriptions
- Vaccination and immunisation
- Reassessments due

Control screens (e.g. hypertension)
Referrals
Archive data (e.g. hospital discharge letters)
Surname index
Practice management (leaving list, register addendum etc.)

(B) Medical Records:
Full community index
Registration
Surname index
Waiting list management
Admissions, discharges and transfers
Case note labels
Out-patient diaries

(C) Ward systems:
Creation of nursing orders from standard phrases
Care plans
Reporting
Ward layout
Admission, discharge, transfers to/from wards
Nursing procedures
Trainee nurse allocation
Nursing management information
Nursing dependency
Pharmacy information
Pathology laboratory urgent results

(D) Other systems
Password allocation and control
System measurement
Pathology laboratory – histopathology and cytology
Nursing personnel

1.3 Equipment

The Project uses an ICL 1904A computer with 128 x 1024 words of core storage. Real-time activity is supported by a communications network: an ICL 7903 Communications Processor links remote VDUs (via telephone line and modems) and termiprinters (locally or remotely connected typewriter terminals capable of operating at 30 characters per second) to the central processor. Modems allow operation of remote VDUs at 4800 baud, and remote termiprinters at 300 baud. Sites local to the computer, i.e. the hospitals, use VDUs connected to the central processor via a Cluster Control Unit, and operate via direct cables at approximately 100,000 characters per second. The terminal network extending to four hospitals and two health centres consists of 51 VDUs connected locally, 20 VDUs connected remotely and 27 printers connected to the 7903 both locally and

remotely. The database is held on EDS 60 disc drives (60 million character exchangeable disc units). There are currently 11 drives connected via three disc controllers. The real-time system requires 70k words of core store and a minimum of seven drives, with the database distributed over five disc packs.

In the future the hospital-based system will be transferred to ICL ME29 computers (1 for medical records, 1 for ward based system, and the two interconnected). The primary care system is being re-written for a minicomputer, and a microcomputer implementation is envisaged.

1.4 Experience

In the early days of the project National Health Service users in general were apprehensive about the application of computers to patient care. Over the last decade this attitude has changed completely to one where doctors, nurses, administrative and technical staff within the service have begun to appreciate the additional value that computerisation of medical records brings. The demand for new systems and for improvements to old applications from current and potential users now outstrips the ability of the computing staff in the NHS to satisfy the demands. This has led to a growth in systems packaged for the NHS.

Initially there was considerable anxiety concerning the confidentiality and security of medical data when held on a computer. These fears have largely disappeared with the advent of terminal interactive computing in the Exeter style. Users of the Exeter system, which in storing GP clinical records touches on the most sensitive data held by the NHS, are now convinced that the computer system is more confidential than the old manual system. The British Medical Association and the Royal College of General Practitioners have also been sufficiently convinced to recommend the Exeter level of confidentiality for any future system developed elsewhere.
National and international visitors are a regular part of the Exeter scene. The integrated patient record nature of the project is widely acclaimed by those from abroad although the NHS itself has been somewhat slow to exploit its undoubted potential. All visitors who see for themselves how GPs, and to a lesser extent nurses, have eliminated the need for keeping manual records extol the benefits of the system.

Locally the system has been valued sufficiently to ensure that all the applications currently implemented will be transferred to new equipment giving it at least a further ten years life extension. The project file structure conceived and brought into operation before any packaged database system existed stands up very well in comparison.

2 Software overview

2.1 Strategy

Programs to date have been coded predominantly in PLAN, with some batch programs coded in COBOL. The real-time system was developed using the ICL Driver concept, in which communications processing and disc transfers are processed by

ICL supplied routines. The user is left to code only the processing of the incoming message, and of constructing the reply. This processing is carried out by *beads*, self-contained segments which return control to Driver after processing. Information is passed between beads and Driver via a common area called the TAB (Task Administration Block) and Driver action (go to next bead; read a file record; write a file record; output a message etc.) is determined by the content of the TAB. The sequence of beads and Driver routines between the start of processing of an input message and transmitting the reply is termed a *thread*. In a simple system there is only one TAB (single-threading) and the TAB, once allocated to a message, is unavailable for further messages. Thus one message must be processed completely before processing of another message can commence. If the processing requires file activity, then the real-time program is idle during the file transfers. A multi-threading system has a number of TABs and when activity on one TAB is held up awaiting a transfer, processing can continue on one of the other TABS. At any time processing may be in progress for as many messages as there are TABs. At the present time the system can have up to 6 TABs, there being a facility for dynamically changing the number.

2.2 General overview

The Exeter program has been developed within the framework of standard ICL software, particularly Multi-Threading Driver and Communications Manager. The resulting program occupies 70k words of memory, of which about 10k is for overlaid beads. There are over 1000 beads amounting to more than 400k words of instructions. The program normally operates with three threads (transactions processed in parallel) but can be increased to a 6-thread system. The size of the program is graphically illustrated by Fig. 1.

PROGRAM

1000 + BEADS
50k + WORDS CORE RESIDENT
400k + WORDS OVERLAID
MULTI-THREADING UP TO 6 PARALLEL MESSAGES

FILES/RECORDS

41 DATA FILES
~ 250,000 PATIENT RECORDS CURRENTLY
~ 500,000 PATIENT RECORDS CAPACITY

FILE ACCESSES

200,000 + FILE ACCESSES/DAY TO DATA FILE
(40 + % ARE UPDATES)
100,000 + FILE ACCESSES/DAY TO OVERLAY FILE

Fig. 1

Each thread in the system requires a number of common areas, the most important of which are:-

The Task Administration Block (TAB) which contains all the information necessary to control the processing of a message and is the area used to transfer data between beads and Driver itself. The standard TAB has been expanded by the Project with an additional 13 words.

The Associated Core Area (ACA), Input Output Area (IOA) and Message Area (MA) are addressed from the TAB and may be freely used by beads. The MA is used for incoming and outgoing messages. The ACA is written to a disc file at the end of each message and read at the beginning of the next. This enables data to be transferred between threads.

A common area introduced by the Project is the Terminal Block, one of which is associated with each terminal. This contains details of the terminal, and current user of the terminal. It is addressed from the TAB.

User-written code has been incorporated in Communications Manager to simplify signing-on. The general flow of messages through Communications Manager, Driver and Project beads is shown diagrammatically on the accompanying flowchart and described below.

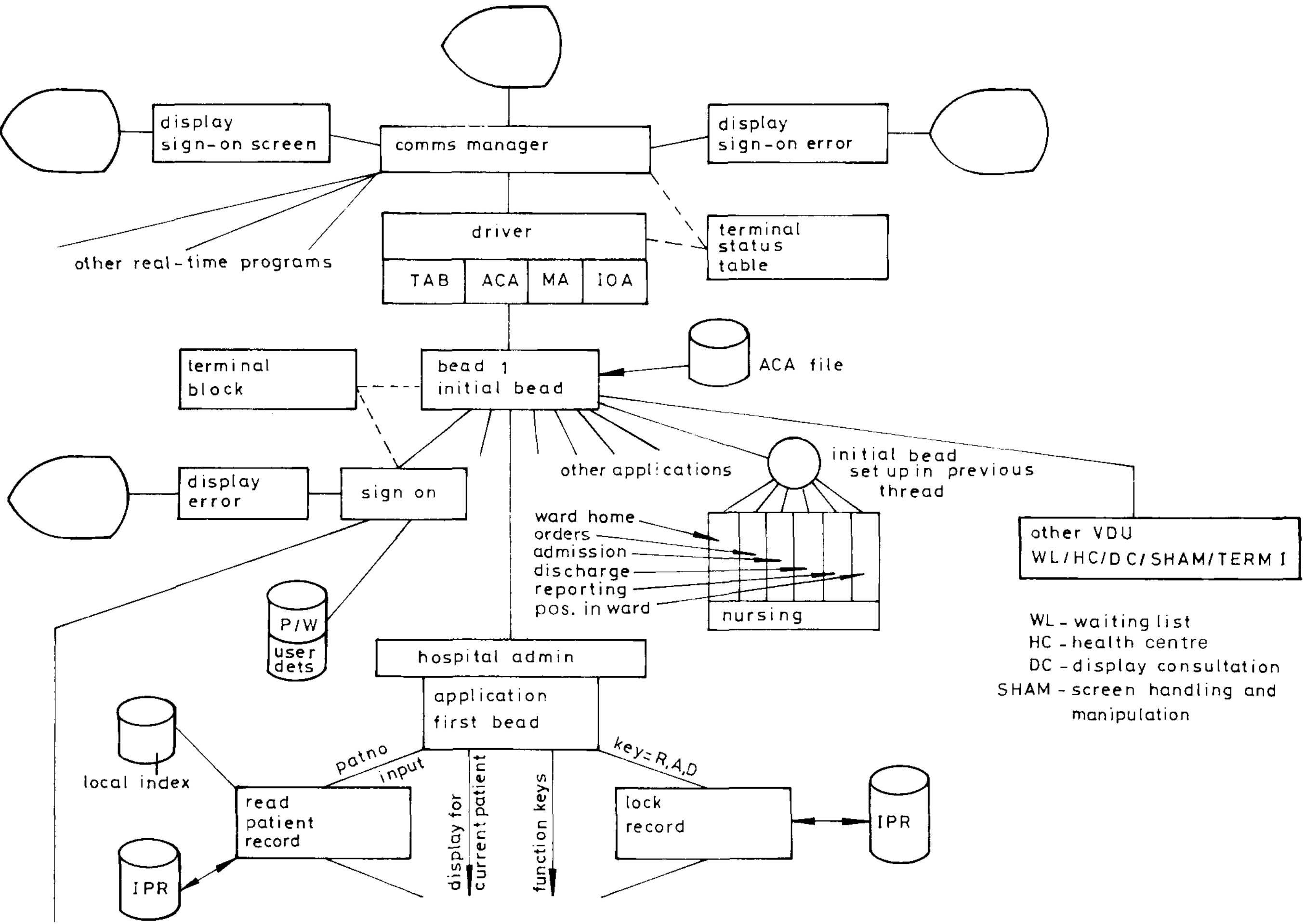
If the input password is all numeric a modification by the Project causes Comms Manager to enter the Current 'Operational' real-time program. If Comms Manager detects an error in the password it will display the appropriate error message.

Once a password has been assigned to an appropriate Driver program, Comms Manager will establish a link to route subsequent messages from that terminal to the same program. When Driver decides that an input message is due for processing it allocates a TAB, ACA, MA and IOA and places the message in the MA. Control is passed to Bead 1.

Bead 1 uses the Project TERMINAL BLOCK to deduce if 'this' terminal is currently signed on. If not, sign-on procedures are entered which, if the password is acceptable, exit to the bead given in the USER DETAILS file after having initialised the ACA with details from this file. If the terminal is already signed on the previous applications thread will have set the appropriate re-entry bead and ACA file bucket number in the Terminal Block. Bead 1 will read in the ACA and exit to the re-entry bead.

At the end of the thread the applications software will set the re-entry bead in the terminal block, roll out the ACA to the ACA file and pass the output message to Driver in the MA. It will then request the TAB be deallocated. Driver will pass the output message to Comms Manager for transmission to the VDU.

This sequence of events, in a much simplified form, is shown on Fig. 2. This shows only one small part of the system in more than outline detail. It gives an indication of the overall size and complexity of the program.



display sign-on screen
comms manager
display sign-on error
other real-time programs
driver
TAB
ACA
MA
IOA
terminal status table
terminal block
bead 1 initial bead
ACA file
display error
sign on
other applications
initial bead set up in previous thread
ward home
orders
admission
discharge
reporting
pos. in ward
nursing
other VDU WL/HC/DC/SHAM/TERM I
WL - waiting list
HC - health centre
DC - display consultation
SHAM - screen handling and manipulation
P/W user dets
hospital admin
application first bead
local index
patno input
key=R,A,D
read patient record
display for current patient
function keys
lock record
IPR
IPR

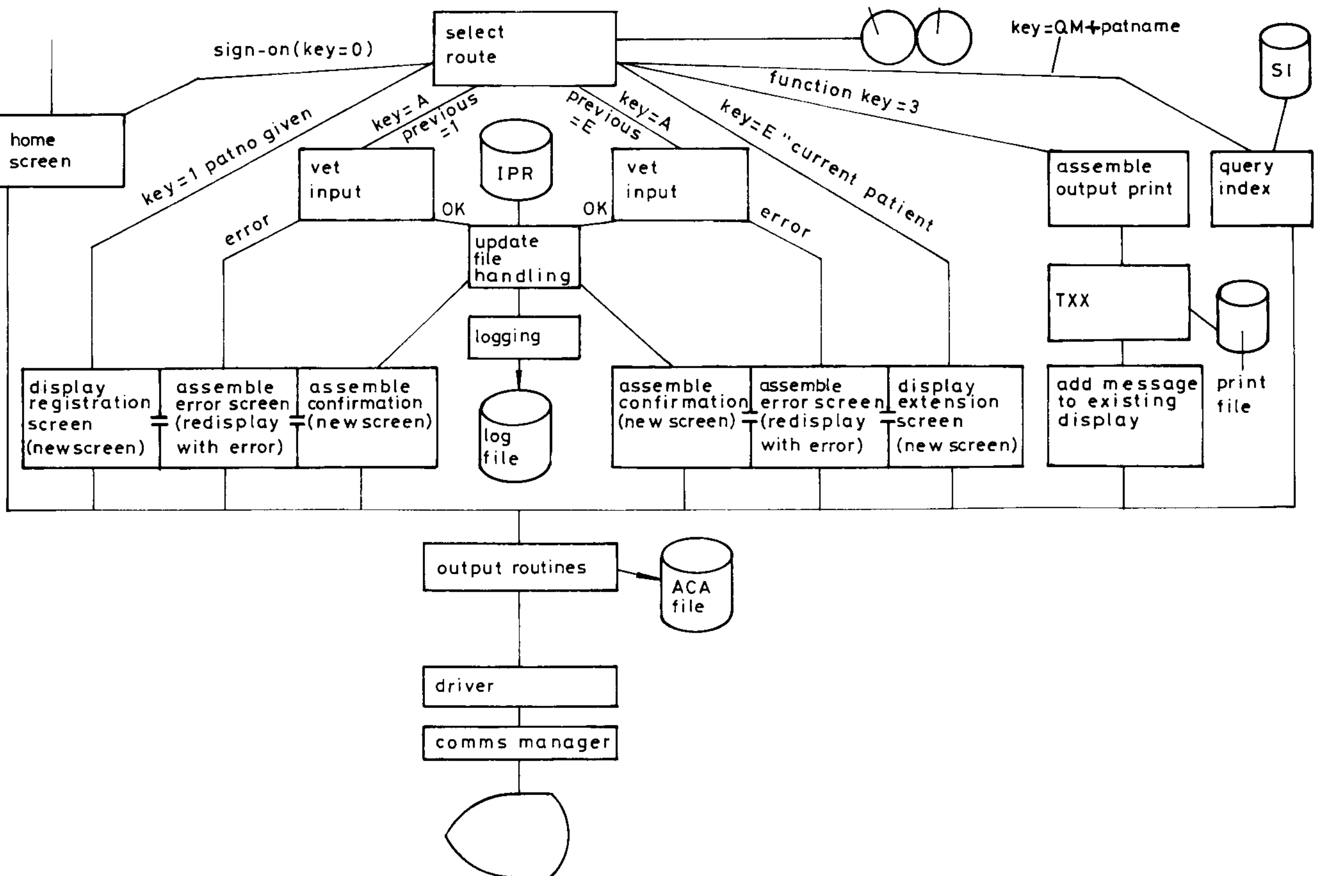


Fig. 2 Schematic of systems

3 Printing Systems

All systems have the need to produce printed reports on demand; here, ICL termiprinters are available for printing. The Project has developed a set of routines known collectively as TXX. This system utilises Print files to store data for printing, or pointers to that data, and to contain a record of each print run from that data.

The processing within TXX falls naturally into two sections:-

(A) Place records on the Print file.

  (*a*) Add new record(s)
  (*b*) Replace existing record(s)

(B) Print from the Print file.

  (*a*) Normal run to print previously unprinted records and end
  (*b*) As (*a*) except that the termiprinter assumes an idle state at the end, allowing further output without the need to sign-on, (Spontaneous Output – SPONTOP)
  (*c*) Repeat last print run
  (*d*) Resume current run after a 'break-in'
  (*e*) Abandon current run.

During a print run the option is given of adding additional information to the print record e.g. date, time, user ID etc.

Fig. 3 illustrates the structure of all print-files.

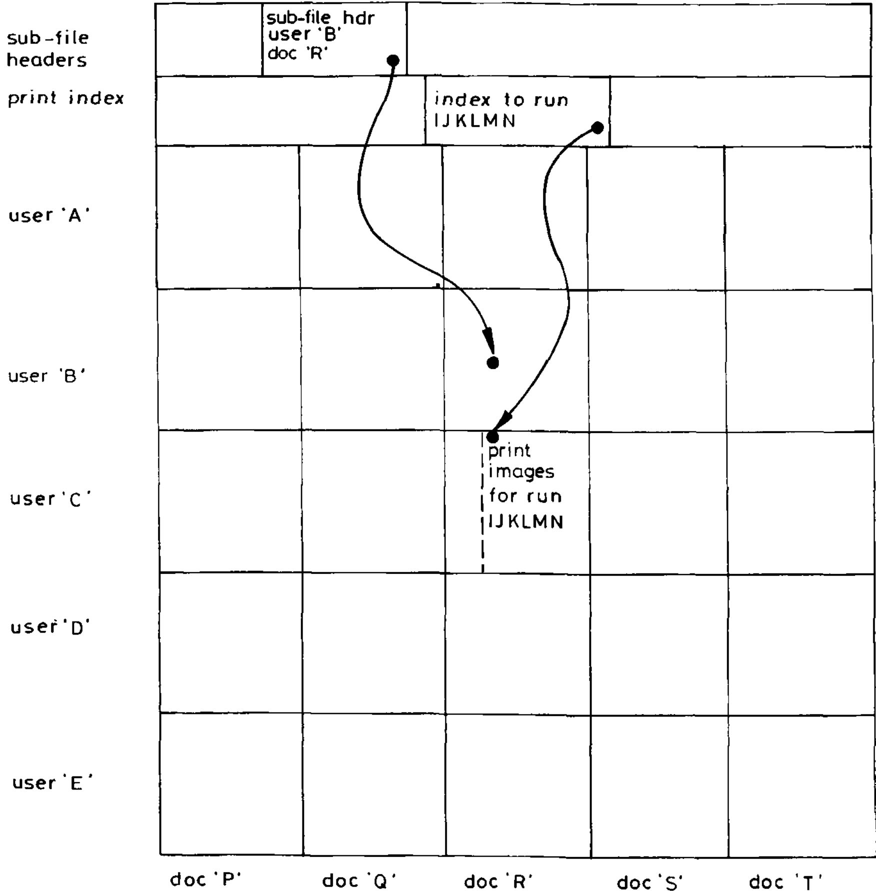
TXX handles the sign-on of a termiprinter, and after sign-on the user will request printing from a particular print sub-file. Any termiprinter will print from any print sub-file but only one termiprinter at a time may access a particular sub-file.

There are two modes of printing run, spontaneous and normal.

– In spontaneous mode TXX will print any outstanding data on the sub-file and then cause the termiprinter to go idle. As soon as more data is added to the sub-file TXX will initiate printing of it.
– In normal mode TXX will print any outstanding data on a sub-file and, having reached the end of data, terminate the print run. Any more data entered into that sub-file will have to be requested afresh by the user.

To maintain activity of the termiprinter each output message must generate a reply to force the continuation of printing. TXX makes use of the 'Answer Back' facility to accomplish this. Each output message is terminated by a control sequence which causes the termiprinter to respond by sending a 'wired-in' message to the computer. In this way printing is accomplished as a pseudo-conversation. Data on the print files is not cleared after printing and is available for repeat prints up to the end of the day. It may be reprinted on the original termi or any other.

Typing 'break-in' at any point during a print run will cause an interrupt, the response to which will be a request for instructions. The run may be restarted by the user merely typing a form number which must be less than or equal to the last form number printed; or it may be abandoned, by giving an 'abandon' request, in which case the run is terminated and the termi will await further instructions. The user may then make a further print request, for any material on the Print file to which he is signed on, and this will be treated in the same way as a print request following a sign-on.



— sub files may be of different sizes
— print images may be added while printing is in progress
— automatic printing as soon as print images are added is possible
— print runs may be repeated

Fig. 3 Print file structure

The termiprinter may be signed-off, which is treated the same as an abandon request except that the termiprinter signs itself off at the end, rather than awaiting intructions. A termiprinter other than the one the user is on may be signed off, but only if it is not currently printing.

Fig. 4 shows a sample of termiprinter output. This would be produced on pre-printed self-adhesive labels.

```
– SIGN ON requested –
  Enter your password & SEND
                                        Password suppressed for confidentiality.
AWAITING INSTRUCTIONS
← PRLNFORTR                                   Parameter requesting print run.

START OF PRINT RUN 812L
PRINTED ON 05.12.80 AT 11.14
ALIGN SATIONERY
THEN TO START PRESS "CTL" AND "A" TOGETHER
```

```
10 00 643
ZZ–HOLES          13.12.42
Raymond Mike      Male   M

10 00 643
ZZ–HOLES          13.12.42
Raymond Mile      Male   M

10 00 643
ZZ–HOLES          13.12.42
Raymond Mike      Male   M

10 00 643
ZZ–HOLES          13.12.42
Raymond Mike      Male   M

10 00 643
ZZ–HOLES          13.12.42
Raymond Mike      Male   M
```

```
ZZ–HOLES             10 00 643       001

Raymond Mike           Mr            812L

20 Pellinore Road    13.12.42        TR N
Beacon Heath
Exeter               Male    M       TRA
Devon
EX1 4QT
          Budleigh 1683 Gp Osm 1277

                Road Roller
A ALEXANDER
The Health Centre
Trial Street
Testown              05.12.80
```

```
PRINTING INTERRUPTED
(IF INSTRUCTIONS DO NOT APPEAR BELOW, PRESS 'CTL" AND "A"
  TOGETHER)

    TYPE NUMBER OF FORM FROM WHICH TO RESTART
OR  TYPE "AB" TO ABANDON THIS RUN
OR  TYPE "OFF" TO SIGN OFF
← OFF                                 Instruction to sign-off termiprinter
RUN ABANDONED
END OF PRINT RUN 812L
002 FORMS PRINTED
THIS TERMIPRINTER IS NOW SIGNED OFF
```

Fig. 4 Example of termiprinter output

4 Error handling

The purposes of error recovery are as follows:-

(i) trapping errors.
(ii) taking corrective action.
(iii) reporting the error as soon as possible to the outside world using the on-line console termiprinter.
(iv) producing as much information as possible concerning the error.
(v) minimising the hold up on the system – during error recovery the program is in single-threading mode.

Bead 0, the error bead in a Driver program, can be invoked in the following ways:-

(*a*) on discovering an error in one of its beads an application may enter it directly.
(*b*) if there is a file handling error a linkage exists to Bead 0.
(*c*) in the case of unanticipated system errors (e.g. transmission failures, unavailabiles etc.) Driver enters Bead O directly.

All information relevant to the error (e.g. registers, address of error, contents of TAB and Terminal Block etc.) is stored and Driver requested to put the program into Exception Mode (i.e. single-threading) thereby the reporting of the error is given the highest priority.

The error bead analyses the exception code associated with each error and determines to which specific error bead control will be passed.

The processing in the specific error beads involves the setting up of appropriate error messages for output to the console termiprinter. Depending upon its gravity each error is assigned an error value in the range from 0-7. Those errors whose error value is less than 2 are not reported at all (e.g. unavailables, availables etc.). All other errors cause a message to be printed on the termi so that immediate action can be taken by operators if necessary.

Only errors with an error value of 3 or greater generate a detailed spooled report. The information is written to a George II disc spooling file in the standard format for printing by George's print module, #OUTA. This method of printing prevents any delay to the real-time program. A George II output file is opened during initialisation in readiness for the first error report generated by the system. After such a report has been written the file is renamed and closed and the next available spooling file is opened by the program.

Control is then passed to the final bead in error recovery, that is, the bead responsible for choosing the bead to which control is to be returned. It is also responsible for certain standard actions and general winding up of error recovery. In certain catastrophic circumstances the program can be suspended by this bead. Also the decision is taken here whether or not to sign off a terminal when a serious fault has occurred.

The structure of Error Handling is shown in Fig. 5.

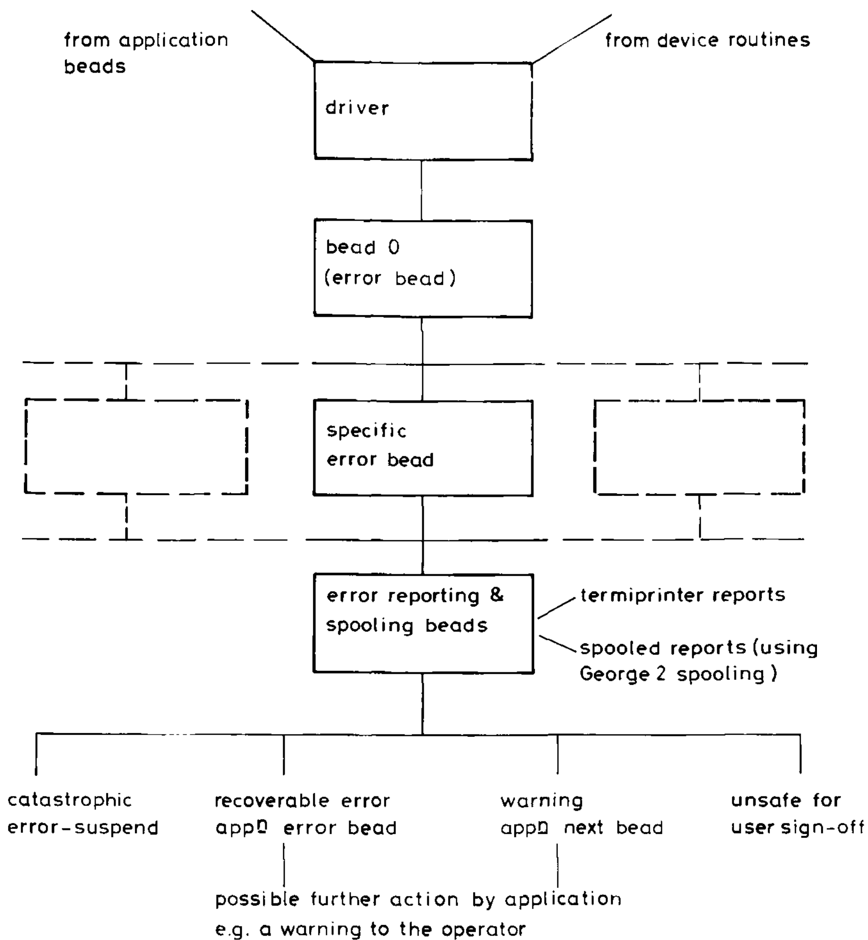


Fig. 5 Error handling

5 File handling

5.1 General

The project's system is very intensive in its use of files. The patient record file is spread over five physical files, contains more than a quarter of a million patient records and has a capacity for up to half a million patient records. There are other files for nursing records, details of health centres, information, logging and many more to a total of 41 separate files. In a typical day there are over 200,000 accesses to these files of which over 40% are updates. This excludes program overlay activity which adds a further 100,000 disc reads. To cope with this range of files and high level of activity the project has developed a sophisticated file handling system.

This system conforms to the requirements of Driver and all accesses are made using standard (DAH3) disc input/output macros using the Direct Response feature, which allows processing to continue while disc transfers are in progress. A common structure has been developed for all files but the special requirements of logging (separate recording of all file updates on a journal), and file locking (reserving a file or part file for the duration of a transaction) have required particular attention and these are dealt with later.

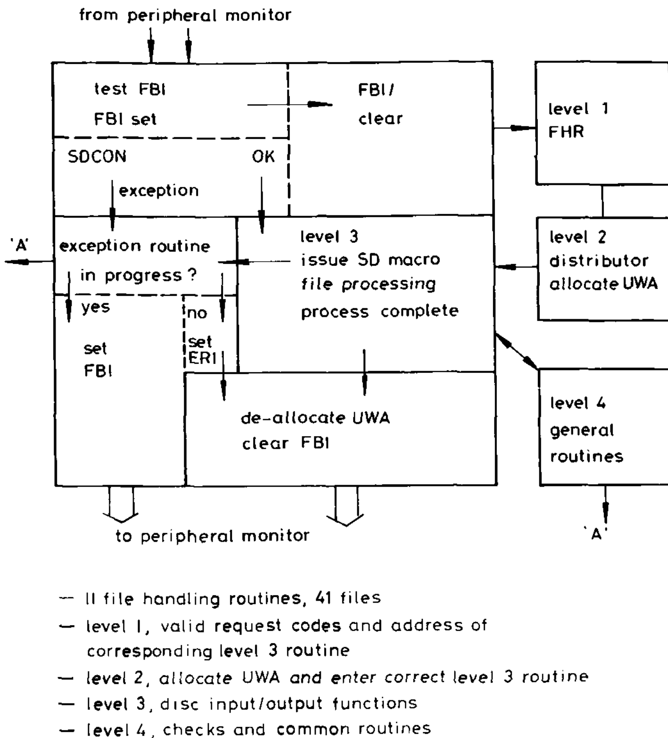


Fig. 6 File handling schematic

The general processing flow through the file handling routines is shown in Fig. 6. The program's 41 Direct Access files are each assigned to a file handling routine. In all, there is a maximum of 11 file handlers. Prior to entry to Driver, after exiting from the bead, a subroutine is called which extracts the driver file number from the TAB and substitutes its appropriate file handling routine number. In fact, one file handler services up to 10 Direct Access files and the necessary queuing for these file handlers is dealt with by Peripheral Monitor on a standard first-in-first-out basis. These queues are controlled by use of the Free/Busy Indicator (FBI).

The following files have their own file handlers:-

(*a*) Update Log
(*b*) 1 handler for each Intermediate Update Log

(*c*) George II Spooling File
(*d*) 1 file handler to queue locking requests.

On entry to file handling from Peripheral Monitor the Free/Busy Indicator is tested. If clear, processing passes via Level 1 to Level 2 where the Unit Work Area for the specified file is allocated and control is passed to the appropriate Level 3 routine.

At this level a disc input/output instruction is issued after which the Exception Routine is entered. If the exception indicates the transfer is still in progress the Free/Busy Indicator is set and control passed to Peripheral Monitor Continuation Routine. For all other exceptions the error reply indicator is set and linkage is set up to proceed to Bead 0 so that a spooled report can be produced for this exception condition.

On subsequent entry to file handling from Peripheral Monitor with the Free/Busy Indicator set the state of the disc transfer is tested; if incomplete, control is returned to Peripheral Monitor, otherwise processing of the request continues.

On exit from the Level 3 routine the file's Unit Work Area is deallocated and the Free/Busy Indicator cleared and finally Peripheral Monitor Continuation Routine is entered in order that Driver can pass control to the next bead.

5.2 Update logging

Every update to major files is separately recorded on a journal file referred to as the Update Log. This contains details of every update, including the 'before update: state of the bucket being written and the 'after update' state of the same bucket. Markers are also placed on the journal to indicate completion of transactions (Thread End Marker (TEM)). Details of updates are held in a core buffer and this is written to disc whenever it changes. The buffer may therefore be written several times before it is filled. To prevent excessive queuing of these disc transfers all intermediate states of the buffer are written to small (single bucket) buffer files with only the contents of the full buffer being written to the journal itself. The management of these intermediate files and of the use of the buffer presents special problems and difficulties.

A schematic of the processing is shown in Fig. 7. The logging routine issues a read macro to transfer the 'before' state of the bucket being logged direct from the disc to the logging buffer. The 'after' state is obtained from the thread's buffer (i.e. MA, IOA or ACA) defined on the TAB request area.

File Handling for the intermediate files (IULs) is entered to record the 'before' and 'after' state. This IUL routine first checks that there is available space in the buffer for the 'before' and 'after' states; if not, the Master Update Log File Handling Routine (FHR) is entered to write away and then clear the buffer.

As file accesses are multi-threaded, processing is interrupted after issuing each disc input/output instruction. Similarly, if processing cannot proceed on a particular

path it can be 'interrupted' to continue on some other task. This is achieved in FHRs by setting the appropriate Free/Busy Indicator and passing control to Peripheral Monitor Continuation routine.

Within logging the interrupt point is determined by an indicator which is always set before entering Peripheral Monitor.

If the buffer is already being written to the Master Update Log then no more can be added to the buffer until that transfer is complete and the buffer cleared.

Whenever a read is initiated a count of reads is incremented. Owing to the physical discontinuities in processing it is possible for the Master Update Log File Handling to be entered while 'before' states are still being read into the buffer. Naturally the logging buffer cannot be written until these reads are complete. After writing Master Update Log the buffer is cleared.

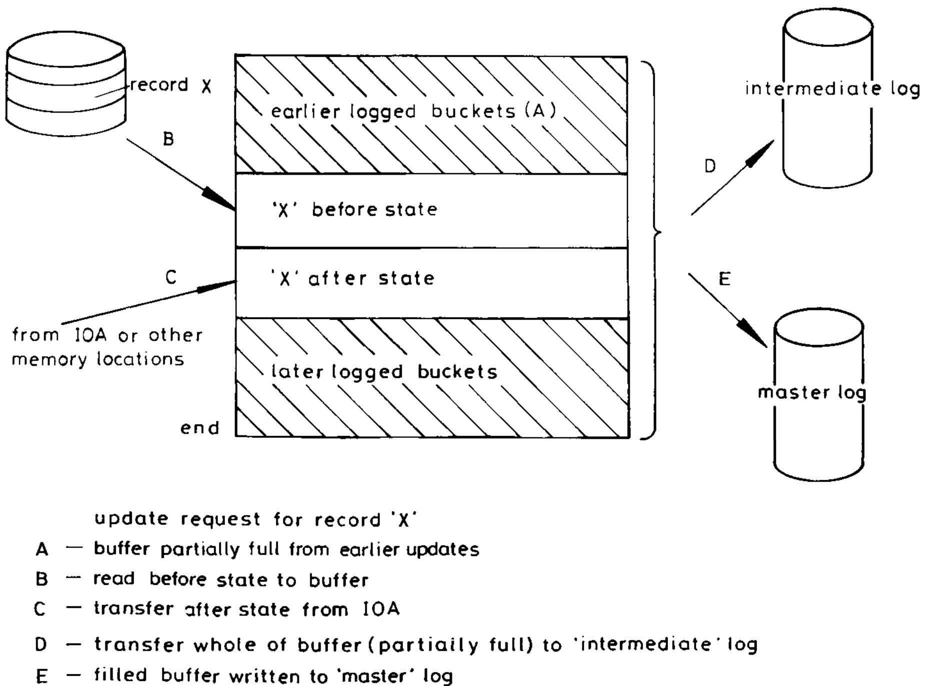


Fig. 7 Update logging

5.3 File locking

As the Project's system allows users to update files, and as requests from several users can be processed in parallel, there is a need for a mechanism to prevent updates of one user counteracting those of another should both be trying to change the same data item (See Fig. 8). This is implemented by only allowing an update if the message requesting the update has reserved the file, or part of file, being

updated for its own use. Details of all locked areas are maintained in a common area, the locking table.

All threads wishing to update data files enter the file locking routine with a request to lock the required files or part files. The threads are able to specify whether the locking request is to be queued if the file is already locked by another thread. At the end of each thread a routine is entered to unlock the files locked by that thread.

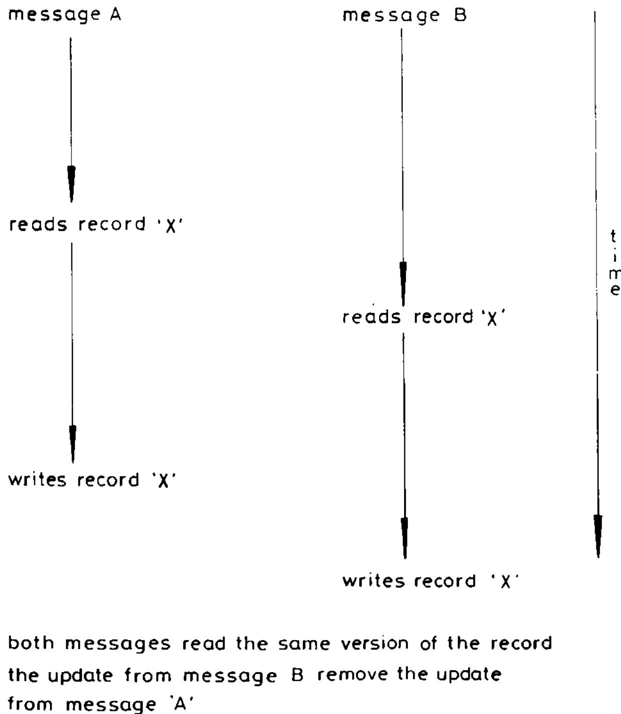


Fig. 8 The multiple update problem

The locking routine, having vetted the parameters giving the files to be locked and having checked that no other thread has any locks on this file, sets up in the locking table the file number and bucket ranges given by the parameters and exits back to the calling routine.

To cope with queuing there is a file handling routine for locking which is entered from Driver in the normal way via its file handling queue. This routine checks to see if the thread which has the locks has unlocked the files yet and if so clears the Free/Busy Indicator to tell Driver that the queued thread can now continue and perform its locks.

The file locking system is shown schematically in Fig. 9.

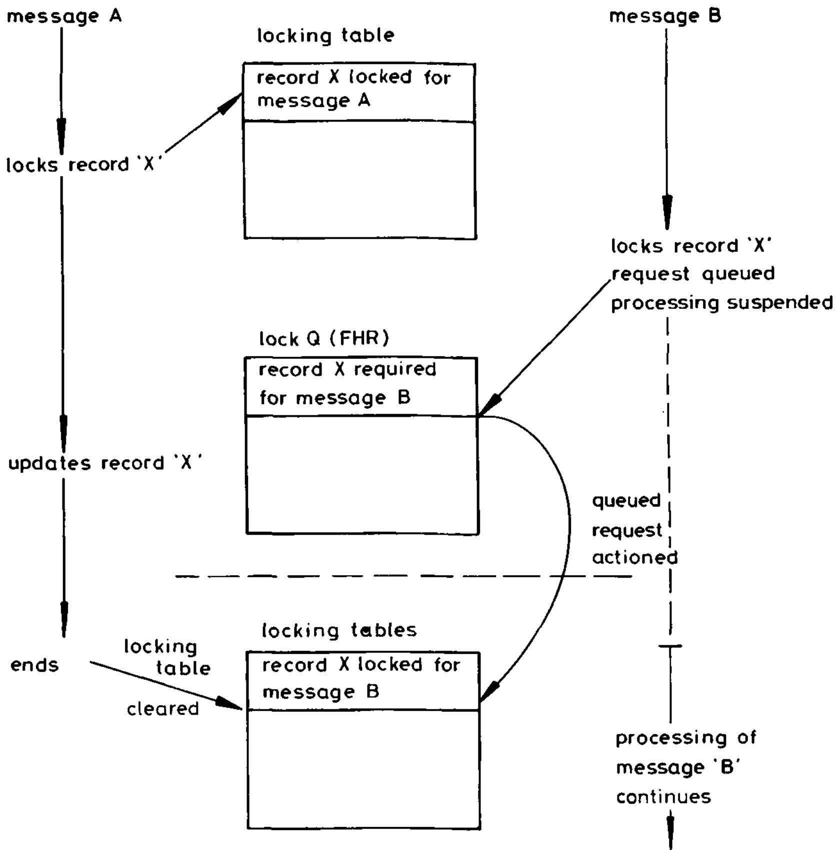


Fig. 9 File locking

6 Database Security

6.1 General

The database maintained by the computer is a vital source of information to many areas of hospital and health centre activity. A large quantity of the data is not readily available in any other form. For some data, particularly medical notes, there is no paper copy of the input. In such circumstances it is a mandatory requirement that no data be lost because of system malfunction, either hardware or software. Further, the service is offered on as close as possible to a 24 hour day basis and any breaks should be kept to a minimum.

To achieve the required level of data security and system availability a number of safeguards and recovery procedures are operated. All rely on the Update Log which has already been described.

All files are dumped daily. If a disc or file becomes unavailable during the day the copy may be restored and brought to its current state by re-applying all updates to it as recorded on the Update Log.

In the event of a system break rapid re-establishment of the service is possible. This requires that all updates outstanding and incomplete at the time of the break are removed. The *stripback* process accomplishes this. File restoring and Stripback are described more fully in the following paragraphs and accompanying diagrams (Fig. 10).

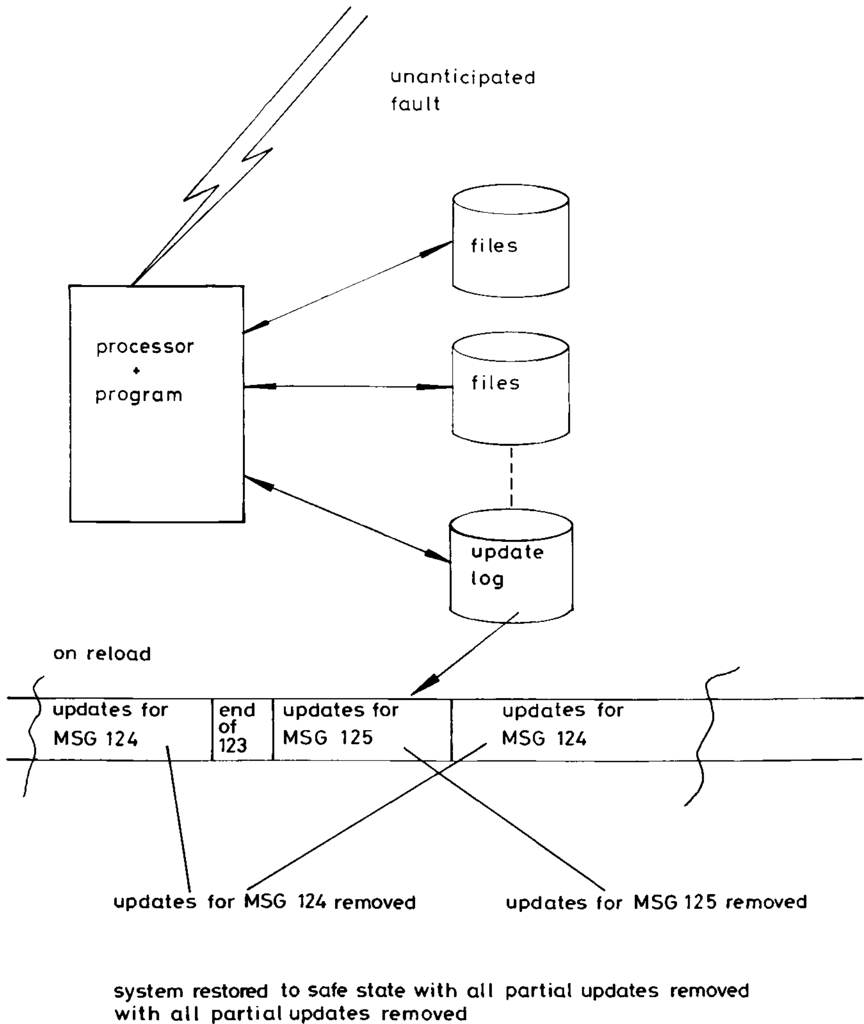


Fig. 10 Stripback

6.2 *Restoring files thread*

Before the start of the real-time day, the complete database is dumped to the copy discs and if, for example, a head crash destroys the data on a disc, its copy is put on-line and the above thread, using the Update Log, restores the files of that disc. This sub-thread is part of the restart path in the initialisation thread.

The 'before' state of the update on the log is compared with the corresponding bucket from the file. If they match (i.e. we have the correct dump on-line) the 'after' state from the log is written to the file. Then the count of updates for this file is decremented and the process continues until that file is completely restored (i.e. count of updates for this file = zero). This process is repeated for all the files affected by the break.

6.3 *Stripback*

Stripback is entered in the restart path of the real-time system. During its processing it can determine which threads if any were incomplete at the time of the break. Such threads may have been updating files at the time the system went down, leading to inconsistencies in the data. Therefore these files have to be returned to an unupdated state.

The bucket of the file being updated is recorded on the Update Log in both its 'before' and its 'after' state. Both buckets on the log are preceded by a bucket descriptor giving details relevant to the update, e.g. file and bucket numbers, message number, update type etc. Certain of these fields are used by Stripback to ensure that the correct buckets are being accessed.

At the end of the updating thread a Thread End Marker (TEM) is written to the log, to signify that all updating requests have finished. Stripback reads back through the log from the last bucket written, comparing and unupdating until a TEM is encountered.

Buckets that have been accessed by Stripback on the log will have a marker set within the six word Envelope Descriptor which precedes the bucket descriptor on each logged bucket. Records within the Update Log bucket that have been used in stripping back will have their record type changed. This is to ensure that the log remains compatible with overnight batch programs which use the Update Log.

7 System functions

7.1 *General*

A number of enhancements have been added to the Driver system to increase its utility, to give greater control over it, and to provide analysis of activity. The functions provided are:-

(i) Long duration tasks
  A system that allows housekeeping tasks (e.g. file integrity checking, and

timing-out interactive terminals) to proceed under the control of Driver but without interfering with message processing activity.

(ii) Add/Delete TABs
Allows TABs (and their associated areas) to be added to or deleted from the program. The number of TABs in the system limits the number of messages that can be processed in parallel. The maximum number of TABs in the project's program is six.

(iii) Operator threads
Provide an interface to the operators who may set closedown time, broadcast messages to all terminals and exercise other controls on the system.

(iv) System measurement
Details of each bead, message, and file access are kept on a magnetic tape file – the System Measurement file.

(v) Testing aids
This system allows a user to view memory contents. With restricted passwords and in test programs memory contents can be altered.

Each of these systems is described in greater detail in the following paragraphs.

7.2 *Long-duration tasks (LDTs)*

Long duration tasks are threads which are not attached to a physical terminal and run in the background whenever activity is low enough on terminal-based threads. At the end of each terminal-based thread a decision is taken at first to see if the message rate is low enough to run LDTs and secondly which LDT requires running. It is necessary that the program is never suspended; if it were, the LDT activity would be dependent on terminal messages. To prevent suspension the LDT software issues a disc read (to the ACA file) whenever there is no other activity taking place. The LDT software keeps a control area consisting of chained records (one per LDT) with details about each task (start time, interval between runs, thread identifier etc.).

The task records are initially set up on file by a batch program; this file is read in during initialisation by the real-time program and kept in core throughout, being written away every time the records are amended. Should the program crash on reload the records will be in the state at which they were last used. This is shown diagrammatically at Fig. 11.

7.3 *Addition and deletion of tabs*

In order to make best use of core and of the multithreading aspect of the real-time program there is a facility to add and delete TABs using #ELASTIC. This means that at peak times the program can have many TABs and at other times can go on with a few TABs on the real time program, releasing core for development and other batch purposes.

Both addition and deletion are initiated from an operator thread. Addition of a TAB is simple in that it is sufficient to perform a GIVE instruction and, having

obtained the memory required, set up the necessary words to make it acceptable as a TAB with Message Area, Input-Output Area and Additional Core Area. Deletion is slightly more complicated because only the TAB in the highest memory locations can be removed. The operator thread to delete TAB does itself require a TAB. If this is the highest TAB it is deleted, but otherwise the thread is passed between TABs until the required one is obtained and which can then be deleted.

To implement this system an additional monitor routine has been written to control memory management. Requests to allocate or de-allocate TABs are set up as requests to this monitor.

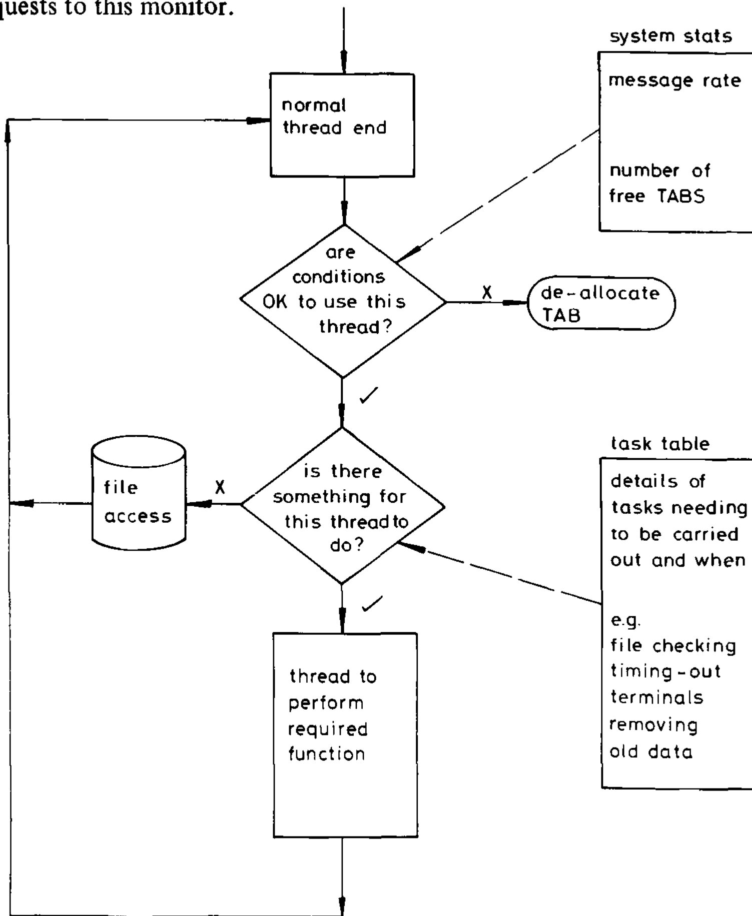


Fig. 11 Long-duration tasks

7.4 *Operator threads*

These threads enable an operator to communicate with and control the system, using a VDU. The available functions are:-

set or unset system closing time,
set or unset a broadcast message,

display/amend details of Long Duration Tasks,
add/delete a TAB
'wrong' or 'right' a terminal
make the surname index file available.

7.5 System measurement

Each bead in the system is entered and left via system measurement routines. These record items such as the bead number, the request code, the file number if a file is being accessed and various other items relating to the system performance. The data is written to a FIND-2 compatible tape file. The file is double buffered and all output is managed from the System Management routines independently of other file handling – file requests are not channelled through Peripheral Monitor.

7.6 Testing aids

The purpose of the testing aids is to allow users access to information in core and to display or alter such information. They are roughly divided into two types: those used mainly to obtain information concerning the real time program, and those used in the development and testing of new systems. However all functions can be used for testing both live and development programs.

The main functions are as follows:-

TAB	– display the contents of the TAB
DISPLAY	– display the contents of a section of memory
FREE	– release a VDU that has become inoperable
WRONG/RIGHT	– make terminals available/unavailable to the system
MESSAGE	– transfer a message from one VDU to another
RSPNS	– performance checks, timed processing and I/O sequences
ALTER	– change contents of a memory location (restricted to few passwords)
TEST AIDS	– invokes various testing procedures, including a trace
SEARCH	– search memory for specified content

8 Review

The foregoing paragraphs have described the real-time system implemented by the Exeter Project and show the complexity that is inherent in a multi-user, multifunction system of this kind. A successful implementation of an undertaking of this magnitude requires a disciplined and orderly approach. The Driver structure lends itself well to a structured design with individual processes being clearly defined and capable of separate development and implementation. The effort required has involved a large team of programmers and analysts and their individual contributions could only be amalgamated into a coherent whole by building on a well defined framework, which Driver provided. Even so it has been necessary to provide a number of functions beyond those provided and as described in these pages they have all been successfully implemented.

The biggest lesson learnt from the experiment is the undoubted need for terminal interactive computer facilities in the field of medical data collection and retrieval.

The need to access the data from multiple places within the District and the need to have up-to-date and accurate information at all times also means that systems must be extremely reliable and operate throughout the 24 hour day, seven days a week. It is encouraging to note that all the current developments in patient care computing, although in the main not going for the fully integrated nature of the Exeter Project, are certainly following the interactive philosophy pioneered in Exeter.

Other papers on various aspects of the Exeter Project are listed in the Bibliography and re-prints of any of these may be obtained from the Project.

Acknowledgments

All members of the Exeter Project have contributed to making it a success. Thanks must go to many of them for providing much of the basic material on which this paper is based and to the Exeter District users of the system who have remained loyal to us from the outset and who have proved the system in day to day operation since 1974.

References

1 Computer Users' Ethical Sub-Committee of the Exeter C.H.S. Computer Project, Submission to the DATA PROTECTION COMMITTEE, October 1976.
2 CLARKE, D.J., FISHER, R.H. and LING, G.: 'Computer-held patient records: the Exeter Project', *Information Privacy*, 1979, **1** (4).
3 CLARKE, D.J.: 'A patient database for the NHS', *Computer Bulletin*, 1978.
4 ROUSE, J.D.: 'Hospital computer registration system with community links', *Medical Record*, 1978, **19** (1).
5 HEAD, A.E.: 'Maintaining the nursing record with the aid of a computer', Conference Proceedings of MEDCOMP 77, pp. 469-483.
6 HEAD, A.E.: 'Nursing systems implications for management and research', November 1977 (Not published).
7 BRADSHAW-SMITH, Dr. J.H.: 'General practice record keeping using a real-time computer', Conference Proceedings of MEDCOMP 77, pp. 303-314.
8 GRUMMITT, A.: Real-time record management in general practice, *Int. J. Bio-Medical Computing*, 1977, (8).
9 Department of Pharmacy, Royal Devon & Exeter Hospital (Wonford), Drug Information System in conjunction with Exeter C.H.S. Computer Project (not published).
10 KUMPEL, Z.: Referral letters — the enclosure of the general practitioner's computerized record, *J. Royal College of General Practitioners*, March 1978, **28**, pp. 163-167.
11 SPARROW, J., FISHER, R.H.: 'Using computers in the NHS – the long term view., March 1976 (Not published).
12 Exeter C.H.S.: Computer project and International Computers Ltd., A Summary of the Exeter Community Health Services Computer Project, 1975, (Not published).
13 LOSHAK, D.: 'Real time for a change of record', *World Medicine*, 1979, **14**, pp. 21-29.
14 ELLIOTT, M.K. and FISHER, R.H.: 'Management audit – the Exeter method', *Nursing Times*, 1979, **75** (22), pp. 89-92.
15 FISHER, R.H.: 'An overall framework for primary care computing', Conference Proceedings, GP-INFO-80.
16 SPARROW, J.: Approximate costs of Primary Care Systems, June 1980. (Not published).
17 SPARROW, J. and KUMPEL, Z.: 'The costs involved in running fully computerized primary care systems for a district', *Medical Informatics* 1980, **5** (3), pp. 181-192.
18 HEAD, A.E. and SAMPSON, N.: 'Learner allocation and statistics system, 9 February 1981. (Not published).

Associative data management system

L.E. Crockford

ICL Office Systems Research Unit, Stevenage, Hertfordshire, UK

Abstract

The needs of database users and their expectations of data management systems are outlined and an associative data management system (ADMS), designed to meet those needs, is introduced. The basic features of ADMS, its use of a content-addressable file store, its flexible message interpreter and its simple model of data are described.

1 Databases – their users and custodians

A database comprises data or facts that have been collected to model some real-world system or organisation. The data are concerned with identifying and qualifying the objects or concepts that have a role in the system and with the relationships that exist between the objects. To be useful as a model the data must represent the true state of the system that is being modelled and, unless the system is static, the database must be either continuously or periodically brought up to date with insertions, deletions and changes.

Those individuals who have collected the facts and those whose duty it is to maintain the database by up-dating have a claim to ownership of the data. Other people are permitted by the owners to examine and make use of the data. It is a convenient generalisation to refer to all these collectively as the 'users'.

Another group owns, or has responsibility for, those devices that store and allow retrieval of the data and thus has custody of the database. Until comparatively recently the custodian group, recognisable as the data-processing department, and user group have been distinct and with sometimes conflicting interests. With the trend to localised use of smaller computers the distinction is often blurred or non-existent as users take custody of their own data.

The database may be used by its owners and others to satisfy simple enquiries about the real-world objects. Another objective may be to enable the users to predict the future behaviour of the system and hence to exercise some control over it. The predictions are made by the processes of induction and deduction from retrieved subsets of the data or from aggregations of some data items. This objective

is served by what is commonly called a management information system. The data collection, updating, enquiry and management information functions are embodied in one or more data management programs provided by the custodians or by a software vendor.

The users have a right to expect that the means provided to access their data are easy to employ and give a 'while-you-wait' service in most circumstances. They expect also some protection against misuses and abuses of their data.

The interests of the custodians are best served by keeping the number of different data management programs to a minimum and by ensuring that those that are supplied to a group of users continue to be useful if and when those users require a change in database form.

2 User expectation

2.1 Users want a simple view of their data

Above all else users desire and need a simple view of their data; a view that is similar to that given by a familiar existing, or replaced, non-computer system. They do not want to be burdened with details of how their data have been mapped into record and file structures or how the files may be linked together if necessary.

A user requesting the selection and retrieval of some of his data should not have to supply such details to the data management program which should, in most circumstances, obtain them by inference. Exceptionally a user's request could be executed in more than one way, giving different results. It is desirable that the user can resolve ambiguities of this kind by supplying with his request, or subsequently, a small amount of additional and meaningful information.

2.2 Users want to express their needs simply

Broadly there are two kinds of user. On the one hand is the user who spends a high proportion of his time at an interactive terminal – updating and accessing data on a regular repetitive basis. At the other extreme is the irregular, infrequent user.

The regular user is likely to be skilled in operating the terminal and to have a good working knowledge of the database – its content and structure. Such users want to be able to state their known needs with the minimum amount of typing effort. To satisfy these users the data management program should support an enquiry-update language with such facilities as form-filling and macro substitution and with useful default actions.

In contrast, the infrequent user is probably unfamiliar with both the database and the terminal that he must operate to gain access to the data. At the same time his needs are likely to be more complex and hence more difficult to express. For his

satisfaction the enquiry language should be rich in facilities and flexible enough to allow expression that comes close to natural language. The program should provide helpful information on request or when the user has made a mistake, should allow the user to examine the structure of the data and should offer a menu choice of alternatives in ambiguous situations.

2.3 Users expect a fast response

Response time is the interval between the sending, from a terminal, of a request to a data management program and that program responding, to the terminal, with some useful information.

In general it is difficult for a user to switch his attention to some other business during the interval which thus represents, to him, lost or unproductive time. Response times are obviously important to the regular user as they directly affect his productivity. The infrequent user whose requirements are often vaguely defined initially may need to issue many requests of an exploratory nature before obtaining something useful. Clearly it is equally important that this user gets fast responses. Response time is the sum of five components; message handling delays, translation and interpretation of request, queueing to access the data, data access and selection and, finally, formatting the response.

Potentially the largest component is that of data access and selection and the data management program will employ indexing techniques to reduce the number of accesses required. Secondary index files have an additional value, particularly to the infrequent user who may wish to 'browse' through them and, since the files have been produced from his data and at his expense, they should be considered as part of the data base.

The data management program should have the ability to recognise requests that will take a long time to service. Such requests should be placed in a separate background queue having first responded to the user with an estimate of the expected delay. He then has the options of aborting that task or allowing it to continue while he proceeds with other matters.

2.4 Users expect protection against misuse

A database is open to misuse in two ways; data may be accessed and information obtained by unauthorised persons and data may be lost or rendered unusable during updating.

Programs that give access to databases via remote terminals should use a combination of personal identity checking techniques to prevent the invasion of privacy by completely unauthorised persons. The data privacy situation may be complicated by the requirement that some persons are authorised to access some but not all of the data. In these circumstances, the program must have the ability to identify a user and present him with a view of the database that omits the data he is not permitted to access.

When a database is being updated, the program must give protection against mechanical or electronic failures by journalising the update transactions and using appropriate error recovery routines.

The program should also protect the database from accidental misuse by its owners or deliberate abuse from others. An obvious safeguard is to permit updating by a few skilled authorised persons only. Each request to erase data should then be vetted to ensure that the removal of the data will not make the database inconsistent. Similarly, the contents of new or changed records should be validated to ensure against duplications and logical conflicts. Questions of privacy and security are discussed in more detail in the paper by Parker in this issue of the *ICL Technical Journal*.[1]

3 General purpose data management

From the point of view of those who supply and maintain programs for the management of data, it is desirable that the number of different programs is small. Ideally there could be a single, general-purpose, program for database updating and retrieval with the backing of a few utility programs for file loading, copying and reorganising and for the creation of index files.

To reach this ideal the general-purpose program must offer a wide range of facilities to the users and must be totally independent of the data to be managed. A high degree of isolation can be achieved by referring to data only via the intermediation of data description tables. Full data independence is obtained by using tables which give not only the physical storage details but describe also the relationships between items of data. Collectively these tables are called a data model – since they contain data about data.

The data model should have sufficient generality to encompass the whole range of database structures and yet be able to offer to the users simple views of their data.

4 Associative data management system

The remainder of this paper is taken up with a description of some of the more important or interesting features of an Associative Data Management System – ADMS.

ADMS has the status of an on-going research project which commenced at Stevenage in 1977. The early aim of the project was simply to demonstrate the effectiveness of the Content Addressable File Store[2] – CAFS – as a component in data management systems. As the work progressed, it became evident that the data-modelling and language interpreting techniques that were being developed were equally as important to the demonstration as CAFS itself. Their joint contributions have made it possible to construct a general-purpose enquiry and update program that goes a long way towards satisfying all the expectations of database users.

The CAFS machine uses purpose-built hardware to achieve high-speed, autonomous and parallel searching through data files. This power, used in conjunction with coarse primary and secondary indexes to files, gives the program the ability to respond quickly to a user's request. The useful contents of secondary index files are made available to user's inspection by including them in the data model.

The hardware File Correlation Unit, described in Reference 3 but available only in enhanced versions of CAFS, is used to facilitate the linking of files in an associative manner, i.e. via common data elements. It will be seen later that this allows users to be completely ignorant of any record and file structuring within their database.

The data model, which will be described later, is basically simple but gives total data independence and supports a number of useful features such as privacy restrictions, update vetting and database navigation by inference.

The user language for ADMS is called AQL – from Associative Query Language. This is partly and provisionally specified in Refernce 4. Of interest here are the structures and algorithms used to interpret messages in AQL.

5 AQL message interpreter

The interpreter for messages in AQL has three basic parts shown schematically in Fig. 1.

The first part divides the incoming message into word units. These are words, as normally understood by the term, or individual non-alphabetic symbols. This task is complicated by the use of macros, i.e. any word that is encountered may have been defined as substituting for a stored sub-message. This necessitates the use of a push-down stack containing pointers into the incoming message and sub-message strings. Whenever a macro requires a parameter this is obtained by a temporary reversion to the original message.

In the second part of the interpreter a classification for the current word is sought.

Any word that was enclosed in quotation marks is a literal value, class 0. Apart from these, and the special case of end-of-message, class 1, the class of a word is sought by looking up the dictionary part of the data model and then, if necessary, a vocabulary table. If the word can be found in the dictionary it is the name of a data element and is in class 2. The vocabulary table contains both class and a sub-classification of all words that are entered in it – see Fig. 2 for an example. A word that cannot be classified is normally regarded as an error although individual users may opt for these words to be classed as literal values, thus avoiding the need for quotation marks.

For efficient and fast classification both dictionary and vocabulary tables are structured as balanced search trees.

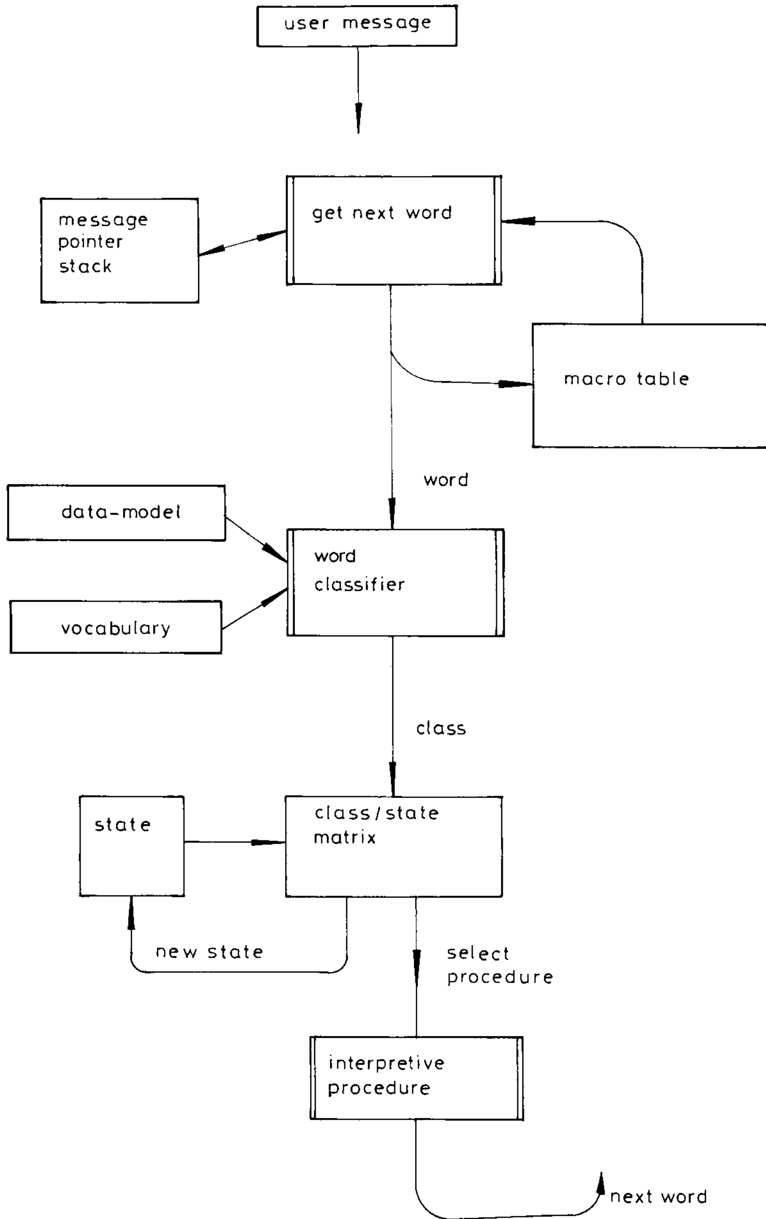


Fig. 1 Message interpretation — schematic

Class	Sub-class	WORD OR SYMBOL AND SYNONYMS
3	0	FOR SUCH WHEN WHENEVER WHERE WITH
	0	COUNT
	1	DISPLAY FETCH GET RETRIEVE
	2	LIST
	3	HOLD
4	4	LOCK
	5	TOTAL
	6	GREATEST LATEST MAXIMUM
	7	EARLIEST SMALLEST MINIMUM
	8	PRINTER
	0	= EQUAL EQUALS IS WAS
	1	< BEFORE EARLIER FEWER LESS SMALLER
	2	> AFTER BIGGER GREATER LATER MORE
	3	UNEQUAL
5	4	(AT) LEAST
	5	(AT) MOST
	6	EXIST EXISTS PRESENT
	7	ABSENT
	8	– (RANGE OF VALUES)
	0	NEITHER NOR NOT
6	1	AND BUT
	2	, OR
	0	(
7	1	)
8	0	AT DOES EITHER FROM PLEASE THAN THAT TO...
	0	ANY SCORE
9	1	BAR
	2	WEIGHT
10	0	ALL
	0	[
11	1	]
12	0	OF VIA
	0	FIRST
	1	LAST
13	2	NEXT
	3	PREVIOUS
14	0	#
15	0	SET
	0	DAY DAYS
16	1	MONTH MONTHS
	2	YEAR YEARS

Fig. 2 Word classification table

The final part of the interpreter uses word-class/state matrices, an example of which is presented in Fig. 3. The matrix is entered for each word, using the class of that word and the current state of the interpreter. The cell that is addressed contains two numbers; the first is used to select one from a set of interpretative procedures – see Fig. 4 – and the second gives the next state to be adopted.

	STATE: 0		1		2		3		4	
WORD CLASS:	P	S	P	S	P	S	P	S	P	S
0	1	0	7	1	9	2	12	1	18	1
1	8	0	8	0	8	0	8	0	0	0
2	3	1	3	1	10	2	3	1	0	0
3	2	1	2	1	2	1	2	1	0	0
4	14	2	14	2	14	2	14	2	0	0
5	4	1	4	1	0	0	4	1	0	0
6	5	1	5	1	2	2	5	1	0	0
7	6	1	6	1	11	2	6	1	0	0
8	2	0	2	1	2	2	2	1	0	0
9	2	3	2	3	0	0	2	3	0	0
10	2	0	2	1	13	2	2	1	0	0
11	0	0	15	1	11	2	15	1	0	0
12	16	1	16	1	16	2	16	1	0	0
13	17	1	17	1	0	0	17	1	0	0
14	0	0	2	4	0	0	0	0	0	0
15	0	0	19	1	0	0	0	0	0	0
16	0	0	20	1	0	0	0	0	0	0

P = PROCEDURE
S = NEXT STATE

Fig. 3 Example of class/state matrix

Example: The receipt of the word LIST in class 4 when the interpreter is in state 0 (its initial state) will invoke the procedure to initialise the formation of a retrieval program and send the interpreter into retriever context – state 2.

Obviously a matrix, as initially set up, can be used to allow context dependent meanings to be attached to words, giving a degree of flexibility to the language. For example, the words AND and OR when encountered in selection context are taken as logical operators but they are ignored in retrieval context. Thus in the message:

> DISPLAY NAME AND ADDRESS WHERE JOB IS CLERK AND AGE MORE THAN 65

the first occurrence of AND is ignored.

Much more flexibility can be obtained by allowing the interpreter procedures to cause an 'escape' to the use of a different matrix or to modify the entries in the matrix in use. So far the potential of the latter has not been explored.

P	PROCEDURE
0	ERROR REPORTING
1	ESCAPE
2	NULL
3	DATA ELEMENT IN PREDICATE
4	RELATIONSHIP
5	OPERATOR
6	PARENTHESES IN PREDICATE
7	SEARCH VALUE
8	END OF ENQUIRY MESSAGE
9	RETRIEVAL FORMAT
10	DATA ELEMENT IN RETRIEVER
11	PARENTHESES IN RETRIEVER
12	WEIGHT OR THRESHOLD
13	UNLIMITED RETRIEVAL
14	RETRIEVER START
15	VALUE SET
16	'VIA' STEERING
17	SET CURRENCY
18	SET SUBSCRIPT
19	SET UNION
20	DATE STEP QUALIFIER

Fig. 4 Interpretative procedures

6 Data model for ADMS

The ADMS data model obtains generality while retaining simplicity by using elementary concepts from set, relation and graph theories.

The database is modelled as a directed graph in which all the sets of data elements appear as nodes and the directed connections or arcs show the recorded relationsships between these sets. A directed graph may be represented pictorially for human assimilation, as in the illustrations to this paper, or alternatively in tabular form by listing the end nodes on each of the arcs. The latter form is used to both define and store the graph.

Supplementary labelling information attached to the nodes serves a number of purposes. The labels are used to divide the graph into a number of *cliques*, each corresponding to a stored set of records, and to give a storage description to each data element within a record set. A further use of labels is to indicate access restrictions on some elements.

Additional information, called weights, on the arcs of the graph may be present to qualify the relation type. The relation may be a correspondence (1:1) or a function (n:1). In the latter case the weight gives lower and upper limits to the value of n. Correspondences and functions normally exist between the sets of attributes of a

real-world entity and its naming or identifying set. Relations between entities are usually of the many-to-many kind ($m:n$). These relations cannot be represented in the ADMS model but there is, fortunately, a simple transform that changes such relations to two functions, $m:1$ and $n:1$.

As an example of this transform, consider a database concerned with sales. Two of the entities are SALESMAN and CUSTOMER and two distinct relations between them are recorded. The first is of the form 'salesman is assigned to customer' and the second 'salesman has obtained order from customer' – see Fig. 5. Both of the

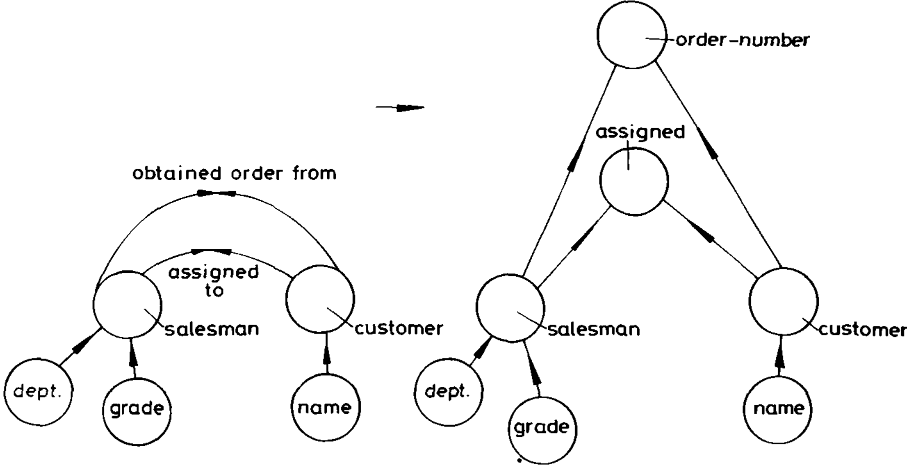


Fig. 5 Transform for many-to-many relations

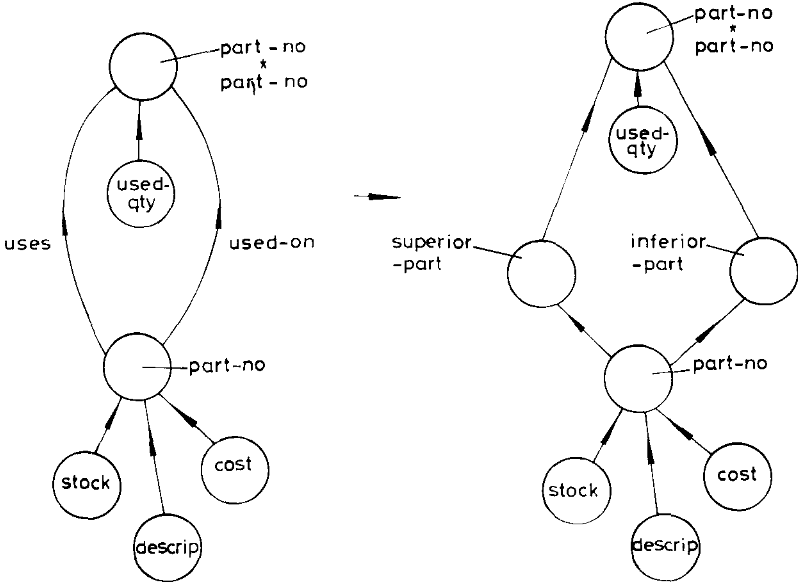


Fig. 6 Transform for a relation on a set

relations are of the many-to-many kind. The first is transformed by the introduction of a compound entity set SALESMAN*CUSTOMER (ASSIGNMENT), or more simply ASSIGNED, each of whose members is an instance of the relation, i.e. the solution set for the relation. A similar treatment of the second relation yields the compound entity set SALESMAN*CUSTOMER (ORDER) but since each instance is documented on a numbered order-form this set may be replaced by an ORDER-NUMBER set.

Two other transforms may be required before the graph model is usable.

The first of these is needed in the situation where there is a relation between an entity set and itself. For example, in a manufacturing database the raw materials, bought-in parts and assemblies are each given a part number and there is a bill-of-materials relation on the PART-NO set as in Fig. 6. The resulting double connection represents an ambiguity which is resolved by inserting a dummy entity set, SUPERIOR-PART and INFERIOR-PART in each arc.

For the implicit navigation algorithm to work successfully, the graph model should have the property that any two or more nodes have at least one common meeting point or upper bound. The graph modelling a job-matching database, shown in Fig. 7, does not have this property - for example, LOCATION and NAME do not have an upper bound in common – until it is transformed by the introduction of the dummy set called SAME-SKILL.

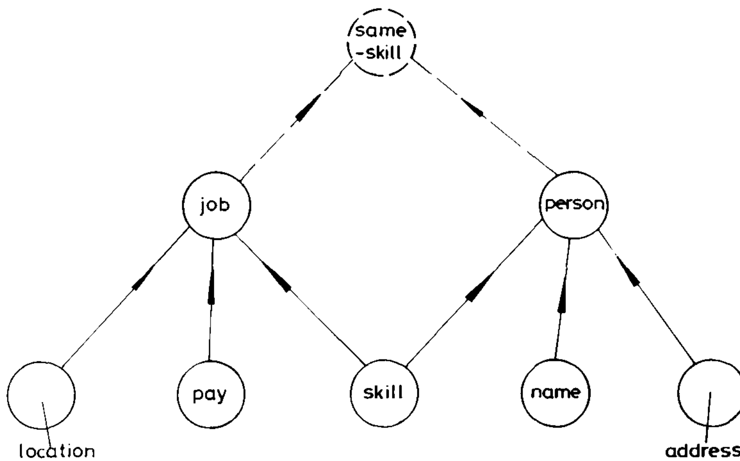


Fig. 7 Transform to provide an upper bound

7 Inferential navigation using the graph model

The graph model allows users to be completely ignorant of the way that their data have been stored; they need only to know the names of the elements of data and to have a broad sense of how they are related.

To illustrate this, a small section of a typical database containing personnel records is modelled in Fig. 8. In this database the static details of staff such as name, sex and date of joining are recorded in one set, while a history of changes to salary is maintained in a set of records containing increase amount, new salary and start and end dates. Similarly, there is a set of records maintaining a history of changes to job title. Users of the database are required to learn the names of data elements that are of interest to them and to be aware that JOB and SALARY are time-dependent.

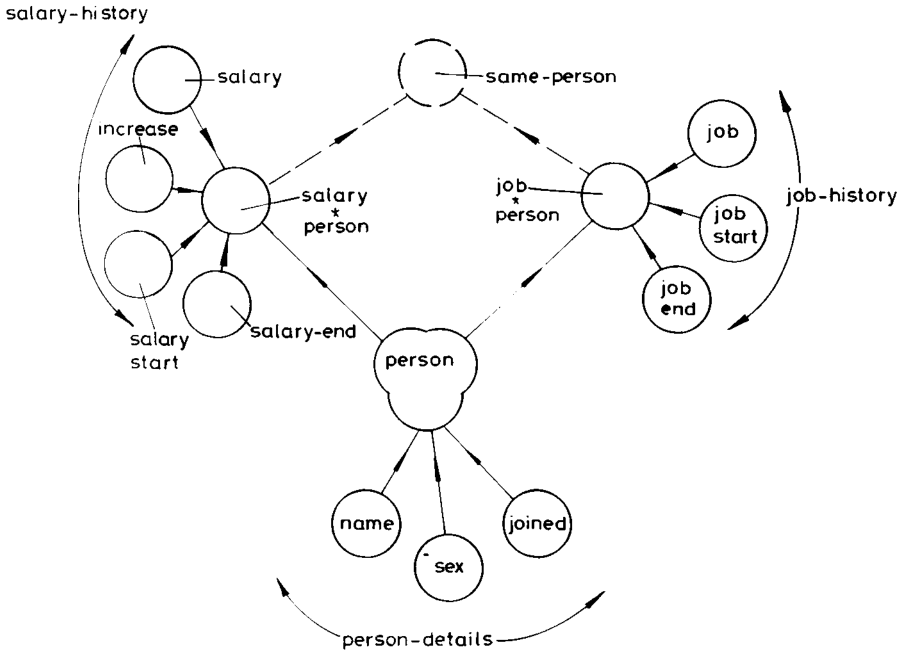


Fig. 8 Simplified model for personnel database

Suppose that a user wishes to discover what was the total salary bill for the company on 1st April, 1981. His requirement would be expressed in AQL as:-

GET TOTAL SALARY AT 01/04/81

The message interpreter would expand this to

GET TOTAL SALARY WHERE SALARY-START IS NOT AFTER 01/04/81
AND SALARY-END IS ABSENT OR NOT BEFORE 01/04/81

by using the macro definition for AT.

Three nodes on the graph have been named; SALARY, SALARY-START and SALARY-END. The navigation algorithm finds that these have a common upper

bound at SALARY*PERSON. Since this node and all those named are present in the clique that forms the SALARY-HISTORY set, it is inferred that a search through this set is necessary and sufficient.

Another user has a more complicated need; to find the current range of salaries for female programmers. In AQL this is:

GET LARGEST AND SMALLEST SALARY NOW WHERE SEX IS F AND JOB NOW IS PROGRAMMER

The word NOW, like AT in the previous example, is a globally defined time-dependent macro and expands to SALARY-END ABSENT at its first occurrence and to JOB-END ABSENT at its second. The set of named nodes contains SALARY, SALARY-END, SEX, JOB and JOB-END with a common meet point at the dummy node SAME-PERSON. Since a search on a single record set is obviously not sufficient, an optimum path through the graph is sought that will visit all the named nodes and terminate at the meeting point. The path is then translated into a chain of searching tasks.

In this example, the first task in the chain is to search the JOB-HISTORY set, identifying the persons currently titled programmer and entering their PERSON identifier into the File Correlation Unit (FCU) of the CAFS machine. Next task is to search the PERSON-DETAILS set looking for records where SEX contains F and where the PERSON number appears in the FCU. The PERSON numbers from records satisfying these criteria are re-entered into the FCU. Finally, the SALARY-HISTORY set is searched and records identified where the PERSON number is in the FCU and the SALARY-END date is missing. From these records the CAFS picks out the maximum and minimum values of SALARY which are then displayed to the user.

There are two situations where a user's request may be ambiguous and he is asked for supplementary information.

Illustrative of the first situation is the AQL request:

LIST ALL CUSTOMER AND NAME FOR DEPT XYZ

in the context of the model in Fig. 5. What is requested is either a list of customers assigned to salesmen in department XYZ or a list of customers who have placed orders with those salesmen. The ambiguity is discovered by the navigation algorithm when it finds the alternative meeting points at ORDER-NUMBER and ASSIGNED. To resolve the ambiguity the user is invited to select, and include in his request, one of the phrases VIA ORDER-NUMBER or VIA ASSIGNED.

The graph of Fig. 6 contains a source of ambiguities in the choice of path offered between PART-NO (and its inferiors) and the uppermost node. Users are asked to remove the choice by using a VIA or OF phrase as in:

GET USED-QTY WHERE DESCRIP OF INFERIOR-PART IS WIDGET

8 Integrity of updates

The graph model may be used to shield off some of the effects of human misuse during updating processes.

Updating is at the record-set level and associated with each set that may be updated is a data element containing unique record numbers. Access to this element is an essential part of any updating transaction and if, therefore, it is indicated in the graph that groups of users are denied such access then users in those groups cannot carry out any of the update functions on the record set concerned.

Fig. 9 shows the skeleton outline of a model for an orders database with three sets of records -- ORDERS, CUSTOMERS AND PRODUCTS. Weights on the arcs of graph give the lower limit for n in each of the $n:1$ relations. These weights show, for instance, that while there must be at least one product for any description, a product can exist even if there are no orders for it.

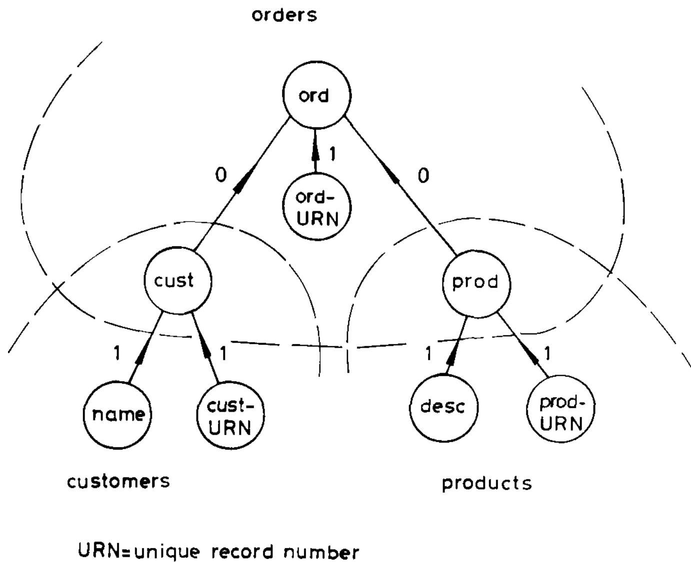


Fig. 9 Skeleton of orders database model

Permitted users can freely insert into the lower sets, i.e. new customer and product records – the system checking for the uniqueness of customer or product number. The insertion of an order, however, will not be allowed if a search of the inferior sets reveals that the order is for a non-existing product or placed by a customer whose details are not recorded.

Conversely, the model indicates that orders may be freely deleted by authorised users but no customer or product record may be removed if a search shows that there are one or more related orders.

9 Conclusions and acknowledgments

A general purpose enquiry program, ADAM, based on that used to demonstrate CAFS and associative data management has been made available to a number of different groups of users[5] who have accepted it with enthusiasm. This is regarded as vindication of the claims for ADMS made earlier. Meanwhile, the research continues with emphasis on the needs and expectations of that large group of users whose data are loosely structured in the form of documents, letters and memoranda.

The author is indebted to the Department of Industry for their funding support of ADMS as an Advanced Computer Technology Project, and to colleagues in the Research Unit and Marketing for their advice, criticism and collaboration.

References

1 PARKER, T.: 'Security in a large general-purpose operating system: ICL's approach in VME 2900', *ICL Tech. J.*, 1982, 3 (1), pp.

2 MALLER, V.A.J.: 'The Content Addressable File Store, CAFS', *ICL Tech. J.*, 1979, 1 (3), pp. 265-279.

3 BABB, E.: 'Implementing a relational database by means of specialised hardware', ACM TODS, 1979, 4(1).

4 CROCKFORD, L.E.: 'AQL – a provisional specification. ICL Office Systems Research Unit', Internal document, November 1981.

5 CARMICHAEL, J.W.S.: 'Personnel on CAFS': 'a case study', *ICL Tech. J.*, 1981, 2(3), pp.244-252.

Evaluating manufacturing testing strategies

M.Small and D.Murray

ICL Mainframe Systems Development Division, West Gorton, Manchester, UK

Abstract

The paper describes a model which has been developed and used in ICL to evaluate the relative performance, in terms of both cost and quality, of alternative strategies for testing in manufacturing. The model is based on an application of basic probability theory to the incidence and repair of faults, the quality of any item with respect to any fault or class of faults being measured by the probability that it is free of that fault or class of faults. The flow process creating a product from components to sub-assemblies to assemblies to final product is studied, also the effects on final quality of the initial qualities of the components, the increasing number involved as the assembly proceeds and the probability of damage by any of the processes including repair of a detected fault. Loading on the test and repair facilities is studied, also total and unit costs. The model has been programmed and run on an ICL 2900 machine.

1 Introduction

This paper arises from the requirement to design manufacturing systems which can achieve a desired level of quality in a product within an acceptable cost. This leads to a need to understand the relationships between the quality of the constituent parts of the product, the processes which produce them and the testing which is applied at various stages. The need to give consideration to these matters is brought about by the rapid increase in complexity of computer systems of a given value, owing to the exploitation of techniques such as the very large-scale integration of electronic circuits, and the potential for more faults which this leads to.

The manufacture of computer systems poses severe testing requirements. Most of the electronic circuits which make up a computer system are manufactured on small slices of silicon known as integrated circuits. This method of making circuits gives rise to significant reductions in power consumption, size and above all cost. However the basic manufacturing processes for these integrated circuits yield few good components. For some circuits a yield of 10% correct devices from the silicon processing stages is normal. On the other hand to achieve a quality level of 99% for a system containing 5000 integrated circuits requires that only about 2 in a million integrated circuits may be faulty at that level. This increase in quality can only be brought about by testing. It must also be achieved in spite of the damage caused by the intervening processes. Therefore the testing strategies adopted are a major aspect of the manufacturing system and hence have an important influence on manufacturing costs.

This paper describes a mathematical model which relates the quality of the product output to the parameters of the processes and testing involved in its manufacture. The model further covers the relationships between quality and the flow rates required at the various stages to sustain the required output of systems. From the flow rates an assessment of the testing capacity required can be made and hence overall testing costs determined. Finally a computer system which incorporates these models is described. The models themselves have been validated in part by observations of existing systems in ICL. The modelling system has been used, within ICL, to evaluate testing strategy proposals and, although the results have met with acceptance amongst other experts, the systems modelled have yet to be fully implemented.

2 Typical products and processes

The major constituent building blocks of ICL products are multilayer printed circuit boards carrying integrated circuits. The printed circuit boards are sandwiches, bonded from mixtures of signal-carrying layers and power-carrying layers, suitably isolated by insulating layers. Interconnections, from one side of a layer to the other, or from one layer to another, are achieved by conventional through-hole plating techniques. Continuity and isolation tests are performed on the layers before bonding takes place, and upon the multilayer board after bonding. Use is made of test jigs comprising many thousands of test probes, known as 'beds of nails', connected to test computers using test data derived from design files contained in computer databases.

Integrated circuits, which have been tested by their manufacturer, are retested using commercial Automatic Test Equipment (ATE), and those which pass this test are assembled onto the multilayer boards. The interconnections are then retested again using the bed-of-nails technique to access the connecting pins to the integrated circuits or other components and the edge fingers of the board, prior to applying power to the board and carrying out a functional test.

These boards are then assembled with other components such as cables and connectors and tested as units. The units are then assembled into products and tested. Finally the products comprising a computer system are brought together and the system as a whole tested.

2.1 Attributes of printed circuit boards

It is pertinent to list some of the many attributes of a printed circuit board which require to be correct for high quality to be obtained:

Printed circuit tracks are continuous and no short circuits exist.
Track widths are to specification and tracks are not resistive.
Insulation thickness is correct.
Decoupling capacitors are correctly mounted and soldered.
Integrated circuits are correctly mounted and soldered.
All integrated circuits function to specification and with safety margins (voltage, temperature, timing).
Physical dimensions are correct so that edge pins locate in sockets.

This list is not exhaustive, but serves to illustrate that a wide range of different

kinds of test together with strict quality control procedures is required to ensure a high quality in the final product.

3 A concept of quality

Quality may be thought of as a measure of the absence of faults in a product. Any complex object, such as a computer system, can be faulty in a very large number of ways, so many ways in fact that some concept is necessary to reduce the problem to manageable proportions. The concept proposed is that all faults which have some similar property be grouped together and the group as a whole, rather than the individual faults, be considered. These groups of faults will be called *fault classes.*

Faults may be grouped into virtually any classification that is considered to be useful, the only important rules are: each fault class must be essentially independent of all other classes and the list of classes should be complete. This latter requirement is far more difficult to achieve and in practice any statement of quality can only be made with respect to some defined list of fault classes.

For the purpose of examining manufacturing systems at least those classes of fault which are inherent in the raw material, or bought-in components, plus those which arise due to the various manufacturing processes must be considered. A base level of quality is set by the inherent material and process fault rates. This level may be improved by the application of tests. Various forms of test may be available, each detecting different classes of faults with different degrees of effectiveness. The final quality, therefore, is a complex function of the fault classes considered, the base quality and the characteristics of the test stages. For the manufacturing system to yield a high quality product the material, processes and tests must be well matched.

3.1 Representation of quality

Quality may be represented numerically as the probability that a product is free from a certain class of fault. This probability can be thought of as the proportion of items which would be fault-free in an infinitely large sample (batch). The observed proportion of fault-free items in any finite batch will show the normal variations due to sampling effects.

The following symbols will be used to represent quality at the input to and the output from a process or test, the output being the derived variable:-

For fault class j:

> Let q_j be the probability of freedom from faults of this class prior to the process or test.
>
> Let q'_j be the probability of freedom from faults of this class among things which have undergone the process or passed the test.

3.1.1 Quality and numbers of faults: As well as knowing about the quality of a product, it is also useful to have information concerning the fault rates. This is because the actual fault rates may be observed during the manufacturing and testing processes and may be used for control purposes; also the repair capacity required will be determined by the number of faults found.

Quality and fault rates are related by a fault distribution. This gives the probabilities that various numbers of faults will occur per class or per product and includes zero faults which is, of course, the quality. The fault distribution may be a set of observed values or be represented by a mathematical model. Two such models are the single fault model in which it is assumed that no more than one fault may occur, and the Poisson model in which it is assumed that there are an infinitely large number of possible faults each of which is equiprobable.

The Poisson model gives the average number of faults as a function of the quality:

$$M_j = \log_e(1/q_j)$$

where M_j is the average numbers of faults of class j.

Neither of the above models is completely satisfactory since faults are not usually equiprobable and although there may be more than one fault there is a finite limit to the number of faults. To deal with this the system described here contains another model which is similar to the binomial model. The assumption made is that a component may have no more than one fault per class. This is useful where components such as integrated circuits are being considered and more than one fault per component is irrelevant. Thus the maximum number of faults per class is limited to the total number of components in the assembly. The average number of faults is given by:

$$M_j = \sum_{i=1}^{k} (1 - q_{ij})$$

where k is the number of components and q_{ij} is the quality of the ith component.

The average number of faults in total (M) over n fault classes is

$$M = \sum_{j=1}^{n} M_j$$

and the probability of no faults:

$$Q = \prod_{j=1}^{n} q_j$$

3.2 Influence of processes on quality

For the purpose of modelling test strategies a manufacturing system may be represented by an implosion tree, see Fig. 1. Basic components are bought in or created by production processes and combined into sub-assemblies by assembly processes. The sub-assemblies and components may be subjected to tests at any point. Thus it is necessary to include the effects of the processes as well as the tests upon quality. As far as quality is concerned there are three effects of a process which must be

modelled: these are the effect of combining items together, the production of new attributes and the introduction of damage.

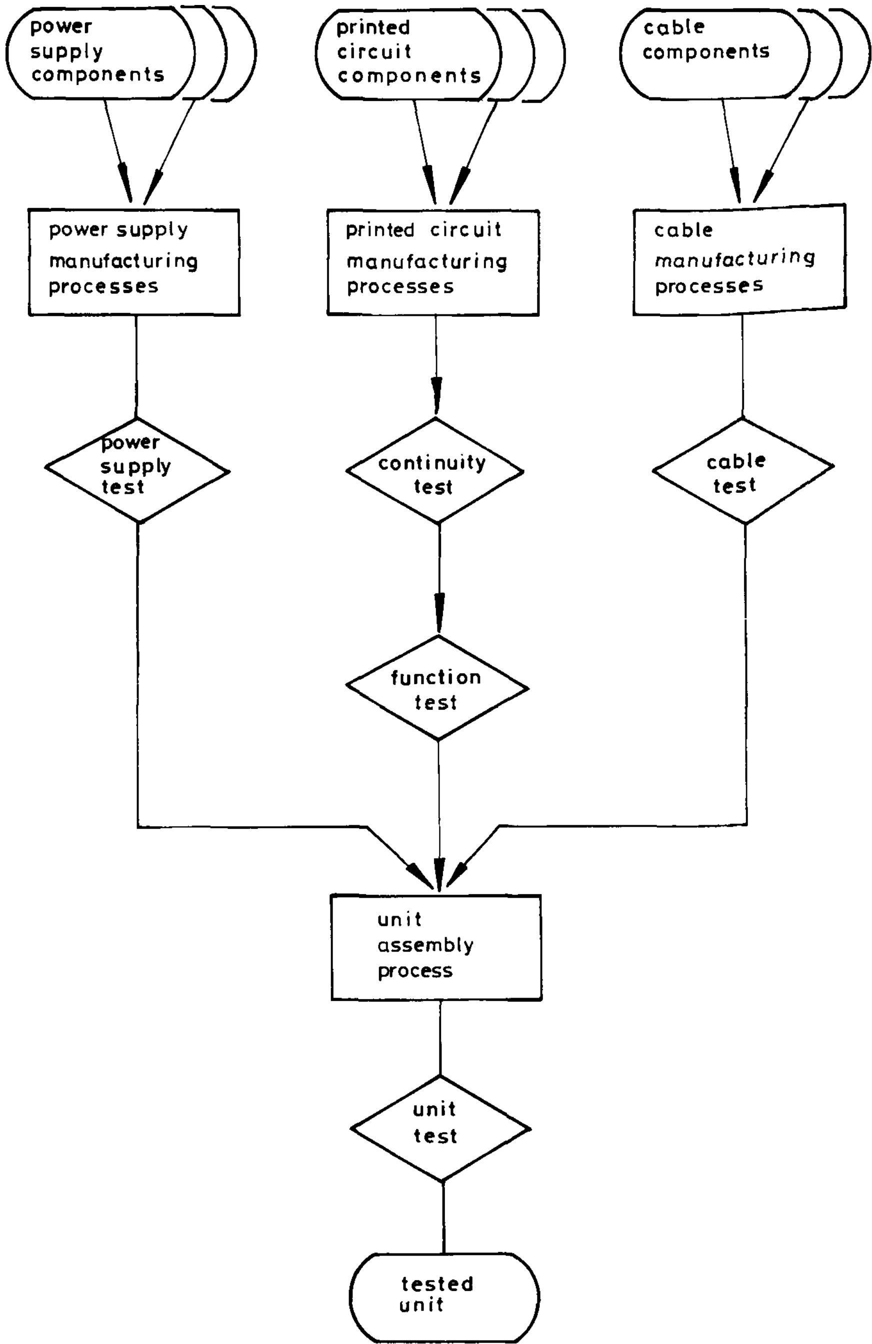


Fig. 1 A manufacturing system

3.2.1 Collection. A process may bring together several items. This can be dealt with as follows: First, where several items have the same fault class the quality of the assembly in respect of that class is the product of the item qualities. Second, the list of fault classes which must be considered for the assembly must be arranged to include all of those applying to the individual items.

Let q'_j be the quality of an assembly of n items in respect of fault class j after assembly but before considering other effects.

Let q_{ij} be the quality of the ith item possessing that fault class:

$$\text{Then } q'_j = \prod_{i=1}^{n} q_{ij}$$

3.2.2 Damage. A process may cause deterioration and hence reduce the quality in respect of some or all of the fault classes. This effect may be described for the process by the probability that faults of each fault class will occur during the process.

Let a_j be the probability that faults of class j may be introduced during the process.

$$\text{Then } q'_j = q_j\,(1-a_j)$$

3.2.3 Production. A production process changes the material input to it. The change may be in the form of the material (moulding etc.) or it may be to create new features (drilling etc.). Both these kinds of change may be considered to create new fault classes. Where the change is small it may be useful to absorb the new classes into existing ones; for example through-hole plating may be absorbed into interconnect quality. Although, strictly speaking, the quality of output from a process depends upon both the quality of the material into the process and the characteristic of the process itself most processes are characterised by their output quality rather than the true process characteristic.

3.3 Testing and its relationship to quality

The quality of items passing a test can be expected to be greater than the quality of those submitted for test providing that the test detects the relevant kinds of faults and is non-destructive. The actual quality of items passing a test depends, for each fault class, upon the quality of items input and the test detection rate for faults of that kind. There are various models which have been described[1,2] relating post-test quality to pre-test quality and test detection. The difference between these models concerns the assumption made regarding the distribution of numbers of faults. Both of the models cited above assume that the test detection rate is the same for all faults and that it is only necessary to detect one of many faults to identify the item as bad. This leads to an increase in the apparent effectiveness of the test when input quality is low and the probability of more than one fault is high. These models are appropriate where items which fail the test are discarded, as are integrated circuits for example.

The system described here contains a model which takes account of the variations in fault rates and test detection rates across the different classes of fault as well as the influence of the repair process. In this model, which is illustrated in Fig. 2, the test is assigned a detection characteristic for each fault class which is the proportion of items faulty in that class which will give rise to a fail result. This characteristic is represented as follows:

Let d_j be the detection rate for class j.

Let p_j be the probability that class j passes the test.

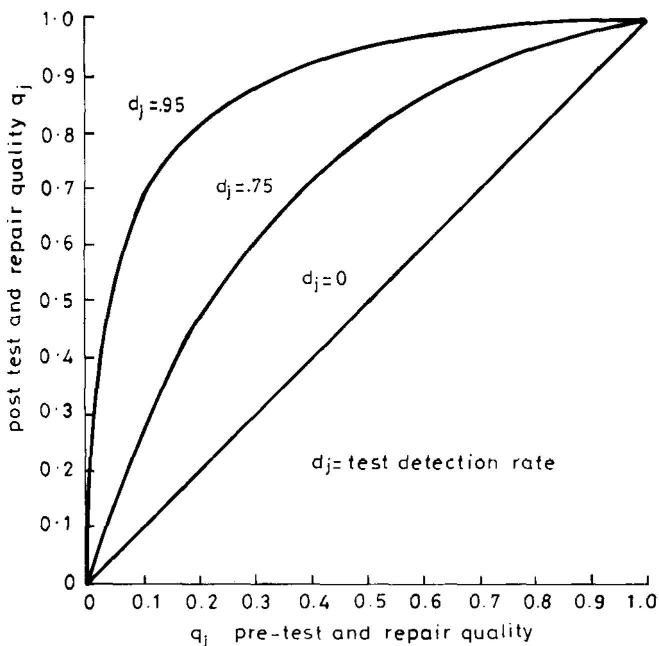


Fig. 2 Improvement in quality from test and repair

If the flow of items into test is taken to be 1:

then $p_j = q_j + (1 - q_j)(1 - d_j)$

If the probability that the item passes the test is P, then assuming that no good items are rejected

$$P = \prod_{j=1}^{n} p_j$$

where n is the number of fault classes and the probability F that the item fails the test *is*

$F = 1 - P$

Since of those passing the test for any fault class only q_j are fault free the quality of those which pass is:

$$q'_j = q_j / [q_j + (1 - q_j)(1 - d_j)]$$

The overall quality of items passing the test Q' when all the fault classes are taken into account is

$$\text{Then } Q' = \prod_{j=1}^{n} q'_j$$

Since the pre- and the post-test qualities are known the number of faults removed by the test can also be determined using the fault distributions.

3.3.1 Feedback through repair

In many cases those items which fail a test are all scrapped, integrated circuits being an example. In this case the above formula for quality out from test clearly applies. However in other cases the items failing the test are sent to a repair stage and thence back into the test as illustrated in Fig. 3. This complicates matters since the quality of input to the test is modified for the repaired items. However, providing certain simplifications can be accepted, it can be shown that the quality of items passing the test is the same with feedback through repair as without.

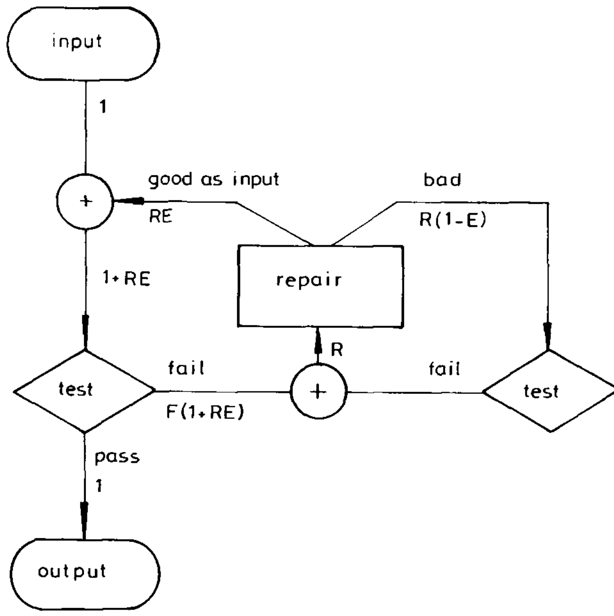


Fig. 3 **Flows through test and repair**

The simple model relies upon the assumptions that the scrap rate and the damage rate during repair are so small as to be ignored. Also, the repair process and the replacement components are assumed to be no better than original.

Because of these assumptions concerning the repair process, the above formula for post-test quality still applies even for those items which failed the test and were repaired and retested. This can be understood as follows: for those assemblies which failed the test for this fault class (j) the repair process raised the quality to the same level as the original input (i.e. q_j) and hence the above formula applies. However for those assemblies which failed the test because of some other factor the repair process will have no effect on this fault class (j) and since retest is the same test no more information can be obtained.

3.3.2 Calculating flows through test and repair. The steady-state flows around a test and repair stage, where failed items are repaired and then returned to be retested at the failing test stage, may be determined by analysis, assuming the input flow to be constant. The method of analysis, which was suggested by the referee, is to separate the flow out of repair into that which contains items which are as good as the untested input and that which contains the bad items. This is shown in Fig. 3.

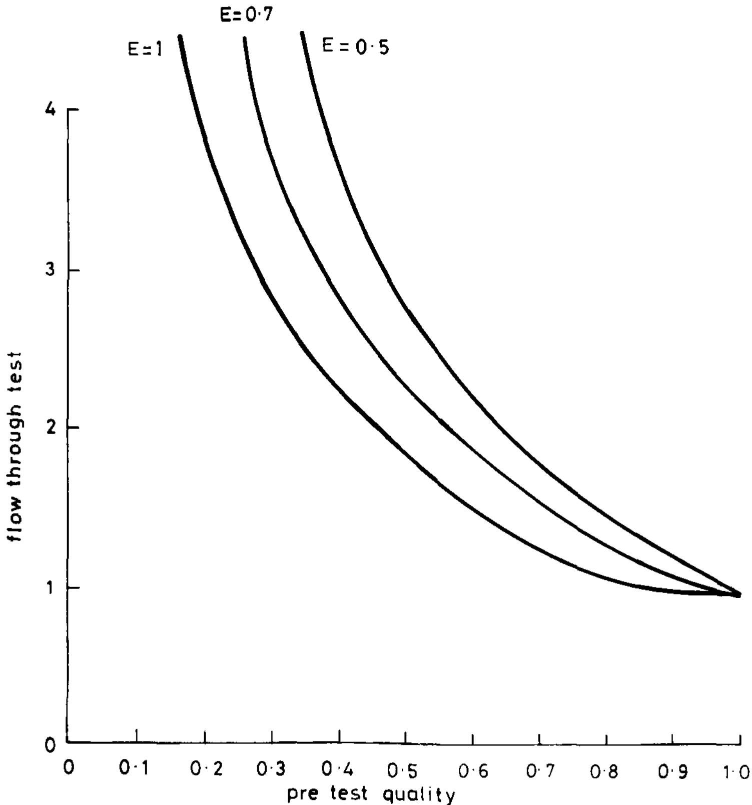


Fig. 4 Influence of quality on tester loading
Test detection = 0·95
E - effectiveness of repair

Suppose that the repair process is capable of raising a proportion E of the items submitted for repair to the same quality as the input prior to test and that the rest of the items submitted for repair remain faulty. Also assume that there is no scrap. By means of these assumptions the quality model, as discussed above, remains valid and the imperfections of the repair process are accounted for by increases in the flows around the system.

If the untested input flow is assumed to be 1, the proportion which fail the test first time is F, and the flow into repair is R. The output from repair can be seen to be divided into two parts, that which is as good as the untested input having a flow $R\,E$ and that which is bad having a flow $R(1 - E)$. The flows around the system are:

flow through repair:

$$R = F\,(1 + R\,E) + R\,(1 - E)$$

i.e. $R = F/E\,(1 - F)$

flow through the test:

$$X = (1 + R\,E) + R\,(1 - E) = 1 + R$$

The way in which flow through the test varies with untested quality for a fixed test detection characteristic and varying repair effectiveness is shown in Fig. 4.

In addition the average number of times a failed item passes around the repair loop can be determined:

$$Z = R/F = 1/E\,(1 - F)$$

3.4 Calculation of costs

Once the flow rates through test and repair have been determined for a given set of circumstances, input quality and test detection, the loading on the test equipment and hence the costs can be calculated. The total cost is the important factor since ignoring some of the less obvious cost factors can easily lead to a distorted view. The total cost of a test strategy includes all the costs associated with acquiring the resources necessary, using and maintaining these resources for a given period of time and, finally, disposing of them.

The total costs of a test strategy form part of the Life Cycle Costs of the products produced using the equipment. Hence they are also of interest from this point of view. In this context the performance of the equipment is also relevant since the quality of the output product has cost implications. However this subject is not dealt with here.

3.4.1 Cost breakdown. The total costs of a test strategy may be evaluated using the cost breakdown structure described by Blanchard[4] in which the total cost is broken down into a research and development cost, an investment cost and an operations

cost. This is illustrated in Fig. 5. For the purpose of comparing testing strategies the following meaning and breakdown of these cost categories is used:

Research and Development Cost. This covers all the costs associated with basic research, feasibility studies, engineering design, the construction and testing of prototypes and the provision of design documentation. These costs apply where all or part of the test equipment being considered is to be developed in house. They must not be neglected since they can exceed the cost of producing the equipment when it has been developed. The cost of developing the control software, test software and writing or generating the test data including the costs of documentation and validation must also be included in this cost.

Investment cost. This covers all the costs incurred in acquiring the required testing capacity and making it operational. If the tester is produced in-house it includes the manufacturing costs, both recurring and non recurring. If the tester is bought in the cost includes the purchase price or an equivalent cost if the tester is leased or hired. Factors also to be included are the cost of installing the equipment, the cost of providing the floorspace and environment required, any support investment necessary (for example spares holding), the cost of test fixtures and training costs.

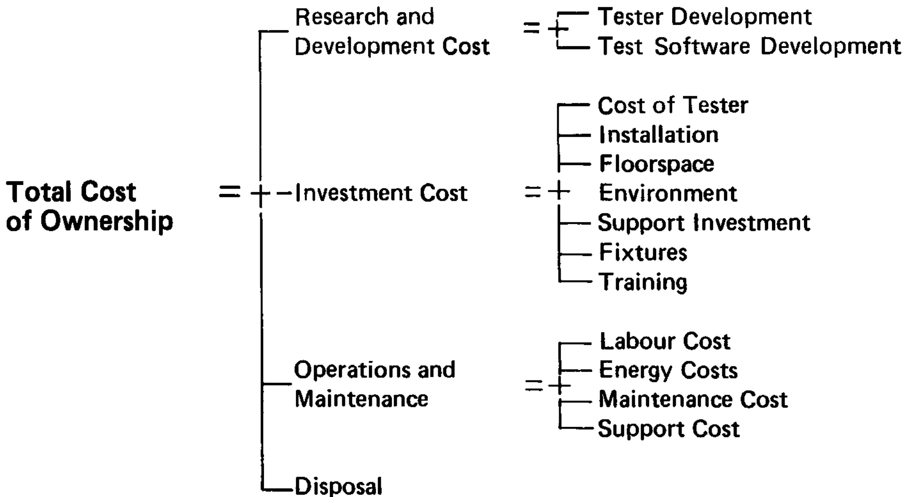


Fig. 5 Test equipment: total cost of ownership

Operations and maintenance cost. This covers all the costs associated with operating and maintaining the equipment. It includes the labour costs of operating the tester and, where applicable, the labour costs of repairing the items failing the test. This latter cost is essential to make the influence of tester diagnosis visible. The operating cost should cover the energy requirements of the testers. Where the equipment is maintained in-house the indirect as well as the direct maintenance costs must be included, for example training maintenance personnel etc. The cost of maintenance of the software and test data, as well as modifications to the equipment, must be included in the support cost.

3.4.2 Total cost and unit cost. The total cost can be determined for a given amortisation period for the test equipment. This is calculated as the total of the development costs, the investment costs and the operations and maintenance costs for that period.

From the total costs the cost per unit of output may also be determined. The unit cost is the total cost divided by the total volume of production output over that period.

3.5 Determination of tester capacity required

The time required on a tester to achieve a given output over a given period of time can be determined from the flow through the tester, the test and diagnosis times and the availability ratio of the tester. (The availability ratio reflects the reliability and repairability of the test equipment.) The number of testers required and hence the total cost follows from the time required.

Let:

D_r be the output required from the tester over the period.
T_a be the tester time for a fault free item.
T_d be the additional tester time required to diagnose a faulty item.
A be the availability ratio of the tester.
R be the flow into repair for unit output.
X be the flow through the tester for unit output.

Then the time required on the test equipment for that period is:

$$T_{req} = D_r (X T_a + R T_d)/A$$

The time available per tester T_{av} is set by the proposed work pattern, (hours per shift, shifts per day, days per period). From the time required and the time available the number of testers needed may be determined:

$$N = T_{req}/T_{av}$$

If N is not an integer this equation holds only if the excess capacity can be used for other work. Otherwise the number of testers must be rounded up. The policy to be used must be decided by the user.

4 Implementation of the modelling system

The modelling system is implemented to run on an ICL 2900 series computer in a VME 2900 environment.[5] It is designed to be used interactively from a video/keyboard but may also be used non-interactively from batch jobs. A model of a particular test system is represented by data provided by the user. The data required comprise three parts:

Definitions of cost and quality parameters for the entities involved, (i.e. the components, processes and tests).

A description of the structure of each strategy to be evaluated.

Data controlling the operation of the system.

The data are set up by means of task-oriented extensions to VME 2900 SCL, the system control language. This makes maximum use of the facilities provided by VME 2900 to assist the user and minimises the programming effort required to implement the system.

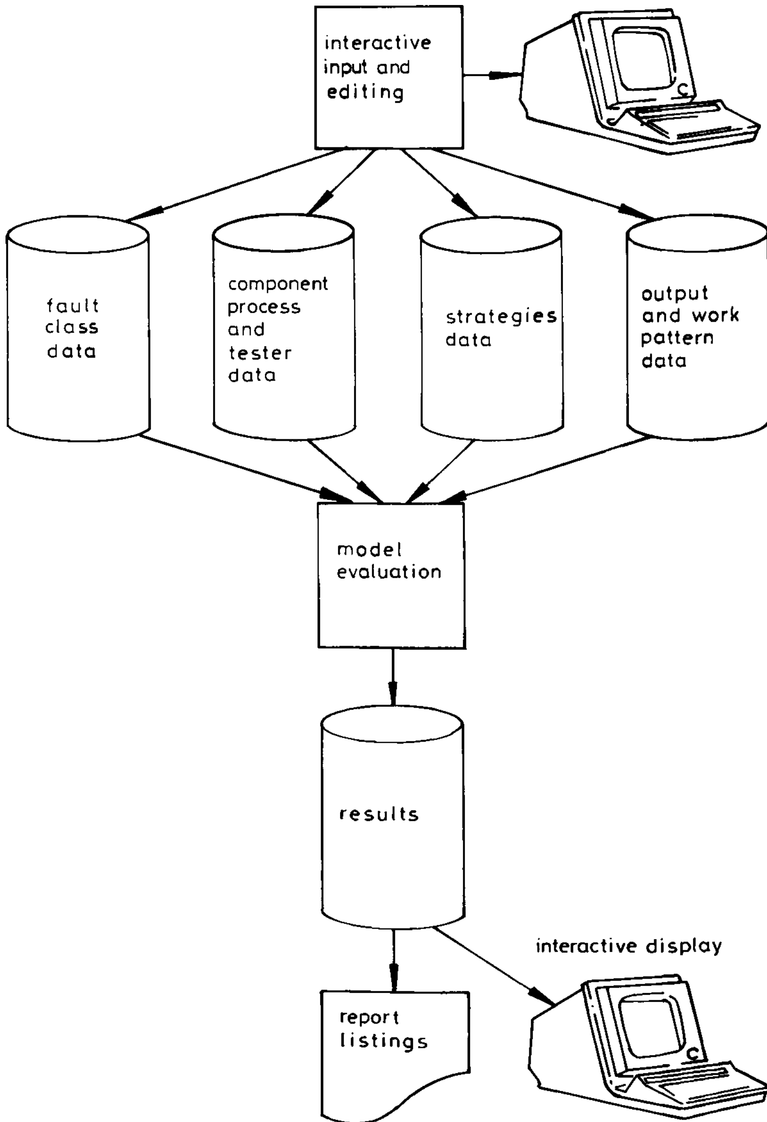


Fig. 6 The modelling system

Data are entered into the system by calling one of a set procedures by an SCL statement. A procedure is, in effect, a small program which performs some function and is complete in itself. Each procedure is known by two names; a full name which is chosen to be meaningful and an abbreviated name to facilitate interactive use. Each procedure has a set of parameters, and the data are passed to the procedure by means of these. The parameters are identified by name. Where practical the system has default values for parameters and these are used unless specifically changed by the user. A template showing the parameters required and their default values may be called up and used to input the data for any command. This is achieved by means of the standard VME 2900 facility known as screen prompting.

While the system is in use data are held in the virtual machine store. This is an ICL 2900 series facility which makes a large amount of program memory available to each user. The data may be set up by procedures executed there and then or may be taken from files set up previously. Entity data and structure data may be filed separately. Hence definitions of standard components, processes and tests may be shared between models, and models may be preserved between sessions.

4.1 Entity definition procedures

DESCRIBE FAULT: enables names to be given for each fault class. These names will be printed out in reports.

DESCRIBE COMPONENT: sets up the quality data for a particular component which forms part of the product to be tested. These data are the quality of the component, as it is received into the system, in respect of each fault class which is relevant. The component is given a name which may be used later, in the appropriate places, to refer to the data which has been set up.

DESCRIBE TEST: sets up the data necessary to describe a particular type of test and its associated repair stage. The data give:

The name of the test type.
The classes of faults detected and their detection rates.
The repair effectiveness.
The test, diagnosis and repair times.
The various costs associated with the life cycle of the equipment.
The work pattern and amortisation assumption to be used.

DESCRIBE PROCESS: sets up the data necessary to describe a particular type of process. The data give the name of the process and the damage rates which the process causes for the various fault classes.

ALTER COMPONENT, ALTER TEST, ALTER PROCESS: these procedures enable data already set up describing components, tests and processes to be altered.

4.2 Describing the structure of the strategy

As mentioned previously the manufacturing system for a product can be considered to be an implosion tree. The method used here for dealing with this structure involves

the use of a push down stack. Data concerning a particular component or subassembly may be entered on the top of the stack, pushing any existing data down. Procedures are then available which operate on the data at the top of the stack to represent the influence of the processes and tests. These procedures replace the data input from the top of the stack with the result leaving this as the topmost element. This is similar to the method of doing arithmetic which is known as reverse Polish.

The procedures which are available are:

COMPONENT: enters the data for a component on the top of the stack. The data may be specified at the time the procedure is called or alternatively the name of a component for which data has been given by the DESCRIBE COMPONENT procedure may be given.

COLLECT SAME: models the effect of there being several of the element currently at top of stack. It has the effect of raising the quality values for each of the fault classes to the power of the number given and multiplying the quantities upon which flow and hence costs will be based. The element at the top of stack may represent a single component or be the accumulated result of previous operations representing an assembly.

COLLECT DIFFERENT: models the effect of combining together several elements starting with the top element on the stack and working down. It multiplies together the values of quality for fault classes which are common between the elements and adds up the quantities to be used for flow computation. The elements may be the same or different and may represent single components or assemblies.

PROCESS: models the effect of a process on the element at the top of the stack. It has the effect of reducing the quality of the various fault classes in proportion to the damage rates for the process. The damage rates may be given directly or the name of a previously described process may be given.

TEST: models the effect of a test stage on the element at the top of the stack. The quality data for the various fault classes are modified according to the fault detection rates for those classes. Flow rates are determined and resource usage is calculated, from which testing costs can be worked out when the total strategy has been evaluated. The data defining the test may be given directly or the name of a previously described test can be given.

Control procedures

TEST STRATEGY MODEL: once the user is logged in, this procedure must be called to connect to the modelling system. It assigns the libraries containing the system and thus makes the various procedures available to the user. The call on this procedure should be preceded by the SCL 'BEGIN' statement to create a new lexical level since the effect of the procedure will continue until the end of the block as signified by a matching 'END' statement or the user logging out.

STRATEGY: this procedure is called to indicate the start of a model to evaluate a particular test strategy. Only one strategy may be evaluated at once but several

strategies may be evaluated in sequence during one session. The parameters to this procedure are the name of the strategy, the name of the entity file and the name of the trace file to be used. The entity file parameter, which is optional, specifies a file containing entity definitions set up by previous runs. Thus a standard set of component, process and test definitions may be used for several models. New entities may be introduced during the evaluation of a strategy and, if required, the amended entity data saved for future use. The trace file is used to contain a record of each step of the evaluation of the strategy and the intermediate results obtained. These data can be used to understand how the overall results summarised in the report were obtained.

OBEYFILE: the procedure calls forming all or part of a test strategy model may be entered into a file in the VME 2900 filestore using the normal file handing procedure provided, (INPUTFILE, SCREENEDIT etc.). This procedure enables such data to be executed. The procedure has one parameter which is the name of the file containing the data.

REPORT: this procedure produces a summary report of the results obtained for the strategy being evaluated up to the point at which it is called. Data specifying the production rate required must be given. The report is written to a file which may be spooled to a printer or displayed on a video.

The report is in tabular form and is divided into sections containing the following information:

A summary of the details of the model as set up by the model definition commands.

Test performance information comprising for each test stage of the test strategy the input and output qualities for each fault class, the average number of faults detected, the number remaining after test, the average test and repair times and the total test and repair times for the strategy.

Overall cost and quality performance of the test strategy giving: the total costs, the cost breakdown, numbers of testers required, output qualities obtained and unit test and repair costs.

SAVE: indicates that the entity definition currently set up are to be saved in the specified file for future use.

HELP: at any time while using the system the user can have a 'mini user guide' displayed on the terminal by calling the macro HELP. A list will be displayed of all the commands together with their abbreviations and a brief indication of their functions. More detailed information on an individual command can then be obtained if required.

DEFAULTS: this procedure changes the general default values used by the system. This allows general policy changes, such as altering the number of shifts worked, amortisation time of equipment etc., to be evaluated quickly and easily.

4.3 Calibration of the model parameters

Data concerning faults found on approximately 10,000 printed circuit boards during manufacturing and installation were analysed to determine process fault rates and test detection characteristics in a form suitable for use in the modelling system. There were 10 board types but, for simplicity, these were divided into three classes on the basis of differences in the test strategy used. These classes were central processor unit boards, store unit boards and peripheral coupler boards. The test strategy for which the data were collected consisted of a post assembly continuity test, a functional test, a unit test and finally installation tests. The functional test applied was different for each board type.

The total fault rate for each class of fault and each type of board prior to cold test was determined from the observed fault detection rates. From the total fault rate and the detection rates for each test the test detection characteristic for single faults together with the pre- and post-test quality levels were calculated. From these data the model gives expected values for the board failure rate at each test stage, this figure was determined and compared with the observations. The predicted board failure rates were found to match the observed values closely as is shown in Fig. 7.

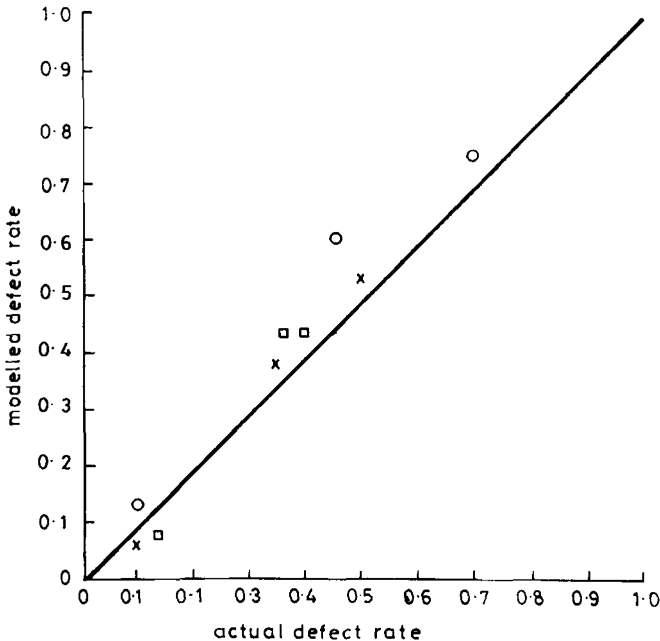


Fig. 7 **Model predictions using observed data**
o CPU PCBs
x Store PCBs
□ PCM PCBs

Since the total numbers of faults detected and hence the size of the sample from which each test detection characteristic was known, it was possible, using statistical

data, to estimate the range of possible detection characteristics. An example of the test detection rates observed for a functional test stage is shown in the Table 1.

Table 1. Observed test characteristics

Tester Type - FUNCTIONAL (Test Rig)
Pcb Type - Central Processor Unit

Fault Class	Detection Characteristic		
	Observed	Range 0·95 Confidence	
		Minimum	Maximum
PROCESS ERRORS			
Open circuit track	0·932	0·89	0·95
Short circuit track	0·909	0·82	0·94
Wiring	–	–	–
Through plated holes	1	0·69	1
ASSEMBLY ERRORS			
Short circuit (solder)	1	0·98	1
Track cut in error	–	–	–
Incorrect Assembly	1	0·99	1
Wiring	0·929	0·85	0·97
Dry Joint	1	0·69	1
Bent pins	–	–	–
Damage	1	0·86	1
COMPONENT FAULTS			
Integrated circuit faults	0·806	0·78	0·83
Miscellaneous Components	0·745	0·68	0·81
Resistor faults	0·952	0·91	0·98
OVERALL	0·881	0·87	0·89

4.4 Use of the system

The system described here was used in the design of the manufacturing test system for a medium sized computer mainframe. The test system in question covered PCBs from bare board test through their assembly into the final product and final system test. Six possible testing strategies were evaluated against a range of possible untested quality values. Results were obtained showing the output quality and cost breakdown for the various strategies and untested quality levels. This information was used to make the final choice of the test methods to be used, the testing/repair capacities required and to set quality targets for the different stages.

The information used was gathered by a combined team comprising design

engineers, manufacturing engineers, test engineers, test programmers and quality engineers. This team defined the test strategies and fault classes to be considered and obtained and agreed the values to be used for the vaious parameters. A target output quality level had been set and it was one of the objectives for the chosen strategy that this target should be achieved.

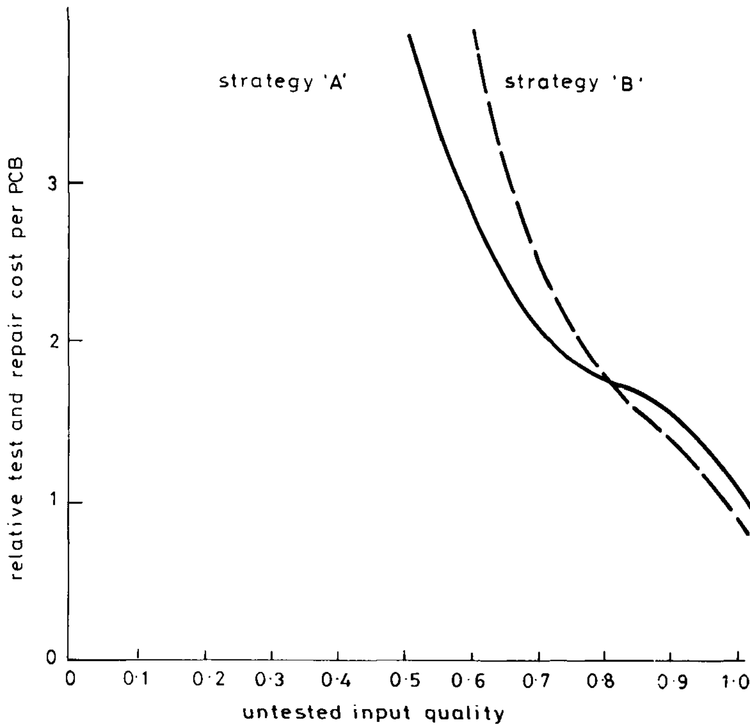


Fig. 8 Influence of quality on cost

The test strategies evaluated comprised various combinations of tests from the following test types:

Component test (of integrated circuits at goods inward).
Bare board continuity test.
Assembled board continuity test.
Board functional test (probe type).
Board functional test (edge finger type).
Store board test.
Assembled processor internal fault detection.
Assembled processor microcode diagnostics.
Assembled processor scan in scan out diagnostics.
Assembled processor order code level function tests.
Assembled processor system software level tests.

Fault profiles for the major subtypes of board and test characteristics for the various test types were estimated. This exercise, in itself, brought to light problems

for which solutions were found. The multidisciplined nature of the team was of great help and promoted useful interchange of views and ideas.

The kind of results obtained from the evaluation are illustrated in Fig. 8. This shows the influence on test and repair cost per board of different test strategies and untested quality levels, and that test and repair costs are very heavily dependent upon untested quality. For the strategies illustrated there is a break-even untested quality level above which one strategy is cheaper and below which the other is cheaper. Hence the choice of strategy becomes dependent upon the confidence which can be placed on the estimates of the untested quality level.

5 Conclusions

A method of modelling the performance and cost of manufacturing test strategies has been described. This model has been programmed for an ICL 2900 series computer in a VME 2900 environment. Data have been collected from which the values of parameters describing the existing processes can be deduced. The modelling system has been used to evaluate several possible test strategies for a computer mainframe. This exercise proved worthwhile since problems were identified, solutions were found and conclusions were reached in a logical and quantified manner. In general it was shown that testing costs are more sensitive to untested quality level than to testing strategy assuming a fixed output quality must be achieved. Hence control of processes and bought in quality is very important in controlling testing costs. Future work will be to include the influence of queuing and the determination of work in progress in the modelling system.

Acknowledgments

To H.Baron who suggested the idea of the computer program, to C.W.Bartlett for the reverse Polish system, to R.Whittaker, P.Dillon and K.T.Burrows who designed and produced the software, to R.F.Guest, J.J.Stewart and K.Hoyle who constructed the models.

Some of the material reported here was presented at the international conferences, INTERNEPCON 1980 and AUTOMATIC TESTING 1981. We are grateful to NETWORK of Birmingham UK, who organised the 1981 meeting and who published the proceedings, for permission to reproduce this material.

References

1 WADSACK, R.L.: 'Fault coverage in digital integrated circuits', *Bell Systems Tech. J.*, 1978, **57**.
2 WILLIAMS, T.W.: 'Testing logic networks and designing for testability, *Computer*, Oct. 1979.
3 SMALL, M. and MURRAY, D.: 'A quality model of manufacturing and test processes, INTERNEPCON, UK 1980;
4 BLANCHARD, B.S.: Design and manage to life cycle cost, M/A Press, USA.
5 WHITTAKER, R.: 'User guide to manufacturing test and repair cost estimation program', ICL Internal Document EDSIN277.
6 SMALL, M.: 'A manufacturing test performance and cost model', *Proc. Automatic Testing 81*.