ICL TECHNICAL JOURNAL

Volume 4 Issue 2
November 1984

ICL
TECHNICAL
JOURNAL

Contents

Volume 4 Issue 2

ICL TECHNICAL JOURNAL

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

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

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

1985 subscription rates: annual subscription £15.00 UK, £17.00 overseas, airmail supplement £7.50, single copy £9.00. Cheques should be made out to 'Peter Peregrinus Ltd.', and sent to Peter Peregrinus Ltd., Station House, Nightingale Road, Hitchin, Herts. SG5 1SA, England, Telephone: Hitchin 53331 (s.t.d. 0462 53331).

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

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

Modelling a multiprocessor designed for telecommunication systems control

R.H. Thompson
British Telecom Research Laboratories, Ipswich, Suffolk

Abstract

The paper describes a sophisticated computer simulation model of a multiprocessor which was designed to control the operation of a telecommunications exchange. The model was initiated in the early design stage of the multiprocessor, and has been progressively enhanced to represent successive developments in the processor design.

The model has been used first as an aid to the evaluation and optimisation of the design of the multiprocessor, and secondly to predict its throughput and performance under normal load, overload and failure conditions.

1 Introduction

Modern stored-program controlled (SPC) digital telecommunications exchange systems are modular in concept, so that each individual module can be separately enhanced to take advantage of improvements in technology without affecting the remainder of the system. The principal hardware modules needed in a telecommunications exchange serving predominantly telephone subscribers are shown in Fig. 1, and are as follows:

- *Line interface units*. These control the state of subscriber lines and the circuits to adjacent exchanges.
- *Signalling units*. These convey signalling information such as call seizure and routing digits, to and from other exchanges.
- *Subscriber concentrator units*. Most individual subscribers generate little traffic, where 'traffic' is interpreted as the proportion of a given time (the exchange 'busy hour') that a line or circuit is occupied with telephone calls. Economics demand that subscribers' traffic be concentrated onto considerably fewer circuits before being switched through the exchange.
- *A routing switch*. This is used to connect a subscriber's call through the exchange either to another subscriber on that exchange or to another exchange.
- *A control processor*. While the other hardware modules will undertake local

processing of their routing functions, in most exchange designs a central control processor will perform the main procedures for handling telephone calls.

This structure will vary for other types of exchange, for example, an exchange simply interconnecting other exchanges would not have subscriber concentrator units. However, these differences are not significant here.

To ensure that a complete exchange system will meet its performance requirements in terms of traffic and call-handling capacities, it is essential that the performance of each hardware module and the software interactions between modules be analysed and quantified. It is also necessary to determine rules so that the optimum amount of equipment needed for any given exchange can be specified; a finite probability of congestion (the 'grade-of-service') is economically necessary.

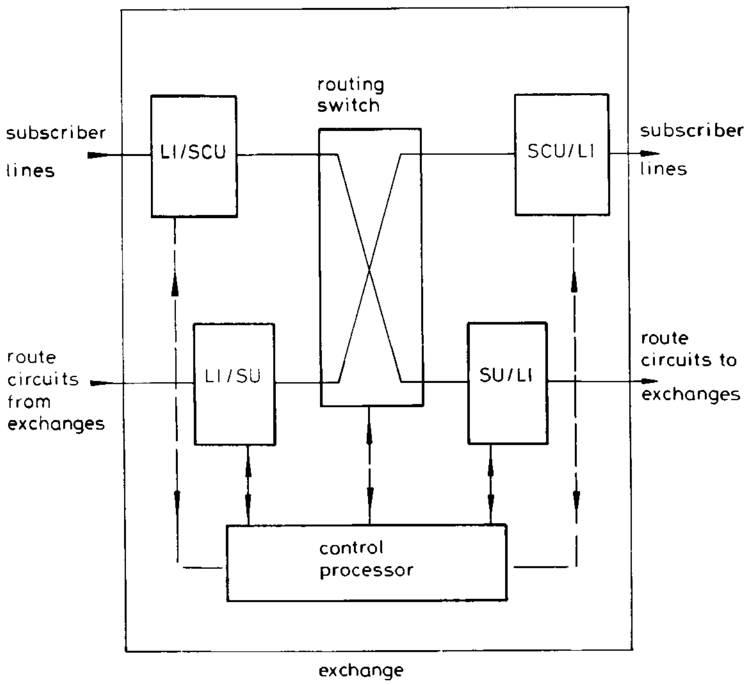


Fig. 1 Simple structure of a telecommunications exchange
LI: line interface unit
SU: signalling unit
SCU: subscriber concentrator unit

Performance analysis and the derivation of rules for specifying equipment quantities are the roles of the engineers in the Teletraffic Division, part of the British Telecom Research Laboratories at Martlesham Heath, Suffolk.

The subject of this paper is the control processor, whose basic performance parameter is the number of call attempts it is capable of handling in the exchange busy hour. The paper describes a simulation program developed by members of the Teletraffic Division to model the hardware and software interactions for a particular multiprocessor design for the control processor. The simulation model has aided the design of the multiprocessor and is currently used to predict its performance under normal call attempt load, under high to severe call attempt overload, and under various failure conditions.

2 General description of the multiprocessor being modelled

Basically, the multiprocessor being modelled consists of up to four central processing units (CPUs) and their associated hardware, collectively known as a

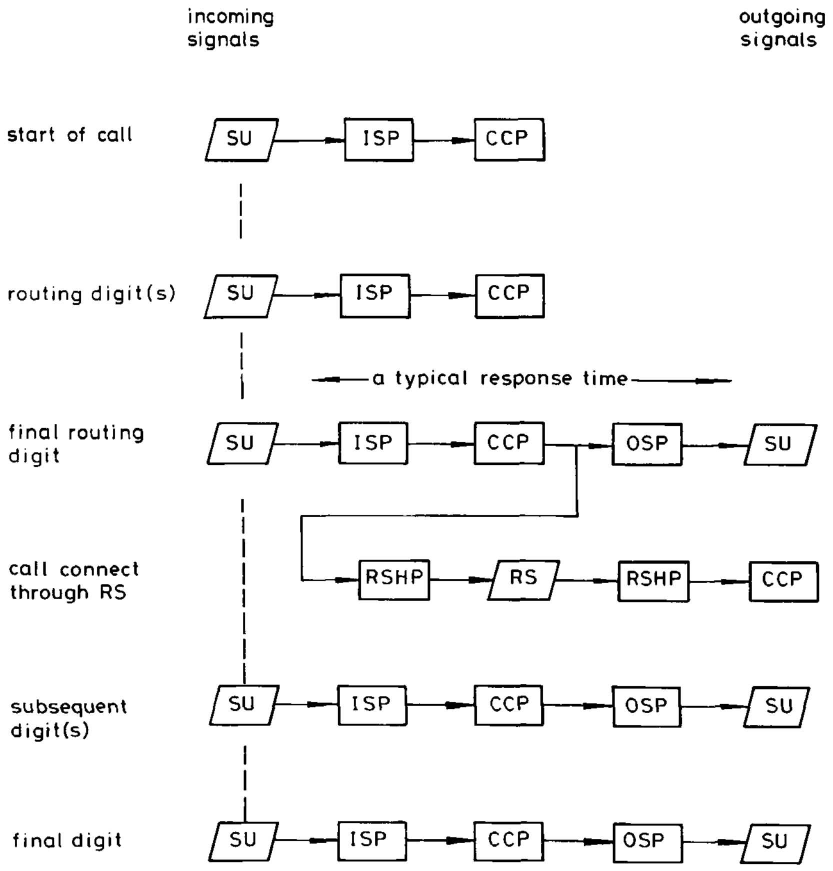


Fig. 2 Typical call setup at main network exchange

CCP:	call control process	SU:	signalling unit
RSHP:	routing switch handler process	RS:	routing switch
ISP:	input from SU process	¦	time intervals between dialled digits
OSP:	output to SU process		

cluster. Each of the CPUs in a cluster has access to a common pool of store blocks. Up to eight clusters can be linked together to increase the processing power if required. The associated exchange hardware modules – for example, the signalling units and the routing switch – appear to the multiprocessor as peripheral units. Each peripheral receives attention from the multiprocessor either periodically or as and when required. However, the simulation model represents these peripherals only as the input/output of work for the multiprocessor, so that further discussion about them is unnecessary here.

Within a cluster, the store blocks form the main memory of the multiprocessor and its software resides in this memory. This software is divided up into a number of units, called processes, each of which is responsible for a well defined function. A process can be one of two types – an application process concerned

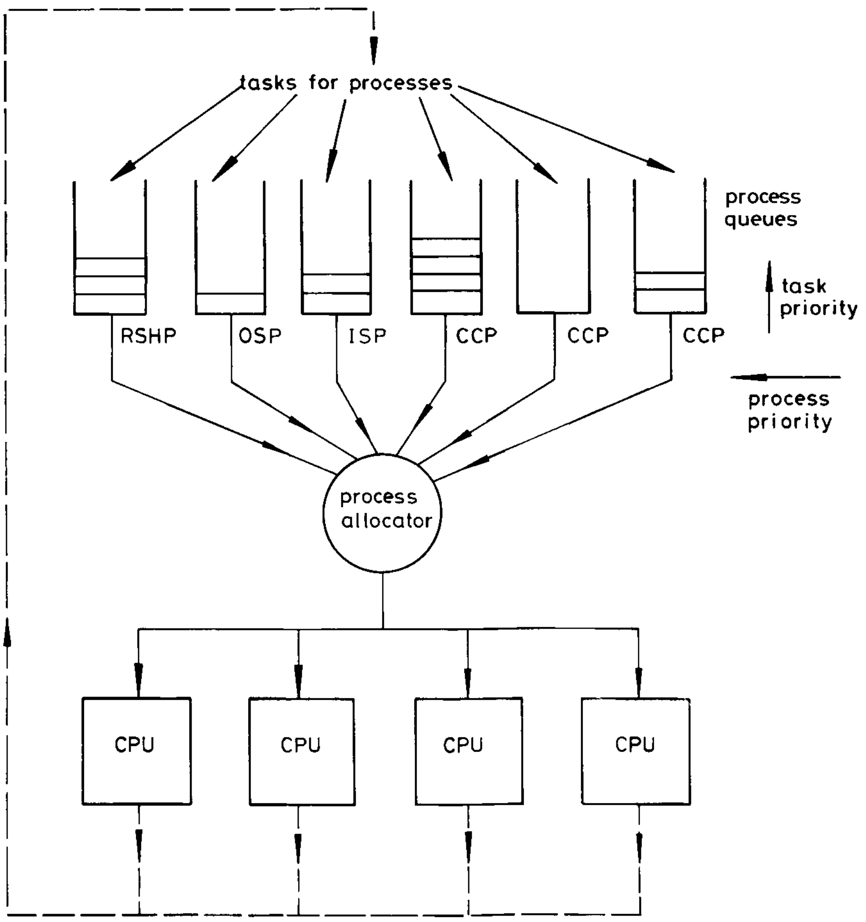


Fig. 3 Process scheduling on the multiprocessor
CCP: call control process (replicated) ISP: input from signalling unit process
RSHP: routing switch handler process OSP: output to signalling unit process

with the handling of telephone or data calls, or a management process which comprises part of the multiprocessor operating system (OS). A number of processes may together comprise a subsystem, and a process may itself be made up of a number of separate modules. For example, the call control process is responsible for the central core of telephone functions required to set up and clear down telephone/data calls. This process is one of many processes forming the call processing subsystem, and the process itself is made up of several modules including the network routing module and the line selection module. However, it is a process as an entity that is recognised by the operating system rather than a subsystem of processes or a process module, so that only processes need be considered here. Processes are scheduled to run on the CPUs, and they can be replicated when more processing power is required to perform a particular function than one copy of the process, running on one CPU, can provide.

The setup and cleardown of a telephone call requires a number of different functions to be performed, i.e. a number of processes to run. Therefore, a sequence of process activations will be associated with each call. Fig. 2 shows, in simplified form, such a sequence of process activations occurring for a typical call setup at a main network exchange. The entire set of such sequences is known as the application program (AP). All transfers of call data between processes are handled by the OS in the form of tasks, so that, in Fig. 2, the OS is required to run at all points corresponding to an interprocess task. Fig. 3 outlines the interaction between the processes and the OS from the teletraffic standpoint. The key feature is a system of queues whereby demands for processing resources (i.e. the tasks) queue for particular processes which, in turn, queue for service by the CPUs. Tasks are serviced by the processes on the basis of task priority (up to 16 levels) and first-in first-out (FIFO) within a priority level.

Any process may, under control of the OS, run on any CPU, but on no more than one CPU at any one time (i.e. processes are non-re-entrant). Processes are allocated to the CPUs on the basis of process priority. Process priorities are pre-emptive, whereas within a process task queue the task priorities are non-pre-emptive. This priority discipline is complicated by the fact that some processes, primarily the peripheral unit handler processes, can be activated periodically; i.e. they are made inactive once their task queues have been emptied and are only reactivated following a periodic 10 ms clock interrupt (this mode of operation facilitates timing functions and reduces OS overheads). The passing of tasks between processes, and the scheduling of both a process to run on a CPU and a task to be handled by that process, are the responsibility of the part of the OS known as the process allocator (PA). Because of its central role in the operation of the multiprocessor and to minimise its runtime, the PA is implemented in microcode within the control store of each CPU. In effect, the PA operates as the highest priority process in the multiprocessor.

Since the processing power of any processor system is limited, there will be a maximum call attempt rate which can be handled successfully. Telecommunications traffic can be highly volatile, and an influx of call attempts beyond the maximum designed rate can occasionally occur. An efficient processor load control is essential to ensure that there is no serious reduction, or even collapse, in the call handling rate of the processor system. In the multiprocessor being modelled, load control operates by monitoring the occupancy of the CPUs and the state of the process task queues, and then by using this information to regulate the rate at which new calls are accepted for handling by the multiprocessor.

3 General description of the simulation model

The model has been designed to simulate the interaction between the operating system, especially the process allocator and load control, and the application program for a single cluster, with the added facility of being able to represent the intercluster communication of a multicluster configuration. It has been progressively developed over a period of some 10 years, from the very early processor design concepts to the current multiprocessor, multicluster configuration. This development of the model has been facilitated because, first, the process structure of the software has been carried forward almost unchanged through the major design changes in the multiprocessor hardware, and, secondly, the decision was taken at an early stage that most of the multiprocessor design details would be defined through input data to the model. New multiprocessor designs have therefore been relatively easy to model by changing this data.

The model has been programmed in PL/1 and uses the event-by-event simulation technique. This technique is the basis of a PL/1 simulation package, called Telesim, which has been specifically designed by the Teletraffic Division for teletraffic performance analysis. The Telesim package undertakes the scheduling of events, and provides for the handling of histograms and confidence interval routines. The model thus consists of a set of event-by-event action blocks, and Telesim schedules the action blocks to run in the order in which they occur in time. By this means, the real-time behaviour of the model is obtained.

Two other main sections of the model concern queue handling and the preparation of input data. There can be up to 128 of the process queues, shown in Fig. 3, where each process queue can have any priority (including the same as another queue) and can consist of tasks having up to 16 priority levels, with the entire set of tasks being buffered in a single storage area. A queueing package consisting of multilinked lists was developed specifically for the model to minimise its runtime and storage requirements, and to allow a variety of service disciplines such as FIFO, LIFO and random-out to be studied. The input data for the model consists of the multiprocessor configuration, details about the processes and tasks which they service, and the set of sequences of application processes which form the Application Program. These data are initially coded as simple alphanumeric structures collectively known as process structured language (PSL), and converted by a PSL translator into data suitable for use by the model. Developed for the simulation model, PSL is a subject in its own right; it is sufficient to say here that it is PSL which provides the model with its versatility.

The model has been written to provide the following main facilities:

(i) a variable number of CPUs per cluster
(ii) a variable number of clusters
(iii) a variable number of software processes

Each process can have a unique priority, or the same priority as another process, and has a queue with 16 priority levels for tasks handed to it.

Each process can be replicated, i.e. there can be more than one copy. A replicated process can have its total traffic balanced or imbalanced among its replicates.

Processes can run on demand, can be initiated periodically, or both. When a periodic process empties its queue, it must receive a timing task, or a task of higher priority, before it can service any tasks which have arrived and are of lower priority than the timing task.

Processes can be interrupted and resumed at a later time. There is no restriction on the number of times a process may be interrupted.

(iv) a variable number of process sequences

A sequence need not necessarily represent a complete telephone call, but any part of a call. A sequence can also represent any form of nontelephony actions.

The model allows parallel processing where this occurs in the sequences, and the passing of control within a sequence can be dependent on a number of conditions having been satisfied, for example, a routing digit cannot be transmitted until it has been received *and* the previous digit has been transmitted.

(v) timing values, which are used for:
- sequence interarrival times
- process task runtimes
- operating system (PA) runtimes.

These may be specified as constant or negative exponentially distributed.

(vi) a variable number of response times may be collected for each sequence, facilities having been provided to start and stop a measurement at any stage in a sequence
(vii) a set of results to give details of the usage and performance of the above facilities, including:
- CPU occupancy
- process occupancies, queueing delays and task queue lengths
- processing effort and time-in-system for each sequence
- response time distributions
- the loss grade-of-service, with the cause(s) for calls being lost, if any
- the behaviour of the processor load control parameters
- a history of the above results during a run of the model.

4 Major elements in the simulation model

A detailed description of the complete model is inappropriate for this paper, instead an outline is given of the major elements in the model. These elements are the process allocator, the set of CPUs, processes and tasks, processor load control and the application program.

4.1 Process allocator (PA)

The PA is modelled as a process per CPU, with each process having the same priority, this priority being the highest in the model. The PA can be considered as re-entrant (more than one process can execute code simultaneously). However, the PA cannot be modelled as a totally re-entrant process, since there are times when data common to all the PAs must be accessed. On these occasions, access to the common data is controlled by a set of lockouts. In the model, when one PA attempts to access data already locked out by another PA, the former PA will be enqueued to await the freeing of the lockout.

The PA is run whenever a process makes a call to it, or an interrupt is received. Under normal conditions, a process will only call the PA if either it is handing a task to another process, or the process has finished the task it was handling. In the latter case, the PA will either find another task in the process's queue for the process to serve, or it will set the process blocked and will reschedule the CPU to set running on the CPU the highest priority process waiting to be processed.

Upon arrival, an interrupt is immediately serviced by that CPU currently running the lowest priority process, this CPU being called the LCPU. Three types of interrupt are incorporated in the model

– a clock interrupt, which is generated at intervals equal to the clock period (currently 10 ms). This enables the PA to perform timing functions and send timing tasks to the periodic processes when required

– an interrupt generated from within a process sequence, which allows a sequence to initiate the running of a process to represent interrupts generated from the associated peripheral units
– a suspended queue interrupt (SQI), which is generated to set running on the LCPU a process which is of higher priority than the process currently running on that LCPU. An SQI may be generated whenever a process has handed a task to another process, or an interrupt has just been serviced.

For a multicluster configuration, the PA additionally performs intercluster task passing. A task destined to be passed from one cluster to another is placed by the PA in a buffer for a time representing the task transmission to the other cluster. After this time, the task is inserted into a receive buffer, which is scanned by the PA every clock interrupt. Thereupon, the task is handed to the destined process.

4.2 CPUs, processes and tasks

Each CPU must be in one of the following states:

– waiting for a lockout to become free
– running a lockout
– running the process allocator
– running a process

Each CPU has associated with it a Background process. This process is run whenever there is no other process awaiting service on a CPU. The set of Background processes have the same priority, which is the lowest priority in the multiprocessor.

Up to 128 processes can be modelled, and each process must be in one of the following states:

– running on a CPU
– making a call to the process allocator
– interrupted, awaiting to resume service
– suspended, awaiting to begin service
– blocked, awaiting the arrival of a task.

A process can be interrupted while running on a CPU, but not while calling the PA. A process must be in the blocked state when it has no tasks to serve, and on the arrival of a task, the process is placed in the suspended state to await scheduling to run on a CPU by the PA.

A task is generated in the model either by the PA as a timing task or for each process occurrence in the application program sequences. Each task is identified with a process and enqueued in that process queue. The limit to the number of tasks in any one queue is the maximum number of tasks allowed in the common storage area for all tasks. Associated with each task is the following set of parameters:

– the priority of the tasks, in the range 1-16. When a process runs, it serves the tasks in its queue in priority order and on a FIFO basis for each priority level.
– the minimum priority of the task which can next be served by the process. This inhibits the process from running the next task should its priority be less that the minimum priority, and the process can only resume running when a task of at least the minimum priority arrives.
– the mean service time of the task, and its service distribution, whether constant or negative-exponential.
– the length, in words, of the task. This length determines the amount of PA time required to enqueue the task, and the task transmission time between clusters in a multicluster configuration.
– whether the task is intra- or intercluster in a multicluster configuration.

4.3 Processor load control

The purpose of processor load control (PLC) is to regulate the work performed by the multiprocessor so that, under call-attempt overload or failure conditions, the number of tasks within the process queues does not build up to excessive levels. Should the total number of tasks exceed the maximum allowed in the common tasks storage area, the consequences would be dire because the multiprocessor would cease handling any work at all and would need to be restarted.

The multiprocessor uses a dual scheme to perform load control. First, workload limits are used to control the acceptance or rejection of new calls (represented by sequences in the application program) such that the maximum number of calls in the setup phase at any instant does not exceed a given workload limit. A monitoring period of approximately 5 s is specified (the periodicity is adjustable) and in each period information is gathered on the CPU occupancies, the calls accepted, the calls rejected and the maximum number of calls in the setup phase. Based on this information, new workload limits are calculated for the next monitoring period to ensure that the average CPU occupancy does not exceed a specified value, say 0·9.

The second part of the scheme involves thresholds on the process queues and the common task area. If the number of tasks within a process queue, or the common task area, exceeds a preset, upper threshold, an overload is indicated. End of overload occurs when the number of tasks is reduced below another preset, lower threshold. There exists a degree of flexibility as to the action invoked when an overload is in progress, but, generally, a process queue overload causes all new calls to be rejected for its duration, and a common task area overload causes the input of tasks from the peripheral units to be inhibited.

This dual load control scheme was largely designed using the model. Its mode of operation is such that, under call-attempt overload, regulation of the CPU occupancy enables the acceptance or rejection of new calls to be controlled in a stable and smooth manner. In more severe circumstances, such as a failure condition, the thresholds provide direction protection against an excessive number of tasks in a process queue or the common task area.

4.4 Application program

The application program is the full set of sequences, where each sequence describes the process activations required for the multiprocessor to perform a particular action, for example, setting up a telephone call. Each process activation is initiated by a task, which is inserted into a queue for that process, so that a sequence progresses from start to finish as a series of tasks handed to, and then served by, the processes in the sequence. A sequence can have the additional features

– a time interval can be specified between any two process activations. A time interval can be either constant or negative-exponentially distributed, and can

be used, for example, to represent the delays between receiving dialled digits from subscribers, or the time taken by the peripheral unit to perform an action as instructed by the multiprocessor
- parallel branches can be specified, so that a sequence can have more than one process activated at the same time. There is no limit to the number of parallel branches, but all branches must finish before a sequence can terminate
- process activations can be conditional. A number of parallel branches can terminate at a process activation, which cannot proceed until all these branches have finished
- an interrupt can be generated to the PA to force the immediate running of a process, if possible
- any number of response times can be specified at any points within a sequence.

When a peripheral unit wishes to communicate with a process, the unit may either generate an interrupt to the PA or wait to be periodically interrogated by the process. Further, the information to be communicated may reside in a buffer, which is external to the multiprocessor, or within its main memory. These possibilities are all modelled as a normal process activation within a sequence, except that the task handed to the process queue is identified as being from external hardware (a peripheral unit) with the following options:

- whether an interrupt is to be generated to serve the task, or the task must await the process to periodically scan its queue
- whether or not the PA is to be invoked for handing the task to the process
- whether or not direct memory access (DMA) is to be used for the task.

These options allow the specific types of peripheral unit communication to processes to be modelled. Communication from processes to peripheral units may be either via a buffer in main memory or direct to the unit. Whichever, the model assumes that the time taken to do this will be included in the service time of the relevant tasks.

Sequences in the application program need not just represent successfully set up and cleared down telephone calls. Rejected or mishandled calls, calls meeting engaged tone or no answer etc. can just as easily be modelled. Sequences can also be used to represent ancillary functions such as updating subscribers' meters, collecting management statistics, handling failure conditions in the peripheral units etc.

5 *Conclusions*

This paper has described a simulation model of a multiprocessor, designed to be the control processor in a telecommunications exchange. Details of the multiprocessor and the work it performs in handling telephone calls are specified entirely by the data input to the model. This allows the user of the model to specify easily, and at will, the type of work for which the performance of the multiprocessor is being analysed.

The model has aided considerably the design of the operating system, especially the process allocator and processor load control. The model has also demonstrated that a telecommunications exchange, controlled by such a multiprocessor, can survive severe call-attempt overload and failure conditions, which cannot be readily simulated on live equipment. Recent measurements on a real system when stimulated by artificial call generators have confirmed the accuracy of early model predictions of the multiprocessor call handling capacity.

Acknowledgments

Acknowledgment is made to the Director of Systems Evolution & Standards of British Telecom for permission to publish this paper. Acknowledgment is also made to colleagues in the Teletraffic Division who made significant contributions to the development of the Simulation Model, in particular Mrs J.W. Davies, Mr A.G. Bruce and Mr A.P. Keogh.

Tracking of LSI chips and printed circuit boards using the ICL Distributed Array Processor

D.J. Hunt
ICL Technical Directorate (Systems Strategy Centre), Stevenage, Hertfordshire

Abstract
Automatic generation of interconnection tracks on LSI chips and circuit boards is a major computation task in design automation. The ICL Distributed Array Processor (DAP), a parallel processor having a few thousand processing elements operating in parallel on a common instruction stream, has been applied very successfully to this problem using the well known Lee algorithm.

1 Introduction

Today's logic designers make extensive use of computer power to help them in their design task, the extent of their problem being apparent when it is realised that a single VLSI chip can have a complexity equivalent to that of several printed circuit boards of just a few years ago. Computers can not only provide the means for capture and storage of a design, but also perform computation-intensive tasks such as simulation, placement (physical mapping of logical elements), tracking (insertion of metal interconnections) and automatic generation of test procedures. The amount of processing needed, and hence the execution time, for the latter group of operations often grows as the square or even the cube of the circuit complexity (measured for example as the number of gates) and hence the time may become excessive.[1]

The use of structured design techniques at the logical or physical level can reduce the processing time, as can developments in algorithms, but there is still considerable demand for processing power. One way of achieving such power is to use a parallel processor; the work described here uses the ICL Distributed Array Processor (DAP). Initial work has concentrated on tracking, but DAP is expected to have a role to play in many other related activities.

2 Principles of DAP

The architecture and principles of this single-instruction-stream/multiple-data-stream (SIMD) machine have been described in several papers.[2] The model used in chip tracking has 4096 processing elements (PEs) arranged 64 x 64, each having 4096 bits of memory, giving a total of 2 Mbytes for the array. Most programming is performed in DAP Fortran[3,4], a Fortran extension incorporating as basic elements matrices and vectors that match the DAP size, as well as scalars; and an assembly language Apal is also available.

The main factors that make DAP suitable for tracking work are its Boolean processing capability and the fact that the two-dimensional array of PEs can be matched to the regular grid on which tracking is performed. The 2 Mbyte storage capacity permits chips with over one million grid points to be represented entirely in DAP memory. Experience shows that excellent performance can be achieved as well as simple mappings and easily understandable programs.

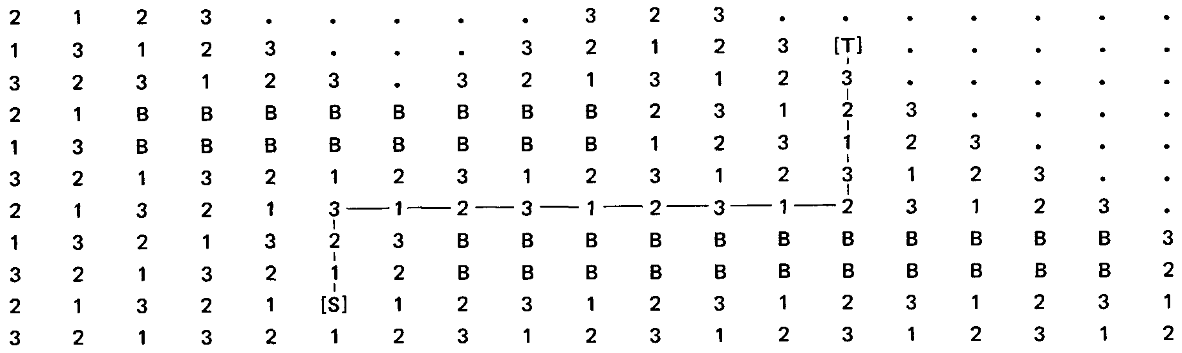
3 Description of tracking algorithm

The Lee tracking algorithm is very well known[5]. Its principle is illustrated in Fig. 1 for a simple single-layer maze, showing the search for a path from source cell S to target cell T when certain cells marked B are blocked. The first step is to mark all free neighbours of S with the value 1 ('free' meaning that the point has neither been reached before nor is blocked). Then free neighbours of those points are marked 2, and so on, values 1,2,3,1, ... being inserted cyclicly. If a path is possible, then eventually the target point will be reached as in the Figure and the number of steps taken is the path length; the algorithm ensures that the shortest path is found.

Having established that a path is possible, a particular path may be marked as shown by starting from the target and following the cyclic sequence 3,2,1,3, . . . (beginning with whatever value is appropriate to the destination) from cell to neighbouring cell. In general, many paths are possible with the same length, the normal procedure being to continue tracing the path in the current direction until a change is found to be necessary. The use of three values for temporarily marking cells is sufficient to enable a minimum length path to be identified by this means.

In a multilayer maze an individual cell may be blocked on any combination of layers, and certain transitions between layers may be prohibited. Thus, it is necessary to mark the 1,2,3 values separately on each layer. Multiple layers are considered in pairs, the Lee algorithm being used to search only for East-West paths on one layer (X) of each pair and only for North-South paths on the other (Y). In chip or board tracking the blocked areas may be blocked to all tracks (for example power lines), or be previously tracked metal. Normally, many points are to be connected together to form a string, or net. This is achieved by tracking individual links to join each pin in turn (starting with the second pin) to the earlier pins in the net. In chips it is usual for the second and subsequent links of a net to be allowed to join onto previously tracked metal of the same net.

A serial implementation of the forward search might maintain a list of currently



2 1 2 3 . . . . . 3 2 3 . . . .
1 3 1 2 3 . . . 3 2 1 2 3 [T] . . . . .
3 2 3 1 2 3 . 3 2 1 3 1 2 3 . . . . .
2 1 B B B B B B B B 2 3 1 2 3 . . . .
1 3 B B B B B B B B 1 2 3 1 2 3 . . .
3 2 1 3 2 1 2 3 1 2 3 1 2 3 1 2 3 . .
2 1 3 2 1 3 1 2 3 1 2 3 1 2 3 1 2 3 .
1 3 2 1 3 2 3 B B B B B B B B B B B 3
3 2 1 3 2 1 2 B B B B B B B B B B B 2
2 1 3 2 1 [S] 1 2 3 1 2 3 1 2 3 1 2 3 1
3 2 1 3 2 1 2 3 1 2 3 1 2 3 1 2 3 1 2

Fig. 1 The Lee tracking algorithm

active points (i.e. those points that were reached for the first time in the previous step) as well as the matrix structure implied by Fig. 1. The next active list is simply the accessible neighbours of points in the current active list.

4 Mapping the problem onto DAP

A hypothetical chip that was small enough to be mapped directly onto the DAP array could clearly be tracked using a very simple DAP program. Separate logical matrices could be used to represent source and target regions, blocked regions, values 1,2,3 and so on. The element-by-element Boolean operations and neighbour shifting available on DAP permit simple expression of all the required functions. In a sense, at any given step only those PEs on the 'active wavefront' (corresponding to points on the active list of a serial program) are doing useful work. However, essential structural information is held in the pattern of that wavefront and the structure is being implicitly handled by the Boolean and neighbour operations. Real chips are very much larger than the DAP dimensions, and this necessitates changing the code in two ways.

First, any given Boolean matrix must be partitioned into sections or sheets that match the DAP size. Although every sheet could be processed at every step, this would clearly involve much unnecessary work. Thus a list of those sheets containing active points is used in a manner similar to the active point list in a serial program.

Secondly, a straightforward allocation of one logical matrix to each necessary state as mentioned above results in a total of about 20 such matrices being needed for two-layer chips or boards, but to accommodate the largest chips in DAP memory it is necessary to code the state of the chip more compactly. In the current program a 9-bit code is used: four bits for each of the two layers and one bit for via information (see Section 5). A decode is performed prior to processing each sheet, but the code is chosen to make this process very simple.

Some details of a program written for tracking a large uncommitted logic array are given in the following sections, concentrating on the parts where most time is spent. The program can run on different sizes of DAP, and accepts as parameters the chip dimensions, adjusting its storage structure accordingly.

5 Forwards scan

The forwards or expansion phase, during which codes 1,2 and 3 are inserted, accounts for most of the execution time, as in a serial program. The DAP code for expansion on a single sheet is considered first; example code is given for the X layer, the Y layer being similar. It is assumed that in DAP-Fortran the 9-bit code for the sheet has been decoded to give logical matrices XLEVEL1, XLEVEL2 etc, and VIAFREE is a matrix which is True where propagation through a via (i.e. a path between two tracking layers, X and Y in this case) is allowed. One step of expansion can be written:

```
XNEW = SHEP (XLEVEL2) .OR. SHWP (XLEVEL2)
                       .OR. (YLEVEL2 .AND. VIAFREE)
```

Here SHEP is the DAP-Fortran function resulting in a Shift East with Plane geometry (i.e. value False being shifted in at the West edge).

Then the points at which the target has been hit, if any, can be identified:

```
XHIT=XNEW .AND. XTARGET
```

A rapid test for completion of this link can then be applied:

```
IF ( ANY(XHIT .OR. YHIT)) GOTO 500
```

The function ANY returns a logical scalar which is the 'OR' function of all bits in its logical matrix argument; this result is then used as the argument of a conditional statement in the normal way.

Finally, the next level marker is inserted, taking into account only those points to which expansion is permitted:

```
XLEVEL3(XFREE) = XNEW
```

This statement is a matrix assignment of XNEW to XLEVEL3, but matrix XFREE which is True at points that have not yet been used acts as a mask, so the assignment is only performed where its element has the value True.

In practice all the sheets that include part of the active wavefront must be processed in this manner, and the active points transmitted across sheet boundaries. To deal with these boundaries each entry in the sheet list includes not only the sheet address, but also four Boolean vectors specifying the inputs at the four edges of the sheet. In fact there are two lists: the 'current' list and the 'next' list. Each step of expansion deals with each sheet in turn in the current list, incorporating the edge vectors in the shifts. The next list is initially empty, but as active points are identified at the sheet edges the address of the appropriate neighbouring sheet is added to the list if necessary, together with the Boolean edge vector. The current sheet is also added to the next list unless it contains no active points. At the end of the current list the next list is copied into the current list; if that list is empty than no path is possible for the link being attempted.

6 Backwards scan

Having established as above that a path is possible from a given pin to a target, the metal is placed by following the sequence of level values 3,2,1,3,2, ... back from the hit point. This is done sheet by sheet, and although a simple method is used it is much faster than Lee expansion because of the smaller number of sheets crossed.

For a given sheet the first step is to evaluate in parallel the permitted directions of travel from every point on the sheet. This involves logical functions of matrices XLEVEL1 etc., and their values shifted one place. The path is followed sequentially using these permitted directions. Where alternative paths exist the current direction of travel is continued whenever possible; otherwise an arbitrary selection is made. When the start point or the sheet boundary is reached the entire path section identified on that sheet is incorporated in parallel into the 9-bit code. This involves setting codes for via, metal and target, the last of these permitting the track to be a target for a subsequent node of the same net.

7 Other tracking operations

Prior to performing any tracking it is necessary to initialise the DAP representation of the chip with codes for pins, blocked areas and metal, perhaps incorporating manually specified tracks. Each such region is built up from a set of rectangles, and DAP-Fortran provides functions to facilitate the generation of these patterns. For example:

```
XMETAL = XMETAL .OR. (COLS(X1,X2) .AND. ROWS(Y1,Y2))
```

Integer scalars X1, X2, Y1, Y2 here define the limits of the required rectangle within a DAP sheet. The function COLS returns a logical matrix which is True in columns X1 to X2 inclusive and False elsewhere; the function ROWS deals similarly with rows.

After tracking, DAP generates descriptions of the tracks in a form suitable for mask making. It can also identify any particular metal on the chip by propagating along tracks from the specified point until the pin is reached.

The 2900 host system is responsible for holding the data structures that represent the initial state of the chip, the interconnections that are to be made and the metal generated by the DAP program. Relevant parts of the database are passed to and from the DAP as required.

8 Results and performance

The program described has been used to track uncommitted logic array chips having up to 8000 logic gates and nearly 1·4 million grid points.[6] Performance is of course very dependent on the number of gates used, the complexity of interconnections and the chip layout. Figures for two example chips are given below, the execution times given being only for tracking.

	Design A	Design B
Number of nets	2200	2400
Metal length (number of grid units)	790 000	930 000
Tracking time (s):		
DAP-Fortran	1700	2100
Apal and DAP-Fortran	410	470

Note that a net may typically contain four pins and hence require three tracking operations; some nets are much more complex than this. Some nets are very long (in terms of the grid length), and it is these that make the major contribution to runtime, since the time for finding a path depends on the sum of the number of active sheets at each step of expansion.

The times quoted are first for a program written entirely in DAP-Fortran, and secondly with just the forwards scan converted to assembly code. The performance advantage of using Apal is obvious: factors of 4·1 and 4·5 overall for the two examples. The advantages are greater for long nets because the fraction of executed code that is converted to Apal is greater.

Comparisons with serial implementations of the same algorithm are more difficult, partly because both codes are subject to alteration and improvement from time to time, and partly because the tracking time given is only one section of the overall design process. Suffice it to say that execution times of a few hours have been needed for these examples on serial mainframes.

9 Comments and conclusions

DAP has proved very effective at tracking the chips mentioned above. The method is, of course, applicable to other tracking cases, such as printed circuit boards, and a number of similar programs have been written to take account of factors such as extra pairs of *XY* layers and variants in the detailed rules specifying which configurations of wiring are permitted.

The Lee algorithm is perhaps the most natural for DAP, but other techniques such as line search and channel allocation are often used on serial machines. No detailed work has been done on these, but similar styles of implementation on DAP are likely to make their use very effective too.

Looking to the future, there are a number of other aspects of design automation that seem worthy of investigation with a view to DAP implementation. The DAP principle is of course applicable to machines smaller than the current 64 x 64 array and is well suited to LSI implementation; a DAP with, say, one-quarter the number of processing elements could achieve about half the performance of the larger machine on the inner work of tracking, for a given technology of implementation. The compactness of such a small DAP together with its high performance could make it a powerful and cost-effective component of a more interactive design automation system.

Acknowledgment

The author is indebted to many colleagues at ICL Manchester for the provision of host software and interfacing to existing design automation systems, without whom the DAP tracking of real chips would have been impossible.

This paper was first presented at the conference Parallel Computing '83. It is reproduced here with the permission of the North-Holland Publishing Company.

References

1 ADSHEAD, H.G.: 'Towards VLSI complexity: the DA algorithm scaling factor: can special DA hardware help?' Proc. 19th DA Conference (ACM/IEEE 1982), 339 ff.

2 HUNT, D.J. and REDDAWAY, S.F.: 'Distributed processing power in memory', in: *'The fifth generation computer project'* SCARROTT, G.G. (Ed.), (Pergamon Infotech, Maidenhead, 1983) 49-62.

3 FLANDERS, P.M.: 'Fortran extensions for a highly parallel processor', in *'State of the art report: supercomputers'.* (Infotech International, Maidenhead, 1979.

4 GOSTICK, R.W.: 'Software and algorithms for the Distributed-Array Processors', *ICL Tech. J.*, 1979, **1**, (2), 116-135.

5 LEE, C.Y.: 'An algorithm for path connections and its applications', *IEEE Trans.*, 1961, **EC-10**, (5), 346-365.

6 ADSHEAD, H.G.: 'Employing a distributed array processor in a dedicated gate array layout system', *Proc. ICCC*, 1982, (IEEE, Oct.) 411 ff.

Sorting on DAP

P.M. Flanders and S.F. Reddaway
ICL Technical Directorate (Systems Strategy Centre), Stevenage, Hertfordshire

Abstract

Sorting data is an important and time-consuming activity in many computer applications. The paper describes how a highly parallel processor, the ICL DAP, can be used effectively for both internal and external sorts.

1 Introduction

In many computer applications, both scientific and commercial, there is a requirement to sort data. Also, sorting may be used to implement other operations on a parallel processor such as scatter-gather, permutations, conformal mappings etc.[1]. In commercial applications the volume of data to be sorted frequently exceeds the size of main memory.

A variety of sorting algorithms have been developed, for the most part with sequential computers in mind. We consider here the application of a parallel computer, the ICL DAP[2], using an algorithm suitable for parallel processing, Batcher's bitonic sort[3]. Sorts for which all data can be accommodated in main memory (internal sorts) and those which require backing store (external sorts) are considered. Consideration is also given to the use of tag sorts, to improve performance and to handle variable length records. Examples of internal non-tag sorts of fixed length records have been implemented, but the remainder of the work is theoretical. More details of topics covered in this paper are given in Reference 4.

2 The ICL DAP

A general description of the DAP is given by Flanders *et al.*[2], and there is a short note in the paper by Hunt in the same issue of this journal as the present paper. A point of detail which is relevant to our work is the difference between what are called 'vertical' and 'horizontal' mode processing, respectively. In the former, which is the principal mode of operation, the successive bits of a data item are held in the store of a single PE, so that each PE produces one result; this has a parallelism of 4096. In the latter a data item is mapped so that a row of PEs is used to produce each result; results are produced more rapidly but the parallelism is reduced to 64. Data are routed between PEs either by shifting 'bit planes' (64 x 64 bits, one per PE) in any of the four directions or by highways which traverse the rows and columns of the PE matrix; for the former, the time taken increases with the distance shifted.

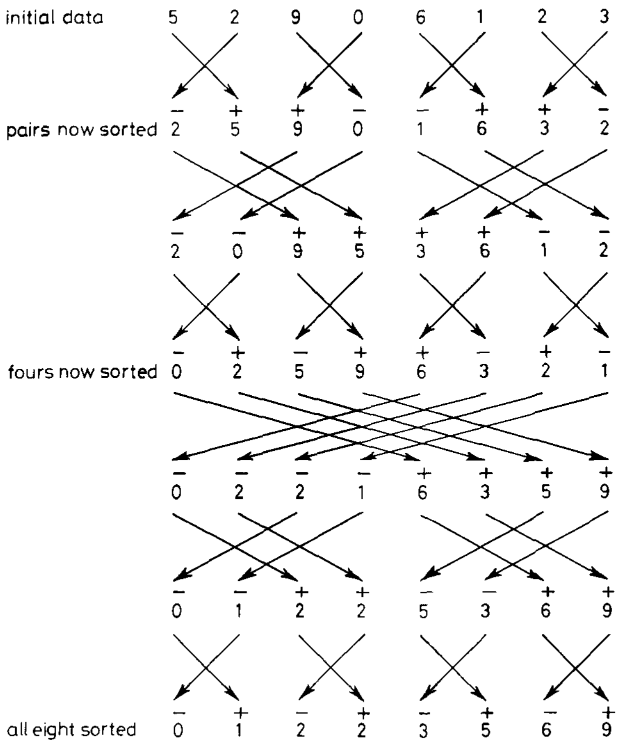


Fig. 1 Sample bitonic sort of eight records

3 Batcher's bitonic sort

Fig. 1 shows the comparisons and exchanges of data in a sample sort of eight records using Batcher's bitonic sort; a brief general description is given below for the more interested reader. In Fig. 1 data are moved as indicated at each step and compared with the data already there; the larger or smaller element, indicated by '+' or '−', is then selected.

Batcher's bitonic sort is based on the idea of a bitonic sequence; a sequence is bitonic if, when considered cyclicly, it has one ascending and one descending portion. Suppose a bitonic sequence of records, A_i where $i = 1,2,\ldots ,N$ is split into the two sequences A_1 and A_2 such that:

$$A_{1i} = \min (A_i, A_{i+N/2})$$

and

$$A_{2i} = \max (A_i, A_{i+N/2})$$

where $1 \leqslant i \leqslant N/2$ and records are compared using the record keys, then it can be shown that A_1 and A_2 are also bitonic sequences and all records of A_2 are greater than or equal to all records of A_1. Thus, if a bitonic sequence of N records is split into two sequences in this way and the resulting sequences are again split, and so on, the result is ordered after $\log N$ steps (all logarithms here are base 2.) This enables a bitonic sequence to be ordered by successive splitting; the ordering may be increasing or decreasing.

To sort an arbitrary set of N records the algorithm to order a bitonic sequence is used as follows. First, consider the N records as $N/2$ sequences, each containing two records. Each sequence is trivially bitonic and is ordered as above; half the $N/2$ sequences are put into ascending order and half into descending order. The $N/2$ sequences are then taken in pairs, one ascending and one descending sequence in each pair, so that each pair is a bitonic sequence of four records. This gives $N/4$ bitonic sequences each having four records; the number of sequences has been halved and the number of records in each sequence has been doubled. Repeating this procedure $\log N$ times leaves the data sorted.

4 Use of DAP for internal sorts

At each step of the algorithm to sort N records there are $N/2$ 'comparison-exchanges' all of which can be performed in parallel. Each of these involves comparing two records and conditionally exchanging them depending on the result of the comparison. In a complete sort there are $\frac{1}{4}(N \log N\ (1 + \log N))$ comparison-exchanges, that is about $\frac{1}{4}\log N$ more basic operations than in a conventional serial sort. However, the Batcher algorithm has the advantage that it is readily implemented in parallel on the DAP with each PE performing one comparison-exchange, for vertical mode processing. For sorts where the number of records is at least twice the number of PEs (or twice the number of rows for horizontal mode processing) the parallelism of DAP can be utilised to the full.

On DAP the conditional exchanges are executed in parallel using activity control; this may be contrasted with implementation on a serial machine using conditional jumps. The time taken on DAP is independent of the initial ordering of data.

Before each stage of comparison-exchanges can be executed, records must be paired up in the same PEs. In general this requires data to be routed among the PEs, and this is done using parallel techniques which shift and merge bit-planes. Unless care is taken this routing can dominate the execution time. Much attention has been given to this in the literature; however, most published algorithms concentrate on the special case where the number of records equals the number of PEs[5, 6]. The more general problem where there are more records than PEs has been studied by Baudet and Stevenson[7] but they use an algorithm involving more comparisons than Batcher's bitonic sort to reduce routing; they also assume independent addressing in each PE is possible.

The algorithm for DAP for the general case where there are more records than PEs is described more fully in Reference 4; it achieves high performance by:

(i) ensuring that all PEs are active and perform distinct comparison-exchanges
(ii) taking into account, in the mapping of data on to the store, the relative frequency of comparison-exchanges at different separations
(iii) allowing the mapping of data to vary during the sort. This almost halves the routing required for an optimal static mapping. It leaves the data in an unnatural order but the overhead to restore natural ordering is small.

With reference to (ii), the data are sorted first within PEs, then over the PE plane starting with nearest neighbours in both directions and moving on to the longer routes. The resulting complexity in data organisation and movement is best handled using the 'mapping vector' techniques described in References 8 and 9. These provide a systematic and readily implemented means of data organisation.

The time taken to sort N records has a contribution from the comparison-exchanges which is proportional to $N \log N$ and a contribution from routing proportional to N. For practical values of N on DAP the contribution from these is of the same order. An estimate of performance is given below for a vertical mode sort of 128 000 records each 8 bytes long; all 8 bytes are used in comparing records. For the sorts implemented on DAP, actual times were typically 10% greater than estimated. Rates of processing data are given in Mbyte s^{-1} to allow easy comparison with disk transfer rates. Figures are given for a DAP having 32*32 PEs; performance on the smaller DAP does not fall in line with the reduction in the number of PEs since routing is less dominant.

	64*64 DAP	32*32 DAP
Comparison-exchanges	536 cycles/bit-plane	536 cycles/bit-plane
Routing	743 cycles/bit-plane	424 cycles/bit-plane
Number of bit-planes	2048	8192
Total machine cycles	$2{\cdot}6 \times 10^6$	$7{\cdot}2 \times 10^6$
Total time at 200 ns/cycle	0·53 s	1·44 s
Rate of processing data	1·9 Mbyte s^{-1}	0·7 Mbyte s^{-1}

5 Tag sorts and variable-length records

If the sort key is a small part of the record, better performance is achieved by using a 'tag sort'. Tags comprising the record key and record address are formed for each record and these are sorted instead of the records themselves. The sorted tags are then used to order the complete records, thereby moving each record once only. This improves performance of the sorting process by a factor roughly equal to the ratio of the record length to the tag length; it does, however, incur the additional costs of tag formation and record movement.

Moving records according to the sorted tags makes effective use of the ability of the DAP to move whole bit-planes of data at a time. The shifting required to produce a new alignment for the moved record is noticeably more with larger DAPs; also the larger number of bits in a bit-plane is only useful for records larger than 128 bytes, and so the smaller DAP may actually be faster. Performance for tag sorts is in many cases dominated by record movement and to a lesser extent by tag formation.

Tag sorts give an effective means of sorting variable-length records. It is assumed that each record contains a byte count for the record and that the key fields are fixed length and at fixed positions relative to the start of record. Despite extra work in forming tags and moving records, DAP is still effective.

Estimates are given below for a horizontal mode tag sort of 32 000 100 byte records having 4 byte keys; figures are given for fixed and variable length records, with 100 bytes being the average length in the latter case. Unless stated otherwise figures are in cycles per bit-plane of record data; the cycle time is 200 ns.

Fixed-length records	64*64 DAP	32*32 DAP
Tag formation	70	21
Sort tags	120	78
Move records	260	29
Total	450	128
Rate of processing data	5·7 Mbyte s^{-1}	5·0 Mbyte s^{-1}

Variable-length records	64*64 DAP	32*32 DAP
Tag formation	200	43
Sort tags	120	78
Move records	410	74
Total	730	195
Rate of processing data	3·5 Mbyte s^{-1}	3·3 Mbyte s^{-1}

6 External sorts

Sorting an amount of data too large to be held in the DAP store is done in two parts, a prestring stage followed by one or more merge passes. The prestring stage forms sorted strings, each of as many records as can be sorted in memory, and is implemented as a sequence of internal sorts. The merge passes combine these strings to form progressively fewer sorted strings until just one string is obtained. The algorithm for a two-way merge pass is similar to that for a conventional two-way merge except it merges blocks using a bitonic merge rather than comparing single records. At each step the next block of records is taken from one of the strings (for ascending order, the one whose next block contains the smallest key) and merged with the 'current block'; the half containing the smallest keys is then output and the remainder left in store as the next 'current block'.

To prove that this algorithm works it is sufficient to show that after each merge the records output from the merge area all have keys less than or equal to those of all records in the input strings (for sorting in ascending order). Clearly this is true for the string from which the last block was selected. We denote this string by S_1, the block selected from it by B_1 and the smallest key in that block by K_1; we denote the other string by S_2, its next block by B_2 and the smallest key in B_2 by K_2. Before block B_1 was selected for merging, the records left in the merge area from the previous merge must have all been less than or equal to the larger of K_1 and K_2; however, since B_1 was chosen in preference to B_2 we know that $K_1 \leqslant K_2$ and consequently all these records must have keys less than or equal to K_2. The result follows immediately.

The merging of two blocks, sorted into ascending order, can be achieved by reversing the order of one of the blocks so that together they form a bitonic sequence. In practice it is not necessary to explicitly reverse the order since the bitonic merge can easily be modified to give the first half of its result in ascending order and the second half in descending order; when the first half is output it is replaced with another block in ascending order so that the two halves to be merged are in opposite order.

The above describes a two-way merge; a number of strings can be merged by forming a binary tree of two-way merges within the DAP memory. The output of each merge at a given level of the tree is input to the next level. Merging a number of strings not equal to a power of two can be done by placing the input from some strings higher up the tree.

For an m-way merge a block of N records is output after each stage of log m two-way merges. The work per two-way merge is approximately proportional to $N(1+\log N)$, which implies that to achieve high performance the block size must be as small as possible while allowing effective use of DAP parallelism. A smaller block size also helps reduce storage requirements. We therefore assume that on the 64*64 DAP, horizontal mode processing is used with a block size $N = 64$. Merge performance varies somewhat with the record parameters, but a typical performance for a 32-way non-tag merge is as follows:

Example of merge performance (32-way merge)

	64*64 DAP	32*32 DAP
Block size (records)	64	32
Number of steps in block merge	7	6
Cycles per merge per bit-plane output	320	220
Total cycles per bit-plane output	1600	1100
Rate of processing data (Mbyte s^{-1})	1·6	0·58

Tags can be used to handle variable length records and, where appropriate, to achieve performance improvements similar to internal sorts. A tag is formed for each record on input and the tags are merged rather than the complete records, which are left in input buffers. As tags emerge from the merge tree the corresponding records are moved to the output buffers.

With the high speeds possible for both prestring and merge stages, disk transfer rates may be the limiting factor in overall performance. The availability of a large DAP memory helps in allowing large disc block sizes which maximise transfer rates.

7 Conclusions

The DAP can be used for high-performance sorting on both internal and external sorts. Tag sorts can be used to improve performance where relevant and to handle variable-length records. Ancillary operations of tag formation and record movement can be effectively implemented on DAP hardware. The performance of smaller DAPs is much better than would be expected from comparing array sizes.

Acknowledgment

This paper was first presented at the conference Parallel Computing '83. It is reproduced here with the permission of the North-Holland Publishing Company.

References

1 SCHWARTZ, J.T.: 'Ultracomputers', *ACM Trans. on Programming Languages & Systems,* 1980, 2 (4).

2 FLANDERS, P.M., HUNT, D.J., PARKINSON, D. and REDDAWAY, S.F.: 'Efficient high speed computing with the Distributed Array Processor', Symposium on High Speed Computer and Algorithm Organisation, University of Illinois, Academic Press, 1977.

3 BATCHER, K.E.: 'Sorting networks and their applications', *Proc. AFIPS*, 1968, *SJCC*, **32**, Montvale, New Jersey (AFIPS press), 307-314.

4 FLANDERS, P.M. and REDDAWAY, S.F.: 'Sorting on DAP', SSC Report CM72, International Computers Ltd., Stevenage, UK (Oct. 1982).

5 NASSIMI, D. and SAHNI, S.: 'Bitonic sort on a mesh-connected parallel computer', *IEEE Trans.*, 1979, **C-28**, (1).

6 THOMPSON, C.D. and KUNG, H.T.: 'Sorting on a mesh-connected parallel computer', *CACM,* April 1977, **20**.

7 BAUDET, G. and STEVENSON, D.: 'Optimal sorting algorithms for parallel computers', *IEEE Trans.*, 1978, **C-27** (1).

8 FLANDERS, P.M.: 'A unified approach to a class of data movements on an array processor', *IEEE Trans.*, 1982, **C-31** (9).

9 FLANDERS, P.M.: 'Languages and techniques for parallel array processing', Ph.D. Thesis, Dept. of Computer Science, Queen Mary College, University of London (July 1982).

User functions for the generation and distribution of encipherment keys

R.W. Jones
ICL Technical Directorate, Bracknell, Berkshire

Abstract

It is generally accepted that data encipherment is needed for secure distributed data-processing systems. It is accepted, moreover, that the enciphering algorithms are either published or must be assumed to be known to those who wish to break the security. Security then lies in the safekeeping of the encipherment keys, which must be generated and stored securely and distributed securely to the intending users.

At an intermediate level of detail of a system it may be useful to have functions which manipulate keys explicitly but which hide some of the details of key generation and distribution, both for convenience of use and so that new underlying techniques can be developed. The paper offers a contribution to the discussion. It proposes key-manipulation functions which are simple from the user's point of view. It seeks to justify them in terms of the final secure applications and discusses how they may be implemented by lower-level techniques described elsewhere. The relationship of the functions to telecommunications standards is discussed and a standard form is proposed for encipherment key information.

1 Introduction

It is generally accepted that data encipherment is needed for secure distributed data-processing systems. It is accepted, moreover, that the enciphering algorithms are either published or must be assumed to be known to those who wish to break the security. Security then lies in the safekeeping of the encipherment keys, which must be generated and stored securely and distributed securely to the intending users. A number of schemes have been proposed, and in some cases implemented, to manipulate keys securely. For example References 1, 2 and 3 describe different methods and offer different but overlapping sets of facilities to the user. It is likely that new methods will be developed and that some part of these methods should be hidden from the user. Since the subject has clearly not reached a stable point it is very likely that any attempt at present to establish a standard user interface will soon need revision. Nevertheless, this paper is written on the assumption that a discussion of such an interface is useful, since it helps to identify the common features of different schemes and to gain some idea of which features will become generic and which become part of the underlying mechanisms.

At some level users do not concern themselves with the manipulation of keys or with explicit commands to encipher and decipher data. They ask for a secure connection to another user or for a securely stored file and can assume that such details are thereby taken care of. At a lower level software and hardware logic exists which deals with things such as how keys are generated, how data encipherment keys and key encipherment keys are kept distinct and the manner of transporting a data encipherment key to a remote user.

At an intermediate level of detail it may be useful to have functions which manipulate keys explicitly but which hide some of the details, both for convenience of use and so that new underlying techniques can be developed. This paper discusses this intermediate level. In doing so it must make assumptions about which functions are primitive at this level. For example, since a digital signature may be achieved by enciphering a message digest, using the secret member of a public key pair, one might decide that it is an application to be programmed in terms of encipherment primitives and does not give rise to specific primitive operations. This view is invalidated by signature techniques which do not depend upon encipherment. Similarly there is implicit in such an interface a judgment of which of the details should be hidden. Reference 4 describes a key-distribution centre. In an appropriate context software at some level submits a request to a key-distribution centre (KDC) for a key which can be used to communicate securely with an intended correspondent. We may wish to produce software which needs no modification when moved from such an environment to one where the system supporting the application user keeps records to enable it to issue keys securely to all members of the community. If this is so we should hide the use or nonuse of the KDC, but we judge in doing so that the user at that level has not lost needed flexibility. Such judgments as these are made in what follows and the reasons for them are discussed.

2 The functions

This section describes a set of functions to generate and manipulate keys. The intention is that they appear simple to the user. The user is somewhat ill defined, but well enough, it is hoped, for the benefit of the discussion. One candidate is certainly an application process which makes use of an application service as defined in the open systems interconnection model[5] and which wishes to perform explicit data encipherment. Another candidate is the logic of a transport layer entity in the open systems interconnection model which offers a secure service to users of the transport service and which, therefore, sends a data-enciphering key to a remote transport entity. The functions are as follows.

2.1 Generate key (t, s)

Generate key (t, s) means generate for me a key or a pair of keys of type t and return to me, as the result of this function, the local name of the item containing the key or keys. The type shows, among other things, whether a symmetric or asymmetric algorithm is involved. In the former case a single key is generated and returned as the result of the function. In the latter case the enciphering and deciphering pair is generated and returned. The local name is subsequently used subscripted by 1 or 2 to indicate an individual member of a key pair thus generated or unsubscripted to mean the single key generated or the complete item containing the key pair. s is a 64-bit string, supplied by the caller, which is to be used by the key-generation function. The caller does not know the cleartext value of the key generated but is assured that the same t and s values in a subsequent call will generate the same key or keys. s may be omitted, in which case the values generated, as far as the caller is concerned, are random. The chance of generating them again is random. The type t is an integer. Possible meanings assigned to its values are:

- a key-enciphering key (KEK) for DEA1
- a data-enciphering key (DEK) for DEA1
- an RSA key pair to be used for enciphering keys.

Other meanings, to which values might be assigned, are discussed in Section 3.

NB. This function and the next two have a result. The assumption is that the users have notations which enable them to write something like

$x :=$ generate key (y, z)

The variable which is to hold the result could be written as another parameter. This is a matter of taste.

2.2 Give key (k, q)

Give key (k, q) means send my key whose local name is k securely to the user known to me as q. Assign to the key a common reference number which we may use in messages to each other and in communicating with our local encipherment services (of which this function forms a part). Make the reference number available to q and return it to me as the result of this function.

NB. The exact manner of making it known to q that the key is available is not considered here. In an implementation it would not be a trivial issue. Similarly, although we may assume that the services at the users' locations acknowledge receipt to each other, there is need to consider whether the end user should do so as well. The assumption here is that if this is done it is separate from the basic functions needed for key distribution.

2.3 Mutual key (t, q, s)

Mutual key (t, q, s) means generate a mutual key for me and user q, use seed s and give the key type t. t and s are as in 'generate key'. s may be omitted to obtain a random key. Assign to the key a common reference number and make it available to q and return it to me as the result of this function.

2.4 Take key (r, q)

Take key (r, q) means make the key whose reference number is r unavailable to user q.

2.5 Destroy key (K)

Destroy key (K) means destroy the key identified by K. K may be a local name of a key, created by 'generate key' or a reference number created by 'give key' or 'mutual key'.

3 Use of the functions

This section considers the functions of Section 2 in the light of applications of encryption and related techniques.

3.1 Connection establishment and user authentication

When establishing a connection between two users so that they may exchange messages protected by encryption (for example if they use an insecure telecommunications link) both users (or their local services) must be provided with a key and the users must be authenticated to each other's satisfaction. 'Give key' and 'mutual key' may both be used to send a key to a remote user (the reason why both exist is discussed in Section 4). A reasonable requirement of either of these functions is that it delivers the key, guarantees to the initiator that the recipient is the user requested, tells the recipient from whom the key came and guarantees that they, in turn, are who they claim to be, i.e. not just a legitimate user of the service. This is illustrated in Fig. 1., where A is one of a number of users of the A service and B is one of a number of users of the B service. The A service is used by A in a controlled environment in which the identity of A is assured (for example the process which represents A has been initiated after the submission of a password to a control program which controls access to resources, one of which is the A service). B has the same relationship to the B service. The route between the A service and the B service is assumed to be insecure in the absence of encipherment.

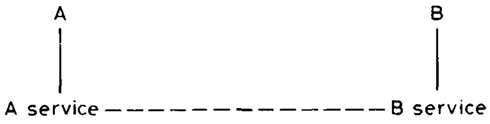


Fig. 1 User communications via encipherment services

After receiving a request from A to deliver a key to B, the A service, having discovered the route, sends it to the B service, suitably enciphered by a KEK. The A service and the B service must authenticate each other. Their manner of doing this depends upon a number of factors, including whether a KDC is involved and whether the KEK is a public or secret key. Methods are discussed, for example, in References 4 and 6. For example, Reference 4 describes protocols for sending a DEA1 key, first when it is protected by DEA1 encryption and secondly when it is protected by public key encryption. In both cases the protocol is described in terms of a user A who wishes to send a key to another user B, with the aid of a KDC (see Fig. 2).

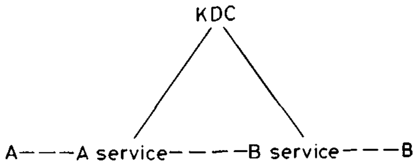


Fig. 2 Use of key-distribution centre

In the first case the protocol has three logical parts, namely:

(i) A obtains securely from KDC two copies of the key, one enciphered by A's KEK and the other enciphered by B's KEK.
(ii) A sends to B the copy enciphered by B's KEK
(iii) A and B use the key to exchange authentication protocol.

In the second case the protocol has four logical parts, namely:

(i) A obtains securely from KDC B's public key and the key to be used.
(ii) A sends to B the key enciphered by B's public key.
(iii) B obtains securely from KDC A's public key.
(iv) A and B exchange authentication protocol.

(For details of the values exchanged to cope with particular security problems see Reference 4.)

Either of these methods may be hidden from the users at the level proposed for them here. The appropriate interchanges are initiated by the function 'mutual key'. A possible improvement in underlying protocols to remove as yet unknown security flaws is also hidden from them.

Once the two services have authenticated each other they may trust each other to have authenticated the users they serve and therefore to give A and B a service which authenticates the remote user.

Having obtained a mutual key, the two users, if they are particularly suspicious, may wish to exchange further messages to convince themselves of each other's genuineness. This must depend upon further secret information, which becomes vulnerable if it is sent to the other, as yet untrusted, party, using the newly established connection. They may, for example, exchange passwords using the protection of the connection they do not quite trust. If a correct reply password is not received within the permitted number of attempts the first one is compromised and there is a suspicion that the key distribution service is in error. The users may, on the other hand, have private encipherment keys, previously delivered, which they use only to protect their private authentication protocol. If the protocol reveals a doubt of correct identity no secret user information is compromised but, as before, the trustworthiness of the key distribution service is in doubt. This kind of consideration is inevitable if there is a standard service which distributes keys and attempts to guarantee that the sender and recipient are genuine. An alternative is that the service does not use encipherment to authenticate the users, but leaves it to them. Another is that the identity of the recipients is guaranteed but that they are only sure that the originator is an authorised user of the key distribution service. Neither of these possibilities seems as useful since one or both users must either risk compromising secret information or must hold a key personally. They may well do so but they should not be forced to.

Another point to consider is that a user who wishes to connect to a remote resource may not be directly identifiable by that resource. For example, a database interrogation service may contain no check of its users' authoritity, assuming that their identity was established as part of the identification procedure when they logged in and that the resources at their disposal, including the interrogation service, were thereby decided. There will then be an entity, at the same location as the users who wish to connect, that is concerned with resource allocation, that knows which users are allowed to use which resources and that checks permission before allowing the users' connections to be made.

This entity has a privileged position in remote user authentication in that it is trusted by remote parts of the service (entities of the same kind as itself) to guarantee that the users it serves are only given authorised connection. It is useful to build into the service some mechanism to guarantee to such privileged entities that they are communicating with their own kind. The simplest way of doing this is to design the control software so that all connections to remote processes are handled by such entities and that they check access permission at one or both of the sites involved. If we assume that this is not the case and that there is a need to make connections between processes which will do their own checking of authorisation then a possible way of identifying the entities which are to be given more trust is to allocate exclusively for their use a special type of key. The encipherment service guarantees to the remote encipherment service that such a key may only be used successfully by such an entity. Reference 2 introduces the idea of type values which it is useful to bind securely to keys (e.g. DEK or KEK). A useful type value which is not mentioned there is one which guarantees that the key may be used only by an entity authorised to check access rights.

There are applications where it is useful to be able to generate the same key at two remote sites rather than sending the key from one to the other and without sending values used to generate it via the telecommunications link. For example, customers are supplied with plastic cards which are used to help identify them.

The cards contain values which are to help generate the key to be used in sending information to a central installation. In addition they are required to type in a PIN value which also contributes. Another contributory value comes from the terminal into which they insert their cards (the terminal value may be changed periodically for greater security). The central installation holds these values. When it is told in clear who the customers and the terminals claim to be it generates a key using the stored values, knowing that the genuine terminals can generate the same on behalf of the genuine users. For this and similar cases the key-generation functions in Section 2 contain a seed value, with the assurance that the same seed will generate the same key. When an unrepeatable key is wanted the seed is omitted. There is, of course, a danger in this facility and it may well be that it should be denied to some users.

In making a request for a transport connection, as described in the open systems interconnection model, it is envisaged that a user may ask that it be secure. The details of what this means are not yet spelled out but it certainly implies encipherment. A connection request message may contain 'security parameters'[7] and we may suppose that they will indicate the key to be used, either as the actual key (suitably enciphered) or as a reference to a key already known to both parties. We may then consider the applicability of the functions described here. First, if the two parties have an established mutual KEK used to encipher keys they wish to send each other the functions are not applicable. The key to be used for the connection is enciphered by a call on the sender's encipherment service. It may then either be placed in the connection-request message or it may be sent beforehand (for example as one of a batch of keys to use that day) and a reference to it may be placed in the connection-request message. If the two parties do not have such a mutual KEK and do not have a supply of session keys to choose from then the function 'mutual key' applies. However, it cannot be used to encipher the key which is then placed in the connection request because that is not its function. Its function is to deliver the key. Neither is it reasonable to suppose that a key should be extracted from the connection request as it passes from one KEK domain to another (and there may be such separate domains for security purposes). The use of 'mutual key' in this case is to establish a mutual key for the transport entities that they may use to encipher the keys to be used subsequently for transport connection protection. It must be done as a separate previous operation and, at least the first time, must be sent over an 'insecure' transport connection. This does not matter as the function handles its own security.

3.2 Data privacy and data authentication

Once keys have been successfully exchanged by the two end users of a telecommunications link or by their local services on their behalf data privacy may be achieved by data encipherment and decipherment. Each local service must therefore provide enciphering and deciphering functions. The user may also wish to encipher and decipher keys using key-enciphering keys to produce and make use of key hierarchies. These topics are dealt with, for example, in References 1 and 2, which describe means of protecting keys such that they never appear in clear outside a trusted encipherment environment. They are relevant to this paper in that the user of the key-manipulation facility needs the ability to operate explicitly upon keys of a chosen type, but should not need to know how the types are indicated or need to be wary of operations upon keys of a particular type which might prejudice security. Data authentication and greater assurance of privacy are obtained by using particular modes of operation of encipherment (for example cipher block chaining or cipher feedback when using block ciphers) and by the addition of checking information (e.g. enciphered sum checks to reveal illicit modification and various identifying values to reveal illicit insertions and replays). These functions are not directly concerned with key generation and distribution and are not dealt with in this paper.

3.3 Digital signatures

A digital signature depends upon a sender using a key that no one else has and the receiver being able to demonstrate that the key has been used. To do this the sender may use the secret key of a public key cipher, such as RSA, and make the public key available to the receiver.[8] Using the functions described here a type value would be assigned to mean a public key pair. The effect of a public key cipher may be achieved by adding type information, meaning 'encipher only' or 'decipher only' to a symmetric cipher key in a trusted environment, with the knowledge that it can only be removed and acted upon in a trusted environment.[2] Another possibility is to use an algorithm which has an associated public and private key but which transforms the text to be signed by some means other than encipherment. Such keys can also be indicated by type information in the functions described in Section 2.

3.4 Stored secure files

The key-generation function may be used to generate a key which enciphers a file stored locally or whose medium is to be physically removed from the computer environment. If a file is stored for a long time or is transferred to a separate site it will be necessary to re-encipher. Reference 1 points out that a hierarchy of keys is needed in such a case. References 1 and 2 discuss how this may be achieved securely. The exact method is hidden at a lower level and visible in the functions described here only in the fact that keys are generated with an explicit type which indicates key-enciphering key or data-enciphering key.

3.5 Protection of software copyright

Reference 2 points out that type information securely attached to a key may be used, given a secure execution environment, to safeguard copyright. Software to be protected would be enciphered by the key and the key would be supplied to the user enciphered by a KEK which was available only inside the secure execution environment. When the software was used it would be deciphered as an implicit part of the loading operation. This idea anticipates the commercial availability of such an execution environment. However, when appropriate, a type value could be assigned in the functions of Section 2.

4 Relationship to detailed key manipulation schemes

This section discusses how the functions described in Section 2 can be implemented using a number of techniques described elsewhere. The functions are dealt with in turn.

4.1 Generate key

Let us assume we are using one of the key management schemes described in References 1-3. Each scheme, when it generates a key and makes it available outside the trusted encipherment facility, protects it by enciphering it. The schemes differ in how they do this and in how they ensure that the keys may not be misused (for example that a DEK may not be deciphered and made available outside the encipherment facility in clear form). They differ in the amount of protection they give the keys. The key notarisation scheme guarantees that a key can only be used successfully by the intended users by making the encipherment and decipherment of the key a function of the identities of the users for whom the key is intended. Since users must establish their identity in a way which satisfies security criteria (for example by supplying a password) they cannot successfully use someone else's key. The IBM scheme protects the key from exposure and ensures that some different types of key cannot be confused. To do this different master keys at an installation are used to encipher KEKs, session keys and keys used to encipher files. The operating system is relied upon to ensure that the keys are used by the intended users. The ICL scheme enciphers a key, together with type information indicating how it may be used, by a KEK (in some cases by an installation master key). It can, therefore, potentially restrict keys in ways which may be defined and could include the equivalent of the key notarisation scheme. The functions supplied in terms of key type therefore overlap and where they coincide they are not implemented in the same way. The functions described in Section 2 may be mapped on to any of the three, with the proviso that some of the key types envisaged are not present in some cases.

The local name produced by 'generate key' is then in the context of Reference 1 the form enciphered by KM_0, KM_1 or KM_2 according to its type. In the context of Reference 2 it is the key and concatenated type enciphered by the master key. In the context of Reference 3 it is the form supplied by the key notarisation facility.

If a key is to be associated securely with its users as in Reference 3 then extra associated software is needed if the basic encipherment facility does not provide it. Whether it is always desirable to tie a generated key immediately to particular users is a debatable point.

4.2 Give key

Assume that the user to whom the key is to be given is at a site which uses a similar system in terms of References 1-3. If the first site has the necessary KEK it can re-encrypt the generated key and send it directly to the second site. There the service re-encrypts it for the second user if the key used to protect it in transit is not the one which protects it when it is stored there. There may, on the other hand, be a series of re-encipherments en route because of the need to cross different key domains. The user of the 'give key' function may remain unaware of this.

As in Reference 4, a key-distribution centre may be used to generate the key in a form suitable for transmission to another site. This also may be hidden from the user of the 'give key' function.

If the sender and recipient are encipherment services which differ in the way they encode keys for protection (as in References 1-3) more manipulation is needed to effect the transfer. There must be a transformation function, which operates in an environment as secure as the one used to encipher the data in the first place, which deciphers and re-enciphers, reformatting as necessary. This also can be hidden from the user of 'give key', although a standard way of formatting keys and their associated information is clearly desirable.

4.3 Mutual key

In some cases this may be only a shorthand way of writing 'generate key', followed by 'give key'.. However, consider the following cases:

- when a KDC is used to generate the key it may be necessary to tell it the identity of the other partner in the connection so that it may encipher it appropriately[4]
- the generation of the key may need the involvement of the encipherment services at both ends of the connection (e.g. when using the Diffie/Hellman algorithm[9]).

For such reasons 'mutual key' may be needed as a primitive function at this level.

4.4 Take key and destroy key

If the underlying implementations are those of References 1-3 these functions are barely necessary. If a generated key is stored by the encipherment service and a reference to it passed back to the user then an explicit destruction of keys is needed. 'Take key' may also be used to inform the service that a particular user is no longer entitled to use a key.

5 Relationship to communications standards

We may expect the emerging open systems interconnection standards to provide secure services. For example, as already mentioned, an enhancement of the transport service is likely to provide authentication of users, data privacy and data authentication. The two entities which communicate to provide this service must establish jointly agreed keys and initialisation variables and would make use of functions such as those described in this paper. The form of the transmitted key and its accompanying information is an obvious candidate for standardisation and would avoid the need to transform the key en route, other than to change its key-encryption key. In seeking a standard form we have to consider:

– the length of the key
– the permitted users (if this is to be declared explicitly)
– information about the type of use permitted.

The methods referred to in this paper do not all allow the same restrictions of key use to be described. Moreover, in some cases, the restriction is implied in the manner of enciphering the key (e.g. the key notarisation scheme). A standard which explicitly stated the users could therefore be considered redundant in this case. However, if the basic key-manipulation method does not involve the user's identity (as in Reference 1 and in Reference 2 in its simplest form) the addition gives added security.

The basic encipherment algorithm affects both the length of the key and the type information which is relevant. For example, an indication of 'encipherment' or decipherment' is irrelevant to an RSA key.

Reference 2 has suggested that the 'parity' bits in the DES key could be used to indicate typing information. This may be unacceptable as an international standard. The typing information must then be held separately from the 64-bit key variable.

Bearing these points in mind the following is a tentative suggestion for a standard form for a key and associated information. First, the clear form. It has the format:

key length, key, key type, users

where 'key length' is an integer which gives the length of the following key;
'key' is the key as a binary string;
'key type' is a binary string whose bits have the following significance:

1st bit	DEK or KEK
2nd bit	enciphering key or not
3rd bit	deciphering key or not
4th bit	software protection key or not
5th bit	key usable by any process or only by one authorised to check access rights

(meanings for other bits are likely to prove useful);

and where 'users' consists of either one or two alphanumeric strings which identify the permitted user or users.

If such a composite item is to be transmitted over an insecure telecommunications line it must be enciphered. The form this takes depends upon the enciphering method. Using a 64-bit block cipher, for example, one must use some method of ensuring that the separate blocks which form the item cannot be changed unnoticed. One might, for example, form an enciphered sum check of the whole item and send it with it. A method which enciphered a block as long as the composite item could dispense with this.

6 Conclusions

This paper has discussed a number of issues related to the standardisation of the interface to an enciphering service at a particular level.

Several ways of providing basic key-manipulation features have been considered. It would be logically possible to evolve a standard way which made use of the best features of those considered. This would make standardisation of the form of the key and associated information easier.

An enciphering service may or may not make use of a separate key-distribution centre, depending on the number of communicating locations and the complexity possible in each. This design option is likely to survive. The functions suggested here deliberately hid this choice, taking the view that it is a part of the service implementation which the user should be able to ignore.

When a key is sent to a remote user it may need to be transformed because a different way of protecting it is needed. It may need to be enciphered by the remote user's location master key. During its journey it may need to be enciphered by a KEK used only for transportation. It may need to be re-enciphered by several such keys in the course of its journey. Such transformations should be hidden from the user at as low a level as possible so that logic can be written irrespective of the context created by the way the network of users is organised.

New methods of enciphering are likely to be developed. We should attempt to protect users from the need to know the underlying changes they bring. This is, of course, an aim which cannot necessarily be fulfilled. At the level chosen for the functions of this paper we reveal the essential difference between symmetric and asymmetric ciphers. New methods may bring their own characteristics which should not be hidden.

New applications of encipherment and related techniques are likely. Two mentioned here are digital signatures, which do not use encipherment of a form which can be used for data privacy, and a new key type dedicated to controlling resource use.

For such reasons the subject is one which will continue to develop and the points made in this paper are offered as part of the discussion needed to find functions and techniques which may develop as our knowledge of the subject grows.

References

1 EHRSAM, W.F., MATYAS, S.M., MEYER, C.D. and TUCHMAN, W.L.: 'A cryptographic key management scheme for implementing the data encryption standard', *IBM Syst. J.*, 17 (2).

2 JONES, R.W.: 'Some techniques for handling encipherment keys', *ICL Tech. J.*, 1982, 3 (2), 175-188.

3 SMID, M.E.: 'A key notarisation system for computer networks', NBS Special Publication 500-54, US Dept. of Commerce.

4 PRICE, W.L. and DAVIES, D.W.: 'Issues in the design of a key distribution centre', NPL Report DNACS 43/81, National Physical Laboratory, Teddington, Middlesex, UK.

5 International standard ISO/IS 7498. 'Information processing systems – open systems interconnection – basic reference model'.

6 NEEDHAM, R.M. and SCHROEDER, M.D.: 'Using encryption for authentication in large networks of computers.' Communications of the ACM, December 1978.

7 Draft International Standard ISO/DIS 8073. Information processing systems – open systems interconnection – connection oriented transport protocol specification.

8 RIVEST, R.L., SHAMIR, A. and ADDLEMAN, L.: 'A method of obtaining digital signatures and public key cryptosystems', Communications of the ACM, February 1978.

9 DIFFIE, W. and HELLMAN, M.E.: 'New directions in Cryptography', *IEEE Trans.*, IT-22 (6).

Analysis of software failure data (1): adaptation of the Littlewood stochastic reliability growth model for coarse data

P. Mellor
ICL Group Quality, Stevenage, Hertfordshire

Abstract

Measurement of software reliability requires three things: a conceptual model, an inference procedure and a prediction procedure. The conceptual model describes software failure in probabilistic terms. The inference procedure gives an estimate of the model parameters based on analysis of the failure history of a software product. The prediction procedure uses those parameter values to forecast future behaviour.

The conceptual model used here is Littlewood's stochastic reliability growth model, published in 1981. This is a three-parameter, fault-counting model. It has two types of uncertainty: each fault is a Poisson source of failure with its own rate, and this rate is itself a gamma-distributed random variable. The shape and size parameters of the gamma distribution and number of faults are the parameters to be estimated. Littlewood's maximum likelihood estimation of the parameters has been adapted to apply to a data set consisting of fault count and running time at several sample points. A standard for data capture is given which will yield a 'clean' set: one which is not influenced by events outside the time period, system sample and product version being studied.

The model has been programmed in Pascal for the ICL Personal Computer. The results of analysing several data sets are presented and assessed.

Put not your trust in optimistic modellers. If the advocate of a model will tell you openly its drawbacks as well as its strengths, cherish him.

Bev Littlewood

1 Introduction

1.1 Software quality assurance: the need for measurement

The need for reliable software is one of the main challenges to the computer industry. To meet this, large sums of money are spent on design, development and testing techniques to ensure that software will not fail in use. Our confidence in its not failing is what we mean by its reliability.

To judge the success of reliability-improvement methods we must have a measure which expresses our confidence as a number. Without an agreed measure, any programme of improvement is pointless. To know what has been achieved, we must be able to test the finished product and estimate reliability from the failures observed. The technique of reliability measurement is an essential complement to the techniques of developing reliable software.

We cannot achieve what we cannot measure.

1.2 Aims of reliability measurement

Typically, software goes through several stages in production: statement of requirements, design, specification, coding, testing, trial and release.

Statement of requirements requires knowledge of the market and the capabilities of current design methods. The conformance of design to requirements, specification to design and code to specification (prior to test and during early testing) is assured by inspection.

Measurement of reliability begins when a stable product is available, in later testing or trial. ('Stable' means that functional enhancement is at an end: repair of faults found during trial can still be carried out.) The purposes of applying the measurement technique at this stage are:

- to allow reliability to be quoted on delivery. A certified minimum reliability may be one of the requirements for certain applications, e.g. flight control, nuclear reactor safety systems
- to enable the vendor to forecast the cost of support
- to assess the point at which the product is fit for release (based on limits of minimum reliability or maximum support cost)
- to estimate the further testing time required until the product is fit for release.

1.3 Defects and failures

It is worth making clear a distinction between the defect count kept during most development procedures which use design and code inspections (notably the Fagan method), and the statistics of software failure used to measure reliability.

The defect count is kept for all stages of development from design through to testing. It provides a measure, typically expressed as 'defects found per thousand lines of code', of development progress. By comparison with similar products at the end of their lifecycle it is possible to estimate 'defects remaining per thousand lines of code'.

Faults found as a result of failure during test will be counted as defects for inspection purposes. There is therefore a common area between the two sets of statistics. In fact, to investigate the relationship between 'estimated defects remaining' and reliability is an interesting task for research. However, the defect count is a static measure: it takes no account of running time. Since it does not tell how likely a given defect is to cause a failure, it provides no measure of reliability.

1.4 The pursuit of perfection

We can never be certain that software will not fail. Even if no failure at all has been observed during trial, this would give less than total certainty that the product would run without failure for any subsequent period. Even though the aim of software production is zero defects, reliability measurement is still needed to assign a number to our confidence that the product in fact contains zero defects.

A target of zero defects must not be confused with an assumption of zero defects. Safety measures, backup and fault-reporting procedures need to be in place even for highly reliable products. As the engineering adage has it: the magnitude of a disaster is proportional to the designer's certainty that it cannot happen.

1.5 Hardware and software reliability concepts

The concepts of hardware reliability must not be applied unthinkingly to software. Hardware reliability theory depends on being able to observe the frequencies of failure among many identically produced items. Software is only produced once, even though many copies are run. The 'probability of nonfailure' is therefore a measure of our confidence in it (this is probability in the Bayesian sense).

There is no direct equivalent in software of physical component failure. The standard measure 'mean time to failure' (MTTF) has limited applicability to software. (The reasons for this are considered later.)

'Mean time to repair' (MTTR) is also not a useful concept in software. Typically, after a software failure, the system can be restarted immediately and the diagnosis of the fault takes place away from the system using evidence gathered at the time of failure. The repair may be applied much later, again without requiring a significant amount of system downtime.

On the other hand, the concepts of software reliability also describe the detection and removal of design faults in hardware systems and some aspects of repairable hardware reliability.

The differences are discussed at greater length elsewhere[1].

1.6 Specific problems of the vendor

Certain problems of software reliability are peculiar to the vendor of computer systems. In particular:

- many systems will simultaneously run the same software product, certainly after release, but probably also during validation
- as a result, one fault may cause many failures in the field before it is dealt with
- the vendor's support costs depend on the number of reports of failures of all degrees of severity, not just those causing total system crash
- postrelease data tends to be poor, since it must be collected from many systems in the field. Automated collection may improve the quality of this data.

Most software reliability models deal with a single instance of a program running in a controlled environment with perfect data capture. The model used here has been adapted to cope with the above problems. Instead of requiring running time up to detection of each fault, it uses total running time and faults found in each of several periods of time.

Published models assume each fault is seen only once, since it is then removed. Here the same effect is achieved by disregarding all but the first failure caused by any fault. Note that data on repeat occurrences would in theory be useful in estimating the contribution made to the overall failure rate by the individual fault. However, the Known Error Log is published to users to suppress the reporting of failures due to faults already logged. Since the repeat data are censored to an unknown extent by this mechanism, all but first occurrences are discarded.

The 'memoryless' nature of the model justifies the addition of running time from several systems running simultaneously.

1.7 Basic method

To measure software reliability requires three things:

- *a conceptual model of how software failure occurs.* This is expressed in terms of the probability of given faults causing failure. It depends on the value of certain parameters. The model used here is Littlewood's stochastic reliability growth model[2] (referred to as LSRG). This assumes that faults in the product cause random failures (i.e. each fault is a Poisson source of failure), but each fault does so with its own rate. As the software is run, faults are found and removed and those remaining are fewer and each is less likely to cause a failure, hence the decreasing rate of occurrence of failures observed during test. The three parameters are the number of faults, and two parameters describing the distribution of their individual rates.
- *an inference procedure.* This applies the model to a particular product by estimating the values of the parameters to fit the observed failure statistics. Maximum likelihood estimation is most frequently used.
- *a prediction procedure.* This predicts behaviour of the product by combining the results of the inference procedure with the conceptual model.

We proceed as follows:

- A frozen version of the product is run under trial conditions as close as possible to those of actual customer use.
- All running time is recorded. (Some other suitable measure of product use may be substituted.)
- All incidents are recorded, with date and time of occurrence. An incident is any software failure, user error or detection of a usability problem.
- Each incident is diagnosed to determine its cause, or 'source'.
- The first incident due to each source is counted, the rest are discarded.
- Data as above are collected for several periods of running. This gives a set of pairs of numbers: the running time and new sources found in each period.
- A graph is drawn of accumulated new sources found against running time. The slope of the 'smallest' curve that can be drawn over this (in a sense defined later) is a measure of the failure rate at any point.
- The three model parameters are evaluated (inference procedure).
- The expected number of sources found and failure rate can then be calculated for any future time (prediction procedure).

The result of this method applied to a typical data set is shown in Fig. 1 and 2 and Tables 1-4 in Section 6.

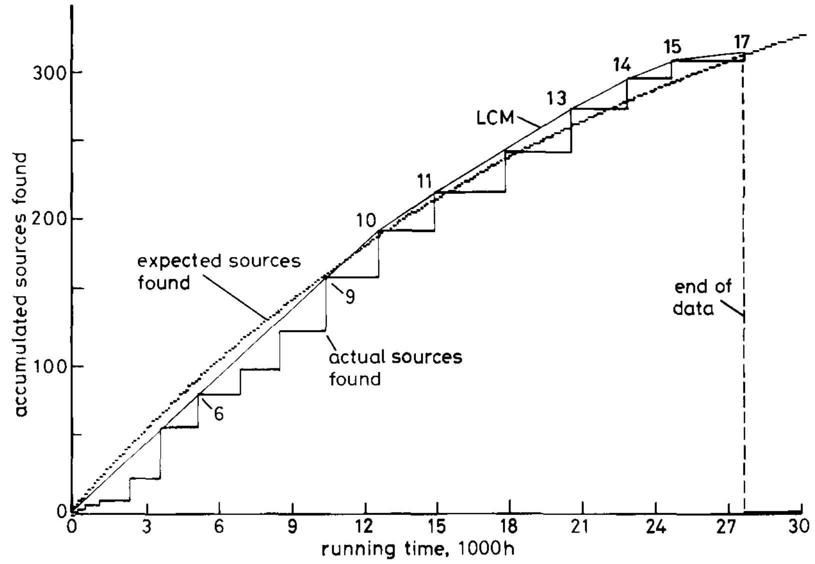


Fig. 1 B200A data set: sources found

Care is necessary when interpreting the results:

The predictions are of the finding of *new* sources. Repeat incidents due to known sources will also occur, depending on the number of systems simultaneously running the product and the repair policy adopted. The software service cost model[1] will take these into account when predicting support load.

The predictions are of incidents over *running* time. For use on the released product, a factor representing 'average use per week' (or similar) must be applied.

Although one of the parameters is 'number of faults in the product' this does not justify a manager saying to a programmer 'Right, Smith! The model shows 394 bugs in your program. Don't come back till you've found them all!!'. Most of those faults would be very infrequent causes of failure. It would make more sense to set targets in terms of rate of occurrence of failures or reliability at several points in time and continue testing until the predicted rates or reliabilities were within target for each point chosen.

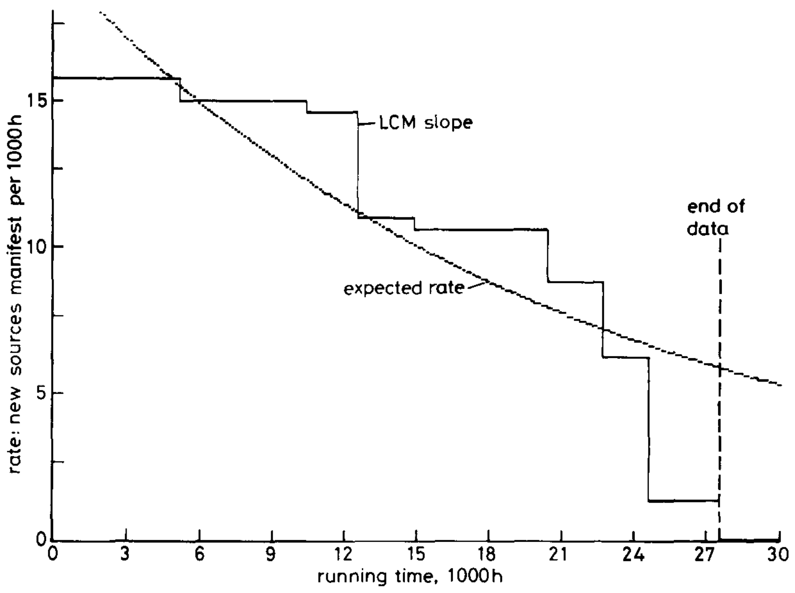


Fig. 2 B200A data set: rate

Care must also be taken in setting up the trial environment. The predictions will only be valid if the type of usage during measurement is not significantly different from that during actual use.

Lastly, care must be taken with data collection. Statistics must come from a given frozen version of the product, run on a given sample of systems over a given period of time. The data set must be 'clean': there must be no 'crosstalk' from outside, e.g. the first time a source is found *in the trial* it must be counted as new, even if known from incidents prior to the trial.

1.8 Structure of the paper

Terminology is defined in Section 2. Section 3 describes the Littlewood stochastic reliability growth model in mathematical terms, including the formulae for prediction. Section 4 describes the inference procedure. Section 5 outlines the minimum data capture standard. Section 6 presents some examples of data sets and their analysis. Section 7 examines the significance of different parameter values, Section 8 considers alternative approaches and Section 9 looks at further work required.

2 Terminology

failure	unacceptable behaviour of the product
query	user response to detection of difficulty (other than an actual failure) in using the product: includes 'usability' problems
incident	covers both failure and query: generally observed by the vendor when an incident report is raised by the user
fault	result of a mistake in design or coding which causes failure when encountered during running
source	anything which causes an incident: either a fault, or feature of the product which causes a query to be raised (see note below)
instance	where a product is run on several systems, the 'copy' of a source on one particular system.
manifestation	detection of a source, resulting in an incident. May occur at any instance of the source.
known source	one which has manifested itself at any of its instances, as opposed to a 'new' source
KEL	Known Error Log: central file of all known sources
u	product use: a measure of how much the product has been exercised and the sources in it exposed to risk of manifestation. Usually running time. However, for certain types of product a different measure may be required. For example, for an intermittently used interactive program the number of transactions may be used. Rarely identical with elapsed time. $u = 0$ at start of trial.
$Pr\{x\}, Pr\{x \mid a\}$	Probability that x it true, conditional probability that x is true given a
$S(t \mid u)$	reliability = $Pr\{$ No incident before further product use $t\}$ as measured after product use u

n	number of sources of incident in the product	parameters of the model
h	scale parameter of gamma distribution	
s	shape parameter of gamma distribution	
$\hat{n}, \hat{h}, \hat{s}$	estimates of n, h, s	
$x!$	$\int_0^\infty y^x \exp(-y)\,dy,\ x > 0$. For integral x, this $= \prod_{i=1}^{x} i$ – the factorial function. For nonintegral x, the gamma notation is often used: $\Gamma(x) = (x - 1)!$	
X	random variable X. Random variables are denoted by capitals and their realisations by lower case	
$pdf(x)$	probability density function of random variable X	
$E\{X\}, E\{X \mid u\}$	expected value of random variable X, expected value after use u	
C	accumulated sources manifest (random variable)	
$M(u)$	$E\{C \mid u\}$: expected number of sources manifest after use u	
$R(u), R(u \mid c)$	rate of occurrence of incidents from whole product after use u, rate conditional on c sources having been found already. (Referred to in the literature as ROCOF: rate of occurrence of failures, but note that we wish to include incidents of both types.)	
Z	rate of manifestation of individual source (random variable)	
$\binom{n}{c}$	the binomial coefficient $n!/c!(n-c)!$, n and c integers	
LCM	least concave majorant: the smallest curve (actually a series of straight line segments) that can be drawn over a graph of sources manifest against product use	
exterior point	point on the graph at which it touches the LCM	

Note: We shall be mainly concerned with large operating system software. The complexity of such software means that product behaviour may be to specification, but unacceptable, or acceptable although outside specification. The above terminology sidesteps any argument about whether 'queries' are 'failures' or not.

The rest of this paper will refer to source manifestations and incidents, except where a deliberate distinction is made between faults and query sources when analysing data sets.

3 Littlewood stochastic reliability growth model

3.1 General remarks

The model is fully described in Littlewood's 1981 paper[2]. It was chosen since it avoids the rather dubious assumption of earlier models, such as those of Musa[3] and Jelinski and Moranda[4], that all sources have an equal contribution to the total incident rate. It is also mathematically tractable. In addition, for calculation of the vendor's lifecycle costs, a model is needed which predicts both the total number of incidents and the number of incidents which are first manifestations of sources.

It is a 'fault-counting' model: one which has as one of its parameters the 'number of sources in the product'. A further refinement is to make this a random variable as in the 'Poisson-gamma' model. This has been compared to the Littlewood model by Moek[5] and found to give almost indistinguishable results.

That the model takes the incident as its fundamental observation accords well with experience in software development and support. We do not observe a rate, we observe an incident. (After product release, the vendor usually observes just the report of an incident.) In fact, without a conceptual model, we cannot define what we mean by the rate.

The cost forecast model[1] hypothesised a different rate for each instance of a source due to the application of a 'stress factor'. This is not relevant here and is omitted for simplicity.

3.2 The multisystem problem

Since sources are random in the Poisson sense, the stream of incidents from any source is 'memoryless': if we observed all manifestations we would see the same distribution of incidents over product use whatever point we took as our origin. This would justify adding together product use from several systems for the purposes of analysis, even with arbitrary shifts relative to one another of the points at which they started running. We could, for example, compile the first week of our combined data set from statistics of week 1 of all systems, even though they did not start running simultaneously. However, we are observing first manifestation only, and since manifestation of a source on system A before system B starts up would censor any manifestation on system B from our records, we must preserve the ordering of system startup. For a given calendar period, then, we combine statistics of product use and incidents across all systems *in that period.*

In what follows, it is assumed this has been done and all references are to manifestations of a source, not of instances of a source, as though dealing with a single system.

3.3 *Assumptions*

1 A software product contains a given number n of discrete sources of incident.

2 Each source generates incidents randomly, i.e. as a Poisson process, during product use.

$$Pr\{c \text{ incidents generated over product use } u\} = \frac{(zu)^c}{c!} \exp(-zu)$$

where z = manifestation rate of this source

3 Only the first incident from each source is observed, i.e. each source is removed immediately and perfectly on manifestation.

4 Sources are independent of each other. The total incident rate from the product is therefore the sum of the manifestation rates of all the sources still present at the time.

5 The manifestation rate Z of an individual source is a random variable with a gamma distribution:

$$pdf(z) = \frac{1}{(s-1)!} h^s z^{s-1} \exp(-hz)$$

where h and s are the scale and shape parameters.

Note: h has the dimensions of product use. s is dimensionless.

3.4 *Choice of gamma distribution*

This is chosen to model the uncertainty about individual source rates since

- it is mathematically tractable
- it is flexible enough to fit any actual distribution we may expect to meet in practice
- it is conjugate with the Poisson: after a period of use, when some sources have been removed, provided the initial *pdf* of the rate was gamma, the final one will be also. The shape parameter remains constant, but the scale parameter h increases after use u to $h + u$. This represents the selective depletion of the population of sources: those with high rates tend to be eliminated earlier.

3.5 *Formulae for reliability and other measures*

For the purpose of estimating these measures, we assume that the parameters n, h, and s are known. Refer to Section 2 for the notation. The derivations are in the Appendix, or else in Littlewood[2]. Note that 'incident' here is 'manifestation of a new source': repeats are ignored (assumption 3).

expected number of sources manifest in $(0, u)$:

$$M(u) = E\left\{ C \mid u \right\} = n\left(1 - (h/(h+u))^s\right) \qquad (1)$$

conditional manifestation rate at u, given c sources manifest in $(0, u)$:

$$R(u \mid c) = (n - c)\, s/(h + u) \tag{2}$$

expected manifestation rate at u:

$$R(u) = M'(u) = nh^s s/(h + u)^{s+1} \tag{3}$$

conditional reliability at $u = Pr$ $\{$ No incident in $(u, u + t) \mid c$ in $(0, u)$ $\}$:

$$S(t \mid u, c) = \left[\frac{h + u}{h + u + t}\right]^{(n - c)s} \tag{4}$$

expected reliability at u:

$$S(t \mid u) = \left[1 - \left(\frac{h}{h + u}\right)^s + \left(\frac{h}{h + u + t}\right)^s\right]^n \tag{5}$$

conditional expected time to next incident at u, given c incidents in $(0, u)$:

$$E\left\{ T \mid u, c \right\} = (h + u)/((n - c)s - 1) \tag{6}$$

Note that the expected manifestation rate of an individual source remaining after use u is $s/(h + u)$.

We shall use mainly the unconditional expressions 3 and 5. This differs slightly from the treatment of Littlewood[2], who usually quotes measures given some observed number of incidents.

The reason is this: if, after estimating $\hat{n}$, $\hat{h}$ and $\hat{s}$ from an observed data set (see Section 4), further observations are made, then instead of calculating the measures with the old $\hat{n}$, $\hat{h}$ and $\hat{s}$ but conditional upon the extra incidents we calculate new values of $\hat{n}$, $\hat{h}$ and $\hat{s}$ using the increased data set.

When monitoring a software trial, therefore, the parameters are always estimated on all available data at any time and used to predict product behaviour beyond the last observation. The incident rate, for example, is therefore shown as smoothly decreasing instead of having the step drops at each incident which the original model gives[2].

It is relevant to this approach that maximum likelihood estimation as described in Section 4 always gives a curve of expected number of sources found against product use which passes very close to the last data point.

3.6 *Confidence intervals*

Formula 1, for the expected number of sources found, is derived from the binomial *pdf* of C, as given in the appendix, where it is used to derive the likelihood function. From this *pdf* can be derived the quantiles of the observed future incidents, and hence of the conditional incident rate and reliability. This will be covered in a later paper on applying formal methods of assessing predictive accuracy.

Note that no confidence intervals can be quoted for the estimates of n, h and s. Maximum likelihood estimation in this case provides no means of calculating them. The uncertainty of our predictions is expressed solely by the confidence intervals of C.

3.7 Nonexistence of MTTF

From eqns. 6 in Section 3.5, it is evident that $E\left\{ T \mid u, c \right\}$ is not always meaningful.

The condition for $E\left\{ T \mid u, c \right\}$ to exist is:

$$s > 1/(n - c) = 1/(\text{number of sources remaining}) \qquad (7)$$

To obtain an unconditional expected time to failure after use u, $E\ \{ T \mid u \}$, we would have to evaluate:

$$\sum_{c=0}^{n} E\left\{ T \mid u, c \right\} Pr\left\{ C = c \right\}$$

Since $E\ \{ T \mid u, n \}$ is obviously infinite (no further incidents are possible after removal of last source) and $Pr\ \{ C = n \}$ is finite (it is just possible that all sources have been removed), the sum always contains at least one infinite term, and the unconditional expected time to next incident, or 'MTTF', is meaningless.

Because of this, manifestation rate and reliability must be used as software measures instead of MTTF.

4 Parameter estimation

4.1 General

This is the inference procedure referred to earlier. The object is to find the best estimate of the parameters n, h and s for a given set of failure data.

The method used is maximum likelihood estimation (MLE). This essentially asks the questions:

1 For a particular set of values of the parameters, how likely are we to observe the given data?
2 For which parameter values is that likelihood greatest?

The first question requires a likelihood function which gives the probability. The second requires a search procedure to enable a computer program to look for the maximum of that function (since an analytical solution is not possible).

4.2 *Likelihood function*

Suppose the data set consists of m pairs of numbers (c_i, u_i), where c_i is number of sources found after product use u_i, $1 \leqslant i \leqslant m$, $c_0 = u_0 = 0$

Then (see Appendix)

$$Pr\{\text{ particular source not manifest in } (u_{i-1}, u_i)\} = q_i = ((h + u_{i-1}) / (h + u_i))^s \tag{8}$$

Define $p_i = Pr\left\{\text{source } \textit{is} \text{ manifest}\right\} = 1 - q_i$

Then

$Pr\ \{\ c_i$ sources manifest by $u_i \mid c_{i-1}$ manifest by $u_{i-1}\ \}$

$= Pr\ \ \{\ (c_i - c_{i-1})$ sources manifest in (u_{i-1}, u_i) out of $n - c_{i-1}$ remaining at $u_{i-1})\ \}$

$$= \begin{pmatrix} n - c_{i-1} \\ c_i - c_{i-1} \end{pmatrix} p_i^{(c_i - c_{i-1})} q_i^{(n - c_i)} = l_i \tag{9}$$

and so:

$$Pr\{\text{ given data set being observed }\} = \prod_{i=1}^{m} l_i \tag{10}$$

This is the likelihood function to be maximised to give $\hat{n}$, $\hat{h}$ and $\hat{s}$. In practice, since the value of this function is very small, its log is used to avoid floating-point underflow.

4.3 *Search procedure*

Many procedures are available. The one used here is the 'naive hill-climbing method'.

1 set precision $p = 0{\cdot}5$
2 select first dimension, say n
3 until likelihood function stops increasing, increment n by step pn
4 until likelihood function stops increasing, decrement n by step pn
5 repeat 2-4 for other dimensions h and s
6 if the set of values has changed since 2, repeat 2-5
7 set precision p to new value $p/2$. If $p < 2^{-7}$ end, else go to 2.

The start point for the search may be set by the user at runtime: by default it is $h = u_m$, $s = 1$ and n = best value as defined below. There is an option to search with a fixed value of s, varying n and h only.

4.4 Simplification of search

It is possible to express the best value of n for given h and s as a function of h and s. To do this, we differentiate the log of the likelihood function (eqn. 14) partially with respect to s and equate to 0 for stationary value. The resulting value of n is given by

$$n = \sum_{i=1}^{m} [\log q_i \ (c_i - c_{i-1}\, q_i) \,/\, (1 - q_i)] \,/\, \sum_{i=1}^{m} \log q_i \qquad (11)$$

This reduces the number of dimensions of the search to two (or one if s is fixed). With this reduction, the search procedure yields an estimate for a set of 150 points in 1 h on the ICL Personal Computer. Regarding $\hat{n}$ being a function of $\hat{h}$ and $\hat{s}$: see remarks in Section 7 on interpreting the parameters.

Compare eqns. 10 and 11 with eqns. 46 and 47 in Littlewood[2].

5 Data-capture requirements

5.1 General

The following standard must be observed when collecting data for analysis, otherwise meaningful estimates cannot be made. Note that the same standard applies to data for any model capable of estimating reliability and incident rate and not solely to data for the Littlewood stochastic reliability growth model.

5.2 Data required (ideal)

1 Total use of product on all sample systems up to every incident.
2 Is the incident a first manifestation or a repeat?
3 What is the type of the incident (e.g. failure or query) and its severity (e.g crash)?
4 In which product, version and module is the source located?
5 Data set must be clean (See 5.4)

Note

(i) Product use up to incident is often derived from the date and time of the incident, which is then mapped on to running-time records. All incident reports, including queries and documentation errors, must therefore record the date and time of occurrence.

(ii) Usually a trial will involve several products running together. The above data are required for each product separately. In addition, if failure rate is to be broken down between modules, sources must be located down to module level.

If ideal data are available, it is possible to use the model with the original likelihood function (eqn. 46 in Littlewood[2]).

5.3 Minimum data required

As above, but with 1 replaced by

1(*a*) Total use of product on all sample systems in each of several periods during the trial
1(*b*) Within which period did each incident occur?

This type of data is often the best that is available. The adaptation of the inference procedure is necessary to cope with it.

5.4 Clean data

Whether ideal or minimum data is collected, the data set must be 'clean'. This means that it applies to a single product version, run on a known sample of systems for a given time period.

Three kinds of crosstalk from outside the data set must be avoided:

5.4.1 Time period: The trial should start with a 'clean slate' i.e. the first time a source is manifest *in the trial* it must be treated as new, even though it is known from incidents prior to the trial.

5.4.2 System sample: Manifestation of a source on a system not included in the trial must also not prevent it being counted as new at its first manifestation on a system in the trial.

5.4.3 Version: The trial must be done with a specific frozen version of the product. Sources manifest in other versions and presumed to be present in the trial version must still count as new when manifest in that version.

Crosstalk will result in failure data being censored, and give rise to optimistic reliability assessments.

5.5 Practical concerns

5.5.1 Delay: There will be a time lag before dependable analysis of a given data set can be performed. This is due to the time required for incident diagnosis, and, in the case of field data, time for reports of product use to be sent in.

In the interim, incidents can be assigned to their 'best guess' category. To err on the side of pessimism, unresolved reports should be treated as manifestations of new sources.

5.5.2 Databases and data extraction: It must be possible to find the earliest incident in which a given source manifested itself.

The database structure must therefore allow the following procedures:

1 For each known source, extract all incidents
2 Of these, select the earliest by date of occurrence.

This implies that 'known source' records must be held separately to incident records, with appropriate cross-references.

5.5.3 Product use recording: Complete records, in the form of system logs or other records, must be kept. Automated recording, particularly if some such measure as 'number of transactions' is being used, is an advantage.

This data is the most frequently missing or incomplete. It cannot be too highly emphasised:

Without records of product use no statement about reliability can be made.

6 Examples of data sets and their analysis

6.1 General

Assume that the data have been gathered to the standard described in Section 5. If it has not, this analysis is useless (in fact, any analysis is useless). We have then a set of pairs of numbers: product use and sources found. (Actually product use is measured by running time in all the examples.)

The first step is to accumulate totals and plot a graph of accumulated sources against accumulated running time. This alone gives a good picture of how the product is progressing. It is surprisingly seldom done.

The least concave majorant is the smallest envelope of straight-line segments which lies over this graph. Its slope at any point is an estimate of the total manifestation rate of new sources. The points at which the graph touches the LCM are referred to as exterior points. Estimation of total rate due to several independent sources with different individual rates by this procedure is known as isotonic regression. See Barlow *et al.*[6] for the theoretical background. It has been applied to the detection of faults in complex systems by Campbell and Ott[7].

The failure data analysis program generates the graphs and reports the exterior points and LCM slope at each.

Following estimation of the parameters n, h and s by MLE graphs can be drawn of the unconditional expected source accumulation and manifestation rate.

In all the examples except Moek83A, MLE was performed on h and s, with n calculated as in Section 4.4.

6.2 Data sets B200

These relate to an operating-system field trial. Accumulated product use and sources manifest are given at the end of each of 17 weeks. Product use is measured in units of 1000 running hours, and all rates are quoted as incidents/1000 h. Exterior points are marked by having the LCM slope shown against them. The results are shown in Tables 1-3.

Three sets have been analysed: B200P: product faults only; B200Q: query sources only; B200A: all sources taken together.

Table 1 B200: input data, exterior points and LCM slopes

Point	Accumulated running time kh	B200P sources	LCM slope	B200Q sources	LCM slope	B200A sources	LCM slope
1	0·312	0		1		1	
2	0·620	3		1		4	
3	1·216	5		2		7	
4	2·450	14		8		22	
5	3·725	31		26	6·98	57	
6	5·146	47		34	5·63	81	15·74
7	6·962	61		38		99	
8	8·583	73		51		124	
9	10·484	99		62	5·25	161	14·99
10	12·598	120	9·52	72	4·73	192	14·66
11	14·960	135	6·35	83	4·65	218	11·01
12	17·776	150		96	4·62	246	
13	20·412	168		108	4·55	276	10·64
14	22·772	183	6·14	114		297	8·90
15	24·685	190	3·66	119	2·57	309	6·28
16	25·847	190		119		309	
17	27·658	193	1·01	120	0·34	313	1·35

Table 2 B200: parameter values and derived rates

Set	n	h	s	s/h	ns/h	$s/(h+u_m)$	$R(u_m)$
P	301	222·522	8·7287	0·0392	11·81	0·0349	3·78
Q	175	170·020	7·6570	0·0450	7·88	0·0387	2·14
A	475	211·336	8·7378	0·0413	19·64	0·0366	5·93

s/h and $s/(h + u_m)$ are the mean individual source manifestation rates at the start and end of the data set. ns/h and $R(u_m)$ are the corresponding rates for the whole product.

Table 3 B200: expected accumulated manifest sources and rates

Use, kh	10		30		100		300	
Set	Acc.	Rate	Acc.	Rate	Acc.	Rate	Acc.	Rate
P	96	7·70	201	3·45	289	0·32	301	0·003
Q	62	4·81	125	1·93	170	0·14	175	0·001
A	158	12·52	326	5·39	459	0·45	475	0·004

Note that n for B200P and Q add to give n for B200A almost exactly. The rates also add. All are type 2 sets (see remarks in Section 7.1 regarding long-term predictions from such sets). The expected accumulations are rounded in Table 3. Given this rounding, 'all sources manifest' is predicted by 300 000 running hours.

The LCM is drawn on the graph of B200A (Fig. 1) and the exterior points are numbered. This is not generally done, since the table of exterior points and slopes suffices. The P and Q graphs are not included, being similar to A.

Note that the rate curve (Fig. 2) disagrees with the last LCM slope. The estimation procedure has ignored this final improvement in favour of the earlier steady growth in manifest sources.

There is a period of increasing manifestation rate at the start. This suggests that early usage is untypical, i.e. it took time for users to install the product fully and start hitting the problems. An analysis was therefore done using the last 13 points only (Tables 4 and 5). This yielded very different parameter estimates (particularly of h and s), but short-term rate estimates which were only about 20% down on the original.

Table 4 B200 (5-17): parameter values and derived rates

Set	n	h	s	s/h	ns/h	$s/(h+u_m)$	$R(u_m)$
P	240	167·190	9·6915	0·0580	13·91	0·0503	3·10
Q	146	79·039	5·2263	0·0661	9·65	0·0272	1·72
A	399	87·973	5·1763	0·0588	23·48	0·0269	4·95

Table 5 B200 (5-17): expected accumulated manifest sources and rates

Use kh	10		30		100		300	
Set	Acc.	Rate	Acc.	Rate	Acc.	Rate	Acc.	Rate
P	98	8·68	199	2·72	251	0·012	254	0·0002
Q	63	5·47	123	1·50	152	0·065	154	0·0006
A	160	14·12	324	4·36	413	0·234	421	0·0026

Note that the accumulated sources in table 5 have been corrected by adding in those already manifest before point 5, and the use is reckoned from point 1, not 5. Tables 3 and 5 are therefore directly comparable.

This shows some resilience to suspect early data.

6.3 Data sets B150

These data were presented in the earlier paper[1], when they were analysed by a relatively crude method assuming $s = 1$. The input data sets are large and are not reproduced here. There are three sets: P: product faults; Q: query sources; C: major faults (subset of P).

B150 is an earlier version of the same product as B200. All rates are therefore converted to manifestations/1000 running hours to enable comparison. Note that the timescale on the graphs (Figs. 3-5) is still calibrated in running days (= 24 running hours), however. The total length of the trial was 22·769 x 1000 running hours. Results are shown in Tables 6 and 7.

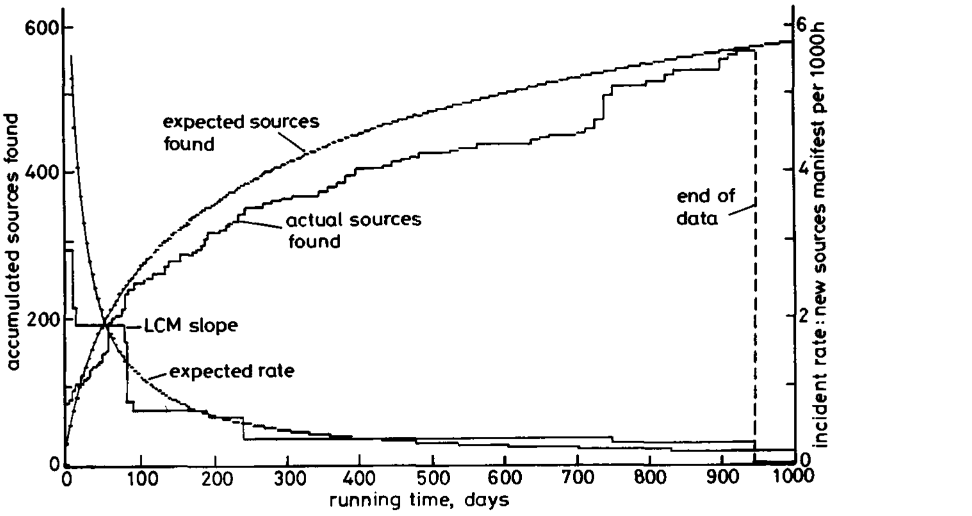


Fig. 3 Data set B150P: sources found, LCM slope and unconditional expected rate and sources

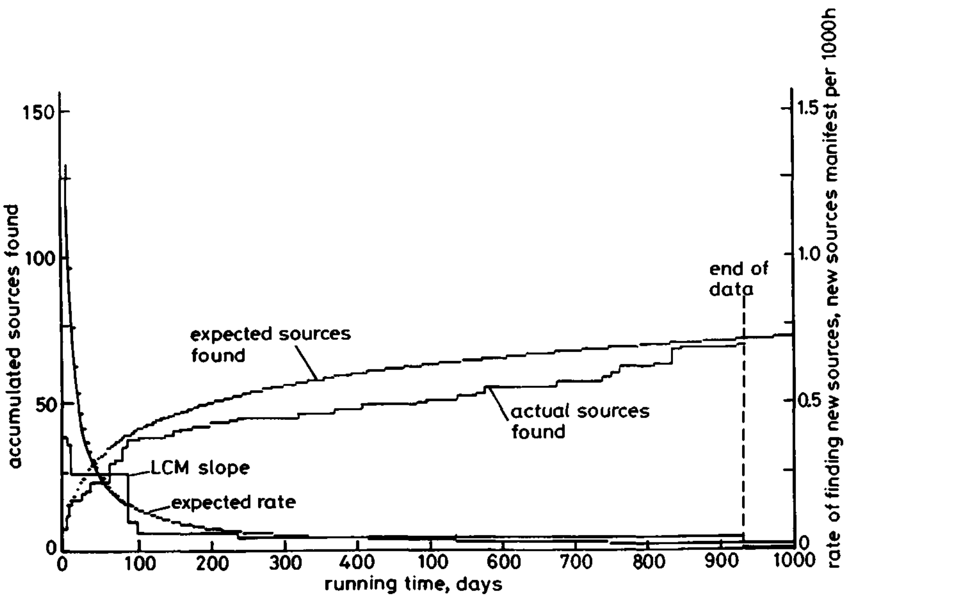


Fig. 4 Data set B150C: sources found, LCM slope and unconditional expected rate and sources

Note that P and its subset C are of type 1, Q of type 2.

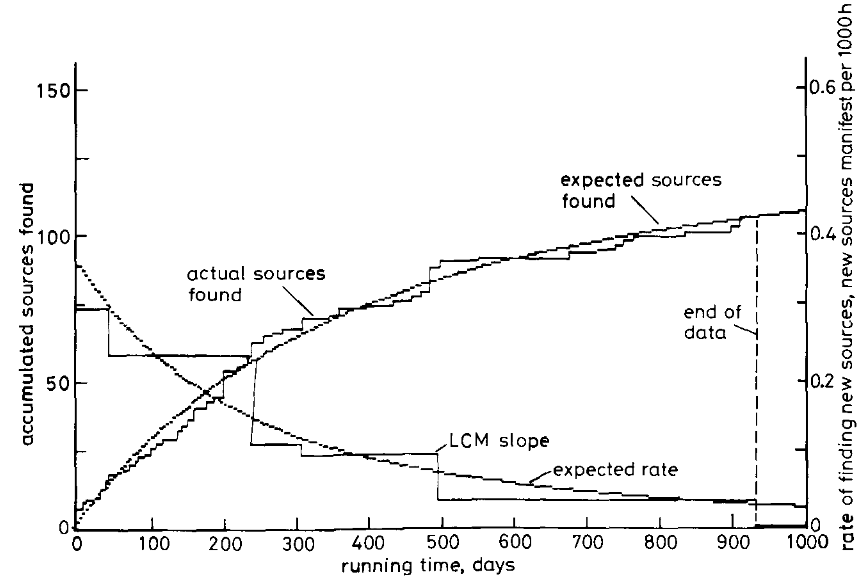


Fig. 5 Data set B150Q: sources found, LCM slope and unconditional expected rate and sources

Table 6 **B150: parameter values and derived rates**

Set	n	h	s	s/h	ns/h	$s/(h+u_m)$	$R(u_m)$
P	7397	0·498	0·02086	0·0419	309·84	9×10^{-4}	0·15
C	1567	0·189	0·00961	0·0509	79·70	$4{\cdot}2 \times 10^{-4}$	0·02
Q	136	14·577	1·6315	0·1119	15·22	0·0437	0·03

Table 7 **B150 : expected accumulated manifest sources and rates**

Use kh	10		30		100		300	
Set	Acc.	Rate	Acc.	Rate	Acc.	Rate	Acc.	Rate
P	455	13·79	608	4·64	775	1·37	924	0·45
C	60	1·42	76	0·475	93	0·141	108	0·047
Q	78	3·85	114	0·804	131	0·067	135	0·005

6.4 Data sets Moek83

These are due to Moek[5]. Set A relates to a large real-time system providing information about aircraft movements, and B to a program for performing complex aerodynamic computations.

Note that time is in units of 10^5 CPU seconds (i.e. time clocked up on the central processing unit) for set A and 10^3 CPU seconds for B.

Table 8 Moek83 data sets: input, exterior points and LCM slopes

Point	Accumulated CPU time 10^5 s	Set A sources	LCM slope	Accumulated CPU time 10^3 s	Set B sources	LCM slope
1	0·0088	1	110·00	0·00052	1	192·31
2	0·0431	2		0·00104	2	192·31
3	0·0717	3	30·00	0·00356	3	
4	0·1893	4		0·00510	4	
5	0·2368	5		0·00585	5	62·37
6	0·2392	6		0·00785	6	50·00
7	0·2622	7		0·00994	7	47·85
8	0·3479	8		0·01226	8	
9	0·3941	9		0·01458	9	
10	0·4047	10		0·01690	10	43·10
11	0·4429	11	20·00	0·02015	11	
12	0·5909	12		0·02243	12	36·17
13	0·6086	13		0·02554	13	32·15
14	0·8513	14		0·02871	14	31·55
15	0·8993	15		0·03196	15	
16	0·9040	16		0·03520	16	30·82
17	0·9044	17		0·04411	17	
18	1·0061	18		0·05306	18	
19	1·0173	19		0·05380	19	16·13
20	1·0271	20	20·00	0·06312	20	
21	1·2701	21		0·07192	21	11·04
22	1·2876	22		0·08117	22	10·81
23	1·3321	23		0·10308	23	
24	1·3807	24		0·11353	24	
25	1·3871	25		0·12320	25	
26	1·4270	26	20·00	0·13280	26	
27	1·6954	27		0·14309	27	
28	1·7181	28		0·15381	28	
29	1·7201	29	10·00	0·16453	29	
30	2·1119	30		0·17483	30	
31	2·2610	31		0·18534	31	
32	2·4077	32		0·19640	32	8·68
33	2·5708	33		0·24909	33	1·90
34	2·9549	34		0·31866	34	
35	2·9661	35		0·38723	35	
36	3·2717	36		0·46023	36	
37	3·3318	37		0·52954	37	
38	3·3350	38	5·57	0·59831	38	
39	3·5371	39	4·95	0·66723	39	
40	3·8011	40	3·79	0·72879	40	
41	4·1791	41	2·65	0·79817	41	
42	4·9213	42	1·35	0·86726	42	
43	5·7657	43	1·18	0·87682	43	1·59
44	6·6157	44		0·95177	44	
45				1·02657	45	
46				1·10127	46	1·34
47				1·16827	46	

Set A is of type 2, and a particularly atrocious example. When *s* is constrained to fixed low values, the prediction of 'imminent perfection' is avoided. B is of type 1.

Results are shown in Tables 8-10 and Fig. 6-8.

Table 9 Moek 83: parameter values and derived rates

Set	n	h	s	s/h	ns/h	$s/(h+u_m)$	$R(u_m)$
A	45	6·13841	4·009	0·6531	29·39	0·3143	0·75
A	49	2·32582	1·5	0·6449	31·60	0·1678	1·09
A	54	1·70560	1·0	0·5863	31·66	0·1202	1·33
A	67	0·97667	0·5	0·5119	34·30	0·0659	1·58
A	339	0·46335	0·05	0·1079	36·58	0·0071	2·10
B	132	0·013	0·09481	7·2797	960·92	0·0803	6·90

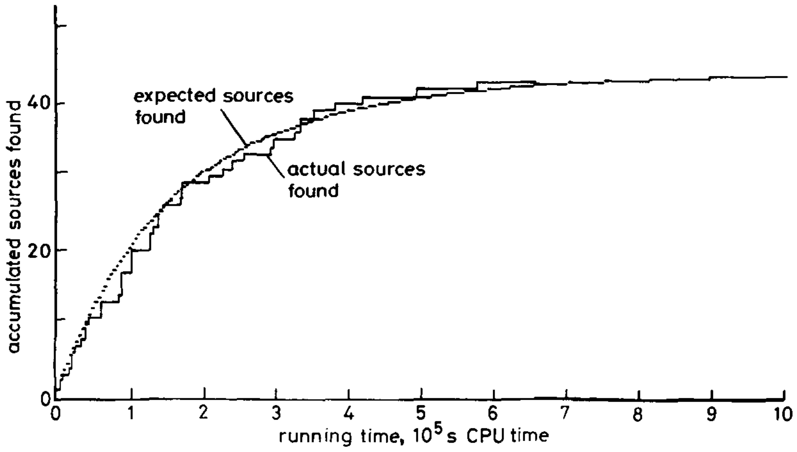


Fig. 6 Moek83A actual and expected accumulated manifest sources against running time: s = 4·009

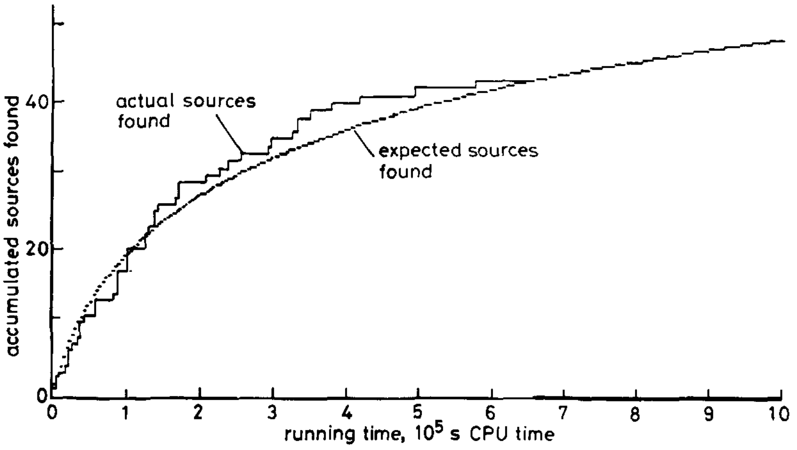


Fig. 7 Moek83A actual and expected accumulated manifest sources against running time: s = 0·05

The data are actually of the 'time to failure' variety (i.e. ideal: see Section 5), but have been treated as 'failures within period' without any great difference from the results obtained by Moek[5].

For set A, the best s from an unconstrained search = 4·009. Other results for set A were obtained by fixing s at the given value and searching on the other parameters only. For set B, s = 0·09481 is also the best obtained in an unconstrained search. Moek does not quote a best s, but calculates n for various fixed values of s and takes a simple average. This procedure seems to have no theoretical justification.

The interfailure running time at point 15 of the published data set[5] is incorrect. The accumulated running time of 31·96 is correct, however, and is used in this analysis.

Table 10 Moek83: expected accumulated manifest sources and rates

Use 10^5 s	1		3		10		30	
Set	Acc.	Rate	Acc.	Rate	Acc.	Rate	Acc.	Rate
A(s = 4·009)	20	13·8	36	4·00	44	0·232	45	0·004
A(s = 1·5)	20	12·9	35	3·98	45	0·489	48	0·044
A(s = 1·0)	20	12·6	34	4·16	46	0·672	51	0·092
A(s = 0·5)	20	11·4	34	4·17	47	0·910	55	0·192
A(s = 0·05)	19	10·9	32	4·43	49	1·386	64	0·451
Use 10^3 s	1		3		10		30	
B	45	8·18	53	2·47	62	0·666	69	0·200

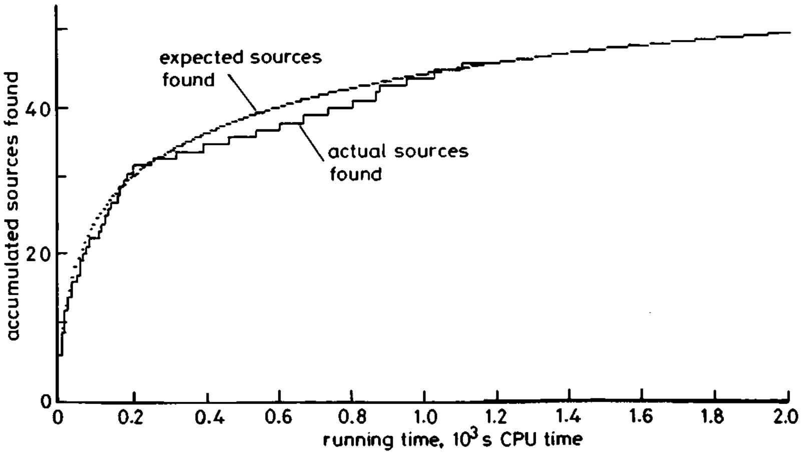


Fig. 8 Moek83B actual and expected accumulated manifest sources against running time: s = 0·09481

6.5 Data sets Misra83

These are from the Space Shuttle flight control program and were published by Misra[8]. They consist of running time (here given in units of 1000 h) and faults found in each of 38 weeks of testing. The faults are divided into major and minor, and are analysed both separately and combined.

Results are shown in Tables 11-13 and Fig. 9.

Note that all three sets are of type 2. It can be seen from Table 12 that the values obtained for *n* from the major and minor sets sum approximately to that obtained from the total set, but the agreement is not as good as that seen in Table 2.

Table 11 Misra83 data sets: input, exterior points and LCM slopes

Point	Accumulated running time, 1000h	Major sources	LCM slope	Minor sources	LCM slope	Total sources	LCM slope
1	0·0625	6	96·00	9	144·00	15	240·00
2	0·1065	8	45·46	13		21	
3	0·1465	9		20	130·95	29	166·67
4	0·2145	11		26	88·24	37	
5	0·2765	14		31	80·65	45	123·08
6	0·3425	15		34		49	
7	0·4155	17		36		53	
8	0·4890	20		41		61	
9	0·5810	22		45		67	
10	0·6524	22		47		69	
11	0·7169	25		51		76	
12	0·7816	26		58		84	
13	0·8176	29		58		87	
14	0·8716	29		63		92	
15	0·9111	31		66		97	
16	0·9791	36		69		105	
17	1·0401	41		72		113	
18	1·1027	43		76		119	
19	1·2014	45		86		131	
20	1·2264	47		89		136	
21	1·2384	48		90		138	
22	1·2934	51		92		143	
23	1·3424	53		96		149	
24	1·4064	57		101	61·95	158	100·01
25	1·4324	58	37·71	101		159	
26	1·4984	60		103		163	
27	1·5474	62		103		165	
28	1·5994	64	35·93	105		169	
29	1·6694	65		108		173	
30	1·7539	68		114		182	
31	1·8369	71	29·47	117		188	69·69
32	1·8969	71		118		189	
33	1·9694	73		119		192	
34	2·0594	75		123		198	
35	2·1174	78	24·96	126		204	
36	2·1774	79		128		207	
37	2·3454	82	17·54	139		221	
38	2·4569	83	8·97	148	44·74	231	69·36

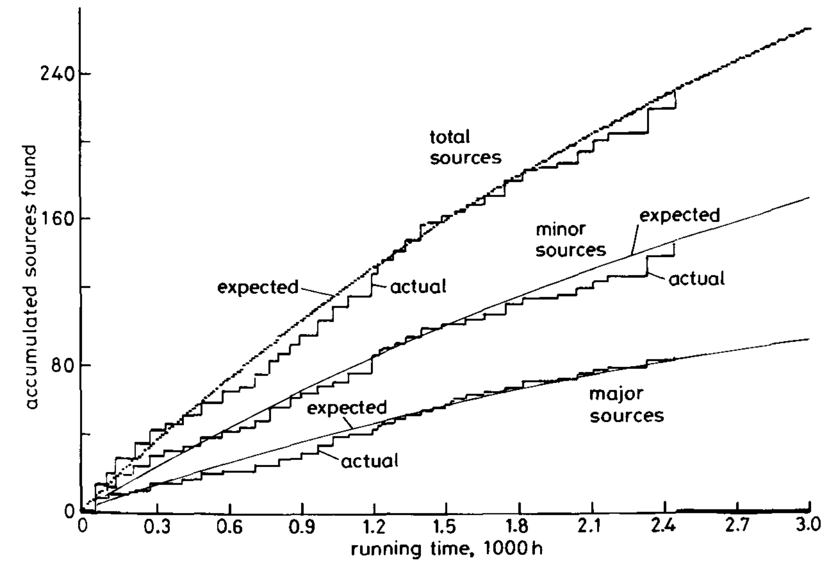


Fig. 9 Misra 83: actual and expected accumulation for three sets

Table 12*a* **Misra83: parameter values and derived rates**

Set	n	h	s	s/h	ns/h	$s/(h+u_m)$	$R(u_m)$
Major	177	17·1570	4·721	0·2752	48·71	0·2407	22·65
Minor	497	5·9503	1·022	0·1718	85·36	0·1216	42·43
Total	727	6·5261	1·195	0·1832	133·15	0·1331	66·03

Table 12*b* **Misra83: results using Goel-Okumoto model[13]**

Set	a	b
Major	163·813	0·28759
Minor	315·551	0·25756
Total	597·887	0·20988

Table 12*b* shows the results obtained by Misra using the Goel-Okumoto model[9]. *a* is the expected number of faults initially present (corresponding to *n* in LSRG) and *b* is the individual manifestation rate of each source (the same for all sources – compare with s/h).

Table 13*b* shows the predicted and actual faults found for the 200 h mission following the collection of the data set. The predictions using the Goel-Okumoto (G–O) model are quoted from Misra[8]. The expected value is optimistic, but the actual figures lie within the 90% confidence interval. The expected value from the LSRG model is very close to the observed numbers of faults. The 'Duane' figure is obtained by fitting a simple power curve to the data (see Section 8.1) and is closer still. On some sets, though, (e.g. B200 – see 8.1 again) this simple procedure is defeated by anomalous data.

It must be borne in mind that this is a very short-term prediction.

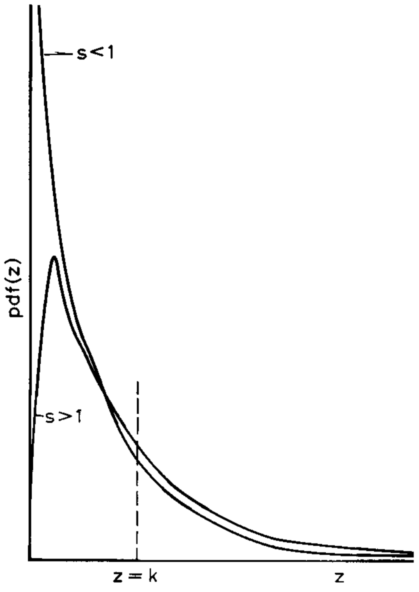


Fig. 10 Distribution of source manifestation rate z for different values of shape parameter s

Table 13*a* Misra83: expected accumulated manifest sources and rates

Use kh	1		3		10		30	
Set	Acc.	Rate	Acc.	Rate	Acc.	Rate	Acc.	Rate
Major	42	35·2	94	19·4	157	3·52	176	0·15
Minor	73	62·3	170	37·4	316	11·6	418	2·24
Total	114	97·4	264	58·0	487	17·3	634	3·04

Table 13*b* Misra83: predictions for next 200 h by various models

Prediction: Set	G-O Acc.	G-O(90%le) Acc.	LSRG Acc.	Duane Acc.	Actual Acc.
Major	3	6	4·4	5·3	5
Minor	7	11	8·3	8·6	9
Total	13	19	12·8	13·8	14

7 Interpretation of results

7.1 Types of data set

As analysed by this MLE procedure, on the basis of the LSRG model, there seem to be two distinct types of data set.

Type 1 is characterised by

- large n, of the order of 10 c_m or 100 c_m
- small h, of the order of 0·01 u_m
- small s, of the order 0·01.

As the ML search proceeds, n increases and s and h both decrease from the default starting values given in Section 4.3.

Type 2 is characterised by

- small n, of the order of 1·5 c_m
- large h, of the order of 10 u_m
- large s, between 2 and 10.

In this case, as the search proceeds n decreases and s and h both increase.

It is not clear why this should be. It is presumably a shortcoming of the inference procedure. Small data sets and those with less obvious reliability growth tend to be of type 2. Medium-term predictions from both types can be similar, but long-term predictions from type 2 are usually 'all faults found, zero rate'. If there is good reason for doubting this, e.g. if it implies an absurdly low number of faults per line of source code, we should consider fixing s at a low value during MLE. This is an interesting point of contact between inspection defect counting and reliability measurement.

In some cases (e.g. Moek83A), $\hat{n}$ may be around c_m + 1. Moek[5] presents graphs of $\hat{n}$ as it changes as the data set grows. It is interesting to note that in set A, $\hat{n}$ comes down to meet the graph of c_i, whereas in B it increases with c_i as the set grows.

7.2 n

It is unfortunate that some researchers using fault-counting models have regarded this as of primary importance. Although a large n indicates that hypothetically very many faults are there to be found, the fact that the corresponding value of s is always small means that most of those faults would have such low manifestation rates that they would not be seen in the lifetime of the product.

n is, in theory, an upper limit on the number of sources manifest. In practice, a re-estimate after further observation generally yields an increased n, and values of n derived from the earlier part of a trial may be exceeded by the number of actual sources found later.

n should be regarded merely as one parameter for predicting the number of sources that will be observed in some specified period of future use.

7.3 h

h has the dimensions of product use. It can be interpreted as 'product exposure prior to start of trial'[1]. From equation 3, it is evident that at product use $-h$, the failure rate of the product was infinite. However, the widely differing value of h between type 1 and 2 data sets means that we should be wary of interpreting this parameter too literally, also.

7.4 s

$s < 1$ for type 1 data sets, $s > 1$ for type 2.
s governs the shape of $pdf(z)$, Z being the manifestation rate of an individual source.

$$s > 1 \Rightarrow pdf\ (0) = 0, pdf\ (z) \text{ unimodal}$$
$$s < 1 \Rightarrow pdf\ (z) \rightarrow \infty \text{ as } z \rightarrow 0$$

For a given period of observation, we can draw a line z = k on the graph of $pdf(z)$ such that the chances of seeing any source to the left of the line are less than 1%, say.

$$\text{i.e. } Pr\left\{\text{source manifest} \mid z < k\right\} < 0{\cdot}01$$

As the period of observation increases, the line moves to the left. When it passes the hump of the unimodal pdf, the product behaviour in the two cases diverges (Fig. 10).

In the type 1 case, we expect to see 1% of a rapidly increasing set of very-low-rate sources. The total manifestation rate, although declining, never gets close to zero. The number of sources is effectively infinite. (In the limiting case of this model, with infinitely many sources each of infinitesimal manifestation rate, no reliability growth would be observed. There may be a variant of the model in which growth is seen, but which imposes no limit on the number of sources.)

For type 2, the manifestation rate falls sharply, and after a long period of observation the expected number of sources manifest is very close to the hypothetical total. We have effectively achieved zero defects.

It must be emphasised that the above is what we would in theory see on the basis of the LSRG model. In practice, particularly for large operating system software, what is seen resembles type 1 behaviour rather than type 2. Products exhibiting type 2 data, instead of yielding perfection with increased observation, yield larger n.

I conjecture:

(i) The hump is an artefact of the model, due to the choice of the gamma distribution to describe individual source rates.
(ii) For $s > 1$, we should not rely on a prediction so far ahead that it takes us 'over the hump'.

7.5 Length of trial

Once sufficient data have been gathered to give reasonable n, h and s we can predict the length of further trial required to achieve a target reliability at release (assuming that sources manifest in the trial are removed, or declared as known deficiencies, or otherwise do not contribute toward the incident rate in the field).

If u_t and u_l are product use during trial and life, respectively, and $M(u_t)$ and $M(u_t + u_l)$ the expected sources manifest up to end of trial, end of life, then the number of sources we may expect to 'miss' during a trial is:

$$M(u_t + u_l) - M(u_t) = n\ [(h/(h + u_t))^s - (h/(h + u_t + u_l))^s] \qquad (12)$$

This provides an estimate of the expected effectiveness of a trial of given length for a given product life.

8 Other models and techniques

8.1 Models

There are very many published models. The reader requiring an overview of the field is referred to the excellent survey paper by Dale and Harris [10].

One class of model, usually known as the Duane type, involves fitting a straight line to the graph of log(accumulated sources) against log(accumulated use). This in effect fits a power curve to the data

$$E\left\{C\right\} = au^b \qquad (13)$$

The corresponding formula for the rate is:

$$R(u) = abu^{\,b-1} \qquad (14)$$

The justification for this procedure is empirical, and for some data sets it gives remarkably good fit and accuracy of prediction. For others it fails badly. For example, its sensitivity to the dubious early data in B200A leads it to predict reliability decay, i.e. $b > 1$, whereas LSRG is fairly resilient to this.

A variant of this procedure[11], used in ICL, involves estimating the failure rate r week by week by dividing reported failures by running time and fitting a straight line to the graph of log(r) against log(accumulated running time). This has been done in the past using only system-crash incidents, and with no distinction between new sources and repeats. Comparison with the LSRG model on the basis of data now available is therefore difficult.

Littlewood[12] has conjectured that the Duane model may be reducible to an underlying stochastic model of the LSRG type.

8.2 Least-squares estimation

An alternative routine to MLE has been devised to estimate n, h and s by minimising the square of the distances of the data points from the graph of unconditional expected accumulation.

The function minimised is:

$$D = \sum_{i=1}^{m} [c_i - M(u_i)]^2 = \sum_{i=1}^{m} [c_i - n(1 - (h/(h + u_i))^s)]^2 \qquad (15)$$

Use has been limited, but results so far have been found to be similar to those with MLE.

8.3 Discrete classes of source

The cost-forecasting program[1] models the differing individual source manifestation rates by assigning various numbers of sources to different classes, all sources in one class having the same rate. If for w classes we assign

$$\begin{aligned} &\text{number of sources} = nPr\{x_{i+1} < z \leqslant x_i\} \\ &\text{class rate} \qquad\quad\;\; = E\{Z \mid x_{i+1} < z \leqslant x_i\} \end{aligned}$$

where $x_w = 0$ and x_1 is large enough to enable us to ignore the right-hand tail of $pdf\ (z)$ beyond that value, then the result gives a fair simulation of the actual gamma source rate distribution.

9 Conclusions and further work

The extent of our predictions and their accuracy are limited by the length of trial and similarity of product usage to real life. No conceivable estimation technique can make up for an inadequate trial. Measurement of reliability must be a part of the trial plan, and mechanisms must be in place to provide data to the necessary standard.

The value of a stochastic model is that it enables us to define metrics which in turn can be used to express trial targets. These targets can be related to expected support cost and customer-perceived reliability. They should be expressed as reliability or failure rate, both as estimated at end of trial and expected after given periods of field use. MTTF is misleading when applied to software, and should not be used.

The ultimate test of a model is the accuracy of its prediction. Formal methods are available for assessing the accuracy of 'time to next failure' predictions[13]. Similar methods must be devised for predictions of 'sources manifest in the next period', and will be the subject of a future paper.

Since the inference procedure described here gives dubious results on certain data sets, particularly small ones, improved procedures should be investigated. The use of simulated data sets will provide a check on inference procedure performance independent of the vagaries of real data.

Acknowledgments

I am once again indebted to Dr. Bev Littlewood of The City University, not only as the author of the model here applied, but also for his continued help and advice on the theory and comments on and corrections to the first draft of the paper (needless to say, any mistakes that remain are my own). Thanks are also due to his assistant Pee Yuaw Chan; to Dr. Barbara Kitchenham of the Software Engineering Centre, ICL Kidsgrove, for her support; to Dr. G. Moek and Dr. P.N. Misra for the use of their previously published data; to the Editor of the *IBM Systems Journal*, J.A. Lacy, for permission to reprint the data; and to John Loomis, John Marriott and Mike Drury of Group Quality management for encouraging the academic involvement which has made this work possible. Lastly, since every author owes a debt to his family and loved ones, I would like to thank mine for giving me all the time I needed to write.

References

1 MELLOR, P.: 'Modelling software support', *ICL Tech. J.*, 1983, **3**, (4), 407-438.
2 LITTLEWOOD, B.: 'Stochastic reliability growth: a model for fault removal in computer programs and hardware designs', *IEEE Trans.*, 1981, **R-30**, 313-320.
3 MUSA, J.D.: 'A theory of software reliability and its application', *ibid*, 1975, **SE-1**, 312-327.
4 JELINSKI, Z. and MORANDA, P.B.: 'Software reliability research', in *'Statistical computer performance analysis'*, Academic Press, New York, 1972.
5 MOEK, G.: 'Software reliability models on trial: selection, improved estimation, and practical results'. Available from National Aerospace Laboratory NLR, Informatics Division, P.O. Box 90502, 1006 BM Amsterdam, The Netherlands, ref. NLR MP 83059.
6 BARLOW, R.E., BARTHOLOMEW, D.J., BREMNER, J.M. and BRUNK, H.D.: *'Statistical inference under order restrictions – the theory and application of isotonic regression'*, John Wiley & Sons, 1972.
7 CAMPBELL, G. and OTT, K.O.: 'Statistical evaluation of major human errors during the development of new technological systems' *Nuclear Sci. & Eng.*, 1979, **71**, 267-279.
8 MISRA, P.N.: 'Software reliability analysis', *IBM Syst. J.*, 1983, **22**, (3), 262-270.
9 GOEL, A.L. and OKUMOTO, K.: 'Time-dependent error detection rate model for software reliability and other performance measures.' *IEEE Trans.*, 1979, **R-28**, (3), 206-211.
10 DALE, C.J. and HARRIS, L.N.: 'Reliability aspects of microprocessor systems', 1981. Available from Department of Industry, report T816164.
11 DRURY, M., STRASS, P., BOWMER, R.A. and GODDING, J.F.: 'A model for software failure rates', ICL Quality Reliability internal report CQA/A/R293, 28.06.76.

12 LITTLEWOOD, B.: 'Rationale for a modified Duane model', *IEEE Trans.*, 1984, **R-33**, April.

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

Appendix

The formulae of Section 3.5 and other results are derived from the assumptions of Section 3.3 as follows (see Section 2 for notation).

Basic result

The manifestation rate of a given individual source is a random variable Z (z denotes a realisation of it).

Since each source is Poisson:

$$Pr\{c \text{ incidents caused by source during product use } u \mid Z = z\}$$

$$= \frac{(zu)^c}{c!} \exp(-zu)$$

In particular

$$Pr\{\text{source not manifest during use } u \mid Z = z\} = Pr\{C = 0 \mid Z = z\} = \exp(-zu)$$

$$Pr\{\text{source } is \text{ manifest during use } u \mid Z = z\} = Pr\{C \neq 0 \mid Z = z\} = 1 - \exp(-zu)$$

Also, at the start of the trial, when $u = 0$, Z has a gamma *pdf*, shape parameter s and scale parameter h:

$$pdf(z) = \frac{1}{(s-1)!} h^s z^{s-1} \exp(-hz)$$

Applying Bayes' Theorem for a continuous distribution with discrete observations (the 'observation' has two possible outcomes: the source manifests itself or it does not):

$$pdf(z \mid \text{source not manifest in } (0, u))$$

$$= \frac{pdf(z)\, Pr\{\text{source not manifest in } (0,u) \mid Z = z\}}{\int_0^\infty pdf(z)\, Pr\{\text{source not manifest in } (0,u) \mid Z = z\}\, dz}$$

Denominator = Pr { source not manifest in $(0, u)$}

(i.e not conditional on rate)

$$= \int_0^\infty \frac{1}{(s-1)!} h^s z^{s-1} \exp(-hz) \exp(-uz) dz$$

$$= \frac{h^s}{(h+u)^s} \int_0^\infty \frac{1}{(s-1)!} (h+u)^s z^{s-1} \exp(-(h+u)z)\, dz$$

The integrand is a gamma *pdf*, hence integral = 1, denominator = $h^s / (h+u)^s$ and *pdf* $(z$ | source not manifest in $(0, u))$

$$= pdf(z)\, Pr\left\{ \text{source not manifest in } (0,u) \mid Z = z \right\} (h+u)^s / h^s$$

$$= \frac{1}{(s-1)!} (h+u)^s z^{s-1} \exp(-(h+u)z) \qquad \text{(A1)}$$

i.e. the pdf of an individual source manifestation rate Z for sources remaining after product use u is a gamma pdf with shape s and scale h + u.

This basic result underlies all the others.

Failure rate

The *pdf* of rate Z applies to each source. If c sources have manifested themselves during use u, so that $n - c$ remain, then the total rate R (i.e. manifestations of *new* sources) from the whole product is the sum of $n - c$ independent identically (gamma) distributed random variables (IIDRVs): i.e. the sum of the rates of all sources remaining. The *pdf* of $n - c$ gamma IIDRVS with shape s is also gamma, with the same scale, but shape $(n - c)\, s$.

Hence:

$$pdf(r \mid c \text{ sources manifest}) = \frac{(h+u)^{(n-c)s}}{[(n-c)s - 1]\,!} r^{(n-c)s-1} \exp(-(h+u)r) \qquad \text{(A2)}$$

where r denotes a realisation of the random variable R.

Since the mean of a gamma distribution = shape/scale, we have formula 2 for the expected rate R $(u \mid c)$ of occurrence of incidents after product use u and c sources found:

$$R(u \mid c) = (n-c)\, s / (h+u) \qquad \text{(A3)}$$

Accumulated sources found and reliability

The probability, not conditional on rate, that any source will not manifest itself during use u was shown above to be $h^s/(h+u)^s$. So:

$$Pr\left\{\text{source is manifest during use } u\right\} = 1 - (h/(h+u))^s$$

Since there are n sources, it follows immediately that

$$M(u) = n\left(1 - (h/(h+u))^s\right) \tag{A4}$$

which is formula 1 from Section 3.5. Similarly, we can show that

$$Pr\left\{\text{given source not manifest in } (u, u+t) \mid \text{not manifest in } (0,u)\right\}$$

$$= ((h+u)/(h+u+t))^s \tag{A5}$$

(Putting $u = u_{i-1}$, $t = u_i - u_{i-1}$, gives the starting point for the likelihood function quoted in Section 4.2.)

It follows that if $n - c$ sources remain at u, then:

$$Pr\left\{\text{no incident in } (u, u+t) \mid c \text{ incidents in } (0,u)\right\}$$

$$= S(t \mid u, c) = \left(\frac{h+u}{h+u+t}\right)^{(n-c)s} \tag{A6}$$

which is formula 4 in Section 3.5.

The unconditional reliability after use u is derived as follows:

$$S(t \mid u) = E\left\{S(t \mid u, c)\right\} = \sum_{c=0}^{n} S(t \mid u, c)\, Pr\left\{C = c\right\}$$

$$= \sum_{c=0}^{n} \left(\frac{h+u}{h+u+t}\right)^{(n-c)s} \binom{n}{c} \left(\frac{h}{h+u}\right)^{s(n-c)} \left(1 - \left(\frac{h}{h+u}\right)^s\right)^c$$

$$= \sum_{c=0}^{n} \binom{n}{c} \left(\frac{h}{h+u+t}\right)^{s(n-c)} \left(1 - \left(\frac{h}{h+u}\right)^s\right)^c$$

$$= \left[1 - \left(\frac{h}{h+u}\right)^s + \left(\frac{h}{h+u+t}\right)^s\right]^n$$

which is eqn. 5 in Section 3.5

For this, note that C is binomially distributed, i.e.

$$Pr\left\{ c \text{ incidents in } (0, u) \right\} = \binom{n}{c} p^{n-c} (1-p)^c$$

where $p = Pr\,\{$ any given source *not* manifest in $(0, u)\} = \left(\frac{h}{h+u}\right)^s$ as above.

The binomial theorem is applied to sum the series into the final expression. This can be interpreted as:

For no incident to occur in $(u, u + t)$, all n sources must manifest themselves *either* before u *or* after $u + t$.

It is interesting to note that eqn. A4 can be derived from a model embodying the following two assumptions (treating the expected number of sources manifest after use u, $M(u)$, as a deterministic quantity and the expected rate of occurrence of incidents as its derivative, $M'(u)$):

(*a*) rate of occurrence of incidents is proportional to the number of sources remaining
(*b*) rate of occurrence of incidents is inversely proportional to product use, u + some constant, h.

From (*a*) and (*b*)

$$M'(u) = s\,(n - M(u))/(h + u) \tag{A7}$$

where s is the constant of proportionality (i.e. the equivalent of eqn. A3 is an immediate consequence of the assumptions).

Applying the standard solution of the linear first-order differential equation, we obtain

$M(u) = K(h + u)^{-s} + n$ where K is the constant of integration.

Since $M(0) = 0$, $K = -nh^s$ and eqn. A4 follows immediately.

Product use to next incident

Let the random variable T denote 'product use to next incident' and t denote its realisation.

$$pdf(t \mid c) = \int_0^\infty pdf(t \mid R = r)\, pdf(r \mid c)\, dr$$

$$= \int_0^\infty r\exp(-rt)\,\frac{(h+u)^{(n-c)s}}{[(n-c)s-1]\,!}\,r^{(n-c)s-1}\exp(-(h+u)r)\,dr$$

$$= \frac{(n-c)\,s\,(h+u)^{(n-c)s}}{(h+u+t)^{(n-c)s+1}((n-c)s)!} \times$$

$$\int_0^\infty \frac{(h+u+t)^{(n-c)s+1}}{((n-c)s)\,!}\,r^{(n-c)s}\exp(-(h+u+t)r)\,dr$$

$$= (n-c)\,s\,(h+u)^{(n-c)s} \,/\, (h+u+t)^{(n-c)s+1} \qquad \text{(A8)}$$

since integral = 1, the integrand being *pdf*. Hence

$$E\left\{T\,|\,u,c\right\} = \int_0^\infty t\,pdf(t)\,dt = \int_0^\infty \frac{(n-c)\,s\,(h+u)^{(n-c)s}\,t}{(h+u+t)^{(n-c)s+1}}\,dt$$

Integrating by parts, we have

$$E\left\{T\,|\,u,c\right\} = -\left[t\left(\frac{h+u}{h+u+t}\right)^{(n-c)s} + \frac{(h+u)^{(n-c)s}}{((n-c)s-1)\,(h+u+t)^{(n-c)s-1}}\right]_0^\infty$$

If $(n-c)\,s > 1$, this converges to $(h+u)/((n-c)s-1)$, otherwise $E\left\{T\,|\,u,c\right\}$ does not exist. (A9)

All the above results and more are contained in Littlewood[2]. They are repeated here (in more detail) for completeness and since a somewhat different notation has been used.

Towards a formal specification of the ICL Data Dictionary

B. Sufrin
Oxford University Computing Laboratory, Programming Research Group, Oxford

Abstract

In this paper we present a formal specification of the ICL Data Dictionary System, paying particular attention to the facilities it provides for controlling the retrieval and updating of dictionary elements. We conclude by suggesting some modifications to the design which would render the system simpler while retaining its full power. The specification notation Z, which is based on set theory, is used, and familiarity with the mathematical notions of predicate, set, relation and function is assumed throughout. A glossary of symbols is provided.

1 Preface

The potential benefits of applying formal, or at least mathematically rigorous, methods to the design and production of software are currently topics of much discussion and have been eloquently expounded elsewhere[1,2]. In common with many others, we believe that the time is ripe, perhaps even over-ripe, for the application of these methods in an industrial context. This paper arose out of a challenge from ICL to work with a group of their practising programmers to investigate the applicability of the methods to a real commercial product, the ICL Data Dictionary System (henceforward DDS). The company sponsored a ten-day pilot project, during which we used mathematical techniques to investigate two areas of DDS which its designers believe to be difficult to understand and explain, namely the means provided for controlling access to dictionary elements and the support provided for multiple versions. This paper is a report of our investigation of access control.

2 Introduction

It is by now a matter of common knowledge that a substantial proportion of the total costs of 'bugs' discovered during the lifetime of a computer-based system can be attributed to mistakes made during the earliest stages of its development. It is for this reason that we believe that the most appropriate time to construct – and to use – a formal specification for a large system is before the system is built. Why, then, attempt a mathematical specification of parts of a software product which is already several years old? First, the DDS documentation of access control is rather difficult to understand, partly because it is not all in one place. A document from which the answers to questions about access can *easily* be deduced will be useful in its own right, and perhaps provide a basis for better documentation. Secondly, a byproduct of the specification activity will be the construction of a conceptual framework within which the consequences of simplifying the design of the system can be investigated. Thirdly, even though we hardly expect the majority of DDS users to be able to read a mathematical specification, to produce it at all we are forced to ask questions of the implementers of the system which the standard documentation fails to answer adequately. These questions and their answers will certainly be of use to the authors of subsequent editions of the documentation. Finally, if used properly the specification should also help to ensure compatibility of successive versions and new implementations of the Dictionary.

3 Overview of the Data Dictionary System

The potential application areas of the Data Dictionary System are outlined in detail in the first chapter of its reference manual[3]. In essence it is a database system specially adapted to the needs of supporting the construction, documentation and maintenance of collections of programs which must be kept mutually consistent. Such collections are to be found at computer installations everywhere, and without computer-based support they can quickly become very difficult to manage.

For the purposes of this paper it is enough for us to note that the system provides a means of naming, storing, manipulating, and enquiring about any number of *elements*, each of which may have several named *properties* which possess *values*. Some properties are possessed only by certain types of element, for example the *TITLE-PAGE property of REPORT–PROGRAM elements. Other properties may be possessed by any or all elements but are never acted upon by DDS, for example the *DESCRIPTION and *NOTE properties, whose values are uninterpreted text. Finally, there is a class of properties which are administrative in nature and may be possessed by any or all elements and which *are* interpreted (acted upon) by the DDS, for example the *PRIVACY and *AUTHORITY properties which are possessed by almost all elements. It is by setting administrative properties such as these that an administrator may control aspects of the behaviour of a Data Dictionary at a particular installation, and users may control how the elements they define can be manipulated by others.

To begin constructing a mathematical model we must first introduce some nomenclature. Let P denote the set of all possible names for *properties* possessed by elements in the database, and let V denote the set of all possible *values* which these properties may take. For the moment we need not investigate the internal structure of the sets P and V, although we will do so later.

In the manual, the term 'element' means an object consisting of a collection of named properties which have values. Such an object may be modelled as a finite mapping from property names to values. Let E denote the set of all possible elements storable in a DDS database; we define it formally by:

$$E \mathrel{\hat{=}} P \mathrel{\twoheadrightarrow\!\!\!\!\!\!\!\;+} V$$

Notice that this definition merely explains the essence of 'elementhood', but gives us no clues about how to represent elements using the data structures which are available in a conventional programming language.

If we take some liberties with the way in which we write values, and suppose that *authority, description, note* and *privacy* are among the possible property names, then we can give a couple of (possibly untypical) examples of elements:

$$\{ \mathit{authority} \mapsto \text{Bernard}; \mathit{description} \mapsto \text{'Formal documentation'} \}$$

$$\{ \mathit{authority} \mapsto \text{Bernard}; \mathit{privacy} \mapsto 99; \mathit{note} \mapsto \text{'What is a note for anyway?'} \}$$

The term 'element identifier' means the *name* by which an element is known to the DDS. If we let EI denote the set of *all possible* element identifiers, then the current state of a DDS may be modelled as an object from the set

$EI \mathrel{\twoheadrightarrow\!\!\!\!\!\!\!\;+} E$ which can be expanded to $EI \mathrel{\twoheadrightarrow\!\!\!\!\!\!\!\;+} (P \mathrel{\twoheadrightarrow\!\!\!\!\!\!\!\;+} V)$

i.e. a mapping from element identifiers to elements, which are themselves mappings from property names to storable values. Notice that to describe the state of a DDS at this level of abstraction it is not necessary for us to give details of the internal structure of element identifiers, although we will be forced to reveal this structure later.

4 Access control

In the first part of this section we present a sequence of successively more accurate descriptions of the abstract information structures of a DDS which support access control. This is done without making explicit the fact that the dictionaries are self-describing – in the sense that these information structures are completely described by elements present in the dictionaries themselves. In the second part the state of a running DDS is considered, and it is shown how some simple commands issued by users affect that state. In the third part we demonstrate in detail the relationship between the abstract description and the stored elements with which the dictionary implements the structures introduced in the first. Finally several more complex commands, whose effects depend on the details just demonstrated, are described.

The technical terms and the notation used are explained in the Glossary (Appendix 2).

4.1 Abstract information structures of the DDS

Our first approximation is rather simple; we simply observe the set of element names known to the system and the correspondence between these names and their stored values. More formally, we define a schema *DD* which characterises the possible states of a DDS.

$$\begin{array}{|l}
\hline
\textit{DD} \\
\textit{elements}: \; \mathbb{F}\, EI \\
\textit{store}: \quad EI \nrightarrow E \\
\hline
\textit{elements} = \textit{dom store} \\
\hline
\end{array}$$

The predicate below the bar records the invariant relationship expected to hold between the two observations: the element names which are known to the system are exactly those which correspond to elements in the store.

A formalisation at this level of abstraction could serve as the basis of a model for the data stored in *any* entity-attribute database! Since it fails to take into account the *specific* characteristics of DDS that we are trying to explain, we shall discard it.

In our second approximation the fact that some elements have owners is recorded. These are called authority elements in the documentation, and are the basis for one of the methods by which access to elements is controlled. When ordinary users run one of the DDS programs, they may do so under the aegis of an authority. Their access to elements in the dictionary depends amongst other things upon this authority.

We begin to formalise this by introducing additional observations, namely the finite set of authority element identifiers which have been introduced by the dictionary administrator into the system, and a mapping from the identifiers of stored elements to those of their owners.

$$\begin{array}{|l}
\hline
\textit{DD} \\
\textit{store}: \quad EI \nrightarrow E \\
\textit{elements}: \; \mathbb{F}\, EI \\
\textit{auth}: \quad \mathbb{F}\, EI \\
\textit{owner}: \quad EI \nrightarrow EI \\
\hline
\textit{elements} \; = \; \textit{dom store} \\
\textit{ran owner} \subseteq \textit{auth} \subseteq \textit{elements} \\
\textit{dom owner} \subseteq \textit{elements} - \textit{auth} \\
\hline
\end{array}$$

The new predicates below the bar record the additional invariant relationships between the observations: all owners of elements must be authority elements, all authority elements must be present in the store, but no authority element has an owner.

Notice that the last predicate is such that not all elements need an owner; indeed it is consistent with a state in which *no* elements have owners. It turns out that it is easier to explain the system if a special 'mythical' authority is invented – the nil authority – and insist that every nonauthority element has an owner (which might be the nil authority). This is formalised in two small steps; first we introduce a constant element identifier, *nil*, to stand for the identifier of the mythical nil authority.

$$\boxed{\; nil : EI \;}$$

Next we give a new description of a DDS state, which incorporates the second approximation but strengthens the invariant with two additional predicates: the first states that *nil* is an authority element, and the second that *all* nonauthority elements must now have owners.

$$\begin{array}{|l}
\textit{DD} \\
\hline
\textit{DD} \\
\hline
nil \in auth \\
\mathrm{dom}\ owner = elements - auth \\
\hline
\end{array}$$

Note: In our formalism, this description is not self-referential or recursive, but is an extension of *DD*, i.e. a redefinition which incorporates the prevailing definition. Of course, for such an extension to be useful, the predicates which are added must be consistent with those already present.

We now engage in a little speculation (with the best of pedagogical motives): if the DDS designers had had an authoritarian or individualistic cast of mind, they might have stopped their design activity at this point and insisted that only the owner of an element can retrieve or update it. Under these circumstances we would have been able to end our modelling activity by making just one more observation of the state of a dictionary: the relation *canaccess*, which can be completely determined by the values of the remaining observations of *DD*, holds between an authority and an element exactly when a user running under that authority can retrieve or update the element.

$$\begin{array}{|l}
\textit{AuthoritarianDD} \\
\hline
\textit{DD} \\
canaccess : EI \leftrightarrow EI \\
\hline
\forall\ user{:}auth;\ elt{:}elements. \\
\quad user\ \mathbf{canaccess}\ elt \Leftrightarrow owner\ elt = user \\
\hline
\end{array}$$

Note: in this extension we have added a new observation as well as a new invariant to the definition of *DD*.

A marginally more libertarian group of designers might have interpreted ownership by the nil authority somewhat differently and allowed anybody to retrieve or update such elements:

NotQuiteSoAuthoritarianDD

DD

$canaccess: EI \leftrightarrow EI$

$$\forall\ user{:}auth;\ elt{:}elements.\quad user\ canaccess\ elt \Leftrightarrow owner\ elt \in \{user, nil\}$$

As might be expected, the ICL designers wanted to make their system a little more flexible than either of these descriptions indicate, and we find both that they have included a number of ways in which elements may be shared between authorities and that they distinguish between retrieval and updating.

An authority may delegate rights to retrieve an element to one or more other authorities. We formalise this by introducing the relation *delegates*, which holds between an authority *a* and an authority *a'* if and only if *a* has taken steps to permit *a'* to retrieve *all* the elements *a* owns.

DD

DD

$delegates: EI \leftrightarrow EI$

$$delegates \in (auth \leftrightarrow auth)$$

To formalise the fact that rights to a *single* element may be given by its owner to another authority we introduce another relation, *mayretrieve*, which holds between an authority *a* and an element *e* only if *e*'s owner has explicitly taken steps to allow *a* to retrieve it.

DD

DD

$mayretrieve: EI \leftrightarrow EI$

$$\begin{aligned} &dom\ mayretrieve \subseteq auth \\ &ran\ mayretrieve \subseteq elements - auth \end{aligned}$$

As we shall see in the next section, part of the stored description of an authority element is a description of the types of element which it may not update. Since our description is not yet at a level of detail which includes types, we can discuss the right of an authority to *update* an element in the database only in rather general terms. To begin the discussion we add a relation *maynotupdate* to our observations. This relation holds between a declared authority and the names of elements which it has explicitly been forbidden to update; these can include elements which are not yet in the store.

$$
\begin{array}{|l}
\textit{DD} \\
\hline
\textit{DD} \\
\textit{maynotupdate}: EI \leftrightarrow EI \\
\hline
\textit{dom maynotupdate} \subseteq \textit{auth} \\
\hline
\end{array}
$$

Note: Readers familiar with DDS will recognise that *delegates* is closely related to the *RETRIEVE property of authority elements, that *mayretrieve* relates to the *RETRIEVE property of nonauthority elements and that *maynotupdate* is related to the *INHIBIT properties of authority elements.

If the designers had stopped here, we would characterise an authority's rights to retrieve and update elements by beginning to define the relation *canretrieve* which holds between an authority and the elements it is permitted to retrieve, and the relation *canupdate* which holds between an authority and the elements which the system will allow it to update. At this stage the only thing we can say about the updating is negative: namely that if an authority has explicitly been forbidden to update an element then the system will prevent it from doing so.

$$
\begin{array}{|l}
\textit{DD} \\
\hline
\textit{DD} \\
\textit{canretrieve}: \quad EI \leftrightarrow EI \\
\textit{canupdate}: \quad EI \leftrightarrow EI \\
\hline
\forall\ \textit{user:auth; elt:elements}. \\
\quad (\ \ \textit{owner elt} \in \{\textit{user, nil}\} \quad \vee \\
\qquad \textit{(owner elt)}\ \textbf{\textit{delegates}}\ \textit{user} \quad \vee \\
\qquad \textit{user}\ \textbf{\textit{mayretrieve}}\ \textit{elt} \quad) \Rightarrow \textit{user}\ \textbf{\textit{canretrieve}}\ \textit{elt} \\
\\
\forall\ \textit{user:auth; elt:elements}. \\
\qquad (\textit{user}\ \textbf{\textit{mayretrieve}}\ \textit{elt}) \quad \Rightarrow \quad (\textit{user}\ \textbf{\textit{canupdate}}\ \textit{elt}) \\
\hline
\end{array}
$$

Although the system we have described above might have satisfied many designers, it turns out that orthogonal to the system of ownership the DDS has a notion of *levels of privacy*. Stored elements have a privacy level, which is a number between 0 and 99. Irrespective of the possibilities for retrieval afforded by the ownership system, a user running under an authority may retrieve any element whose privacy level is less than that of the authority. (Incidentally, the documentation indicates that the privacy level of an element may be higher than that of its owner, a fact which we find puzzling). The nil authority is given a privacy level of 0, which reflects its role as the 'owner' of elements intended to be universally retrievable. More formally

DD

DD

$privacy : EI \text{⇻} \mathbb{N}^{99}$

$dom\ privacy = elements$

$privacy\ nil = 0$

$\forall\ user: auth;\ elt: elements.$

$\quad privacy\ elt \leqslant privacy\ user \Rightarrow user\ canretrieve\ elt$

A dictionary administrator may decide for operational reasons to nominate one authority as the master authority. This authority (if one has been nominated) may retrieve and update any element in the dictionary, irrespective of ownership. A master authority should not be confused with the dictionary administrator: although the two roles might be played by the same person in many organisations, their functions are entirely different.

The only element which does not have a privacy level in the range 0..99 is the master authority (if there is one). For our purposes it will simplify matters if we attribute privacy level 99 to a master element if one exists: while this is not strictly in accordance with the choice of representation made by the ICL designers, its consequences are precisely the same. Note that the master authority (if there is one) may not be forbidden to update elements. Note also that by virtue of its privacy level the master authority has the right to retrieve any element at all.

DD

DD

$master: \mathbb{F}^1\ EI$

$master \subseteq auth$

$privacy\ \llbracket master \rrbracket \subseteq 99$

$master \cap (dom\ maynotupdate) = \{\ \}$

This almost concludes the first part of the description of the information structures which characterise a DDS and the invariant relations which hold between them. In Fig. 1 these information structures are summarised, formally simplifying some of the predicates and recording two more things. The first of these is that the conditions hitherto outlined are the *only* conditions under which retrieval can take place. The second characterises updating more positively: an element may be updated by its owner or by the master authority. Notice that it is possible for a nonmaster authority which owns an element to be prevented from updating it.

Since we have not yet given any details of the structure of values, we have no way yet of recording the fact that the owner of a stored element is stored as its *authority* property. Nor can we record the fact that the set of authorities to

$$
\begin{array}{|l}
\hline
\textit{DD} \\
\begin{array}{ll}
\textit{store:} & El \nrightarrow E \\
\textit{elements:} & \mathbb{F}\, El \\
\textit{auth:} & \mathbb{F}\, El \\
\textit{owner:} & El \nrightarrow El \\
\\
\textit{delegates:} & El \leftrightarrow El \\
\textit{mayretrieve:} & El \leftrightarrow El \\
\textit{maynotupdate:} & El \leftrightarrow El \\
\\
\textit{privacy:} & El \nrightarrow \mathbb{N}^{99} \\
\textit{master:} & \mathbb{F}^{1}\, El \\
\\
\textit{canretrieve} & El \leftrightarrow El \\
\textit{canupdate:} & El \leftrightarrow El
\end{array} \\
\hline
\begin{array}{l}
\textit{elements} = \textit{dom store} \\
\textit{ran owner} \subseteq \textit{auth} \subseteq \textit{elements} \\
\textit{dom owner} \subseteq \textit{elements} - \textit{auth} \\
\\
\textit{nil} \in \textit{auth} \\
\textit{dom owner} = \textit{elements} - \textit{auth} \\
\\
\textit{delegates} \in (\textit{auth} \leftrightarrow \textit{auth}) \\
\textit{dom mayretrieve} \subseteq \textit{auth} \\
\textit{ran mayretrieve} \subseteq \textit{elements} - \textit{auth} \\
\textit{dom maynotupdate} \subseteq \textit{auth} \\
\\
\textit{dom privacy} = \textit{elements} \\
\textit{privacy nil} = 0 \\
\textit{master} \subseteq \textit{auth} \\
\textit{privacy}\, [\![\textit{master}]\!] \subseteq 99 \\
\textit{master} \cap (\textit{dom maynotupdate}) = \{\ \} \\
\\
\forall\ \textit{user: auth; elt: elements}\ . \\
\quad \textit{user}\ \underline{\textit{canretrieve}}\ \textit{elt} \Leftrightarrow \\
\qquad\quad \textit{user} = \textit{owner elt} \quad \vee \\
\qquad\quad (\textit{owner elt})\ \underline{\textit{delegates}}\ \textit{user} \quad \vee \\
\qquad\quad \textit{privacy elt} \leqslant \textit{privacy user} \\
\\
\forall\ \textit{user: auth; elt: El}\ . \\
\quad \textit{user}\ \underline{\textit{canupdate}}\ \textit{elt} \Leftrightarrow \\
\qquad\quad \textit{user} \in \{\textit{owner elt}\} \cup \textit{master} \quad \wedge \\
\qquad\quad \neg\, (\textit{user}\ \underline{\textit{maynotupdate}}\ \textit{elt})
\end{array} \\
\hline
\end{array}
$$

Fig. 1 Summary of the information structures of a DDS

whom an authority has delegated all its access rights is stored as the *retrieval* property of that authority element, and that the set of authorities which have been given the right to access an element by its owner is stored as that element's *retrieval* property. We will only be able to go into these details after revealing more of the internal structure of stored values, element identifiers, and property names in Section 3.3.

4.2 DDS dynamics: Part 1

In this section and its companion we describe the way in which certain commands affect DDS information structures. In fact there are two distinct kinds of DDS run: administrative runs during which administrative commands may be issued, and ordinary runs during which they may not. For the purposes of this note, the main difference is that certain kinds of element (in particular *AUTHORITY elements) may be inserted during administrative runs, but otherwise may not.

Although one might expect the administrator to be the same as the *master* authority, this turns out not to be so. For simplicity, therefore, we have avoided consideration of administrative commands and concentrated on the most important nonadministrative commands. Later we suggest a small simplification of the design which would avoid the distinction between administrative and ordinary runs.

4.2.1 State of a running DDS: In characterising the state of a running DDS we must account for the fact that users 'log in' under the aegis of an authority, which is deemed in our model to be the nil authority for those users who run under 'no authority'. In fact a password-based scheme is used to check that an individual has the right to log in as a particular authority, but the details of logging in are beyond the scope of this note.

Once the running authority is established, elements are processed by establishing the 'element context', i.e. the identifier of the element to which subsequent commands will refer. One thing which the documentation does not make quite clear is whether or not the element context identified by the user must always refer to an already defined element. This seems plausible, however, since there is a special element whose identifier is *null*; the *null* element context prevails at the beginning of a run, after certain classes of errors, and after the stop command has been issued. We therefore introduce a constant

$$\boxed{\;\mathit{null}\colon \mathit{EI}\;}$$

and characterise the *a* state of a running DDS by:

$$
\begin{array}{|l}
\hline
\textit{DDS} \\
\mathit{DD} \\
\mathit{user}\colon\ \mathit{EI} \\
\mathit{elementcontext}\colon\ \mathit{EI} \\
\hline
\mathit{user} \in \mathit{auth} \\
\mathit{null} \in \mathit{elements} \\
\forall\, a\colon \mathit{auth}\ .\ a\ \boldsymbol{\mathit{mayaccess}}\ \mathit{null} \\
\mathit{user}\ \boldsymbol{\mathit{canretrieve}}\ \mathit{elementcontext} \\
\hline
\end{array}
$$

To capture the effect of a command on a DDS, we relate the observations made before the command is performed (denoted by the unprimed names) to those which can be made afterwards (denoted by primed names). As is customary we factorise our description of the commands into those properties which all the commands have in common and those which are peculiar to particular commands.

The schema ΔDDS summarises the common characteristics of the effects of nonadministrative commands on a DDS: the set of declared authorities may not change, nor may the master authority, nor may the authority of the running user.

$$
\begin{array}{|l}
\hline
\Delta DDS \\
DDS \\
DDS' \\
\hline
auth' = auth \\
master' = master \\
user' = user \\
\hline
\end{array}
$$

Some commands change the element context, or just display parts of the stored dictionary, but don't change the information structures concerned with access control; their additional common properties are summarised in the schema $\square DDS$

$$
\begin{array}{|l}
\hline
\square DDS \\
\Delta DDS \\
\hline
owner' = owner \\
delegates' = delegates \\
privacy' = privacy \\
mayretrieve' = mayretrieve \\
maynotupdate' = maynotupdate \\
\hline
\end{array}
$$

4.2.2 Display command: The simplest command to describe is the DISPLAY command, which displays parts of the current element in a readable form. Below we just indicate that the user must supply some property names, and that on completion of the command he is presented (*readable'*) with the values of the required properties as stored for the element currently in context. We do not concern ourselves with the precise form in which the display is presented.

$$
\begin{array}{|l}
\hline
Display \\
properties: \mathbb{F}\ P \\
\square DDS \\
readable': P \mathrel{+\!\!\!\!\twoheadrightarrow} V \\
\hline
\#properties = 1 \vee properties = P \\
readable' = (store\ elementcontext) \upharpoonright properties \\
store' = store \\
\hline
\end{array}
$$

Those familiar with DDS should note that the ALLPROPERTIES variant of the DISPLAY command corresponds to *properties* = *P* whereas the *property-keyword variant corresponds to *#properties* = 1. To simplify our description we have not formalised either the PROMPTLIST or the EXPLOSION variants of this command. To do either would require a more detailed model than we have so far presented, although the detail is not complex.

4.2.3 Setting the element context: One command which plays at least three roles in the system is the FOR command: the three variants of which we are aware all set 'context' in one way or another, they are:

> FOR VERSION *<version element identifier>*
> FOR AUTHORITY *<authority element identifier>*
> FOR *<element identifier>*

Since we do not treat version control here, we shall not consider the first of these. The second corresponds to the beginning of a DDS run: it just sets the user authority after checking a password. The third command is used to set the current element context: it sets the null context if presented with an element name which the current authority is not allowed to retrieve, or one which has not yet been defined.

```
┌─FOR──────────────────────────────────
│ eid: EI
│ ΔDDS
├──────────────
│ store' = store
│
│ user canretrieve eid          ∧
│ eid ∈ elements                ∧
│ elementcontext' = eid
∨
│ eid ∉ elements                ∧
│ elementcontext' = null
∨
│ ¬ (user canretrieve eid)      ∧
│ elementcontext' = null
└──────────────────────────────────────
```

This concludes the account of the commands which have no effect on the stored information. Before describing the remaining commands it will be necessary to take a small detour and explain the relationship between the stored elements and the access-control information.

4.3 Representation of access-control information

In this section we formalise the fact that data dictionaries are self-describing. To put this another way, the abstract information structures which were introduced to explain the control of access to elements are actually represented by means of elements and properties stored in the dictionary.

In a DDS the properties named *AUTHORITY, *PRIVACY, and *RETRIEVAL are defined for most elements, and so distinct constants are introduced into the specification which will henceforth stand for these property names:

```
┌──────────────────────────────────────────────┐
│ authority, privacy, retrieval: P
├──────────────────────
│ authority ≠ privacy ≠ retrieval ≠ authority
└──────────────────────────────────────────────┘
```

Properties may take on a variety of values, but in the first part of this section we shall be concerned only with the numbers, $\mathbb{N}$, single element identifiers *EI* and finite sets of element identifiers $\mathbb{F}$ *EI.* The set of storable values *V* contains the disjoint union of these three sets

$$Number \ll \mathbb{N} \gg \mid Elements \ll EI \gg \mid Elements \ll \mathbb{F}\, EI \gg \subseteq V$$

Note: A detailed explanation of the notion of disjoint union is presented in Reference 4. For our purposes it is sufficient to understand that the above disjoint union contains one value for each number *n*: $\mathbb{N}$, one value for each single element identifier *ei: EI*, and one value for each set of element identifiers *eis:* $\mathbb{F}$ *EI*. These values are distinct and denoted respectively by the terms *Number n, Element ei,* and *Elements eis.*

All that now needs doing is to strengthen the invariant of the existing DDS description. In Fig. 2 the qualities inherent in the use of the term 'self-describing' are summarised.

```
┌─DD────────────────────────────────────────────────────────────────────
│ DD
│ has: EI ↔ P
├──────────────
│∀ ei: elements  .
│  ei has privacy          ∧    Number(privacy ei) =  store ei privacy   v
│ ¬(ei has privacy)        ∧    privacy ei = 0
│
│∀ a: auth  ..
│  a has retrieval         ∧    Elements (delagates ⟦ a ⟧ ) =  store a retrieval   v
│ ¬ (a has retrieval       ∧    delegates ⟦ a ⟧ ) =
│
│∀ ei: (elements  –  auth)   .
│  ei has retrieval        ∧    Elements (mayretrieve⁻¹ ⟦ ei ⟧ ) = store ei retrieval   v
│ ¬ (ei has retrieval)     ∧    mayretrieve⁻¹ ⟦ ei ⟧ =  {   }
│
│∀ ei: (elements –auth)   .
│  ei has authority        ∧    Element (owner ei) = store ei authority   v
│ ¬ (ei has authority)     ∧    owner ei = nil
│
│∀ ei: EI; p: P   .
│  ei has p ⇔ ei ∈ elements ∧ p ∈  dom(store ei)
└───────────────────────────────────────────────────────────────────────
```

Fig. 2 Data dictionaries are self-describing

To simplify our account of the default values provided by the system an additional observaton is introduced: the relationship *has*, which holds between an element identifier *ei* and a property name *p* exactly when *ei* has been stored with a property named *p*. The first additional predicate records the fact that the privacy level of each stored element is represented by the numeric value of its privacy property. The second predicate records that the retrieval property of each author element represents the set of authorities who stand in the relationship of *delegates* to it. The third predicate records that the value of the retrieval property of each nonauthority element represents the set of authorities which may retrieve it, and the fourth predicate that its authority property represents its owner. Notice the different interpretations given to the *retrieval* property of an authority element and the same property of a nonauthority element.

Hitherto we have given a rather abstract characterisation of the information structure which prevents certain authorities updating certain types of element, but we are now in a position to give a fuller account. To do so we need to examine the structure of the space of element identifiers (*EI*) in a little more detail. In fact this space is two-dimensional; an element is identified by an element-type-identifier (known in the documentation as an element keyword) and a within-type identifier. If we let *K* denote the set of element type identifiers, and *I* denote the set of within-type identifiers, then we can define

$$EI \triangleq K \times I$$

Note: All elements with the same type have property names drawn from the same set of names; these are stored as the *SYSTEM-PROPERTIES and *USER-PROPERTIES properties of the ELEMENT element which describes the type. A complete formalisation of this is possible within the framework we have already established, but goes beyond the scope of this paper.

An authority is prevented from updating certain types of element if the element which describes it has a property called *INHIBIT. The value of this property is the set of element type identifiers to which the authority is denied update rights – despite any retrieval rights it may have. First another constant is introduced:

$$\begin{array}{|l}
\textit{inhibit: P} \\
\hline
\textit{inhibit} \notin \{\textit{authority, retrieval, privacy}\}
\end{array}$$

and indicate that sets of element type identifiers may also be stored:

$$\textit{Types} \ll \mathbb{F}\, K \gg \subseteq V$$

then strengthen the *DD* invariant a little further:

$$\begin{array}{|l}
\textit{DD} \\
\hline
\textit{DD} \\
\hline
\forall\ a\!: \textit{auth};\ et\!: K;\ i : I\ . \\
\quad a\ \textbf{\textit{maynotupdate}}\ (et, i) \Leftrightarrow \\
\qquad a\ \textbf{\textit{has inhibit}} \wedge et \in \textit{Types}^{-1}\ (\textit{store a inhibit})
\end{array}$$

Note: The obscurely formulated predicate $et \in Types^{-1}$ (*store a inhibit*) means: 'the type keyword of the element identifier is one of the set of keywords which are stored as the *INHIBIT property of the authority'.

Our account of self-description is now as detailed as it needs to be for the purposes of this note. Interested readers may care to take the account further and formalise the fact that details of the properties possessed by elements of each type are recorded in the database, as are details of the representation of each property.

As a hint we will show how the authority elements in the dictionary are identified. First introduce the constant:

$$\boxed{AUTHORITY : K}$$

to denote the AUTHORITY element keyword. All that remains is to strengthen the *DD* invariant yet again.

$$\begin{array}{|l}
\text{— } DD \text{ —} \\
DD \\
\hline
auth = \{(k, i) : elements \mid k = AUTHORITY\} \\
\hline
\end{array}$$

In other words, the authority elements are exactly those whose keyword is *AUTHORITY.*

4.4 DDS dynamics: Part 2

In this section the description of the DDS is completed with a formalisation of the behaviour of the INSERT and DELETE commands. The REPLACE command is simply a combination of DELETE followed by INSERT, so we leave its formalisation as an exercise for interested readers. The current formalisation is partial, in the sense that it accounts only for the behaviour of successful commands. Although behaviour in the case of erroneous commands can easily be described within the present framework, doing so would not be particularly useful, especially in view of the simplifications we have made (Appendix 1).

4.4.1 Inserting elements: Judging by its documentation, the INSERT command appears to have two variants. The first takes a new element identifier and a set of property-name, property-value pairs – in other words, an element – and stores the element as the value of the identifier. If no *authority* property is given, then the owner of the new element will be the current user; if no *privacy* property is given, then the element will be given the privacy level of the current user. The command sets the current element context.

```
┌── InsertNewElement ──────────────────────────────────
│  eid            EI
│  newelement:    P ⇸ V
│  ΔDDS
├──────────────────
│  eid ∉ dom store
│  store' = store ⊕ {eid → newelement'}
│  elementcontext' = eid
where
│  newelement' = {authority ↦ user; privacy ↦ privacy user} ⊕ newelement
└──────────────────────────────────────────────────────
```

This description is such that if the *privacy* and *authority* properties are specified in such a way that *elementcontext'* is no longer accessible, then the insert operation will fail (the last invariant of *DDS* will not be satisfied). It seems to indicate that a user running under one authority can add an element to the database but give ownership rights to another authority. While this was rather difficult to rationalise, we could not discover anything in the documentation which forbids it. On asking the designers what really happens – an option which might not be open to the average DDS user – we discovered that only the master authority can give ownership rights to another authority when creating a new element. When the current user is not the master and *authority* and *privacy* properties are supplied, then they must be the ones which the system would provide by default anyway. Formalised concisely we have:

```
┌── InsertNewElement ──────────────────────
│  InsertNewElement
├──────────────────
│  user ∉ master ⇒
│       newelement ⟦ authority ⟧ ⊆ {user}   ∧
│       newelement ⟦ privacy   ⟧ ⊆ {privacy user}
└──────────────────────────────────────────
```

The second variant of the insert command takes a set of property-name, property-value pairs and incorporates them into the current element, providing that it has no existing property with one of the names supplied.

```
┌── InsertNewProperties ───────────────────
│  newprops: P ⇸ V
│  ΔDDS
├──────────────────
│  user canupdate elementcontext
│  ¬(∃ p: (dom elt) . elementcontext has p)
│  store' = store ⊕ {element ↦ ((store element) ∪ newprops)}
│  elementcontext' = elementcontext
└──────────────────────────────────────────
```

The documentation does not make it clear whether or not administrative properties may be added to an element by authorities other than its owner once it has been inserted. On asking the designers, we discovered that the only administrative property for which the description given above fails to account is the *authority* property: the master authority can give ownership of an unowned element to any authority, but nonmaster authorities can only take the ownership of such elements for themselves. More formally:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{InsertNewProperties}} \\
\textit{InsertNewProperties} \\
\hline
\textit{authority} \in \textit{dom newprops} \Rightarrow \\
\quad \textit{user} \in \textit{master} \vee \\
\quad \textit{newprops authority} = \textit{user}
\end{array}$$

4.4.2 Deleting elements: DELETE appears in two variants: in the first, the user explicitly mentions an element for the command to delete:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{DeleteElement}} \\
\textit{eid: EI} \\
\Delta \textit{DDS} \\
\hline
\textit{user canupdate eid} \\
\textit{store}' = \textit{store} \setminus \{\textit{eid}\} \\
\textit{elementcontext}' = \textit{null}
\end{array}$$

In its second form the user mentions some properties to be removed from the current element. Unfortunately the documentation does not make it clear whether or not users may delete administrative properties from elements to which they have update rights, nor is it quite clear what happens if the last remaining property of an element is deleted. At first we assumed, albeit uneasily, that administrative properties could be deleted and that elements with no properties could remain in the store, so to that extent the formalisation was inaccurate. Discussions with the implementation team proved my unease to be well founded; we learned that if administrative properties are deleted from an element by the user then they revert to the default values which the system would have provided if the element had just been inserted.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{DeleteProperties}} \\
\textit{props: } \mathbb{F}\ P \\
\Delta \textit{DDS} \\
\hline
\textit{user canupdate elementcontext} \\
\textit{store}' = \textit{store} \oplus \{\textit{elementcontext} \mapsto \textit{element}'\} \\
\textit{elementcontext}' = \textit{elementcontext} \\
\textbf{\textit{where}} \\
\textit{element}' = (\textit{store elementcontext}) \setminus \textit{props} \oplus_{1} \\
\qquad\qquad \{\textit{authority} \mapsto \textit{user}; \textit{privacy} \mapsto \textit{privacy user}\}
\end{array}$$

5 Prospects

While we would have liked to go on to describe the control of multiple versions in DDS, the present design proved too hard to formalise simply. The specification therefore has its limitations and it would be unwise of users to rely upon formal deductions from it to discover the consequences of actions they might take while running the system itself. It nevertheless remains useful as a pedagogical tool because it provides a discursive introduction to the concepts which underlie access control.

In our view the principal benefit of constructing the formal specification is the fact that a framework has been developed within which designs of future dictionaries can easily be investigated. Whereas it has been an interesting challenge to build a mathematical model of a software system such as the Data Dictionary System, the enterprise would remain simply an academic exercise if it were to stop at this point, so we have tried to indicate how to use the framework by using it to make a tentative proposal for simplifying the system. This is presented in Appendix 1.

Acknowledgments

ICL sponsored the ten-day pilot experiment in technology transfer which led (*inter alia*) to the production of this note. It has been a pleasure to work with Roger Stokes who despite the multiplicity of demands imposed on his time and talents, always remained interested enough in the experiment to convince me that it was worthwhile.

References

1 HOARE, C.A.R.: 'Programming is an engineering profession', Technical Monograph 27, Programming Research Group, Oxford, 1982.
2 JONES, C.B.: '*Software development: A rigorous approach*', Prentice-Hall International, 1980.
3 Reference manual for the ICL Data Dictionary System (DDS.600) ICL Document RPO120, May 1982.
4 SUFRIN, B.: 'Mathematics for system specification, Computation MSc Course Notes, Programming Research Group, Oxford, 1983.
5 SUFRIN, B.: 'Notes for a Z handbook. Part 1: the mathematical language', Software Engineering Working Paper, Programming Research Group, Oxford, 1984.
6 MORGAN, C. and SUFRIN, B.: 'Specification of the Unix filing system', *IEEE Trans.*, 1984, SE-10, (2).

Appendix 1
Potential simplifications

Those familiar with DDS will have noticed that an important simplification has been made already, by ignoring the 'facility' to refer to as-yet-underfined authorities when adding or modifying properties. Although we have no definite knowledge about the operational consequences of this facility, we hazard a guess that it causes more aggravation than it saves: readers who have been victims of implicit declarations in Fortran may care to comment on this.

The most obvious additional simplification would be to drop the independent notion of privacy level, which seems to be orthogonal to authorities and ownership. We are tempted to wonder if there are any DDS installations where both privacy and authority are employed within the same dictionary.

A further simplification would be to remove the distinction between the system administrator and other authorities. This might well pay dividends in terms of enhancing the functionality of the system and reducing the complexity of its documentation and implementation. Our design goal is based on a new interpretation of the meaning of an authority element, which we prefer to think of as a role, or locus of responsibility, rather than a particular person. Indeed it is often the case that one individual plays several distinct roles in an organisation.

In the design outlined below every authority is made subordinate to ('owned by') some other authority; the root of this tree of authorities is the system administration authority (which owns itself). Power to alter properties of elements reposes ultimately in the administrator, which is able to delegate them to subordinate authorities, which in turn can delegate them further if need be. Any element which several authorities need to retrieve or to update should be owned by an authority which is higher in the tree than all of them and which delegates its retrieval or update rights to them all.

To formalise this design, we first need to introduce the idea of a 'loop-free' function, sometimes called a 'tree'. Consider a homogeneous function

$$f: X \nrightarrow X$$

We say that an element $x' : X$ is *reachable via f* from an element $x : X$ if there is at least one non-zero number, $n: \mathbb{N}_1$ for which $x' = f^n\, x$. When this is the case, we write

$$x f^+ x'$$

More formally, we can define:

$$\begin{array}{|l}
\_^+ : (X \nrightarrow X) \rightarrow (X \nrightarrow X) \\
\hline
\forall f : X \nrightarrow X;\ x, x' : X\ . \\
\quad x\, f^+ x' \Leftrightarrow \exists n : \mathbb{N}_1\ .\ x' = f^n\, x \\
\hline
\end{array}$$

A function f: $X \nrightarrow X$ is said to be *loop-free*, or a *tree*, if there is no x: X which reachable from itself via f. More formally:

$$\begin{array}{l}
[X] \\
\quad Tree \mathrel{\hat=} \{ f : X \nrightarrow X \mid \neg\ (\exists x : X\ .\ x f^+ x) \}
\end{array}$$

Our first approximation to a description of the design outlined above recalls the description of the standard data dictionary: the main difference is that all elements (including authority elements) are owned, and that if we confine our attention to authorities other than the administrator, the ownership function is a tree.

The administrator is reachable from every element via the ownership function, i.e. the administrator is ultimately responsible for everything in the dictionary.

$$
\begin{array}{|l}
\hline
\textit{DD} \\
\textit{store}: \quad EI \twoheadrightarrow E \\
\textit{elements}: \quad \mathbb{F}\, EI \\
\textit{auth}: \quad \mathbb{F}\, EI \\
\textit{owner}: \quad EI \twoheadrightarrow EI \\
\textit{admin}: \quad EI \\
\hline
\textit{elements} = \operatorname{dom} \textit{store} \\
\operatorname{dom} \textit{owner} = \textit{elements}: \\
\operatorname{ran} \textit{owner} \subseteq \textit{auth} \subseteq \textit{elements} \\
\textit{admin} \in \textit{auth} \\
\textit{owner} \setminus \{\textit{admin}\} \in \textit{Tree}[EI] \\
\forall\, ei: \textit{elements} \,.\; ei \; \textit{owner}^{+} \; \textit{admin} \\
\hline
\end{array}
$$

The information structures from which the relations *canretrieve* and *canupdate* will be derived are similar to those in the original design, except that there is no longer a role for privacy levels, the *nil* authority no longer exists, and update permission is characterised positively rather than negatively.

$$
\begin{array}{|l}
\hline
\textit{DD} \\
\textit{DD} \\
\textit{delegates}: \quad EI \leftrightarrow EI \\
\textit{mayretrieve}: \quad EI \leftrightarrow EI \\
\textit{mayupdate}: \quad EI \leftrightarrow EI \\
\hline
\textit{delegates} \in \textit{auth} \leftrightarrow \textit{auth} \\
\operatorname{dom} \textit{mayretrieve} \subseteq \textit{auth} \\
\operatorname{dom} \textit{mayupdate} \subseteq \textit{auth} \\
\textit{delegates} \subseteq \textit{owner}^{-1} \\
\hline
\end{array}
$$

The last predicate states that an authority may only delegate its rights to authorities for which it is responsible.

An authority can retrieve an element if it or any of its subordinates own the element, or if it has been given explicit permission to retrieve it, or if it has been delegated rights to retrieve the element. An authority can update an element if it or any of its subordinates owns the element, or if it has been given permission to update the element.

$$
\begin{array}{l}
\textbf{DD} \\
\hline
DD \\
canretrieve: EI \leftrightarrow EI \\
canupdate: EI \leftrightarrow EI \\
\hline
\mathrm{dom}\ canretrieve \subseteq auth \\
\mathrm{dom}\ canupdate \subseteq auth \\
\\
\forall\ user: auth;\ elt: elements\ . \\
\quad user\ \mathbf{canretrieve}\ elt \Leftrightarrow \\
\qquad\quad elt\ \mathbf{owner}^{+}\ user \qquad\qquad \vee \\
\qquad\quad (owner\ elt)\ \mathbf{delegates}\ user\ \vee \\
\qquad\quad user\ \mathbf{mayretrieve}\ elt \\
\\
\forall\ user: auth;\ elt: elements\ . \\
\quad user\ canupdate\ elt \Leftrightarrow \\
\qquad\quad user\ \mathbf{canretrieve}\ elt \qquad \wedge \\
\qquad\quad elt\ \mathbf{owner}^{+}\ user \qquad\qquad \vee \\
\qquad\quad user\ \mathbf{mayupdate}\ elt
\end{array}
$$

It might be nice if no authority possessed any capabilities that its owner does not also possess. In other words, if within a dictionary:

$$
\begin{array}{|l}
canretrieve \subseteq owner\ ;\ canretrieve \\
canupdate \subseteq owner\ ;\ canupdate
\end{array}
$$

Interested readers may care to check whether or not this is the case, and if not, to modify my formalisation so that it is.

Finally we propose a small project for the interested reader. Devise representations (along the lines suggested by Fig. 2) for the abstract information structures *mayretrieve*, *mayupdate*, and *delegates*. These should make the relations *canretrieve* and *canupdate* simple to compute, and also make the system invariant simple to check.

Appendix 2
Glossary

The specification notation Z, based on set theory[4,5,6], is used throughout the paper.

$\_ \triangleq \_$	syntactic equivalence	_ is defined to be _
$\wedge$	logical conjunction	and
$\vee$	logical disjunction	or
$\_ \Rightarrow \_$	logical implication	if _ then _

$\_ \Leftrightarrow \_$	logical equivalence	_ if and only if _
$\forall$	universal quantification	for all ...
$\exists$	existential quantification	for some ...
$\mathbb{N}$	the natural numbers (non-negative integers)	
$\mathbb{N}_1$	the strictly positive natural numbers.	
$m..n$	the numbers between m and n	$m..n = \{i : \mathbb{N} \mid m \leqslant i \leqslant n\}$
$\mathbb{N}^{99}$	0..99	
$\subseteq$	set inclusion	
$\subset$	strict set inclusion	
$\{\}$	empty set	
$-$	set difference	
$\mathbb{P}\,S$	all subsets of S	$ss \in \mathbb{P}S \Leftrightarrow ss \subseteq S$)
$\mathbb{P}_1\,S$	nonempty subsets of S	
$\mathbb{F}\,S$	finite subsets of S	
$\mathbb{F}_1\,S$	nonempty finite subsets of S	
$\#\_$	cardinality (number of elements of a finite set)	
$\mathbb{F}^1\,S$	Subsets of S with no more than one element	$\mathbb{F}^1\,S \mathrel{\hat=} \{ss : \mathbb{F}\,S \mid \#ss \leqslant 1\}$
$\cup$	set union	
$\cap$	set intersection	
$\in$	set membership	
(a, b)	ordered pair a b	
$\{s{:}S \mid pred\}$	set comprehension: the set of elements s of S which satisfy *pred*	
$\{\,term\,.\,s{:}S \mid pred\}$	term comprehension: the set of *term* generated by the elements s of S which satisfy the predicate *pred*.	
$S \leftrightarrow T$	the set of binary relations between S and T	$S \leftrightarrow T \mathrel{\hat=} \mathbb{P}\,(S \times T)$
$S \rightarrowtail T$	the set of partial functions from S to T	$S \rightarrowtail T \subseteq S \leftrightarrow T$
$S \twoheadrightarrow T$	the finite functions (mappings) from S to T	$S \twoheadrightarrow T \subseteq S \leftrightarrow T$
$\{a \mapsto b\}$	the mapping $\{(a, b)\}$ which takes a to b	
$a\,R\,b$	relationship R holds between a and b	$a\,R\,b \mathrel{\hat=} (a, b) \in R$
$f\,x$	application of f to x	$(x, f\,x) \in f$
dom	the domain of a relation or function for $R : S \leftrightarrow T$	$dom\,R \mathrel{\hat=} \{s{:}S \mid (\exists t{:}T\,.\,sRt)\}$

ran	the range of a relation or function	
	for $R{:}S \leftrightarrow T$	$ran\ R \mathrel{\hat{=}} \{t{:}T \mid (\exists s{:}S\ .\ sRt)\}$
R^{-1}	inverse of a function or relationship	$s\ R\ t \Leftrightarrow t\ R^{-1}\ s$
$[\![\ ]\!]$	image:	
	for $R{:}\ S \leftrightarrow T; SS{:}\ \mathbb{P}S$	$R[\![\ SS\ ]\!] = \{t{:}T \mid (\exists s{:}SS\ .\ sRt)\}$
	for $R{:}\ S \leftrightarrow T; s{:}\ S$	$R[\![\ s\ ]\!] \mathrel{\hat{=}} R[\![\ \{s\}\ ]\!]$
$\backslash$	domain restriction of a function or relationship	
	for $R{:}\ S \leftrightarrow T; SS{:}\ \mathbb{P}S$	$s\ R \backslash SS\ t \Leftrightarrow sRt \wedge s \notin SS$
$\restriction$	domain restriction of a function or relationship	
	for $R{:}\ S \leftrightarrow T; SS{:}\ \mathbb{P}S$	$s\ R \restriction SS\ t \Leftrightarrow sRt \wedge s \in SS$
$\oplus$	over-riding of relationships or functions	
	for $f, g{:}\ S \leftrightarrow T$	$f \oplus g = f \backslash (dom\ g) \cup g$
	for functions $f, g{:}\ S \nrightarrow T$	$x \in dom\ g \Rightarrow f \oplus g\ x = g\ x;$
		$x \in (dom\ f) - (dom\ g) \Rightarrow f \oplus gx = f\ x$
;	forward relational (or functional) composition	
$\circ$	relational (or functional) composition. (NB: $f\,;g \mathrel{\hat{=}} g \circ f$)	
f^N	N-fold composition. $f^0 = id; f^{(n+1)} = f \circ f^n$	

Notes on the authors

Dr. P.M. Flanders Sorting on DAP

Dr. P.M. Flanders received the B.Sc. degree in Physics and the Ph.D. degree in Computer Science from Queen Mary College, University of London. He has since been engaged in research and advanced development at the ICL Systems Strategy Centre in Stevenage, where his main interests have been in high-level language machines, language design and parallel processing. Much of this work has been centred on the ICL DAP and includes the design of DAP-Fortran and methodologies for using array processors.

D.J. Hunt Tracking of LSI chips and printed circuit boards using the ICL Distributed Array Processor

David Hunt read Mathematics and Physics at Trinity College, Cambridge, where he was a senior scholar. He has worked for ICL ever since graduating in 1968 and is a Senior Research Consultant at the Systems Strategy Centre. He joined the DAP project at its inception, and has been involved with most aspects, but especially hardware and applications.

Dr. R.W. Jones User functions for the generation and distribution of encipherment keys

Roy Jones is a security consultant in ICL's Defence Technology Centre. He has been concerned with secure systems and the use of encipherment in computer system architecture since 1975, first as a consultant working for CCTA and then, since 1977, within ICL. He is a member of the BSI committee 0IS21 whose task is to produce standards for the use of encipherment and of the corresponding international committee ISO/SC20.

P. Mellor Analysis of software failure data: 1 adaptation of Littlewood stochastic reliability growth model for coarse data

Peter Mellor joined ICL in 1968 after leaving Cambridge where he studied Mathematics. He has worked for ICL ever since, man and boy, apart from a short time out in the cold hard world of commercial programming. For several years he worked in software development, and later in support, and designed and wrote databases for incident report management, an expert system for software fault diagnosis, and remote diagnostic tools. This led to involvement in life-cycle costing of software support and to his present interest in the estimation and prediction of software reliability. At present he is employed within the Group Quality organisation.

Dr. S.F. Reddaway Sorting on DAP

Dr. Reddaway graduated from Cambridge, and after doing a Ph.D. at Durham joined English Electric Computers in 1965. He progressed from hardware technology to novel architectures, and since originating the DAP he has for the last 10 years managed an array processing research group. This group built the pilot DAP system and then co-operated on the first generation DAP product. In 1981 work was redirected to smaller second generation DAPs. A major activity has been applications analysis for array processing, recently mostly in signal and image processing. He is with the Systems Strategy Centre at Stevenage, and is working on ideas for a third generation.

B. Sufrin Towards a formal specification of the ICL Data Dictionary

Bernard Sufrin has a degree in Mathematics from the University of Sheffield and a degree in Computer Science from the University of Essex. He joined the Programming Research Group at Oxford University in 1978, previously having been Chief Research Officer in Computer Science at Essex University, Research Fellow at Bolt Beranek and Newman Inc. and Senior Research Officer in Computer Science at Essex. His research for the past six years has focused on making practical use of mathematics in the description and development of computer systems, and he has lectured extensively on this topic. He holds a Fellowship at Worcester College and is University Lecturer in Computation.

Dr. R.H. Thompson Modelling a multiprocessor designed for telecommunications systems control

Dr. Thompson graduated from Liverpool University in 1966, and, after some years at the Corporate Research Laboratories of ICI, he went to the Control Systems Centre at UMIST (University of Manchester Institute of Science and Technology). Having graduated from the University of Manchester with a Ph.D. (in 1973) he joined the Teletraffic Division of British Telecom, where he is currently engaged in studying the design and performance of processor-controlled telecommunications exchanges.

Museum and archive for the history of the computer

There is general agreement with the view that the electronic digital computer is one of the most important inventions of mankind, possibly the most important. It has been developed, and continues to develop, with extraordinary speed, its history extending little beyond the career span of individuals still active or recently retired; and Britain has made contributions of fundamental importance to this development. While a certain amount of historical material is preserved in museums and other collections, there is a serious risk that unless steps are taken to capture the documents, artefacts and personal experiences from this recent past they will be irretrievably lost or dissipated. This would be to the disadvantage both of scientific historians and of present-day workers in the field, and also would lose an opportunity to interest and increase the awareness of the public in a field which is becoming increasingly important in their everyday work and leisure activities.

At a meeting arranged recently by ICL to discuss this situation, attended by representatives of the computer industry and profession and of a number of relevant public bodies, there was strong and unanimous support for the general idea of a British national museum and archive of computer history. The meeting resulted in an agreed aim to formulate a specific proposal as quickly as possible, probably by early 1985. Meanwhile, all the participants stressed the seriousness of the risk of loss or destruction of potentially valuable material. This note, therefore, is being circulated widely, to ask anyone, whether as an individual or as a member of a commercial, industrial or other organisation, who has material which might be of historical value, to preserve this and to send a descriptive note – as brief as possible – to

Dr. J. Howlett
International Computers Limited
ICL House
Putney
London SW15 1SW
England

'Potentially valuable material' could include, for example, handbooks, logic diagrams, house journals, and personal writings such as notebooks and correspondence; also historic items of equipments and other artefacts. It can be assumed that copies of all open publications, such as learned and professional societies' journals, are already available. If there is a risk of destruction of material because of lack of storage space, ICL may be able to arrange temporary storage for limited amounts.